DB2 Information Management Software

IBM

DB2 UDB for z/OS: Application Design for High Performance and Availability

- Develop applications for concurrency, performance, and availability
- Learn useful programming techniques
- Check your table and index design

Paolo Bruni
Patric Becker
Jan Henderyckx
Joseph Link
Bart Steegmans

Archived

Redbooks

ibm.com/redbooks

Archived

International Technical Support Organization

DB2 UDB for z/OS: Application Design for High Performance and Availability

December 2005

Archived

SG24-7134-00

Note: Before using this information and the product it supports, read the information in “Notices” on page xxi.

Archived

First Edition (December 2005)

This edition applies to Version 8 of IBM Database 2 Universal Database for z/OS (DB2 UDB for z/OS), program number 5625-DB2.

© Copyright International Business Machines Corporation 2005. All rights reserved.
Note to U.S. Government Users Restricted Rights -- Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Figures xi

Tables xv

Examples xvii

Notices xxi
Trademarks xxii

Preface xxiii
The design challenge xxiii
You are not alone in the system xxiv
Technology is always improving xxiv
Recommended Service Update xxv
Assumptions xxv
Contents xxvi
The team that wrote this redbook xxvii
Become a published author xxix
Comments welcome xxix

Part 1. Database design 1

Chapter 1. Table design 3
1.1 Implementing the logical model 4
1.1.1 Domains and data types 4
1.1.2 Nullability 13
1.1.3 The effect of column placement on performance 18
1.1.4 Key design 18
1.1.5 Choosing an encoding scheme 23
1.1.6 Volatile tables 23
1.2 Deviating from the third normal form 24
1.2.1 Adding columns or tables to support application requirements 24
1.2.2 Aggregation 27
1.2.3 Normalization and denormalization 34
1.3 Partitioning 43
1.3.1 Benefits of partitioning 44
1.3.2 Partitioning considerations 44
1.3.3 Identifying and evaluating candidate partitioning key 45
1.3.4 Determining the number of partitions 50
1.4 Using DB2 compression 51
1.4.1 When should you use DB2 compression? 51
1.4.2 When does the compression dictionary get built? 53
1.4.3 When should the compression dictionary be rebuilt? 54
1.5 Choosing a page size 55
1.5.1 Maximum record size 55
1.5.2 Rows per page 55
1.5.3 Using page size to optimize storage 56
1.5.4 Other page size considerations 58

Chapter 2. Index design 59

© Copyright IBM Corp. 2005. All rights reserved.

2.1 What is an index? . . . 60
2.2 Why build indexes? . . . 61
2.2.1 Uniqueness . . . 61
2.2.2 Improve performance . . . 61
2.2.3 Index design for concurrency . . . 62
2.3 Cost of an index . . . 63
2.4 How does DB2 use the index? . . . 65
2.4.1 (Matching) indexable predicates . . . 65
2.4.2 Index screening predicates . . . 66
2.4.3 List prefetch . . . 67
2.4.4 Sort avoidance . . . 69
2.4.5 Index-only access . . . 71
2.5 Will DB2 use the index? . . . 71
2.6 Which indexes to build . . . 72
2.6.1 General index design guidelines . . . 72
2.6.2 Must build indexes . . . 74
2.6.3 Highly recommended index candidates . . . 74
2.6.4 Indexes that require analysis . . . 75
2.6.5 A multi-column index or multiple indexes . . . 76
2.6.6 Index considerations for join operations . . . 79
2.6.7 Less attractive index candidates . . . 80
2.6.8 UNIQUE WHERE NOT NULL . . . 83
2.6.9 NOT PADDED and PADDED indexes . . . 83
2.6.10 Indexing long columns . . . 84
2.7 Clustering index . . . 88
2.7.1 What is a clustering index? . . . 88
2.7.2 Choosing the clustering index . . . 90
2.7.3 Clustering and the impact on asynchronous I/O . . . 95
2.7.4 Clustering and the impact on other indexes . . . 95
2.7.5 Clustering and the impact on synchronous I/O . . . 95
2.7.6 Clustering and the impact on join access . . . 95
2.7.7 Changing a clustering index . . . 97
2.8 Indexing partitioned tables . . . 98
2.8.1 Partitioned partitioning index . . . 98
2.8.2 Nonpartitioned indexes . . . 98
2.8.3 Data-partitioned secondary indexes . . . 100
2.8.4 Page range screening . . . 101
2.8.5 Data-partitioned secondary indexes at work . . . 103

Chapter 3. Model validation . . . 109
3.1 Defining process requirements and service level agreements . . . 110
3.1.1 Reasons to estimate . . . 110
3.2 Determining the size of your objects . . . 111
3.3 Determining the duration of your process . . . 113
3.3.1 Elements of the estimation . . . 113
3.3.2 Based on previous measurements . . . 114
3.3.3 Estimating without tools . . . 114
3.3.4 Quick upper bound estimate . . . 120
3.4 Impact of the access path and table design on cost . . . 121

Part 2. Application enabling components . . . 127

Chapter 4. Constraints . . . 129
4.1 Uniqueness constraints . . . 130

4.2 Referential constraints . . . 130
4.2.1 To enforce or not . . . 131
4.2.2 RI enforced by DB2 or enforced by application . . . 132
4.2.3 Informational referential constraints . . . 133
4.3 Check constraints . . . 134
4.3.1 Check constraint benefits . . . 134
4.3.2 When to avoid check constraints . . . 134
4.3.3 Check constraints as an option to implementing referential integrity . . . 135
4.4 Code and look-up tables . . . 135
4.4.1 RI considerations . . . 136
4.4.2 Code table alternatives . . . 136

Chapter 5. Techniques to generate numbers . . . 137
5.1 Introduction . . . 138
5.2 Generating nonunique ordered numbers . . . 138
5.3 Generating unique not ordered nonconsecutive numbers . . . 139
5.3.1 GENERATE_UNIQUE() . . . 139
5.3.2 ROWID . . . 139
5.4 Generating unique ordered nonconsecutive numbers . . . 140
5.4.1 Using CURRENT TIMESTAMP . . . 141
5.4.2 Identity columns and sequences . . . 141
5.5 Generating consecutive numbers . . . 144
5.5.1 Using the MAX() function . . . 145
5.5.2 Using a single row table . . . 145
5.5.3 Handling multiple counters . . . 147
5.5.4 Using direct row access to access the counter table . . . 148
5.5.5 Handling consecutive numbers with very high transaction rates . . . 148
5.6 Generating numbers without using DB2 . . . 148

Chapter 6. Database side programming . . . 151
6.1 Overview of server side programming options . . . 152
6.2 Using stored procedures . . . 153
6.2.1 Why use stored procedures? . . . 153
6.2.2 What flavors can we use? . . . 156
6.2.3 Good housekeeping for stored procedures . . . 156
6.2.4 Performance expectations and tips . . . 158
6.3 Using user-defined functions . . . 162
6.3.1 Why use user-defined functions? . . . 162
6.3.2 What flavors can we use? . . . 162
6.3.3 Performance expectations and tips . . . 167
6.4 MQListener in DB2 for OS/390 and z/OS . . . 168
6.5 Using triggers . . . 168
6.5.1 Why use triggers? . . . 168
6.5.2 What flavors can we use? . . . 169
6.5.3 Good housekeeping for triggers . . . 169
6.5.4 Performance expectations and tips . . . 169

Chapter 7. XML support . . . 177
7.1 Introduction . . . 178
7.2 Storing XML intact . . . 179
7.2.1 Store intact with no need to process information contained in the XML . . . 179
7.2.2 Store intact, but process information contained in the XML documents . . . 179
7.3 Decomposing or shredding XML documents . . . 181
7.4 XML publishing . . . 183

Archived

7.5 The future 186

Part 3. Embedding SQL in the application 187

Chapter 8. SQL fundamentals 189
8.1 Number of SQL calls 191
8.2 Number of rows searched 192
8.3 SELECT column list 193
8.4 ORDER BY clause 193
8.4.1 ORDER BY sort avoidance 194
8.5 Subquery to join transformation 195
8.6 Indexable, stage 1, and stage 2 predicates 196
8.6.1 Mismatched numeric types 198
8.6.2 Mismatched string types 199
8.6.3 Performance 200
8.6.4 Recommendations 201
8.7 Considerations on the CAST function 201
8.8 Selective predicates 201
8.9 Subqueries 202
8.10 QUERYNO in SQL statements 202
8.11 Reduce trips to DB2 203
8.11.1 OPTIMIZE FOR *n* ROWS 203
8.11.2 FETCH FIRST *n* ROWS ONLY 204
8.11.3 Existence checking 204
8.12 Cursor versus searched update and delete 205
8.12.1 SQL precompiler option NOFOR 207
8.13 COMMIT frequency 208
8.13.1 SAVEPOINTs 210
8.14 Temporary tables 210
8.15 Retrieving values generated at insert 212
8.16 Current statistics 213

Chapter 9. Programming templates 215
9.1 Cursor versus singleton select 216
9.2 UPDATE and DELETE using cursors 217
9.3 Browsing techniques 217
9.3.1 Integrity of displayed data 218
9.3.2 Performance 218
9.3.3 A browsing technique solution using DB2 repositioning 222
9.3.4 Backward browsing 226
9.3.5 Considerations for browsing on clients 226
9.4 Advanced inquiry techniques 227
9.4.1 Many criteria available, but only a few selected 227
9.4.2 Predicates and repositioning columns do not match 230
9.5 Scrollable cursors 232
9.5.1 Static scrollable cursors 233
9.5.2 Dynamic scrollable cursors 234
9.5.3 FETCHing options for scrollable cursors 235
9.5.4 Updating using static scrollable cursors 237
9.5.5 Change of underlying data for scrollable cursors 238
9.5.6 SQLCODEs for scrollable cursors 238
9.5.7 Summary of scrollable cursors 239
9.6 Using multi-row FETCH 239

9.6.1 Application preparation . . . 240
9.6.2 Determining the result set . . . 242
9.6.3 Application code changes . . . 242
9.6.4 Multi-row update and multi-row delete . . . 244
9.7 Data integrity . . . 245
9.7.1 Concurrent updates . . . 245
9.7.2 Last update column . . . 246
9.7.3 Work in progress tables . . . 249
9.8 Intelligent exception handling . . . 251
9.8.1 Using DSNTIAR . . . 251
9.8.2 GET DIAGNOSTICS . . . 252
9.8.3 Retry logic . . . 256
9.8.4 Error handling control table . . . 258
9.8.5 Determining the correct load module . . . 259
9.9 Data access topics . . . 260
9.9.1 Restricting an application's data access . . . 260
9.9.2 Applications to switch between isolation levels . . . 260

Chapter 10. Advanced programming techniques . . . 263
10.1 Massive insert, update, and delete . . . 264
10.1.1 Insert considerations . . . 264
10.1.2 Multi-row insert . . . 266
10.1.3 LOAD SHRLEVEL CHANGE . . . 270
10.1.4 LOAD . . . 270
10.2 Delete considerations . . . 272
10.2.1 Mass delete . . . 272
10.2.2 Deleting only qualifying rows . . . 273
10.2.3 Update considerations . . . 275
10.3 Special data accesses . . . 276
10.3.1 Data access using ROWID . . . 276
10.3.2 Never-ending update . . . 277
10.4 Optimizing repeat processes . . . 278
10.4.1 Prefetch mechanisms . . . 278
10.4.2 Split programs into two parts . . . 279
10.4.3 Sorted input . . . 280
10.4.4 Parallelism exploitation . . . 280
10.4.5 Hot spots . . . 281
10.4.6 Influencing the access path . . . 281
10.4.7 Caching SQL results . . . 281
10.5 Batch process design . . . 282
10.5.1 Maximizing throughput . . . 283

Chapter 11. Infrastructure topics . . . 289
11.1 Database BIND options . . . 290
11.1.1 CURRENTDATA . . . 290
11.1.2 ACQUIRE . . . 290
11.1.3 RELEASE . . . 291
11.1.4 ISOLATION . . . 292
11.1.5 VALIDATE . . . 293
11.1.6 DEFER(PREPARE) . . . 293
11.1.7 ENCODING . . . 294
11.1.8 DEGREE . . . 294
11.1.9 Handling collections . . . 295

Archived

11.2 Thread reuse . . . 296
11.2.1 IMS message processing regions . . . 297
11.2.2 IMS/ESA fast path regions . . . 298
11.2.3 Thread reuse in CICS . . . 298
11.2.4 Thread reuse in a distributed environment . . . 300
11.3 Blocking in a distributed environment . . . 301
11.4 Plan and package authorization . . . 303
11.4.1 DB2 caching authorization IDs . . . 303
11.4.2 Minimize the cost of plan authorization checking . . . 303
11.4.3 Minimize the cost of authorization checking for packages . . . 304
11.4.4 Routine authorization cache . . . 304
11.5 Static SQL, dynamic SQL, and reoptimization . . . 305
11.5.1 Dynamic versus static SQL . . . 305
11.5.2 Why do we need reoptimization? . . . 306
11.5.3 Reoptimization as a solution . . . 307
11.5.4 When to use reoptimization . . . 307
11.5.5 Summary . . . 308
11.5.6 Performance coding with dynamic SQL . . . 308
11.6 CICS Open Transaction Environment . . . 309
11.7 Rebinding plans and packages . . . 310
11.7.1 Review bind parameters . . . 311
11.8 The role of catalog statistics . . . 311
11.8.1 Getting the proper statistics . . . 311
11.9 System parameters . . . 314
11.9.1 System parameters that affect locking . . . 315
11.10 Considerations when using Java to access DB2 . . . 317
11.10.1 Which Java programming interface should I use? . . . 317
11.10.2 Which driver type to use and when . . . 318
11.10.3 Java application coding considerations . . . 319

Part 4. Designing for availability and performance . . . 325

Chapter 12. What you need to know about locking . . . 327
12.1 Serialization mechanisms used by DB2 . . . 329
12.1.1 Locks . . . 329
12.1.2 Claims and drains . . . 333
12.1.3 The effect of table space parameters on locking . . . 333
12.1.4 ISOLATION levels . . . 336
12.1.5 Lock avoidance . . . 340
12.2 Lock suspensions, timeouts, and deadlocks . . . 341
12.2.1 Timeouts . . . 341
12.2.2 Deadlocks . . . 341
12.2.3 Deadlock avoidance . . . 342
12.2.4 Application action after a deadlock or timeout . . . 344
12.3 Lock duration . . . 345
12.3.1 When will the lock be acquired? . . . 345
12.3.2 Will a lock be required? . . . 346
12.3.3 How restrictive will the lock be? . . . 346
12.3.4 When will the lock be released? . . . 346
12.4 Unit of work . . . 347
12.4.1 Commit . . . 348
12.4.2 Commit frequency . . . 350
12.4.3 Cursors WITH HOLD . . . 350

12.4.4 Savepoints . . . 352
12.5 Hot spots . . . 353
12.5.1 Hot data page problem . . . 354
12.5.2 Hot row problem . . . 355
12.5.3 Sequential numbers in DB2 tables . . . 357

Chapter 13. Achieving 24x7 . . . 359
13.1 Maintaining the schema . . . 361
13.1.1 Issuing DDL . . . 361
13.2 Managing application programs . . . 363
13.2.1 Avoid the BIND if the SQL did not change . . . 363
13.2.2 Use packages with versions . . . 363
13.2.3 Performing REBIND . . . 364
13.3 Designing for application parallelism . . . 365
13.3.1 Locking considerations . . . 366
13.3.2 Conclusions . . . 367
13.4 Managing application data . . . 368
13.4.1 Supporting multiple versions of the data . . . 368
13.4.2 Keeping the data in good shape . . . 369
13.4.3 What if your insert rate is not compatible with your select requirements? . . . 373
13.4.4 Backing up the data . . . 378

Chapter 14. Determining the application quality . . . 379
14.1 Performance expectations . . . 380
14.2 Verifying during program preparation . . . 380
14.3 Verifying at run time . . . 381
14.3.1 Using RLF . . . 381
14.3.2 Using DSNZPARM options . . . 383
14.4 Verifying after the fact . . . 385
14.4.1 Buffer pool efficiency . . . 385
14.4.2 Accounting and statistics . . . 386
14.4.3 Accounting traces . . . 387
14.4.4 Statistics trace . . . 397
14.4.5 SMF 30 and job output . . . 398
14.4.6 Real-time statistics . . . 398

Part 5. Data sharing . . . 401

Chapter 15. Data sharing specifics . . . 403
15.1 Data sharing goals . . . 404
15.1.1 Provide improved availability . . . 404
15.1.2 Provide scalability . . . 404
15.2 Data sharing overhead . . . 405
15.2.1 Infrastructure impact . . . 406
15.2.2 Inter-DB2 data coherency control cost . . . 407
15.2.3 Global locking cost . . . 408
15.3 Reducing data sharing overhead . . . 409
15.3.1 Reduce locking cost . . . 409
15.3.2 Avoid overhead on system pages in your user objects . . . 411
15.3.3 Create member affinity . . . 412
15.4 IMMEDWRITE . . . 413
15.5 DML considerations . . . 414
15.5.1 Sequences and identity columns . . . 414

Archived

Part 6. Appendixes 419

Appendix A. DB2 access techniques 421
A.1 Data access methods 422
A.2 Prefetch mechanisms 423
A.3 Sequential detection 424
A.4 Index look-aside 426
A.5 Repetitive SQL procedures 429

Appendix B. Detailed performance reports 431
B.1 Language Environment storage 432
B.2 Trigger performance 433
B.3 Sequences in data sharing 435

Appendix C. Additional material 439
Locating the Web material 439
Using the Web material 439
System requirements for downloading the Web material 439
How to use the Web material 440

Related publications 441
IBM Redbooks 441
Other publications 441
Online resources 442
How to get IBM Redbooks 442
Help from IBM 442

Abbreviations and acronyms 443

Index 447

Archived

Figures

1-1 Relocated row 8
1-2 Accident driver table 10
1-3 Accident driver statement table using multiple rows to store long text 11
1-4 Accident driver statement table with overflow 12
1-5 Storing the first 40 bytes of each statement on the base table 13
1-6 EMP table with effective date as part of the primary key 21
1-7 EMP and EMP_HIST tables. 23
1-8 Data model requiring aggregation 28
1-9 New tables to store aggregated data 31
1-10 Code table 35
1-11 Multiple logical code tables denormalized into a single table 36
1-12 EMP table and EMP_PHONE attributive table 39
1-13 EMP table with WORKPHONE denormalized 40
1-14 EMP table with all phone numbers denormalized 41
1-15 Vertical table split. 42
1-16 TRAN table layout 48
1-17 Redesign of TRAN table with hash table to assign PART_NR 49
2-1 Sample three-level index structure 60
2-2 Index matching and screening predicates 66
2-3 Nonunique index 81
2-4 Application-managed sparse index 82
2-5 Application-managed sparse index: One table per status 83
2-6 Index on Last_Name, First_Name 86
2-7 Index on the anagram 87
2-8 A clustering versus a non-clustering index 89
2-9 Joined tables in a different clustering sequence 96
2-10 All joined tables clustered the same 97
2-11 Data-partitioned secondary index 101
2-12 Page range screening using an NPI 102
2-13 Page range screening using a DPSI 103
2-14 DPSI design. 105
3-1 Definition of table sizes and attributes 112
3-2 Definition of index sizes and attributes 112
3-3 Quick estimations of the number of pages and levels 113
3-4 Table-based partitioning by MONTH 122
3-5 Table-based partitioning by PRODUCT with 120 partitions. 123
3-6 Table-based partitioning by product with as many parts as products 124
3-7 Table-based partitioning by month and group of products. 124
5-1 COUNTER_TBL design. 147
6-1 Overview of server side programming options. 152
6-2 Example of a stored procedure executing four SQL statements 154
6-3 Summary of row versus trigger CPU cost per inserted row. 172
6-4 Total CPU cost per row insert for row and statement trigger. 173
6-5 CPU of the trigger package for row and statement trigger. 173
6-6 Nonnested CPU for row and statement trigger 174
6-7 Trigger CPU cost when adding a condition 175
7-1 XML column. 180
7-2 XML collection for shredding 182

© Copyright IBM Corp. 2005. All rights reserved.

7-3 XML publishing . . . 184
8-1 Analysis of stage 1 and stage 2 predicates . . . 197
8-2 Example of mismatched numeric types . . . 199
8-3 Example of mismatched string types . . . 200
9-1 Program structure for most likely retrieving one row . . . 216
9-2 Retrieve and store the result set in temporary storage . . . 219
9-3 DB2 repositioning with efficient access path . . . 220
9-4 DB2 repositioning without efficient access path. . . . 221
9-5 Cursor repositioning. . . . 222
9-6 Screen with many selection criteria . . . 227
9-7 Filter factor of the repositioning predicates . . . 231
9-8 Temporary result set for static scrollable cursors . . . 233
9-9 Comparing single-row fetch and multi-row fetch . . . 240
9-10 Using multi-row FETCH inside your application. . . . 244
9-11 Concurrently updating processes . . . 246
9-12 Coding for online concurrency . . . 247
9-13 Work in progress table. . . . 250
9-14 When to use GET DIAGNOSTICS. . . . 253
10-1 Application logic for multi-row INSERT . . . 267
10-2 Mass change activity using LOAD REPLACE . . . 271
10-3 Never-ending update using matching index scan . . . 278
10-4 Program logic for commit and restart. . . . 284
10-5 Sequential output files . . . 285
11-1 DB2SystemMonitor . . . 323
12-1 Locking hierarchy for the different types of table spaces. . . . 330
12-2 Data inconsistency caused by rollback . . . 337
12-3 Data inconsistency between index and data page. . . . 338
12-4 Lock avoidance . . . 340
12-5 Deadlock situation . . . 342
12-6 Using FOR UPDATE OF to acquire a U lock. . . . 343
12-7 Deadlocks due to unordered updates . . . 344
12-8 Changing processing sequence to reduce lock duration . . . 345
12-9 Logical units of work and DB2 units of recovery . . . 348
12-10 Rollback to a savepoint . . . 352
12-11 A hot page and a hot row. . . . 354
12-12 Updating summary table . . . 356
12-13 Spreading updates in summary table . . . 356
13-1 Sample versioning of stored procedures in a DB2 subsystem . . . 364
13-2 Number of getpages per insert in relationship to REORG. . . . 370
13-3 Life cycle of data . . . 371
13-4 Two table implementation for concurrent insert and select . . . 373
13-5 Using a queue to keep two tables in line . . . 374
13-6 Using a Q-Replication to keep two tables in line . . . 375
13-7 Table select process when using multiple tables. . . . 376
13-8 Bind DBRM into multiple collections with different QUALIFIER. . . . 377
14-1 Processing for predictive governing. . . . 383
14-2 Buffer pool monitoring options . . . 385
14-3 Schematic overview of accounting classes . . . 387
14-4 Table with overview of plan and package efficiency . . . 388
14-5 DB2 accounting report: Part 1 . . . 389
14-6 DB2 accounting report: Part 2 . . . 394
14-7 Pivot table based on IFCID 125 records . . . 395
14-8 Spreadsheet format of SMF 30 record. . . . 398

Archived

15-1 Putting multiple CF instances on a single ICF 407
15-2 Synchronous GBP page write activity per five minutes 414
15-3 Elapsed time breakdown per FETCH with NO CACHE and CACHE 50 417
A-1 Initial page ranges to determine when to use prefetch 426
A-2 DB2 index structure 427
A-3 DB2 table structure 428

Archived

Archived

Tables

1-1 Relative cost of DB2 data types for SQL processing from lowest (top) to highest. . . . 5
1-2 Byte counts of columns by data type 5
1-3 DB2 and Java data types. 6
1-4 Comparison between CHAR and VARCHAR 7
1-5 The difference between EQUAL and IS NOT DISTINCT FROM. 17
1-6 DB2 logging of updated rows. 18
1-7 Contents of the denormalized code table 36
1-8 A table with multi-valued columns to store aggregated data 37
1-9 PARTKEYU values and partitioning key update behavior 46
1-10 Default DSSIZE values 51
1-11 Maximum record size supported by each page size, in bytes 55
3-1 Overview of partitioning design alternatives 125
5-1 Characteristics of unique number generation 138
6-1 Stored procedure only vs. enabling non-Java subprogram for result set passing . . 158
6-2 A subset of the DB2 MQ scalar functions 164
6-3 DB2 MQ table functions. 165
7-1 Storing and retrieving XML data 178
8-1 Moving predicates to a better performing stage. 198
8-2 Comparing CTTs and DTTs. 211
9-1 Relevant values for multiple cursor repositioning 225
9-2 Dependencies on DECLARE and FETCH statements for scrollable cursors 235
9-3 FETCH orientation keywords for scrollable cursors. 236
9-4 Comparing scrollable cursors 236
9-5 SQLCODEs for scrollable cursors 238
9-6 Candidate SQLCODEs for application retry logic 256
9-7 Error handling parameters 258
10-1 Differences between online and batch applications. 279
12-1 Table space, partition, and table lock types. 331
12-2 Page and row lock types 331
12-3 Table space, table, and partition lock compatibility matrix. 331
12-4 Page and row lock compatibility matrix 332
12-5 Locks acquired for operations on LOBs. 332
12-6 Claim classes. 333
12-7 Termination of a unit of recovery 350
13-1 Overview of locks acquired by various SQL statements 361
13-2 Flow of nonconcurrent batch chain 378
15-1 P-lock modes 407

© Copyright IBM Corp. 2005. All rights reserved.

Archived

Examples

1-1 SQL to find the minimum EDLEVEL if defined as NOT NULL WITH DEFAULT 0 . . 14
1-2 SQL to find the minimum EDLEVEL if null values are allowed 15
1-3 Date range SQL with nullable columns 15
1-4 Date range SQL with default values in not nullable columns. 16
1-5 Two equivalent ways to code indicator variables. 17
1-6 Replacing null values with a literal. 17
1-7 SQL to replace data value with sequence for ordering 26
1-8 Sales by store calculation 30
1-9 Sales by district calculation 30
1-10 DDL to create materialized query tables 32
1-11 Populating the HASH table 49
1-12 Daily process to populate the next TRAN table partition 50
1-13 Sample output from the DSN1COMP utility. 52
1-14 Sample output from the LOAD utility showing compression statistics 53
1-15 Sample output from the REORG utility showing compression statistics 53
1-16 Calculating the maximum record size for a table. 55
1-17 Optimum page size for T1 56
1-18 Optimum page size for T2 57
2-1 Determining the number of duplicates per truncation length 85
3-1 Retrieving the employees with the top 10 wages. 116
5-1 Using the GENERATE_UNIQUE() function. 139
5-2 Using a ROWID data type 140
5-3 Deadlock using SELECT MAX with RR. 145
5-4 Using a single row counter table with UPDATE first 146
5-5 Using a single row counter table with retrieval first 146
5-6 Separating the counter update from its use. 146
6-1 Storage report produced by run-time option RPTSTG(ON) (extract). 160
6-2 Coding the MQSEND function for the Employee table 164
6-3 Singleton SELECT that puts the Employee table in a queue. 165
6-4 Create statement for a trigger that invokes a MQ XML publishing function. 166
6-5 Row trigger to copy Employee inserts to new table. 170
6-6 SQL to insert many rows into the Employee table. 170
6-7 Row trigger accounting extract 170
6-8 Statement trigger to copy Employee inserts to new table 171
6-9 Statement trigger accounting extract. 171
7-1 Using XMLELEMENT and XMLATTRIBUTES. 185
7-2 Result of XMLELEMENT and XMLATTRIBUTES 185
8-1 Applying functions using application programming language 191
8-2 Applying functions using SQL 191
8-3 Constant values in host variables 193
8-4 Constant values in SQL statement 193
8-5 Logically removed ORDER BY columns 194
8-6 Using backward index scan. 194
8-7 Candidate query for subquery to join transformation. 195
8-8 Specifying FOR FETCH ONLY to improve optimizer's choices. 195
8-9 Dictating join sequence 201
8-10 Control execution of subqueries 202
8-11 Including QUERYNO in a SQL statement 203

© Copyright IBM Corp. 2005. All rights reserved.

8-12 Specifying OPTIMIZE FOR *n* ROWS. 203
8-13 Existence checking with singleton SELECT 204
8-14 SELECT INTO using ORDER BY 205
8-15 Declaration of FOR UPDATE OF cursors 205
8-16 DELETE using FOR UPDATE OF cursors 206
8-17 UPDATE using FOR UPDATE OF cursors 206
8-18 Use of USE AND KEEP UPDATE LOCKS 206
8-19 Using SAVEPOINTs 210
8-20 Table including generated values by DB2 212
8-21 Retrieving values during INSERT 213
9-1 Cursor declaration for restart: The common way. 223
9-2 Cursor declaration for restart: Using AND NOT predicates 224
9-3 Cursor declaration for restart: Using OR and additional predicates. 224
9-4 Program logic for browsing using multiple cursors 225
9-5 Declaring a SENSITIVE DYNAMIC scrollable cursor 235
9-6 Host variable definitions for fetching 10 rows at a time 240
9-7 Cursor declaration for multi-row fetch 240
9-8 New FETCH syntax 241
9-9 Multi-row cursor definition with FOR UPDATE OF 244
9-10 Updating a result set of a multi-row cursor 244
9-11 Updating a single row of a result set of a multi-row cursor 245
9-12 DDL for table T1. 246
9-13 Update using optimistic locking 248
9-14 Concurrent update on independent columns. 248
9-15 Concurrent update on dependent columns 249
9-16 Implementation of DSNTIAR 251
9-17 GET DIAGNOSTICS on multi-row INSERT. 254
9-18 Example of GET DIAGNOSTICS ALL output 255
9-19 Message text for -913 256
9-20 Error handling table 258
9-21 Consistency token from SYSIBM.SYSPACKAGE 259
9-22 Result set for package associated to module PGM1 259
9-23 Checking the load module 260
9-24 Logical locking table. 260
9-25 BIND PACKAGE statements 261
9-26 Host variables for SET CURRENT PACKAGESET. 261
9-27 Setting CURRENT PACKAGESET 261
10-1 Inserting *n* rows at once. 269
10-2 Inserting *n* rows at a time combined with scalar host variables. 270
10-3 Sample statement for using ONLINE LOAD 270
10-4 Delete affecting many rows: The ugly way 273
10-5 Delete affecting many rows: The good way. 273
10-6 Cursor declaration for update 274
10-7 Online REORG using DISCARD option. 274
10-8 Using LOAD REPLACE 274
10-9 Updates with different WHERE clauses. 275
10-10 Update using CASE expression. 275
10-11 Selecting a ROWID column 276
10-12 Query using direct row access. 277
10-13 Eliminating duplicate records using DFSORT 286
10-14 Cursor repositioning: A common approach 286
10-15 Cursor repositioning: Minimizing getpages 287
11-1 Use different statements for different host variable values 306

11-2 Performing a batch query. 321
12-1 Overriding the isolation level of a single SQL statement 336
12-2 Sequential numbers in DB2 table: Case one 357
12-3 Sequential numbers in DB2 table: Case two 357
12-4 Sequential numbers in DB2 table: Case three. 358
13-1 Table DDL for base table and control table 368
13-2 View definitions for current and future view 368
13-3 Two cursor definitions to read future and current data 369
14-1 DSNI031I message after lock escalation 384
14-2 DSNR035I output after not committing during four checkpoints 384
14-3 Sample query to determine if a table space needs to be REORGed. 399
14-4 Sample query to determine if an index needs to be REORGed or rebuilt 399
15-1 Creating MYSEQ1 with NO CACHE 416
15-2 Selecting the next 42 values of MYSEQ1 416
15-3 Account extract for NEXT VALUE with NO CACHE 416
15-4 Changing the sequence to CACHE 50 416
15-5 Account extract for NEXT VALUE with CACHE 50 417
A-1 Index screening predicate 423
A-2 Sample sequential processing program. 428
B-1 Storage report produced by run-time option RPTSTG(ON). 432
B-2 Row trigger accounting 433
B-3 Statement trigger accounting. 434
B-4 Accounting report for SELECT with NO CACHE 435
B-5 Accounting report for SELECT with CACHE 50. 436

Archived

Archived

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:
IBM Director of Licensing, IBM Corporation, North Castle Drive Armonk, NY 10504-1785 U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the products and/or the programs described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:
This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

© Copyright IBM Corp. 2005. All rights reserved.

Trademarks

The following terms are trademarks of the International Business Machines Corporation in the United States, other countries, or both:

AIX®
CICS/ESA®
CICS®
Database 2™
Distributed Relational Database Architecture™
DB2 Connect™
DB2 Extenders™
DB2®
DFSMShsm™
DFSORT™
DRDA®
Enterprise Storage Server®
@server®
Extended Services®
FICON®
Geographically Dispersed Parallel Sysplex™
GDPS®
IBM®
IMS™
IMS/ESA®
iSeries™
Language Environment®
MQSeries®
MVS™
OmniFind™
OS/390®
Parallel Sysplex®
PR/SM™
QMF™
RACF®
Redbooks™
Redbooks (logo) ™
System z9™
System/390®
Tivoli®
VTAM®
WebSphere®
z/OS®
z9™
zSeries®

The following terms are trademarks of other companies:

EJB, Java, Java Naming and Directory Interface, JDBC, JVM, J2EE, and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Excel, Microsoft, Visual Basic, Visual Studio, Windows, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

Archived

Preface

Conducting business via the Web and remaining open for business 24 hours a day, seven days a week is now commonplace. Customers come in with unexpected workloads through the Web and operate in a self-service fashion with mostly context-sensitive metadata to guide them. The strong requirement is availability. However, even with new Web applications, most of the core business systems considerations still apply, and performance is critical.

Technology has been accelerating for mainframe systems. They had become adept at providing business resiliency, accommodating strategic software that has been around for the last several decades, such as IMS™, DB2®, and CICS®, and they have also become a host for developing and running Web applications built in Java™ accommodating the latest business requirements. Businesses need to leverage, extend, and integrate the strategic assets which represent multi-year investments to support leading edge technology.

DB2 for z/OS® has come a long way and provides facilities to exploit the latest hardware and software technologies, accommodating a majority of user requirements. However, special considerations are required to build high performance applications. If you want to achieve high performance or high availability, you must use the design, programming, and operational techniques applicable to DB2.

In this IBM® Redbook, we discuss many of these techniques and provide guidelines for database and application design. We expect the best practices described in this IBM Redbook will help DB2 professionals design high-performance and high-availability applications.

The design challenge

Figure 1 represents a visual summary of the challenges in application design.

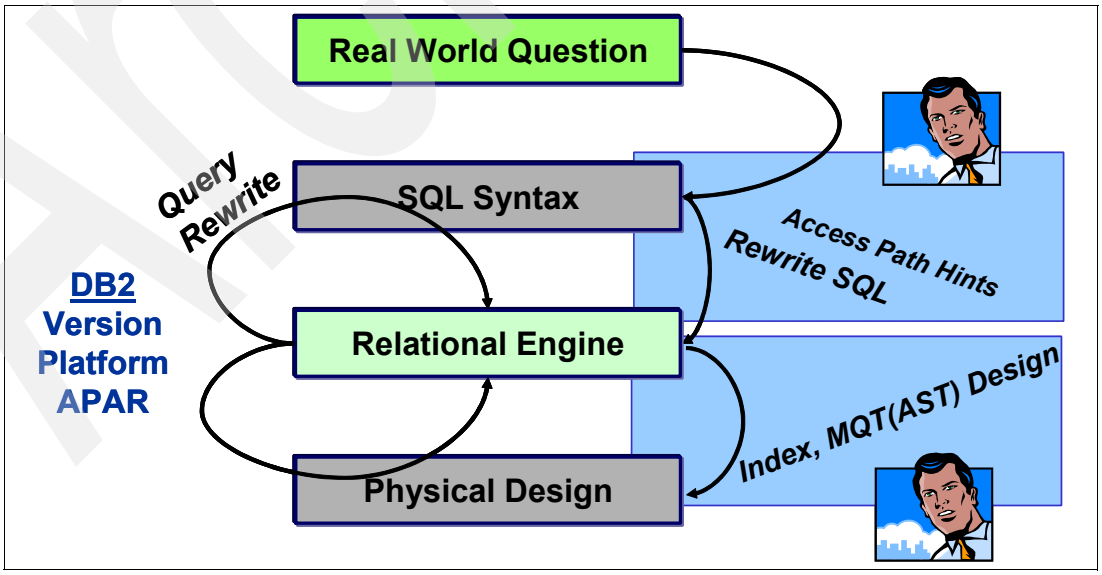


Figure 1 The tasks involved

© Copyright IBM Corp. 2005. All rights reserved.

Assuming that the business needs have been clearly understood (the real world question), the logical data model is completed, and the service levels have been agreed upon, we jump into the physical design of the tables and start coding the application.

The functional design of the application should be implemented within the boundaries of reality, like the DB2 version utilized, the operating system and hardware available, even the maintenance level, and it has to comply with several levels of sanity checks. Any system must have a ROI that is in line with the business value of the application or requirement. Very early in the design, compare the application needs with system limits and the environment available, avoiding surprises and setting realistic performance expectations. You do not want to find out too late in the process that there are requirements that cost too much to implement.

Interaction between the DBA and the application designers becomes critical for the return on investing in MQTs and verifying the performance of complex queries.

You are not alone in the system

The advantage of DB2 for z/OS and z/OS as a platform is it is very good at providing a runtime environment with reliability, availability, and serviceability characteristics that are unmatched in the industry. It is very likely that yours is not the only program running on the platform. Memory, I/O, and CPU are shared resources among all the processes. Although prices are decreasing significantly, none of the shared resources are infinite.

One aspect of sharing the system is that applications compete for resources and there is a potential for bottlenecks. The DB2 active log is a serialization point in DB2. It is required from a data consistency point of view to identify every data change uniquely and in sequence. This means that the log has its limits in terms of number of I/O operations that can be performed per second. The I/O bandwidth on a file that always needs to be appended is determined by two dimensions of the I/O, the size of the data that is written, and the number of I/Os that are performed per second. So, an application that performs 100 COMMIT operations per second might meet the performance objectives, but has a significant impact on the overall system performance when integrated with the other applications in the DB2 subsystem.

Technology is always improving

A proven design technique could be to simply wait until the machine is fast enough to process the program you wrote or the table that you designed. Unfortunately, your business is probably not going to be able to wait until the machine becomes faster.

IBM has made a lot of progress in several areas over the last few decades by introducing:

- Parallel Sysplex®

  The ability to combine multiple machines into a single virtual machine.

- Faster processors

  The net result is the ability to process more instructions in the same time or to put it even more simply: double the machine speed and your time is half of what it used to be.

- Parallel Access Volumes

  Parallel Access Volumes (PAV) is the ability to read or write on a single disk volume in parallel, which eliminates many of the data placement issues.

- FICON® channels

  This new technique to connect the mainframe to the disks provides a much bigger bandwidth, hence increasing the volume of data that you can read or write from a single data set.

- More real memory

  It is not uncommon to see production LPARs with 20 GB of real storage or more (up to 128 GB maximum). This means that you can cache more data in memory than ever before.

- Larger virtual storage

  Simplifying the storage model with the 64 bit addressing and again increasing the amount of data that can be cached, reducing the probability that a physical I/O is required.

Does this mean that you can assume that the available resources are infinite? Not really. There are always two factors to take into account: The return on investment (ROI) and the basic laws of nature.

A typical example of the laws of nature is the problem of consecutive number generation. How do we build an application that needs to generate 1,000 new rows per second and the primary key of these rows is a consecutive number? The 1,000 inserts are performed by separate transactions, so batching is not an option. An option is to use a table that keeps track of the last number assigned. So every transaction performs an update of the table with the last number, performs the insert, and issues a COMMIT. The slowest component in this process is most likely to be the COMMIT, which might take between 2 to 6 milliseconds. The net result of that design is a transaction rate of 166 to 500 per second and not the required 1,000 transactions per second. See Chapter 5, “Techniques to generate numbers” on page 137.

Recommended Service Update

The IBM Consolidated Service Test mission is to provide a consolidated, tested, and recommended set of services for z/OS and key subsystems, such as DB2, on a quarterly basis. When you order your current service deliverable, you receive a recommended service package with a tested level of service for all included products.

Check the quarterly and monthly recommendations, including HIPER and PE PTFs, at the Web site:

http://www.ibm.com/servers/eserver/zseries/zos/servicetst/mission.html

Assumptions

A properly designed and normalized logical model is a necessary raw material of any physical database design, but there are still many processes that must be applied in order to achieve the required functionality and best performance. The logical model is much like bread dough. All of the ingredients have been added and mixed, but it is too soon to tell if the end result is pizza crust or a loaf of bread. The decisions made and processes applied by the chef determine that outcome.

The assumption of this redbook is a fully normalized logical model for which every entity has one or more candidate keys identified and documented.

This redbook is based on DB2 for z/OS Version 8. There are features and functions of DB2 that depend on which version you use. You can get different results when running on older versions of DB2. We used Version 8 as a base assumption because Version 8 has been in the market for some time now and has proven to be stable. It is always possible that future versions of DB2 may introduce new features that might change some of the recommendations in this redbook. We have also added APARs on Version 8 that are necessary in order for you to get the results that are described in this redbook. It is important to stay current on maintenance since IBM is constantly improving performance based on the feedback of customers.

The CPU estimates in this book are based on a z990 (2084-301) machine which has 480 MIPS. If any other machine is used for a test, we specify the model with the example.

We do not cover the aspects of system tuning. We assume that your system's DBA has tuned DB2 correctly, so that the buffer pools are set up correctly, that the log is not a bottleneck, and the WLM dispatching priorities are set correctly. We also assume that the mainframe configuration has sufficient I/O, CPU, and memory available. Hardware can be used to mask some design inefficiencies, but a well behaving DB2 system cannot compensate for hardware under-investment.

This redbook is meant primarily for application developers, but it can be useful for database administrators as well. In these days, rapid deployment is paramount. Vendor solutions, the use of tools, and Web services have shortened the development cycle, but introduced more complexity and increased the workload for DBAs. The developers want to develop and port code with object methods and connectivity avoiding database and platform peculiarities. Problems tend to show up once the application is in production, and by that time, there is less knowledge of database design and application objectives.

It becomes more important to have interaction and cooperation to understand the application issues and choose the infrastructure characteristics. Reviews before production with key representatives of systems support can reduce risks and prepare for better operations.

Contents

The redbook is divided into the following parts:

- Part 1. Database design

  It consists of three chapters dealing with the aspects of table design, implementing logical entities and attributes into DB2 tables, how index design and their influence on physical database design decisions, and model validation as a way to identify potential resource and performance bottlenecks.

- Part 2. Application enabling components

  It consists of four chapters dedicated to functionalities in DB2 that are building blocks in developing the application: support for constraints, routines, identity columns and sequences, and XML.

- Part 3. Embedding SQL in the application

  It consists of four chapters where we look at introducing effective SQL language in operations. We start with the basic rules about SQL programming, we introduce programming templates, then we proceed to more advanced techniques and relevant DB2 environment definitions.

- Part 4. Designing for availability and performance

  In the three chapters contained in this part, we provide background information about locking, we discuss techniques for achieving high application availability, and then provide criteria for determining the application quality.

- Part 5. Data sharing

  This part is a chapter dedicated to data sharing application considerations.

- Part 6. Appendixes

  This part contains relevant additional information and reference material.

The team that wrote this redbook

This redbook was produced by a team of specialists from around the world working at the International Technical Support Organization, San Jose Center.

Paolo Bruni is a DB2 Information Management Project Leader at the ITSO, San Jose Center. He has authored several Redbooks™ about DB2 for z/OS and related tools, and has conducted workshops and seminars worldwide. During Paolo's many years with IBM, in development and in the field, his work has been mostly related to database systems.

Patric Becker is an Application Programmer with Sparkassen Informatik GmbH and Co. KG in Germany. Since joining the company in 1997, Patric has been responsible for several, high availability Customer Relationship Management, DB2, and IMS applications. For DB2, he has also assumed the role of database administrator and he was involved in evaluating and applying the new functions of DB2 for OS/390® Version 6 and Version 7. He is also co-author of the IBM Redbook *DB2 for z/OS Using Large Objects*, SG24-6571.

Jan Henderyckx is an independent consultant, lecturer, and author. He has over 19 years of experience with DB2 and has presented papers at several international conferences and local user-groups, both in Europe and the United States. He is the co-author of the redbook *DB2 for OS/390 Application Design Guidelines for High Performance,* SG24-2233. His areas of expertise include application design, performance, connectivity and information integration, heterogeneous replication, data sharing, data warehousing, and application and system tuning. He often conducts performance audits and design reviews for some of the largest DB2 installations. He is also an IBM Gold Consultant, an IBM certified solutions expert on replication and connectivity and an IBM certified solution designer for DB2 BI V8. He can be reached at Jan_Henderyckx@brainware.be.

Joseph Link is a Data Architect with First Data Corporation in the United States. He has 20 years of experience working with relational databases. He holds a degree in Computer Science from Iowa State University. Since joining First Data in 1999, Joseph has been primarily responsible for the performance tuning of DB2 applications. His areas of expertise include database design, database administration, and DB2 performance.

Bart Steegmans is a Senior DB2 Product Support Specialist from IBM Belgium currently working remotely for the Silicon Valley Laboratory providing technical support for DB2 for z/OS performance problems. Previously, Bart was on assignment as a Data Management for z/OS project leader at the ITSO, San Jose Center, from 2001 to 2004. He has over 16 years of experience in DB2. Before joining IBM in 1997, Bart worked as a DB2 system administrator at a banking and insurance group. His areas of expertise include DB2 performance, database administration, and backup and recovery.

A photo of the team is in Figure 2 on page xxviii.

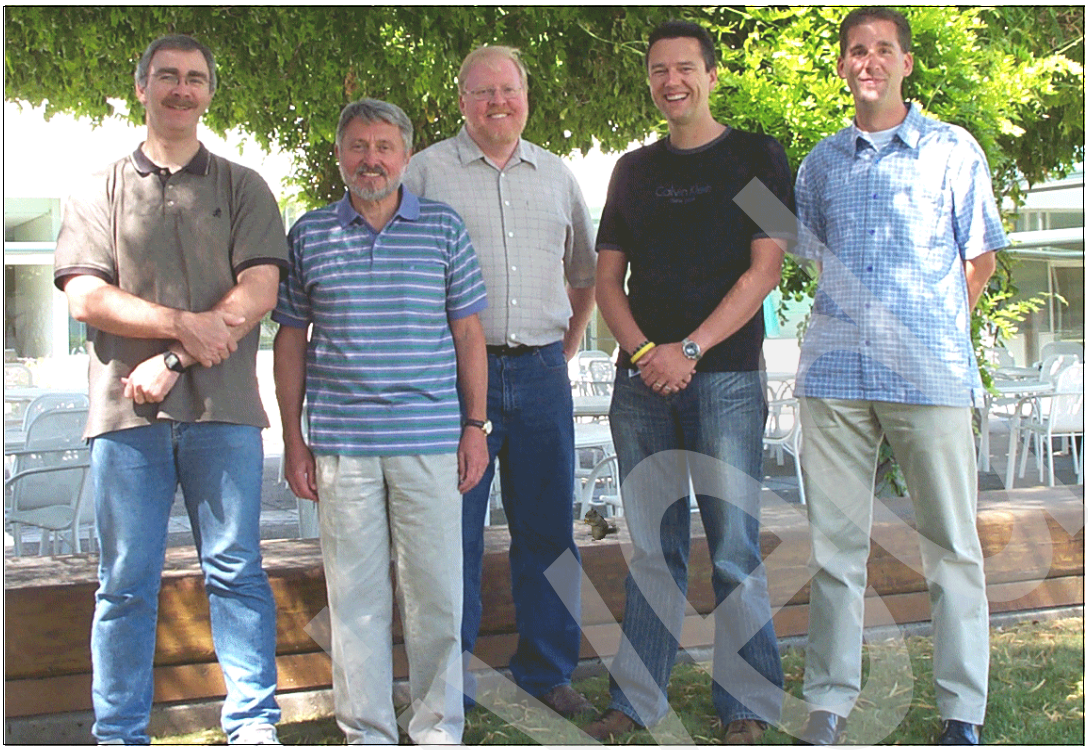


Figure 2 Left to right: Bart, Paolo, Joseph, Jan and Patric (photo courtesy of Bertrand Dufrasne)

Thanks to the following people for their contributions to this project:

Rich Conway
Bob Haimowitz
Emma Jacobs
Leslie Parham
Deanna Polm
Sangam Racherla
International Technical Support Organization

Kristin Blicharz
John Campbell
Julie Chen
Karelle Cornelll
Ramani Croisettier
John Garth
Bill Kyu
Chao-Lin Liu
Roger Miller
Matthias Nicola
Paul Ostler
Terry Purcell
Jack Shedden
Akira Shibamiya
Jim Teng
Julie Watts
Maryela Weihrauch
Jay Yothers

Guogen Zhang
IBM Silicon Valley Lab

Bryce Krohn
Krohn Enterprises, Inc.

Joyce M Lindow
First Data Corporation

Rick Butler
Bank of Montreal (BMO) Financial Group

Become a published author

Join us for a two- to six-week residency program! Help write an IBM Redbook dealing with specific products or solutions, while getting hands-on experience with leading-edge technologies. You'll team with IBM technical professionals, Business Partners, and/or customers.

Your efforts will help increase product acceptance and customer satisfaction. As a bonus, you'll develop a network of contacts in IBM development labs, and increase your productivity and marketability.

Find out more about the residency program, browse the residency index, and apply online at:

ibm.com/redbooks/residencies.html

Comments welcome

Your comments are important to us!

We want our Redbooks to be as helpful as possible. Send us your comments about this or other Redbooks in one of the following ways:

- Use the online **Contact us** review redbook form found at:

  ibm.com/redbooks

- Send your comments in an e-mail to:

  redbook@us.ibm.com

- Mail your comments to:

  IBM Corporation, International Technical Support Organization
  Dept. QXXE Building 80-E2
  650 Harry Road
  San Jose, California 95120-6099

Archived

Archived

Part 1

Database design

The logical model is typically created independently of the RDBMS used, the anticipated size of the objects, and the access methods of the processes that access the database. DB2 for z/OS has particular characteristics and abilities that should influence the way in which the logical model is implemented into a physical design. In addition, the anticipated size of database objects, type, and volume of data access by application processes should influence physical design decisions.

In order to achieve optimum application performance, it is necessary to understand the physical database design options of DB2 for z/OS. These options are influenced by the volume metrics of your particular application. For that reason, it is typically impossible to provide “one size fits all” recommendations. Before proceeding with the physical database design, it is very important to gather or estimate the following volume metrics:

- The number of rows in each logical entity.
- The growth rate of each logical entity.
- The cardinality and source of each logical attribute.
- The auditing and retention requirements of the data.
- The average and peak volume of each application process that creates, retrieves, updates, or deletes rows in each entity. This is referred to as *CRUD information*. Table 1 on page 2 contains a sample CRUD matrix.
- The service level agreements (SLAs) that are in place governing the processes that access and update the database.

© Copyright IBM Corp. 2005. All rights reserved.

Table 1 A sample CRUD matrix for the account entity

Process name	Entity name	Access type (CRUD)	Attributes accessed	Attributes qualified	Access frequency
Online Create Account	Account	Create	All	N/A	avg. 100/hr. max. 500/hr. < 1 sec resp. 24x7
Batch Create Account - Daily job D0001	Account	Create	All	N/A	25,000-40,000/day Sun.-Fri. Job starts at 10pm, must complete by 2am.
Online Account Lookup	Account	Read	FRST_NM LAST_NM BAL_AMT	ACCT_NR = current active	avg. 1000/hr. max. 3000/hr. < 1 sec resp. 24x7
Online Name Search	Account	Read	ACCT_NR BAL_AMT	LAST_NM LIKE and FRST_NM LIKE order by LAST_NM, FRST_NM	avg. 200/hr. max. 600/hr. < 1 sec resp. 24x7
Online Account Update	Account	Create Update	All END_DT, STTS_CD	None ACCT_NR = current active	avg. 75/hr. max. 250/hr. < 1 sec resp. 24x7
Online Close Account	Account	Update	CLOS_DT, END_DT, STTS_CD	ACCT_NR = current active	avg. 25/hr. max. 100/hr. < 1 sec resp. 24x7
Purge Account History - Monthly job M0001	Account	Delete	N/A	END_DT over 5 years old and STTS_CD <> ‘CLOSED’ or END_DT over 10 years old	100,000-300,000 per month. Must complete in 6 hours

This part consists of three chapters:

- In Chapter 1, “Table design” on page 3, we review the process of implementing logical entities and attributes into DB2 tables.
- In Chapter 2, “Index design” on page 59, we discuss how to design indexes to support all of the requirements. Along the way, we demonstrate how the metrics mentioned above influence the physical database design decisions.
- Once a physical design has been created, validation should be performed to estimate the resource consumption of the applications and whether the design satisfies the SLAs. This important step, described in Chapter 3, “Model validation” on page 109, provides a way to identify potential resource and performance bottlenecks early.

1

Table design

In this chapter, we review the table design points and the associated performance implications. We deal with the transition from the logical model to the physical model by discussing the following topics:

- Choosing the appropriate data type for each attribute, including the use of VARCHAR, large objects, and storing XML data
- Deciding when a column should be defined as nullable
- The effect of column sequence on performance
- Considerations when selecting the primary key
- Selecting the encoding scheme of the table
- The VOLATILE attribute and how it should be used
- When and how to change the normalization level of the logical model
- Considerations when deciding whether to partition or not
- Criteria for selecting partitioning columns
- The performance trade-off of using DB2 compression

This chapter contains the following sections:

- Implementing the logical model
- Deviating from the third normal form
- Partitioning
- Using DB2 compression
- Choosing a page size

© Copyright IBM Corp. 2005. All rights reserved.

1.1 Implementing the logical model

This section highlights the decision points when moving from the entities and attributes of the logical model to the tables and columns of the physical model.

1.1.1 Domains and data types

In the logical model, attributes are assigned a domain. This domain has a finite number of possible values for the attribute. DB2 does not have domains, DB2 has data types. Depending on the tool used, these domains may or may not correspond directly to a column data type supported by DB2. In order to enforce a specific domain in DB2, it may be necessary to implement a constraint as described in Chapter 4, “Constraints” on page 129.

Domain verification

DB2 always enforces that the value in a column fits in the domain of the data type. For that reason, it is a good idea to verify the domain from the logical model before selecting a specific DB2 data type. To do this, it is necessary to know the source and use of the data. For example, if the data source of a column is an online screen, you might know that edits are applied before the data is inserted into your transactional table. In this case, you can safely choose a DB2 data type that enforces the rules of the domain. Suppose, however, that you have an input logging table that must capture any input, regardless of whether it passes the edits or not. In this case, it may be necessary to accommodate data values that violate DB2’s rules for a specific type.

Here are questions to ask:

- For numeric data types, are they always strictly digits, or is it possible to receive character or blank data?

  If an attribute can contain only numeric data, it should be defined using one of DB2’s numeric data types. Numeric data types occupy less space than character data, and the optimizer is able to calculate more accurate filter factors for columns using these predicates.

  If an attribute can contain both numeric and non-numeric data, it should be defined using one of the character data types.

- Are the numeric values always a whole number? If so, what are the upper and lower limits? If not, what is the scale, or number of decimal places that must be maintained?

  Whole numbers are most efficiently stored in one of the binary data types, which are SMALLINT or INTEGER. Larger numbers or numbers with digits to the right of the decimal point should be stored using DECIMAL data types. Very large numbers which can afford the loss of accuracy should use a floating point data type.

- DB2 dates, times, and timestamps must contain valid values. Does the source of your data value abide by these rules, or is it possible to receive a date of February 30?

  If possible, the DB2 data types for date and time should be used. Not only do these data types occupy less space, but DB2 also provides a number of predefined functions that can be performed on these columns. It is possible to CAST a character column to a date so that these functions can be used, but this would require that the character column always contains a valid date. If you are certain that the column always contains a valid date, then you should use the date data type.

- How often is the column updated, and when it is updated, is the length of the data changed?

  The answer to this question can influence the decision to use VARCHAR, and is also a key factor in determining column placement.

- For long character columns (longer than 18 bytes), what is the maximum length and average length?

  The answers to these questions should be used to decide whether to define the column as a fixed length character using the CHAR data type, or whether to use one of the variable character data types. This topic is discussed in “Using VARCHAR” on page 7.

  If two data types can equally support the domain of the attribute and the requirements of the application, choose the data type with the lowest cost for best performance. The cost of a data type depends on two factors, the CPU required to manipulate the data and the disk space required to store it. Table 1-1 contains a list of DB2 data types in the order of their relative CPU cost for SQL processing, from lowest cost to highest cost. This is a very rough rule-of-thumb (ROT) since there are differences in input or output, local or distributed, and so on.

Table 1-1 Relative cost of DB2 data types for SQL processing from lowest (top) to highest

Data type
SMALLINT and INTEGER
FLOAT
CHAR
VARCHAR
DECIMAL
DATE and TIME
TIMESTAMP

Table 1-2 contains a list of data types and the number of bytes required to store them.

Table 1-2 Byte counts of columns by data type

Data type	**Byte count**
INTEGER	4
SMALLINT	2
FLOAT(n)	If *n* is between 1 and 21, the byte count is 4. If *n* is between 22 and 53, the byte count is 8.
DECIMAL	INTEGER(p/2)+1, where p is the precision
CHAR(n)	n
VARCHAR(n)	n+2
CLOB	6
GRAPHIC(n)	2n
VARGRAPHIC(n)	2n+2
DBCLOB	6

Archived

Data type	Byte count
BLOB	6
DATE	4
TIME	3
TIMESTAMP	10
ROWID	19 (stored as VARCHAR(17))
LONG VARCHAR or LONG VARGRAPHIC	To calculate the byte count, use this formula: 2*(INTEGER((INTEGER((m-i-k)/j))/2)) Where: m is the maximum row size (8 fewer than the maximum record size) i is the sum of the byte counts of all columns in the table that are not LONG VARCHAR or LONG VARGRAPHIC j is the number of LONG VARCHAR and LONG VARGRAPHIC columns in the table k is the number of LONG VARCHAR and LONG VARGRAPHIC columns that allow nulls To find the character count: 1. Find the byte count. 2. Subtract 2. 3. If the data type is LONG VARGRAPHIC, divide the result by 2. If the result is not an integer, drop the fractional part.

Consider the data types of the application programming language

Not all application languages have data types that match up directly with those allowed in DB2. For example, C++ and FORTRAN do not support a data type that is directly equivalent to the DB2 DECIMAL data type. REXX only processes string data, so when using REXX, DB2 converts the data from a string type to the table column type and vice versa. Table 1-3 maps DB2 data types to those you would have when coding in Java. Be sure to match the Java data type to the DB2 data type in order to avoid unnecessary data conversions.

Table 1-3 DB2 and Java data types

DB2 data type	Java data type	Comment
SMALLINT	short, boolean	no direct mapping for bit in DB2
INTEGER	int	
REAL	float	single precision
FLOAT, DOUBLE	double	double precision
DECIMAL (p,s) NUMERIC (p,s)	java.math.bigDecimal	with p=precision, s=scale keeps scale and precision in Java
CHAR, VARCHAR, GRAPHIC, VARGRAPHIC	String	
CHAR, VARCHAR FOR BIT DATA	byte[]	

DB2 data type	Java data type	Comment
TIME	java.sql.Time	
DATE	java.sql.Date	
TIMESTAMP	java.sql.Timestamp	

Since Java does not support a fixed length character string data type, there are pros and cons when using CHAR or VARCHAR. They are summarized in Table 1-4.

Table 1-4 Comparison between CHAR and VARCHAR

CHAR columns	VARCHAR columns
Requires use of the DB2 RTRIM function (or, second choice, the Java trim() function) to remove the trailing blanks to eliminate trailing blanks	Easier for Java application
Higher CPU cost for Java applications	Higher CPU cost 'in DB2' possible
Non-Java applications are not affected	All applications are affected

When coding your Java application and designing your database, make sure to store numeric data in a numeric data type since numbers are not dependent on encoding schemes. In general, Java applications always execute in Unicode. For character data types stored in the database as EBCDIC or ASCII, a code page conversion is required. See "Choosing an encoding scheme" on page 23 for more information about encoding schemes.

Information about the data types available in DB2, the domains they support, and their space allocation can be found in the *DB2 UDB for z/OS Version 8 SQL Reference,* SC18-7426-02. The *DB2 UDB for z/OS Version 8 Application Programming and SQL Guide,* SC18-7415-02, discusses how to determine the equivalent DB2 data types for most common programming languages.

Note: Mismatched data type comparison is made sargable and indexable in DB2 for z/OS V8 compatibility mode via PK12389.

Implement data types consistently, especially for join columns

When different logical attributes have the same domain, it is a good idea to use the same data type for all of them. In particular, the data type of a foreign key should match the data type of the primary key that it references. However, in Version 8 new-function mode, many unmatched data types can now be indexable.

Tip: If you previously used the CAST or CONCAT function in joins prior to Version 8 to achieve column matching, the function can now be removed. This may allow the optimizer the option of joining the tables in either order.

Using VARCHAR

The longest fixed length character data type supported by DB2 is CHAR(255). While you can specify CHAR(300), any character strings longer than 255 are stored using a variable length data type, such as VARCHAR. When the maximum length of a character attribute is fewer than 256 bytes, there are several factors to consider when deciding whether to implement a CHAR or VARCHAR data type.

Each variable length column includes a two byte length indicator in addition to the data string that is being stored. Therefore, from a pure performance perspective, the savings from using variable length columns must overcome the disk space cost of the additional two bytes per column, the CPU cost of handling it, and the potential additional logging when the row is updated.

There may be an additional performance impact of implementing a variable length row, either through the use of variable length columns or compression. Whenever a variable length row is updated, the new row length could be longer than the original row length and there may not be sufficient space on the page for the updated row. When this happens, DB2 relocates the row on another page and places a pointer to it on the original page in the *record ID* (RID) as logically represented in Figure 1-1. Here the index leaf page shows the key, for example, key value 100 and the RID that points to the table space row. In the data row, the key field is shown as the first column in the row.

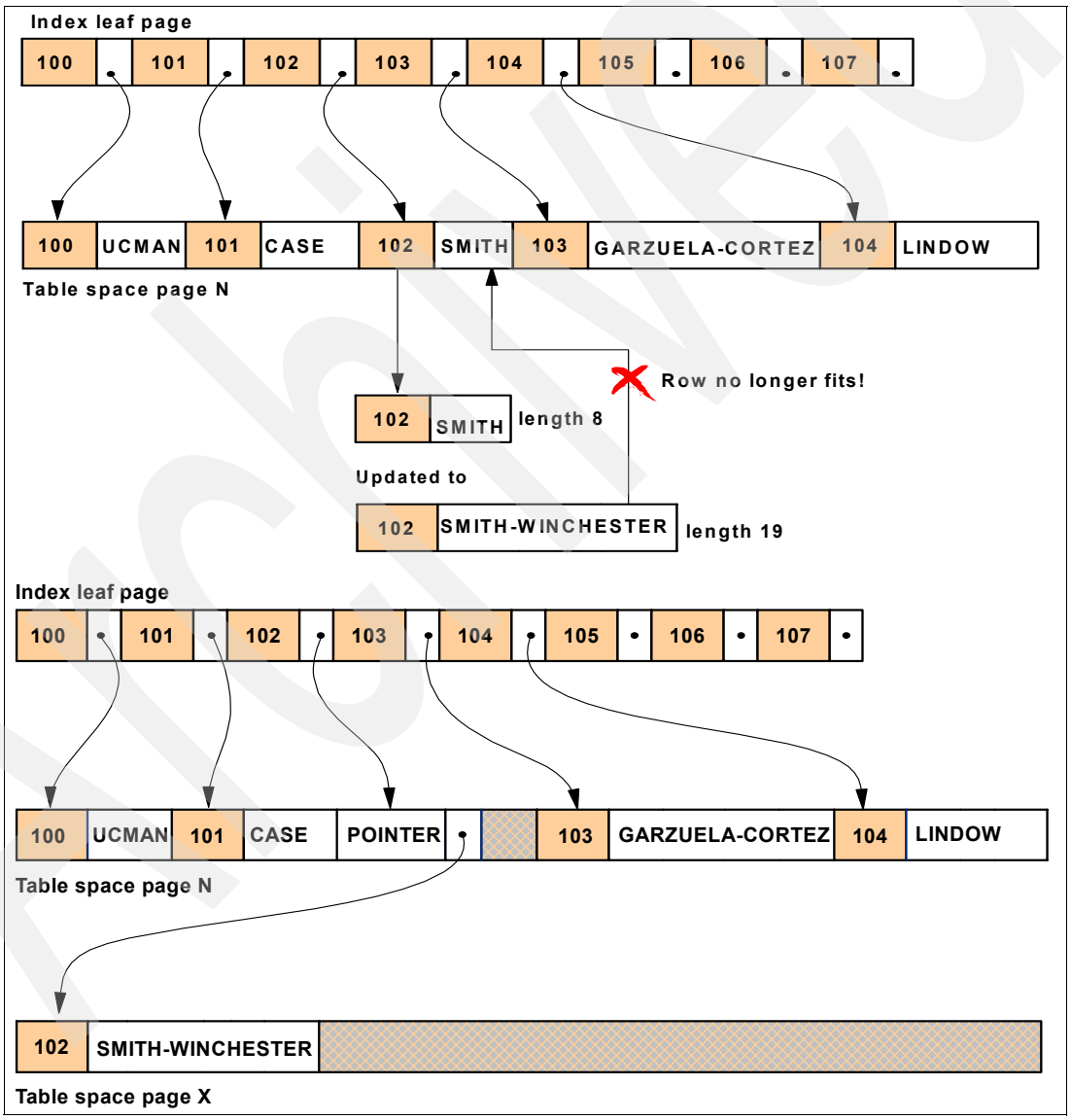


Figure 1-1 Relocated row

This results in an additional getpage and possibly an additional I/O for any reference to this row. This row remains relocated until it is updated to a size that fits on the original destination

page, or until the table space is reorganized. Increasing the PCTFREE parameter of the table space can reduce the chance of relocated rows, but also reduces the disk savings of VARCHAR if this is the only reason for the increased free space.

The performance savings from using VARCHAR come from reducing the average length of the row. While reduced disk space usage is the most obvious benefit, a reduced row length increases the number of rows that fit on a page, which in turn reduces getpages. Fewer getpages translate into fewer locks, less CPU, and less I/O. This can also reduce the storage requirements of buffer pools and increase their efficiency. Compression, which is discussed in "Using DB2 compression" on page 51, also reduces the row length and can achieve similar savings. However, VARCHAR has the additional benefit of reducing the number of bytes of data that must be passed between DB2 and the application. Because there is a CPU cost associated with each byte of data moved, VARCHAR should still be considered even when compression is used.

As a general recommendation, if a column length can vary, start considering using VARCHAR for column length of 18 and above, and use fixed length CHAR for 17 and below. A fixed length CHAR column with the maximum value should also be favored if the length of the attribute is expected to marginally change.

The placement of VARCHAR columns in DB2 tables can also have an effect on performance. The offset of each column following a VARCHAR column in the row must be calculated for each row accessed resulting in additional CPU consumption. The topic of column placement is discussed in more detail in "The effect of column placement on performance" on page 18. There are special considerations for VARCHAR columns that are going to be indexed. Refer to Chapter 2, "Index design" on page 59, for more information.

Tip: Although you can use LONG VARCHAR and LONG VARGRAPHIC, we recommend the use of VARCHAR with the length specified. The LONG option gives you a row length that is the rest of the page after what is left after subtracting all other columns. This means that the maximum length becomes variable and depends on the content of the other columns. It also impacts the online schema options as it is impossible to add a column to the table once there is a LONG VARCHAR or LONG VARGRAPHIC. The table must be dropped and recreated to add a column, and the possibility exists that data in a LONG column no longer fits in the row.

Long columns and LOBs

DB2 provides three different data types that allow the storage of very large strings, they are collectively referred to as LOBs, or large objects. These types are BLOB, CLOB, and DBLOB. However, there are significant restrictions on how LOBs can be manipulated in DB2. Only EXISTS, LIKE, and NULL predicates can be used on LOB columns, and LOB columns cannot be specified in GROUP BY, ORDER BY, or DISTINCT clauses.

If you know that you are going to use one of the LOB data types, a starting point is the redbook *Large Objects with DB2 for z/OS and OS/390*, SG24-6571. However, this redbook was written at the DB2 V7 level. Since then, several restrictions of the standard DB2 utilities regarding objects larger than 32 KB are being lifted. DB2 V8 LOAD and UNLOAD support LOBs with APARs PQ90263 and PK10278.

Consider the following alternatives before choosing to use a LOB data type:

- Use a larger page size.

  DB2 supports page sizes of 4 KB, 8 KB, 16 KB, and 32 KB. With a page size of 32 KB, a VARCHAR or VARGRAPHIC column can be up to 32,704 bytes long. Consider this approach when dealing with strings that are shorter than this size.

- Use multiple rows in the same table.

  If your long strings are too long to be supported in a single row, consider storing the string in multiple rows in the same table.

- Use a base table and an overflow table.

  If the average size of your long string is significantly shorter than the maximum size, consider using a separate table to store just the overflow.

Each of these alternatives is demonstrated in the following case study.

Storing long columns: Case study

Suppose that your logical design contains a table called ACCIDENT DRIVER. This table stores the accident number, the driver number, the name of the driver, and the driver's statement, which can be up to 8,000 bytes long. Your first cut physical model would look something like the table shown in Figure 1-2.

ACCIDENT DRIVER TABLE

ACCIDENT NUMBER	DRIVER NUMBER	DRIVER NAME	STATEMENT LENGTH	TEXT
0001	0001	KATHY	2040	THERE IS NO WAY THIS ACCIDENT IS MY FAULT...
0001	0002	RAVI	630	THAT LADY IS COMPLETELY OUT OF HER MIND...
0002	0001	ANISHA	7630	IT WAS A DARK AND RAINY NIGHT. I TRIED TO S...

Figure 1-2 Accident driver table

Your first option is to define your table space in an 8 KB buffer pool in order to support the table containing the long column.

This option might be acceptable, but suppose that the average length of a driver's statement is fewer than 1,000 bytes.

The second option would be to create an attributive table to store the driver's statement in multiple rows that are fewer than 4 KB each. Figure 1-3 on page 11 shows how the table might look.

The benefit of this design is that the tables can be supported using a 4 KB page size, and since the average statement length is fewer than 1,000 bytes, the average accident driver row only has one accident driver statement row.

The disadvantage of the design is that the total disk space usage is greater than the first option due to the repeating keys in the attributive table and all accesses perform a join in order to get all of the accident driver information. An additional disadvantage is that any time you split the contents of a column in two, the application may have to concatenate the column back together. Splitting the column may also prevent the use of DB2 predicates such as LIKE, since a string that matches the LIKE predicate might be split across two different columns or rows.

Archived

ACCIDENT DRIVER TABLE

ACCIDENT NUMBER	DRIVER NUMBER	DRIVER NAME
0001	0001	KATHY
0001	0002	RAVI
0002	0001	ANISHA

ACCIDENT DRIVER STATEMENT TABLE

ACCIDENT NUMBER	DRIVER NUMBER	SEQUENCE NUMBER	STATEMENT LENGTH	TEXT
0001	0001	0001	1000	THERE IS NO WAY THIS ACCIDENT IS MY FAULT...
0001	0001	0002	1000	LIKE I TOLD THE OFFICER AT THE SCENE...
0001	0001	0003	40	AND THAT'S ALL I HAVE TO SAY ABOUT THAT.
0001	0002	0001	630	THAT LADY IS COMPLETELY OUT OF HER MIND...

Figure 1-3 Accident driver statement table using multiple rows to store long text

Now suppose that we also know that not only is the average statement fewer than 1,000 bytes, but fewer than 10% of statements are longer than 1,000 bytes. The third design option takes advantage of this information by putting the statement column back on the accident driver table, but only at a length of 1,000 bytes. If the statement is longer than 1,000 bytes, the length of the statement on the accident driver is set to zero and an overflow table is used to store the full statement text.

This design option is shown in Figure 1-4 on page 12. The advantage of this design is that the statement is stored with the base table for the majority of rows, and the base table is still able to use a 4 KB page size. However, an additional access is required for the 10% of rows that have longer statement lengths.

Archived

ACCIDENT DRIVER TABLE

ACCIDENT NUMBER	DRIVER NUMBER	DRIVER NAME	STATEMENT LENGTH	TEXT
0001	0001	KATHY	0	
0001	0002	RAVI	630	THAT LADY IS COMPLETELY OUT OF HER MIND...
0002	0001	ANISHA	0	

ACCIDENT DRIVER STATEMENT TABLE

ACCIDENT NUMBER	DRIVER NUMBER	STATEMENT LENGTH	TEXT
0001	0001	2040	THERE IS NO WAY THIS ACCIDENT IS MY FAULT...
0002	0001	7630	IT WAS A DARK AND RAINY NIGHT. I TRIED TO S...

Figure 1-4 Accident driver statement table with overflow

Then, just when you thought you had settled on a design you could live with, you find an additional requirement in the CRUD information. There is a high volume online selection transaction that displays up to 20 accident driver rows on the screen, including the first 40 bytes of the driver's statement. A single row is then selected and the full statement text for this row is then displayed on a following screen.

With the knowledge that over 95% of accesses to the accident driver table are going to be looking for only the first 40 bytes of the statement text, you might consider the design option shown in Figure 1-5 on page 13.

By storing the first 40 bytes of the statement text in a fixed length column on the accident driver table, you are able to support the requirements of the high-volume online transactions with accesses to only this table. Because the row is now shorter, less CPU is consumed when moving the data from DB2 to the application, and with more rows on a page, buffer pool efficiency should be improved and I/O reduced.

In our example, we shortened the length of the long string in the accident driver statement table by 40 bytes since the first 40 bytes of the column are stored in the base table. This approach avoids redundantly storing these 40 bytes. However, as mentioned above, splitting the string has disadvantages. Another option would be to store the full string in the accident driver statement table and accept the fact that 40 bytes of the string have been stored redundantly.

ACCIDENT DRIVER TABLE

ACCIDENT NUMBER	DRIVER NUMBER	DRIVER NAME	STATEMENT40
0001	0001	KATHY	THERE IS NO WAY THIS ACCIDENT IS MY FAUL
0001	0002	RAVI	THAT LADY IS COMPLETELY OUT OF HER MIND
0002	0001	ANISHA	IT WAS A DARK AND RAINY NIGHT. I TRIED TO

ACCIDENT DRIVER STATEMENT TABLE

ACCIDENT NUMBER	DRIVER NUMBER	STATEMENT LENGTH	TEXT
0001	0001	2000	T. I HAD THE RIGHT-OF-WAY, AND THIS MANIAC CAME FLYING...
0001	0002	590	IF SHE THINKS THAT THIS ACCIDENT WAS MY FAULT. I WAS DRI...
0002	0001	7590	STOP, BUT THE CAR SLID. I HARDLY TOUCHED THE OTHER VEH...

Figure 1-5 Storing the first 40 bytes of each statement on the base table

Storing XML

For a detailed discussion of the considerations when storing XML in DB2, see Chapter 7, “XML support” on page 177.

1.1.2 Nullability

When a logical attribute is defined as optional, further consideration is required to determine if the column should allow null values. All data types include the null value. Distinct from all not null values, the null value denotes the absence of a value.

A DB2 column allows nulls by default if the NOT NULL clause is omitted. Logical attributes that are mandatory should always be implemented as columns that specify NOT NULL or NOT NULL WITH DEFAULT.

Allowing null values in a column does not save space. To the contrary, a one byte null indicator is added to the row for each nullable column. For this reason, and because of the additional coding required to handle nulls in the application, they should not be used indiscriminately.

Archived

When nulls should be used

The following cases examine scenarios in which nulls might be required, the alternatives to using nulls in the scenario, and the criteria that should be considered when making the choice.

- Optional foreign keys with *referential integrity* (RI) enforced

  When RI is enforced in DB2, a row must exist on the parent table that matches each not null foreign key on the child table. By defining the foreign key column as nullable, you can support optional RI relationships. Depending upon the design of the application, it may be necessary to define optional RI relationships for child rows that may be inserted before the parent.

  The alternative to supporting an optional RI relationship without a nullable column is to store a dummy value in the parent table that could be referenced by all child rows that do not have a real foreign key value. This typically leads to additional application code or predicates in the SQL to avoid returning the dummy value. In this situation, we recommend the use of a nullable column.

  Another alternative to using a nullable column to represent an optional foreign key is to remove the referential integrity constraint from the database. This is not recommended if the referential integrity is then enforced in the application, but, in some circumstances, this is an acceptable solution. For more information about whether or not to consider this option, see “Referential constraints” on page 130.

- Optional character data

  In most cases, the non-existence of optional character data can be represented by spaces. Only if a value of space falls in the domain of the column and has significance that must be distinguished from non-existence should a character column be defined as nullable. VARCHAR columns can use a string length of 0 to represent no value, and, therefore, should never be defined as nullable.

- Optional numeric data

  For some numeric data, a value of 0 is sufficient to identify the non-existence of a value. However, consider the following before ruling out the use of nulls:

  Are scalar functions, such as MIN, MAX, and AVG, applied to this numeric column?

  If so, realize that a value of 0 is returned for the MIN function and may skew the result of the AVG function if it was meant to calculate the average of non-optional columns. Additional code or SQL predicates are required to exclude the rows with a value of 0 from the function, where a null value is automatically excluded.

  Assume that column EDLEVEL on table EMP represents an optional attribute. Example 1-1 demonstrates two SQL techniques that can be used to find the minimum EDLEVEL if the EDLEVEL column is defined as NOT NULL and a value of 0 is used to represent the non-existence of a value.

Example 1-1 SQL to find the minimum EDLEVEL if defined as NOT NULL WITH DEFAULT 0

```
-- An example SQL to filter out the default value on the WHERE clause

  SELECT MIN(EDLEVEL)
   INTO :HV-EDLEVEL:HV-NULLIND
   FROM EMP
  WHERE WORKDEPT = :HV-WORKDEPT
    AND EDLEVEL <> 0

-- An example SQL to treat the EDLEVEL as NULL if the value is 0

 SELECT MIN(NULLIF(EDLEVEL,0))
```

```
  INTO :HV-EDLEVEL:HV-NULLIND
  FROM EMP
 WHERE WORKDEPT = :HV-WORKDEPT
```

Example 1-2 shows the SQL that can be used if the EDLEVEL column is defined to allow nulls. This simpler statement requires fewer predicates and eliminates the possibility of erroneously returning the default value. Note that both of these statements require the use of an indicator variable, because the result of the MIN function can be null even if the column is not defined as nullable. Example 1-6 on page 17 contains a rewritten version of this SQL statement that does not require indicator variables.

Example 1-2 SQL to find the minimum EDLEVEL if null values are allowed

```
SELECT MIN(EDLEVEL)
  INTO :HV-EDLEVEL:HV-NULLIND
  FROM EMP
 WHERE WORKDEPT = :HV-WORKDEPT
```

Does the value of 0 fall within the domain of the attribute?

Consider a column that is meant to store an optional temperature measurement. It is probably necessary for the application to distinguish between a temperature reading of 0 and the non-existence of a temperature reading. In this case, the column should be defined to allow null values.

- Optional date and timestamp columns

  In DB2, DATE, TIME, and TIMESTAMP columns require that a valid value is provided on INSERT or UPDATE; otherwise, the row is rejected. While a dummy value such as ‘0001-01-01’ or ‘9999-12-31’ can be used to represent the non-existence of a date, it can cause complications similar to those encountered with optional numeric columns.

  When using a dummy value in a DATE, TIME, or TIMESTAMP column, it is recommended to consistently use the same value as the default. However, when two different date columns are used as a range, it might simplify application coding to default the beginning range to ‘0001-01-01’ and the ending range to ‘9999-12-31’. In the case of a date range, defining the column as nullable might complicate the application coding.

  Suppose that table EMP_HIST contains columns EFFDATE and ENDDATE to represent the beginning and ending date range represented by the row, and it is possible that both the EFFDATE and ENDDATE might be unknown. If a null value is used to represent the unknown state of each of these columns, the SQL required to retrieve the row effective on a specific date might look something like that shown in Example 1-3.

Example 1-3 Date range SQL with nullable columns

```
SELECT FIRSTNME,
       MIDINIT,
       LASTNAME,
       WORKDEPT
  FROM EMP_HIST
 WHERE EMPNO = :HV-EMPNO
   AND (EFFDATE IS NULL OR
        EFFDATE <= :HV-DATE)
   AND (ENDDATE IS NULL OR
        ENDDATE >= :HV-DATE)
```

If instead of nulls, a default value of ‘0001-01-01’ is used for the EFFDATE and a default value of ‘9999-12-31’ is used for the ENDDATE, the SQL can be coded as shown in Example 1-4 on page 16.

Example 1-4 Date range SQL with default values in not nullable columns

```
SELECT FIRSTNME,
       MIDINIT,
       LASTNAME,
       WORKDEPT
  FROM EMP_HIST
 WHERE EMPNO = :HV-EMPNO
   AND EFFDATE <= :HV-DATE
   AND ENDDATE >= :HV-DATE
```

Indexing considerations for nullable columns

When creating an index on any set of columns that contains a nullable column, you have the ability to specify UNIQUE WHERE NOT NULL. This clause tells DB2 to enforce uniqueness across all not null values in the index, while any number of null values are allowed. For uniqueness purposes, an index entry is considered null if any column value in that index entry is null. There is no way to duplicate this uniqueness enforcement for not nullable columns that use a default value to represent optionality without expensive application code.

You should also be aware of the effect of null values on index splitting. DB2 uses a special rule to prevent leaf page splits at the end of the index when entries are being inserted into the index in an always ascending order. If the new entry is the highest value in the index and the last index leaf page is full, no page split occurs. A new leaf page is created and the new entry is placed there. In all other cases, when an index leaf page is full, the page is split, which means that half of the entries from the full leaf page are moved to the new leaf page.

When an index nullable column is null, the first byte of the index entry contains X'FF', which means that the null value is the highest possible value in the index. In this case, the presence of a null value in that column results in a split of the last index leaf page, even if the new entry is the highest not null value. Splitting index pages when the insert activity is constantly ascending results in excessive index growth as the split pages are left half-empty and never filled with new values. This same behavior would result from the use of the highest value as the default value for a column.

If the index order is defined as DESCENDING, then the reverse is true. The null value or highest default value would occupy the first position in the index, and a low value default would occupy the last position in the index.

Handling nulls in the application

Null values present a special challenge to the application. It is necessary to understand how to handle null values, even if all of the columns are defined with the NOT NULL clause. Scalar functions, such as the MIN, MAX, AVG, or SUM in SQL statements that do not qualify any rows, return a null value. In addition, columns selected from a null-providing table in an outer join can be null.

When a column contains a null value, the only predicate on that column that qualifies the row is the IS NULL clause. Two columns that both contain a null value are not considered equal to each other. The IS NOT DISTINCT FROM comparison predicate is true if two columns containing null values are compared. Assume that COLA and COLB are nullable columns from two different tables. Table 1-5 on page 17 shows the difference between the EQUAL predicate and the IS NOT DISTINCT FROM predicate when comparing these two columns.

Table 1-5 The difference between EQUAL and IS NOT DISTINCT FROM

Value		Predicate	
COLA	**COLB**	**EQUAL**	**IS NOT DISTINCT FROM**
1	1	TRUE	TRUE
NULL	**NULL**	**FALSE**	**TRUE**
1	NULL	FALSE	FALSE
1	2	FALSE	FALSE

When a nullable column is selected, DB2 passes an indicator variable to the program along with the column. The program must have an indicator host variable defined and coded in the INTO clause, or the SQL statement returns a -305 SQL code. Example 1-5 demonstrates two different ways that indicator variables can be coded in an SQL statement.

Example 1-5 Two equivalent ways to code indicator variables

```
SELECT MIN(EDLEVEL)
  INTO :HV-EDLEVEL:HV-NULLIND
  FROM EMP
 WHERE WORKDEPT = :HV-WORKDEPT

SELECT MIN(EDLEVEL)
  INTO :HV-EDLEVEL INDICATOR :HV-NULLIND
  FROM EMP
 WHERE WORKDEPT = :HV-WORKDEPT
```

The value in the indicator variable is zero if the column value is not null, and is less than zero if the column value is null. Positive values in indicator variables are the result of string truncation, and the indicator variable is set to the original length of the string.

Tip: When checking an indicator variable for a null value, always use a < 0 comparison because the indicator could contain either a -1 or a -2, and other negative values could be added.

As an alternative to the use of an indicator variable, it is possible to avoid the -305 SQL code by using the VALUE, COALESCE, or IFNULL function to replace null values with a literal. Assuming that the EDLEVEL column allows null values, the SQL statement in Example 1-6 shows how the IFNULL function can be used to replace a null value with a literal and you can avoid the need to code an indicator variable.

Example 1-6 Replacing null values with a literal

```
SELECT IFNULL(MIN(EDLEVEL),0)
  INTO :HV-EDLEVEL
  FROM EMP
 WHERE WORKDEPT = :HV-WORKDEPT
```

Refer to the *DB2 UDB for z/OS Version 8 SQL Reference,* SC18-7426-02, for more information about the VALUE, COALESCE, IFNULL, and IS NOT DISTINCT FROM functions.

1.1.3 The effect of column placement on performance

The order of columns in a DB2 table can impact performance in two ways.

- Any column placed after a variable length column in the row has a variable offset. DB2 must calculate this offset every time the column is referenced. The cost of this calculation is 5 to 15 instructions per column.

  For the best performance, place frequently referenced columns before variable length columns in the rows. This is particularly true for columns that are qualified using stage 2 predicates, because the offset of these columns must be calculated prior to stage 2 predicate evaluation. If many rows are evaluated and few rows qualify, this is a significant cost.

- Column placement can affect the amount of data that is logged when a row is updated. This is particularly true for variable length rows. A variable length row is a row that contains a variable length column, or a row that is compressed. The length of a compressed row can change with an update to any column. The length of an uncompressed row only changes if a variable length column changes. Table 1-6 describes DB2 logging of updated rows based on row type.

Table 1-6 DB2 logging of updated rows

Row type	Logging starts with ...	Logging ends with ...
Fixed length row - no compression	first updated column	last updated column
Variable length row with no length change	first changed byte	last updated column
Variable length row with length change	first changed byte	end of the row

Note: The use of DATACAPTURE(CHANGES) and certain advisory REORG conditions caused by online schema changes may result in additional logging.

When choosing column placement, there are several tradeoffs that must be considered. The the column placement guidelines are:

- Place variable length columns at the end of the row.
- When there are multiple variable length columns in the row, order them according to their likelihood to change length. Place the variable length columns that are most likely to change length last.
- Place columns that are frequently updated together in clusters and after the variable length columns if you want to reduce logging.
- Place columns qualified with stage 2 predicates ahead of variable length columns, even if they are updated frequently, if the ratio of select to update access is high and the stage 2 predicates are expected to disqualify a significant percentage of rows.

1.1.4 Key design

Each entity should have a primary key according to relational theory. While this rule is not enforced in DB2, and there are certainly instances where DB2 tables without a primary key can be used efficiently, for the purpose of this section, we assume that every entity from the logical model has one or more candidate keys identified.

A *candidate key* is a set of one or more naturally occurring attributes that uniquely identifies one row in an entity. “Naturally occurring” means that the attributes exist in the logical model as defined business data elements.

A *surrogate key* is a set of one or more attributes that uniquely identify one row in an entity, but this set is not made up entirely of naturally occurring attributes. In other words, a surrogate key can be a single artificially assigned attribute or it can be a sequence number added to a set of naturally occurring attributes to make the row unique.

From a performance perspective, it is best to first consider the naturally occurring candidate keys as primary keys first, and only resort to a surrogate key when necessary. There are several reasons for this recommendation:

- Using a surrogate key increases the row length.

  By definition, a surrogate key contains an attribute that is added to the table strictly for the purpose of uniquely identifying a row. If a candidate key is chosen as the primary key, there is no need to add this attribute to the table.

- Using a surrogate key may require an additional index.

  The candidate keys identified in the logical model are frequently attributes that are used as search criteria. If a surrogate key is chosen as the primary key, there may still be a need to index the candidate key columns for performance reasons.

- Using a surrogate key may cause the child table to be clustered differently than the parent.

  The primary key of a table propagates down into all child tables as a foreign key. If the primary key is also chosen as the clustering index, it is possible to also cluster the child tables using the same values. This can reduce I/O and elapsed time by enabling prefetch.

  A surrogate key, particularly a single column unique identifier, is typically not the best choice for clustering.

Consider a surrogate key when:

- There is no unique candidate key.

  Sometimes the naturally occurring candidate keys do not guarantee uniqueness. For example, consider an entity called PERSON containing attributes LASTNAME, FIRSTNAME, MIDINIT, and BIRTHDATE that have been identified collectively as a candidate key. While this may be unique in most cases, there is no way to guarantee that your application is not presented with two different rows containing identical data in these columns. Unless there is a business rule that these attributes must be unique, you probably do not want to use these columns to uniquely identify a row in your table.

- All or part of the candidate key can be or contains null.

  No column in the primary key of a DB2 table can allow null values. When choosing a primary key, you must eliminate any columns that allow nulls. Remember that a foreign key must allow nulls if the child row can exist before the parent has been inserted. If this rule disqualifies all of the candidate keys, a surrogate key must be used.

- The size of the candidate key is unreasonably large.

  If the candidate key contains a large number of columns to achieve a guaranteed uniqueness set, it may be larger than you would like. Although DB2 can support a unique index of up to 2,000 bytes, significant savings may be achieved if a shorter surrogate key is used to replace the candidate key; particularly, if it is propagated down to child tables as a foreign key.

- All or part of the candidate key is updated.

  While DB2 allows the update of a primary key, it does not allow the update of a primary key that has child rows referencing it. In this case, updating the primary key means first deleting all of the child rows, updating the key, and then re-inserting the child rows with the new value. If your candidate key is updatable, consider replacing it with a surrogate key.

For techniques that can be used to generate a unique surrogate key, see Chapter 5, “Techniques to generate numbers” on page 137.

Version control and keeping history: Case study

A common challenge in the design of DB2 tables is the storage of previous values of attributes as they change over time. There are basically two flavors of this scenario depending upon the business rules and how the data is used by the applications.

The first flavor is version control, in which multiple versions of the data need to exist to support trailing activity, or operational transactions that need to know the values of the attributes at a previous time. An example of this is an automobile insurance policy database. The policy coverage may change over time, but when a claim is presented, it is necessary to find the coverage that was in effect on the date the accident occurred, even if the coverage has subsequently changed. Another characteristic of version control is to support future versions of the data.

The second flavor is historical data, where the requirements are to retain previous values of attributes for auditing or research purposes. In this case, operational transactions are always looking for the current version of the data, and the historical data is used for reporting and research transactions.

Typically, the business requirement of retaining historical data is designed in the logical model as a sequence number or effective date component as part of the candidate key. However, the table design and key structure used to support version control may be very different than that of historical data. In the following case study, we examine the design considerations of both approaches using a table that contains Employee data.

Suppose that Figure 1-6 on page 21 represents the logical design of the Employee Table. We know that the EFFDATE attribute was included in the primary key, because there is a business requirement to keep a history of the previous values of the FIRSTNME, MIDINIT, LASTNAME, WORKDEPT, JOB, EDLEVEL, SALARY, BONUS, and COMM attributes.

If this table were implemented as designed, whenever one of the attributes of an employee is updated, the application would perform the following actions:

1. Insert a new row into the table with the updated attributes, an EFFDATE equal to the date the change is effective, and an ENDDATE of ‘9999-12-31’.
2. Update the ENDDATE of the previously effective row to a value of EFFDATE - 1 DAY.

This design meets the business requirements of both version control and historical data, so what is the big deal? Why would I not just implement this “one size fits all” design? Because a “one size fits all” design leads to “one size fits all” performance. And, sometimes, “one size fits all” performance is not good enough.

EMP			
EMPNO	CHAR(6)	<pk>	not null
EFFDATE	DATE	<pk>	not null
ENDDATE	DATE		not null
HIREDATE	DATE		not null
SEX	CHAR(1)		not null
BIRTHDATE	DATE		not null
FIRSTNME	CHAR(12)		not null
MIDINIT	CHAR(1)		not null
LASTNAME	CHAR(15)		not null
WORKDEPT	CHAR(3)		not null
JOB	CHAR(8)		not null
EDLEVEL	SMALLINT		not null
SALARY	DECIMAL(9,2)		not null
BONUS	DECIMAL(9,2)		not null
COMM	DECIMAL(9,2)		not null

Figure 1-6 EMP table with effective date as part of the primary key

Suppose that our EMP table is a parent table in a referential relationship. With this design, the EFFDATE is propagated down to all dependent tables. When an update is made to the attributes of the EMP table and a new active EMP row is created, an entire new set of dependent rows must be created with the new EFFDATE. This may be acceptable if the data on the dependent rows is all updated at the same time; however, in most cases, the dependent rows are identical, except for the new EFFDATE foreign key. To make matters worse, there may be version control or historical data requirements on the dependent tables as well, leading to another set of EFFDATE/ENDDATE columns. If there is then another level of dependent tables, there are two EFFDATE columns propagated down, and the potential for even more rows that must be replicated when the parent is updated.

What you would really like to do is maintain a single set of dependent rows and code the application to assume that the dependent rows are implicitly related to all subsequent versions of the parent. However, you realize that the old versions of the employee data are deleted after five years. When that happens, what do you do with the dependent rows that are referentially tied to your historical EMP row? You cannot delete them, because they are still active and logically associated with your current EMP row.

Okay, so what if you update the EFFDATE on the dependent rows every time the EMP table is updated? If the foreign key is not part of the primary key of the dependent row, this works. However, that is still an excessive amount of work for the update EMP transaction. Time to drop the referential constraint and enforce the referential integrity in the application? No, not yet.

What if we change the key structure of the EMP table, and put the ENDDATE in the primary key instead of the EFFDATE? Your first reaction might be to say that we cannot do this, because the ENDDATE column is updated. However, we can change the application process, so that the ENDDATE is not updated by using the following process whenever one of the attributes of an employee is updated:

1. Insert a new row into the table using the values from the currently active row and an ENDDATE of the date the change is effective - one day.
2. Update the attributes of the current row to the new values and an EFFDATE of the date the change is effective.

With this design, all dependent tables are always referentially tied to the parent row with the ENDDATE of ‘9999-12-31’. When an update occurs, this row is always updated, and the previous version of the attributes is inserted as a new row. This design supports the requirement of both version control and historical data, and it eliminates the problems

associated with the propagation of the always changing EFFDATE to the dependent rows. There are special considerations for indexing this design, because it has a high value default in an indexed column. For that reason, any index on the ENDDATE column should be defined in descending order.

As we mentioned earlier, there are two flavors of storing previous values. Our latest design supports both, but the requirements for version control are more expensive to support than the requirements for historical data. Since we know that the only requirement for our EMP table is to capture the previous values of the FIRSTNME, MIDINIT, LASTNAME, WORKDEPT, JOB, EDLEVEL, SALARY, BONUS, and COMM attributes for historical purposes, let us see if there is a more efficient design.

From our CRUD information, we discover that over 90% of the accesses to the EMP table are going to be looking for the row effective on the current date. Only one online screen and the monthly batch job that deletes historical rows older than five years are looking for history rows.

With this information, we can make the following physical design changes that have several performance benefits:

- Remove the ENDDATE from the EMP table.

  The EMP table now only contains the current version of the employee data. The primary key is only the EMPNO. The ENDDATE is no longer needed, because the value would have always been set to '9999-12-31' for the currently active row. The EFFDATE is still needed to identify the date that the currently active row became effective.

  - Removing the ENDDATE from the primary key reduces the size of the unique index on EMP.
  - Only the EMPNO column is propagated to the dependent tables, reducing the row length of the dependent table and the size of the index on the dependent table's foreign key.
  - Removing historical data from this table reduces the number of rows and allows more efficient access for the 90% of application accesses that are only looking for the current values.
  - Secondary indexes that are added to the EMP table to support application access do not have entries for the historical rows.
- Create a second table called EMP_HIST containing those attributes which have historical requirements. This table only stores the historical data. An "update" function in the applications inserts a row into EMP_HIST with the old values from EMP, and then updates the current row on EMP.
  - By storing history for only these attributes, we avoid the redundancy of storing multiple instances of data that has not changed.
  - Because the EMP_HIST table stores only history, the monthly process to delete historical data accesses only this table. Locking contention between the monthly delete process and applications accessing the EMP table is eliminated.
- The columns in EMP are rearranged so that the most updated columns are together and at the end of the row to reduce the amount of data logged.

Figure 1-7 on page 23 shows a redesigned EMP table and a new table called EMP_HIST that stores only the historical data.

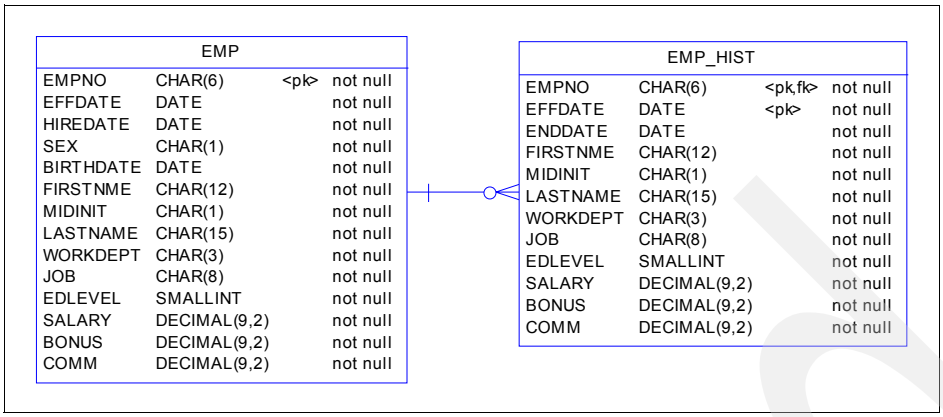


Figure 1-7 EMP and EMP_HIST tables

The purpose of this case study is to demonstrate that, while a simple design can support all of the business rules and application requirements, applying thought to the table design to consider alternative key structures may result in a better performing application. As always, physical database design decisions should be based upon the requirements and metrics of your unique application.

1.1.5 Choosing an encoding scheme

DB2 supports three different encoding schemes that you can specify in the CCSID parameter on the CREATE TABLESPACE statement. These encoding schemes are EBCDIC, ASCII, and UNICODE.

Carefully consider the following when choosing an encoding scheme:

- The client scheme

  The first and most important consideration should be the data requirements of your users. The format of the data before it reaches the database and the format in which it is presented to the actual user should be your primary influence.

  If your client scheme contains multiple different character sets in a single column, UNICODE is the only encoding scheme that can support them all.

  Each of the different encoding schemes sorts data differently. Some installations have used FIELDPROCs to influence the sort order of DB2 data, but, in general, you should favor the encoding scheme that best supports the requirements of your client applications.

- The encoding scheme of related data in your environment

  There is overhead required within DB2 to convert data from one encoding scheme to the other when tables with different CCSIDs are joined. If possible, choose an encoding scheme that avoids this overhead.

- The encoding scheme of the applications

  If the majority of your applications are implemented in Java, you may consider storing user data in Unicode tables to reduce the costs of code page conversions.

1.1.6 Volatile tables

A volatile table is one whose *cardinality*, or volume of rows, varies significantly during its life. For example, a transactional table that begins empty, accumulates data over some period of time, and is then deleted, is a volatile table. Volatile tables present a tuning challenge because the statistics in the catalog can only represent one state of the table as it progresses

through this cycle. For best performance, it is necessary to ensure that the access paths chosen by applications accessing a volatile table satisfy the SLA at any point in the cycle.

Introduced in DB2 UDB for z/OS Version 8, the VOLATILE table attribute specifies that index access should be used on this table whenever possible for SQL operations. By defining a table with the VOLATILE attribute, it is possible to achieve index access without maintaining statistics in the catalog. While this may satisfy the performance requirements of simple SQL access, it is important for you to know that list prefetch and certain other optimization techniques are disabled when the VOLATILE attribute is used. In addition, when a table defined with the VOLATILE attribute is used in a join, having accurate catalog statistics may be necessary to ensure the most efficient join order.

The NPGTHRSH DSNZPARM is a subsystem parameter that influences the optimizer to favor index access for small or volatile tables. While it is true that there are cases when an index access is slower than a table space scan, always using index access for volatile tables is a way to ensure that the worst-case performance scenario is still acceptable. Before considering changes to NPGTHRSH or any other DSNZPARM, remember that it affects all applications in the DB2 subsystem. The value that would provide the best performance for your application may not provide the best performance for the subsystem as a whole. Refer to the *DB2 UDB for z/OS Version 8 Administration Guide,* SC18-7413-02, for the possible values of the NPGTHRSH DSNZPARM and their effect.

Tip: For tables that you know are volatile, consider the use of the VOLATILE attribute rather than relying on the NPGTHRSH DSNZPARM to influence index access.

1.2 Deviating from the third normal form

Sometimes, it may be necessary to make changes to the level of normalization in the logical model in order to achieve optimum performance. In other words, split a logical entity into two tables, combine two logical entities into one table, duplicate logical attributes into columns in multiple tables, or add physical columns that did not exist as attributes in the logical model. Every situation is different and there is no such thing as a single solution that fits all needs, but, in this chapter, we provide case studies showing denormalization and explaining the effect on performance. It is the responsibility of the physical database designer to choose to implement solutions that best fit the needs of your applications.

1.2.1 Adding columns or tables to support application requirements

When performing physical database design, you may encounter application processing requirements that are not entirely supported by the data in the logical model. This occurs because the purpose of the logical model is to represent the business data and the business rules. Keeping this model “pure” means not allowing specific application processing requirements to “pollute” the logical model with attributes that have no meaning to the business.

A simple example of a column added to support application processing requirements is a maintenance timestamp column on every table that is used by the application process to ensure data integrity if blind updates are performed.

Derived columns

A derived column is a column whose value can be derived using the value from other columns in the same or related rows. The reasons to use derived columns are typically to improve the performance of application processes in the following ways:

- Reduce the number of tables or rows accessed
- Improve access path selection and the performance of data retrieval
- Achieve greater control of the point in time when data is derived

These benefits come with an associated price tag. In general, derived columns have the following costs:

- By definition, a derived column is storing data that could be calculated, so the additional disk space required to store the redundant column is the most tangible cost.
- There is a cost associated with updating the derived column any time there is a change to the source data from which the column is derived.
- Any time you use a derived column, the possibility exists that a derived column can become out of sync with the source data from which it is derived.

  In order to ensure data integrity, it may be necessary to code application processes to periodically verify that the derived values are correct. Another option is to perform the maintenance of the derived column in a trigger.

This section discusses several of the most common types of derived columns. Aggregation columns and tables, which are a special type of derived columns, are discussed in "Aggregation" on page 27.

Indicator columns

An indicator column is a column that is typically used to indicate a true or false condition within the data of other related columns or rows. For example, an existence indicator column in a parent table may have a value of 'Y' or 'N' depending upon the existence of one or more rows in a child table. The benefit of the indicator is eliminating the need to perform a SELECT statement on the child table if the indicator is 'N', or avoiding accessing the table altogether if there is an application requirement that simply needs to know if a child row exists.

Another type of indicator may be used to avoid complex calculations. Suppose your database stores information about prepaid gift cards. The CARD table is used to store the information about each particular prepaid gift card and the TRAN table is used to store all transactions that have occurred using that card. A card is considered "active" if it has a positive balance remaining. Once a day, there is an application process that must count the number of active cards in the CARD table. This process could add and subtract all of the debit and credit transactions for each card to determine if the card has a positive balance, but this would require touching each row on the CARD table as well as each row on the much larger TRAN table. Adding an indicator column on the CARD table called ACTIVE_IND and updating the value whenever a transaction is added would prevent the need to access the TRAN table to perform the calculations.

Another option is to store a BALANCE derived column on the CARD table, which would be calculated by summing the initial value of the card and all subsequent debit and credit transactions. This column can be used to answer the number of active cards question as well as the total outstanding balance question.

Transforming data for indexing

In some circumstances, the pure business data does not lend itself well to the application processes that must access it. For example, the business requirements for a name may require that it is stored with any suffixes, such as 'JR' or 'SR', but these suffixes do not lend

themselves well to name searches. Additionally, some applications use phonetic algorithms to search on names. Because the phonetic name is calculated using a fixed formula against the data in the name column, it is considered a derived value. However, without storing this derived value, you cannot index it. Your access path to perform a search where phonetic name = ‘some value’ is:

1. Scan all rows in the table to get the name column.
2. Apply the formula to derive the phonetic name.
3. Compare the derived phonetic name to the search value to determine if the row qualifies.

If this process is only performed once a day in a batch task, the performance may be acceptable. However, in order to support high-volume online searches, the derived column must be added to the table and indexed.

A similar circumstance is when an application must always return rows in a specific order, but that order does not match the natural order of the encoding scheme of the data. As a simple example, consider a column called NUM that contains the values ‘ONE’, ‘TWO’, ‘THREE’, ‘FOUR’, or ‘FIVE’. The meaning of the word implies an order that is different than the order that would result from coding ORDER BY NUM. This problem could be solved using the SQL shown in Example 1-7, but it always results in a sort.

Example 1-7 SQL to replace data value with sequence for ordering

```
SELECT NUM,
       CASE NUM
            WHEN ‘ONE’
            THEN 1
            WHEN ‘TWO’
            THEN 2
            WHEN ‘THREE’
            THEN 3
            WHEN ‘FOUR’
            THEN 4
            WHEN ‘FIVE’
            THEN 5
       END AS SQNC_NR
  FROM T1
ORDER BY SQNC_NR
```

If a derived column called SQNC_NR is added to the table to represent the sequence in which the rows should be ordered, an index can be built on this column so that a sort can be avoided. This is a silly example, but is the same scenario that is frequently found in many data models. In a fully normalized model, the SQNC_NR would replace the NUM column in the table and a look-up table would be used to translate the SQNC_NR to the NUM value.

State transition indicators

When rows in a single table transition from one state to another as application processes are applied to them, it may be necessary to add a state indicator that communicates the current state of the row. State indicators are not derived values, since there may be no other way to identify the state of the row using the values in the other columns. A state indicator is also distinctly different from a status code that is used by business processes.

When state transition always results in a final state, and the majority of the rows in the table are going to always be in that state, an alternative implementation of state transition might be multiple tables, in which the rows in transition are held in a separate table that contains a state indicator. Once they have reached the final state, the rows are moved into the other table that has no state indicator.

Another implementation of state transition is a work in progress table that externally tracks the state of the rows as they are transformed. See “Work in progress tables” on page 249 for a detailed description of this technique.

Type codes

When denormalizing logical entities, as discussed in “Normalization and denormalization” on page 34, the resulting DB2 table may contain multiple different types of rows. In this case, it is necessary to add a type code column to identify the type of each row in the table as shown in Figure 1-11 on page 36 and Figure 1-12 on page 39. These type codes are used to distinguish between multiple logical entities contained in the same physical table.

Creating replicas

In some scenarios, there may be a need to create a replica of an entire table or a portion of a table, meaning selected rows or columns. An example of this may be when ad-hoc queries need to access transactional data that is heavily updated. Allowing long-running ad-hoc queries against the transactional data may result in an unacceptable amount of lock contention. If the latency requirements of the ad-hoc queries are low, creating a replica allows the ad-hoc queries to run against an asynchronously updated copy of the transactional data without any impact to the high-volume processes accessing the primary copy. Additional indexes can then be placed on the replica to support the access requirements of the ad-hoc queries.

When using this type of replication, you should adhere to the following guidelines:

- The replica should be read-only.

  Exception is just the propagation process that moves the data from the primary copy to the replica.

- Constraints should not be defined on the replica.

  Any constraints on the data such as uniqueness, referential, or domain constraints should have been enforced on the primary copy. Replicating the enforcement would only add to the cost of the replication.

1.2.2 Aggregation

Aggregation is the process of summarizing data from multiple rows and possibly multiple tables into a single value or set of values. By design, DB2 can perform aggregation functions simply through the power of SQL. However, the cost in terms of CPU consumption and elapsed time of these aggregation functions may be prohibitive, and, in some cases, there may not be sufficient work space available to complete the query. For that reason, you may need to add derived columns or tables to your physical design to store aggregated values. Another option also discussed in this section is the use of materialized query tables (MQT) to support your application’s aggregation requirements.

When designing tables, it is important to identify the aggregation requirements of the application processes in the CRUD and answer the following questions that determine the effect of aggregation on your table design:

- How often is the aggregated data selected?

  How many times per hour, day, or month is the result of the aggregation going to be selected?

- What are the latency requirements of the select processes?

  Can the aggregation result from last night be used throughout the day, or must the data be current at the time it is selected?

- What are the stability requirements of the select processes?

  Because aggregations functions can take a long time, consider the possibility that the data that is being aggregated might be changing while the aggregation is performed. This means that by the time the result is displayed on the screen, it may be obsolete.

  Performing aggregations that require high stability requires a very restrictive locking scheme that may impact concurrency.

- What is the cost to perform the aggregation using the base data?

  How long does it take to read and summarize all of the data? “Determining the duration of your process” on page 113 demonstrates techniques that can be used to estimate the performance of different design options.

- What is the volatility/frequency of update of the data being aggregated?

If you determine that it is not possible or practical to perform the aggregation function every time the aggregated data is needed, then you should consider one of the following options to support your aggregation requirements:

- Store aggregate information as an additional column on the table at the level which the aggregation is needed.

  Sometimes aggregation is required at multiple levels.

- Create a new table to store the aggregation data.
- Materialized query table.

All of these options are discussed in the following case study, which uses the data model shown in Figure 1-8.

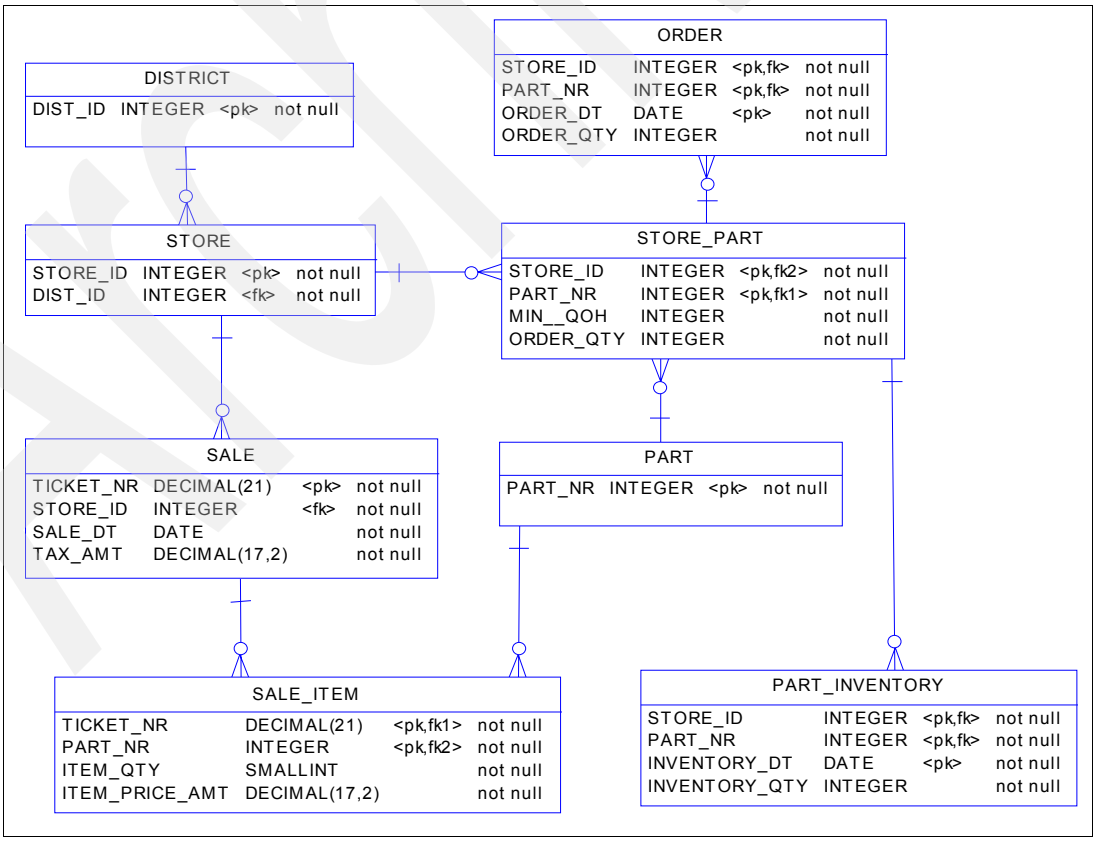


Figure 1-8 Data model requiring aggregation

The following aggregation requirements have been identified:

- After every sale, the Quantity on Hand (QOH) of each part sold must be recalculated and compared to the MIN_QOH on the STORE_PART table to determine if an electronic order needs to be placed.
- The financial applications use the following aggregated data items in online screens and batch reports. These applications do not look at the current day's numbers as the sales are occurring, only previous day and before going back five years. The data in the base tables is retained for five years.
  - Daily sales by store and district
  - Calendar month sales by store and district
  - Calendar quarter sales by store and district
  - Calendar year sales by store and district

First, we examine the characteristics of the quantity on hand (QOH) aggregation:

- The cost to perform the aggregation

  The calculation of QOH is as follows:

  Read the PART_INVENTORY table using the STORE_ID and PART_NR to find the row with the maximum INVENTORY_DT to get the INVENTORY_QTY of the part at the last inventory. This is the beginning QOH value.

  Next, the ORDER_QTY of each ORDER row for the STORE_ID and PART_NR with an ORDER_DT equal to or after the INVENTORY_DT is added to the QOH.

  Finally, a join between SALE and SALE_ITEM is performed to find all sales for the STORE_ID and PART_NR and with a SALE_DT equal to or after the INVENTORY_DT. The ITEM_QTY from each of these rows is subtracted from the QOH value.

  Because inventory is typically performed only once per year, this process could be reading through up to an entire year's transactions. The cost of the aggregation is high.
- Access frequency and volatility

  The QOH value for a part in a particular store is required every time the part is sold in the store, or once for every SALE_ITEM row that is inserted. This means that the update and access of the aggregation occur at the same frequency.

  Our estimated number of SALE_ITEM rows per day is 3.5 million, so that is the number of times the QOH calculation needs to be performed.
- Latency requirements

  The QOH calculation requires zero latency. In other words, it must be performed at the time of the sale to determine if an electronic order needs to be placed.

After estimating the performance of the aggregation process it is clear that the aggregation cannot be performed every time and meet the specified business requirements. One option might be to ask if the business requirements could be changed. For example, if the latency requirement were changed so that the QOH value of each part sold could be calculated once a day or even once every hour instead of after every sale, it might be possible to perform the aggregation.

Assuming that the business requirements cannot be changed, you must alter the physical design to satisfy the requirements. You need to add the QOH attribute to your data model to store the value for each part in each store so that it can be updated by each transaction. There are two ways to add this attribute to the data model:

- Add a new table to store the QOH
- Add the QOH value to an existing table

The QOH needs to be kept for each store and part, so the key of the table should be STORE_ID and PART_ID. There is an existing table with this key - STORE_PART. Furthermore, the STORE_PART table is already being accessed as part of the transaction to compare the updated QOH to the MIN_QOH. Therefore, the best option is to add the QOH column to the STORE_PART table.

Storing the aggregated QOH value on the database means that it must be maintained. This could be done in the application process, but a better option might be to use a trigger on the SALE_ITEM table and the ORDER table that automatically updates the QOH value for a store and part whenever a row is inserted. In addition, because the QOH and MIN_QOH values are on the same row, another trigger could be used to perform the electronic order whenever the QOH is updated to a value that is less than the MIN_QOH. See “Using triggers” on page 168 for more information on using triggers.

Because of the volume of updates that occur to the QOH values it is possible that storing the aggregated value on the STORE_PART table, creates hot spots, or pages that have potential concurrency problems. For more information about working with hot spots, see “Hot spots” on page 353.

The final consideration when storing the aggregated value in the database is the integrity of the data. In some cases, it might be necessary to create a process to periodically verify and update the aggregated value. In the case of our QOH column, it is updated to the INVENTORY_QTY every time a new PART_INVENTORY row is inserted.

Next, we examine the characteristics of the daily, monthly, quarterly, and yearly aggregations required by the financial applications:

- The cost to perform the aggregation

  The calculation of the sales by store is shown in Example 1-8.

Example 1-8 Sales by store calculation

```
SELECT SUM(B.ITEM_QTY*B.ITEM_PRICE_AMT)
  FROM SALE A, SALE_ITEM B
 WHERE A.STORE_ID = :HV-STORE-ID
   AND A.SALE_DT BETWEEN :HV-INTERVAL-START AND :HV-INTERVAL-END
   AND B.TICKET_NR = A.TICKET_NR
```

  The calculation of the sales by district is shown in Example 1-9.

Example 1-9 Sales by district calculation

```
SELECT SUM(B.ITEM_QTY*B.ITEM_PRICE_AMT)
  FROM SALE A, SALE_ITEM B, STORE C
 WHERE C.DIST_ID = :HV-DIST-ID
   AND A.STORE_ID = C.STORE_ID
   AND A.SALE_DT BETWEEN :HV-INTERVAL-START AND :HV-INTERVAL-END
   AND B.TICKET_NR = A.TICKET_NR
```

- Access frequency

  The sales information for any store or district can be requested for any interval through an online transaction. The volume of online financial transactions is approximately 1,000 per day.

  A daily batch report always needs the sales totals for the previous day, the current month, the current quarter and the current year to perform interval-to-date reporting.

- Update volatility

  The update volatility of the aggregated financial data is zero. In our simple example, there are no transactions that can change, add or delete rows that have sales data prior to today.

- Latency requirements

  The first process that uses the aggregated data from the previous day is the batch job, which runs at 3 a.m., three hours after the last sale for the previous day could have completed.

The estimate of the aggregations shows that the daily and monthly aggregations by store, and the daily aggregations by district can be performed within the online SLA if a clustering index on STORE_ID and SALE_DT is defined. However, the quarterly and yearly calculations by store and district and the monthly calculations by district are expected to exceed acceptable online response time.

Again, assuming that the business requirements cannot be changed, the physical design must be changed to satisfy the requirements. Some amount of aggregated data needs to be added to the data model. While it is possible that the SLAs could be met by only aggregating some of the data, we go ahead designing for aggregation of the daily, monthly, quarterly, and yearly totals by both store and district. As before, we have two options:

- Add a new table or tables to store the aggregated data
- Add the aggregated data to an existing table or tables

The aggregated store totals needs to be keyed by STORE_ID and some interval component, and the district totals are keyed by DIST_ID and a time component. Since there are no existing tables with either key structure, new tables are needed. Figure 1-9 shows the design of the new tables.

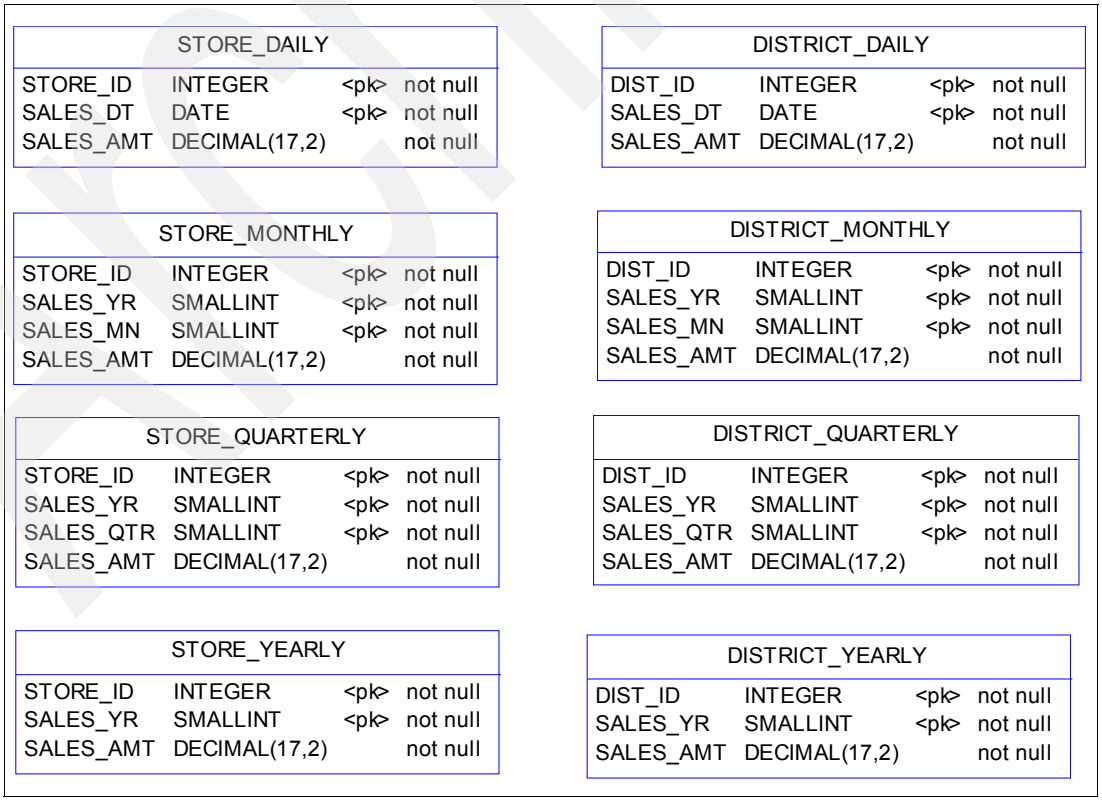


Figure 1-9 New tables to store aggregated data

Maintenance of the aggregation tables requires the following:

- On a daily basis, all sales for the day just ended, are summarized by store and district.
- A new row is added to the STORE_DAILY and DISTRICT_DAILY tables containing the day's sales totals.
- If the day being processed is the first day of a new month, a new row is added to the STORE_MONTHY and DISTRICT_MONTHLY tables. Otherwise, the day's sales totals are added to the SALES_AMT column for the current month in each of those tables.
- If the day being processed is the first day of a new quarter, a new row is added to the STORE_QUARTERLY and DISTRICT_QUARTERLY tables. Otherwise, the day's sales totals are added to the SALES_AMT column for the current quarter in each of those tables.
- If the day being processed is the first day of a new year, a new row is added to the STORE_YEARLY and DISTRICT_YEARLY tables. Otherwise, the day's sales totals are added to the SALES_AMT column for the current year in each of those tables.

To see how these eight tables can be demoralizing into a single table through the use of multi-valued columns, see "Summary tables with multi-valued columns" on page 36.

Using materialized query tables

Materialized query tables (MQTs) are tables that contain information that is derived and summarized from other tables. MQTs can be used to precalculate and store the results of queries with expensive join and aggregation operations.

The advantage of an MQT over an application-defined and application-maintained table to perform aggregation is twofold:

- With MQTs, the REFRESH TABLE command can be issued to run the aggregation query and populate (or repopulate) the table.
- When query optimization is enabled on an MQT, DB2 can use automatic query rewrite to determine if a dynamic query coded to access the base table can be resolved using the MQT. If so, the query is rewritten to access the MQT instead of the base table.

  Static SQL is not eligible for automatic query rewrite. However, static SQL can be coded to access the MQT directly.

An MQT is created using a CREATE TABLE statement that specifies the fullselect that would be used to populate the table. Example 1-10 shows the DDL that you can use to create MQTs for two of the eight aggregation tables in our model.

Example 1-10 DDL to create materialized query tables

```
CREATE TABLE STORE_DAILY (STORE_ID, SALES_DT, SALES_AMT) AS
       (SELECT A.STORE_ID,
              A.SALE_DT,
              SUM(B.ITEM_QTY*B.ITEM_PRICE_AMT)
         FROM SALE A, SALE_ITEM B
        WHERE B.TICKET_NR = A.TICKET_NR
        GROUP BY A.STORE_ID, A.SALE_DT)
       DATA INITIALLY DEFERRED
       REFRESH DEFERRED
       MAINTAINED BY USER
       DISABLE QUERY OPTIMIZATION
    IN DB.TS1;

CREATE TABLE DISTRICT_QUARTERLY (DIST_ID, SALES_YR, SALES_QTR, SALES_AMT) AS
      (SELECT DIST_ID,
```

```
            YEAR(A.SALE_DT),
            QUARTER(A.SALE_DT),
            SUM(B.ITEM_QTY*B.ITEM_PRICE_AMT)
      FROM SALE A, SALE_ITEM B, STORE C
     WHERE A.STORE_ID = C.STORE_ID
       AND B.TICKET_NR = A.TICKET_NR
     GROUP BY C.DIST_ID, YEAR(A.SALE_DT), QUARTER(A.SALE_DT))
    DATA INITIALLY DEFERRED
    REFRESH DEFERRED
    MAINTAINED BY USER
    DISABLE QUERY OPTIMIZATION
  IN DB.TS2;
```

It is also possible to alter existing tables to MQTs through the use of the ADD MATERIALIZED QUERY clause on the ALTER TABLE statement. Likewise, an MQT can be altered back to a base table through the use of the DROP MATERIALIZED QUERY.

Maintained by system or maintained by user

MQTs can be defined as MAINTAINED BY SYSTEM or MAINTAINED BY USER. The only difference between these two options is that an MQT defined as MAINTAINED BY SYSTEM cannot be the target of a LOAD utility, or an INSERT, UPDATE, or DELETE statement. Both options allow the use of the REFRESH TABLE statement.

In our example, we defined our MQTs as MAINTAINED BY USER so that we have the option of maintaining the data in the aggregation tables using our applications as well as through the use of the REFRESH TABLE statement. The data that we are aggregating is never updated, only inserted. Therefore, we can use the REFRESH TABLE statement to initially populate the MQT, and then use application inserts and updates to create and modify only those rows that are affected as new data is added to the base tables.

When maintaining the data in an MQT with an application process, DB2 does not enforce that the data in the MQT is correct with respect to the fullselect that was specified in the table definition. For this reason, the application that is aggregating the data should use SQL that is identical to that in the table fullselect to ensure the integrity of the data. One way to accomplish this is to create a view over the aggregation SQL, and then select from that view in the fullselect of the MQT and in the application that maintains the data.

Tip: When maintaining the data in an MQT with your application, embed the aggregation SQL in a view that can be referenced both in the fullselect of the MQT definition and in the application process that maintains the data in the table.

The REFRESH TABLE statement

The REFRESH TABLE <table-name> statement can be used to refresh the data in an MQT. When a REFRESH TABLE statement is executed, the following actions are performed:

- All rows in the MQT are deleted.
- The fullselect in the MQT definition is executed to recalculate the data from the base tables.
- The calculated result set is inserted into the MQT.
- The DB2 catalog is updated with a refresh timestamp and the cardinality of the MQT.

All of these actions are performed in a single unit of recovery, so be aware of the potential locking and concurrency issues. If the table is defined with LOCKMAX 0 to prevent lock

escalations, the potential for a large number of locks exists, perhaps enough to exceed the NUMLKUS DSNZPARM threshold.

Recommendation: Be very careful when using the REFRESH TABLE statement.

If possible, define your MQT table spaces as segmented to achieve the benefit of the segmented table space mass delete function.

Consider altering the MQT to disable query optimization prior to execution to prevent dynamic queries from accessing the table during the refresh process.

Consider executing a LOCK TABLE statement on the MQT prior to execution of the REFRESH TABLE statement to avoid acquiring an excessive number of locks.

Once the table is initially populated, use application logic to maintain the data in the MQT rather than the REFRESH TABLE statement.

Enable query optimization

The CREATE TABLE statements in Example 1-10 on page 32 specified DISABLE QUERY OPTIMIZATION. This was done so that the optimizer would not begin rewriting dynamic SQL statements to use the MQT right away. Once the table has been populated either by the application or through the use of the REFRESH TABLE statement, the table should be altered to ENABLE QUERY OPTIMIZATION so that automatic query rewrite can occur.

Recommendation: Always create MQTs with DISABLE QUERY OPTIMIZATION initially to prevent automatic query rewrite from directing dynamic queries to the MQT before it is populated. Once the MQT has been populated, alter the table to specify ENABLE QUERY OPTIMIZATION.

1.2.3 Normalization and denormalization

An important issue is the degree to which tables should be normalized. Data normalization considerations should always be part of your database design. Typically, data should be normalized to third normal form. However, if most accesses require a join or the join limits are being stressed, the design may be over-normalized.

On the other hand, over-denormalization should be avoided, because denormalized data may:

- Cause update anomalies, if redundant data is not updated in all places in the database.
- Prevent the use of database enforced referential integrity (RI).
- In general, prevent the use of simple SQL (except when you denormalize to avoid joins, where SQL becomes simpler).

In selected situations, however, you can consider a certain degree of denormalization from third normal form, because it can significantly improve performance by reducing the number of database accesses.

There is, then, a trade-off between performance and the disadvantages of having denormalized data. In general, highly normalized tables are the best for data that has frequent updates. Denormalized tables are often better for data that is mostly queried. Accurately weighing the cost and benefit of denormalization requires complete knowledge of all application access requirements and volumes.

In this section, we investigate several types of denormalization.

Combining tables

One common denormalization technique is to combine tables to reduce the number of table accesses that is required by the application. There are several ways that tables can be combined.

Denormalize look-up table into data table

One type of table combination that you can consider is a look-up table. A *look-up table* is a type of code table that is used to replace one or more column values with a key or code that can be stored on the data table. The benefit of a look-up table is typically a reduction in space in the data table and the ability to update values in the look-up table without having to update each referencing row in the data table.

If your data table has a look-up table that nearly always is accessed in order to retrieve the values, and those values are never or rarely updated, consider denormalization instead. The foreign key to the look-up table could be replaced by the look-up column values on the data table. In this way, accesses to the look-up table could be eliminated.

It is a good idea to still implement the look-up table so that it can be used by applications that need to find all possible distinct values, such as for a drop-down window.

The downside of this denormalization is that if any of the values from the look-up table ever change, all rows in the data table containing that value must also be updated. A failure to update all of the data table rows results in a data anomaly.

Combine multiple code tables into one

A typical logical model may have numerous code tables that are used to define the valid range of values that are allowed in a column on a data table. Implementing each of these different code tables into the physical model with their own table, table space, and index may not be the most efficient approach. A very small code table with a unique index requires at least two physical data sets and three pages, since the index structure always begins with two levels. Figure 1-10 shows the definitions of three small code tables.

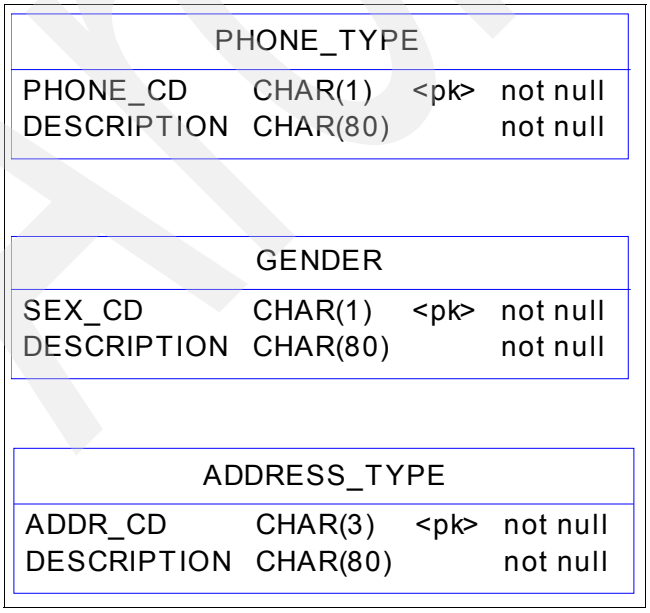


Figure 1-10 Code table

Implemented individually, these code tables would require a minimum of six data sets and nine pages. Even if all three tables were placed into a segmented table space, they would require four data sets and the same nine pages. By combining the three code tables into a single table as shown in Figure 1-11, we can store all of the code table data in two data sets and three pages.

CODE_TABLE			
CODE_TYPE_CD	CHAR(2)	<pk>	not null
TYPE_CD	CHAR(3)	<pk>	not null
DESCRIPTION	CHAR(80)		not null

Figure 1-11 Multiple logical code tables denormalized into a single table

The contents of the denormalized code table would look like Table 1-7.

Table 1-7 Contents of the denormalized code table

CODE_TYPE_CD	TYPE_CD	Description
AD	HOM	HOME ADDRESS TYPE
AD	WRK	WORK ADDRESS TYPE
AD	TMP	TEMPORARY ADDRESS TYPE
AD	VAC	VACATION ADDRESS TYPE
GN	F	FEMALE GENDER
GN	M	MALE GENDER
GN	U	UNKNOWN GENDER
PH	H	HOME PHONE TYPE
PH	W	WORK PHONE TYPE
PH	C	CELL PHONE TYPE
PH	F	FAX PHONE TYPE

This approach would prevent database enforced referential integrity from being implemented between the code table and the data table. However, this is not a disadvantage, because our recommendation is to not implement database enforced referential integrity on code tables. The reasons for this recommendation and the alternatives for enforcing data integrity are discussed in “Referential constraints” on page 130.

Summary tables with multi-valued columns

A multi-valued column is one in which the meaning of the value in the column depends upon some other characteristics in the row, usually a type code. The use of a multi-valued column can allow tables with similar or identical structures to be combined.

For example, the eight tables in Figure 1-9 on page 31 all have a very similar structure. By using a multi-valued column to store STORE_ID or the DIST_ID, and another to store the SALES_AMT, we can create a single table to store the aggregated data from all eight tables. Table 1-8 on page 37 shows what the data looks like.

Table 1-8 A table with multi-valued columns to store aggregated data

INTRVL_DT	INTRVL_CD	STOR_DIST_ID	STOR_DIST_CD	SALES_AMT
2005-01-01	D	123	S	$3,641.65
2005-01-01	D	456	S	$4,423.91
2005-01-01	D	987	D	$8,065.56
2005-01-01	M	123	S	$113,417.80
2005-01-01	M	456	S	$136,823.54
2005-01-01	M	987	D	$236,241.34
2005-01-01	Q	123	S	$326,188.87
2005-01-01	Q	456	S	$401,712.45
2005-01-01	Q	987	D	$727,901.32
2005-01-01	Y	132	S	1,357,986.42
2005-01-01	Y	456	S	$1,864,235.79
2005-01-01	Y	987	D	$3,222,222.21
2005-01-02	D	123	S	$3,394.15

In this table, there are actually three multi-valued columns controlled by two type codes:

- INTRVL_DT

  The intervaldate is a multi-valued column that is controlled by the INTRVL_CD type code. Depending on the value in INTRVL_CD, the INTRVL_DT contains one of the following:

  – When INTRVL_CD = ‘D’, INTRVL_DT contains the date of the daily sales total.

  – When INTRVL_CD = ‘M’, INTRVL_DT contains the first day of a month for a monthly sales total.

  – When INTRVL_CD = ‘Q’, INTRVL_DT contains the first day of quarter for a quarterly sales total.

  – When INTRVL_CD = ‘Y’, INTRVL_DT contains the first day of a year for a yearly sales total.

- STOR_DIST_ID

  The Store/District ID is a multi-valued column that is controlled by the STOR_DIST_CD type code. Depending on the value in STOR_DIST_CD, the STOR_DIST_ID contains either a STORE_ID or a DIST_ID.

  – When STOR_DIST_CD = ‘S’, STOR_DIST_ID contains a STORE_ID.

  – When STOR_DIST_CD = ‘D’, STOR_DIST_ID contains a DIST_ID.

- SALES_AMT

  The sales amount is a multi-valued column that is controlled by the combination of INTRVL_CD and STOR_DIST_CD. The eight possible combinations of these type codes determine what the sales amount is.

  – WHEN INTRVL_CD = ‘D’ and STOR_DIST_CD = ‘D’, SALES_AMT contains a district daily sales amount.

  – WHEN INTRVL_CD = ‘D’ and STOR_DIST_CD = ‘S’, SALES_AMT contains a store daily sales amount.

Archived

- WHEN INTRVL_CD = ‘M’ and STOR_DIST_CD = ‘D’, SALES_AMT contains a district monthly sales amount.
- WHEN INTRVL_CD = ‘M’ and STOR_DIST_CD = ‘S’, SALES_AMT contains a store monthly sales amount.
- WHEN INTRVL_CD = ‘Q’ and STOR_DIST_CD = ‘D’, SALES_AMT contains a district quarterly sales amount.
- WHEN INTRVL_CD = ‘Q’ and STOR_DIST_CD = ‘S’, SALES_AMT contains a store quarterly sales amount.
- WHEN INTRVL_CD = ‘Y’ and STOR_DIST_CD = ‘D’, SALES_AMT contains a district yearly sales amount.
- WHEN INTRVL_CD = ‘Y’ and STOR_DIST_CD = ‘S’, SALES_AMT contains a store yearly sales amount.

Multi-valued columns improve performance by reducing the number of tables a process accesses. With all of the aggregated data in a single table, we can also cluster the table in a way that puts totals that are frequently accessed together in close physical proximity.

When using multi-valued columns it is important to consider the following:

- Exercise care when evaluating column functions such as average.

  In our example above, the SALES_AMT depends upon both of the type codes. If a column function such as average is used on the SALES_AMT and only one of the type codes is specified with an equal predicate, the result of the average function is meaningless because it is calculated across different types of sales totals.

- When a foreign key value is placed in a multi-valued column, database-enforced referential integrity cannot be used.

  In our example above, the STOR_DIST_ID column contains a foreign key to either the STORE table or the DISTRICT table, depending on the value of the STOR_DIST_CD type code. However, database-enforced RI cannot be used to enforce this relationship because DB2 does not support a mutually-exclusive, either-or parent relationship.

  This implementation requires application-enforced RI, which is discussed in “RI enforced by DB2 or enforced by application” on page 132.

- Tables using multi-valued columns cannot be defined as materialized query tables, since there is no way to communicate to DB2 the relationship between the type code and the contents of the multi-valued column.

- SQL accessing the table becomes more complex.

  While accessing a single table should be simpler, the key structure using the type codes may require using more complex SQL.

Attributive table: Case study

Consider the two tables shown in Figure 1-12 on page 39. The EMP table contains most of the attributes of the employee, but, because an employee can have multiple phone numbers, an attributive table, EMP_PHONE, was created. The business requirements are to store the mandatory work phone number and up to one home phone number and one cell phone number for each employee. The PHONETYPE column is included in the primary key to uniquely identify the type of phone number stored in the row.

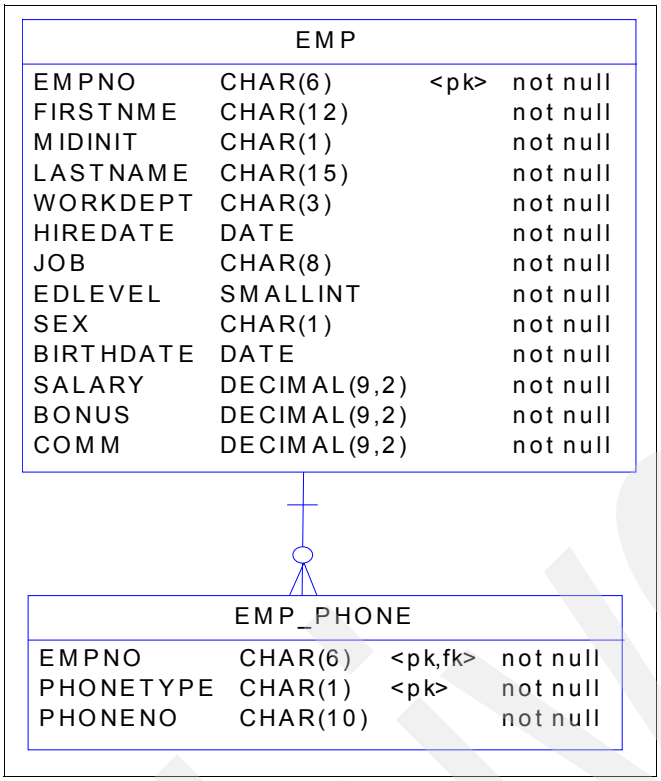


Figure 1-12 EMP table and EMP_PHONE attributive table

From the CRUD information, you can see that all of the online transactions that display employee information need to join to EMP_PHONE to get the work phone number. A much smaller number of transactions are looking for home or cell phone numbers. In order to eliminate the need to join to the EMP_PHONE to get the work phone number, we can denormalize by adding a column to EMP called WORKPHONE as shown in Figure 1-13 on page 40. The EMP_PHONE table can still be used to store the optional home and cell phone values, but the work phone is no longer stored there.

Archived

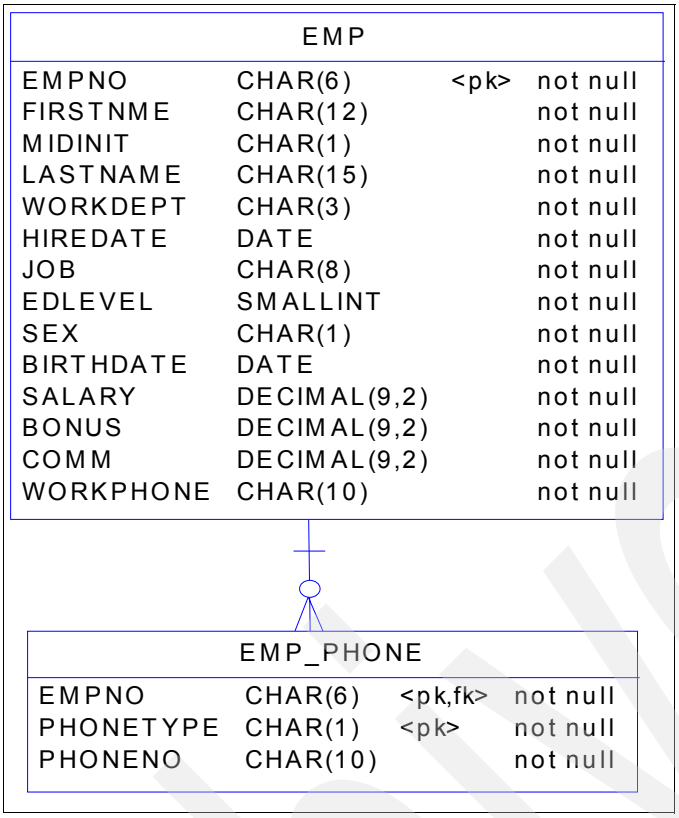


Figure 1-13 EMP table with WORKPHONE denormalized

What is the cost of this denormalization? By adding a column to the EMP table, the row length has increased, and, therefore, the table is larger. However, because the work phone is a mandatory attribute, the number of rows in the EMP_PHONE table is reduced by the number of rows in the EMP table. Often denormalization reduces the flexibility of the application. In this case, the flexibility of the design is the ability to add a new type of phone number without altering the DB2 table design. In this case, the flexibility is retained because the attributive table is still there to support the optional phone number types.

One possible cost of this denormalization is if there is a need to search by phone numbers of any type. With all types of phone numbers in a single column, it is possible to support a phone number search with a single index. By our denormalization, a phone number search of all types now requires two indexes, one on the EMP table WORKPHONE, and one on the EMP_PHONE PHONENO. On the other hand, if the business requirement is only to search on the work phone number, our denormalization may have further improved performance by allowing us to index only the WORKPHONE column.

The model can be further denormalized by placing columns for all three phone number types on the EMP table, and eliminating the EMP_PHONE table all together as shown in Figure 1-14 on page 41.

EMP			
EMPNO	CHAR(6)	<pk>	not null
FIRSTNME	CHAR(12)		not null
MIDINIT	CHAR(1)		not null
LASTNAME	CHAR(15)		not null
WORKDEPT	CHAR(3)		not null
HIREDATE	DATE		not null
JOB	CHAR(8)		not null
EDLEVEL	SMALLINT		not null
SEX	CHAR(1)		not null
BIRTHDATE	DATE		not null
SALARY	DECIMAL(9,2)		not null
BONUS	DECIMAL(9,2)		not null
COMM	DECIMAL(9,2)		not null
WORKPHONE	CHAR(10)		not null
HOMEPHONE	CHAR(10)		not null
CELLPHONE	CHAR(10)		not null

Figure 1-14 EMP table with all phone numbers denormalized

An additional cost of this denormalization is that it has removed the flexibility of the logical design. Now, adding a new type of phone number requires a change to the table. However, altering the table to add a new column to the DB2 table may be the least of your worries if the design of the online screens in the applications has to change to accept and display a new type.

Denormalization of attributive tables should only be considered when the number of occurrences is finite and small. The larger the number of occurrences and the wider the difference between the average and the maximum number of attributive values, the higher the cost of the denormalization.

Once again, this simple example demonstrates the importance of knowing all of the business requirements and table accesses before designing the tables. Without this information, the cost/benefit analysis of denormalization is purely guesswork.

Splitting tables

Sometimes there are performance benefits that can be achieved by splitting tables. Tables can be split vertically or horizontally. This section examines both of these techniques, and when and why you may choose to use them.

Vertical table split

Vertically splitting a table is accomplished by creating two or more DB2 tables, each with part of the columns of the original entity, but repeating the columns that form the primary key in all of them. Figure 1-15 on page 42 shows an example of how a table can be vertically split.

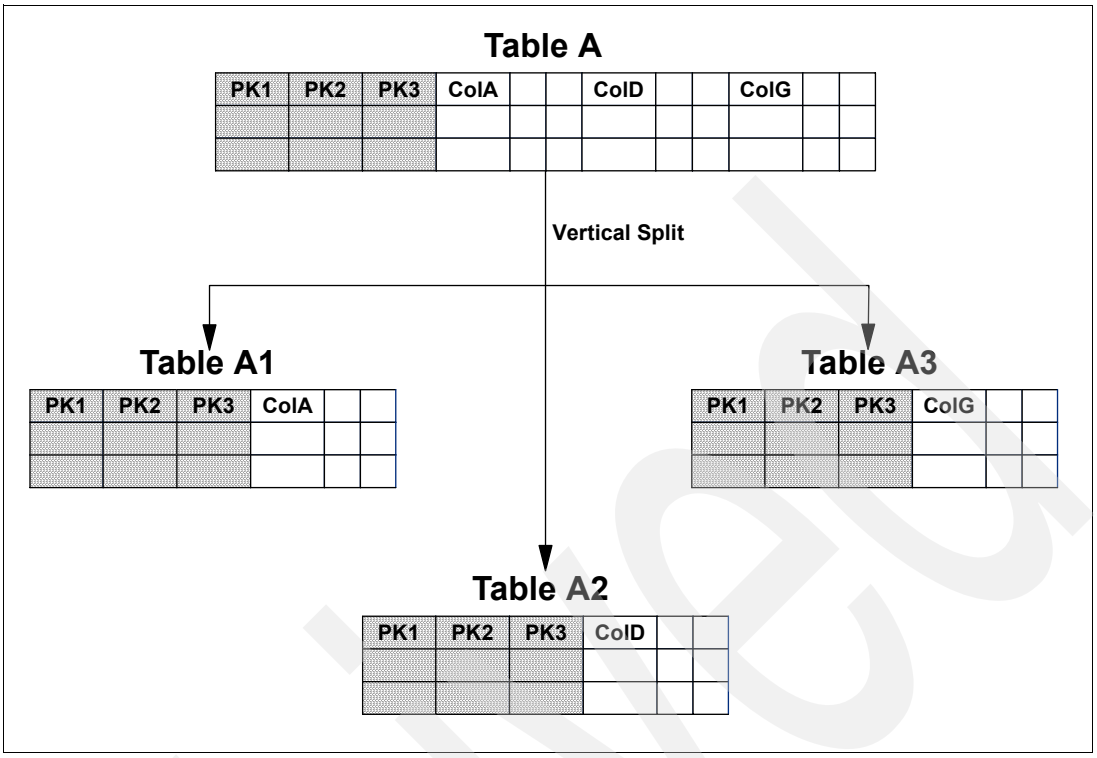


Figure 1-15 Vertical table split

Some reasons you may consider vertically splitting a table are:

- If some of the columns are optional and only populated for a small percentage of the time.

  Another alternative to using nulls to handle optional columns in a table is to vertically split the table by removing the optional columns. If a subset of the columns in the table is present or not present depending upon the same criteria, these columns might be split into a separate table where the columns are not null and the entire row is optional.

  This approach might be considered if the number or size of optional columns is large and the percentage of rows where the columns are present is small.

- If a small percentage of the columns is required for an extremely high volume process.

  If your CRUD information identifies a process that requires a certain set of columns to be retrieved in a very heavy process, splitting these columns into a separate table might provide improved performance. By placing the table with the smaller, highly accessed row in a dedicated buffer pool, you might eliminate all I/O for the high volume process. Figure 1-5 on page 13 is an example of this type of vertical table split.

Vertically splitting a table may have the following disadvantages:

- More storage may be needed because of primary key duplication.
- If a view cannot be used, programs and users see several DB2 tables for only one entity.
- If every occurrence of the primary key is not included in all the tables, outer join operations are required to access the data.
- Data maintenance, mainly for insert and delete, is complex and results in higher resource consumption.

Horizontal table split

Horizontally splitting a table means creating two or more tables with virtually the same columns and dividing the rows between them. We say virtually the same columns because in some cases the placement of rows in the tables may preclude the need for one or more columns.

Horizontally splitting a table for size or availability reasons when each split has the same attributes and indexing requirements is called *partitioning*, which is discussed in the next section.

Some reasons you may consider horizontally splitting a table are:

- If there are two or more distinct sets of rows in the table that are never or rarely accessed together.

  An example of this is a table that was logically designed to contain both current and historical data that is split into a current and historical table as was demonstrated in "Version control and keeping history: Case study" on page 20.

- If there are two or more distinct types of rows in the table that each have their own optional and mandatory attributes.

  If a certain column or set of columns is only present for rows of a specific type, separating those rows into a different table allows for the removal of those columns from all other types. Additionally, if the columns are only mandatory for a specific type, the columns can be defined as NOT NULL or NOT NULL WITH DEFAULT for the table that supports that type rather than using nulls.

- If there are two or more distinct sets of indexing requirements in the table based upon the contents of the row.

  Different indexing and clustering can be defined on each table that are specific to the access requirements of the types of rows contained within each table.

Horizontally splitting a table may have the following disadvantages:

- If a view cannot be used, programs and users see several DB2 tables for only one logical entity.
- If processes exist that need to access all rows across the set of horizontally divided tables, there is no way to present the rows in a common order without sorting.
- Data maintenance, mainly for insert and delete, may be more complex and resource intensive.
- If the criteria that was used to horizontally partition the table is updatable, an update of that attribute requires a delete from one table and an insert to the other.

1.3 Partitioning

In DB2, partitioning a table space means that it is divided up into multiple physical data sets. The placement of rows into the data sets is controlled by the values in one or more columns, called the *partitioning key*.

Archived

1.3.1 Benefits of partitioning

When implemented correctly, partitioning can achieve all of the following benefits:

- Overcome size limitations

  The most obvious reason to partition a table is to overcome the size limitations of segmented table spaces. Aside from LOB table spaces, which are a completely different subject, partitioning is the only way to implement a table larger than 64 GB.

- Improved availability

  The ability to run utilities at the partition level potentially allows the rest of the table to remain available even if destructive utilities are used.

- Improved concurrency

  Locking can be controlled at the partition level, so that a lock escalation only affects the partitions being updated.

- Shorter recovery time

  In the event of a single data set failure, just that partition can be recovered quickly. In the even of a recovery of the entire table spaces, recovery jobs can be run in parallel against multiple partitions simultaneously.

- Better throughput

  Both through the ability to run multiple utilities and applications simultaneously or through query parallelism, partitioning provides the ability to perform large data intensive tasks faster.

- Improved performance

  Each partitioned index has multiple independent index trees that typically have fewer levels than a nonpartitioned index would, reducing the number of getpages required for index access.

- Supporting application requirements

  In some cases, it is possible to partition a table in a way that supports the business requirements for data retention.

- Better space management

  Breaking a large table into more and smaller data sets can alleviate allocation problems and improve I/O performance.

  However, the use of parallel access volumes and disk striping technology can significantly improve I/O performance without partitioning.

  Compression dictionaries built on the data from one partition might be more efficient than a single compression dictionary built for the entire table space. Partitioning also allows the flexibility to compress only certain partitions.

Regardless of which of these reasons you choose to partition your table, the topics discussed in this section are intended to help you get the maximum benefit from all of them.

1.3.2 Partitioning considerations

Before implementing partitioning, you need to be aware of the following facts about partitioned table spaces.

- No *mass delete* function

  A mass delete, which is a DELETE statement with no WHERE clause, explicitly deletes and logs each row. Segmented table spaces have the ability to perform a fast mass delete

by resetting the space map pages of the table space and indexes without deleting and logging each row individually.

If you have an application requirement to perform a mass delete on a partitioned table space, you should consider using a LOAD REPLACE utility with an empty input file instead of using an SQL DELETE.

- Less efficient reuse of space

  Partitioned table spaces have less granularity in their space map and a less sophisticated space management algorithm than segmented table spaces. A segmented table space is better able to reuse the space left by deleted rows. A partitioned table space that experiences heavy delete and insert activity may require more frequent reorganization to maintain clustering order than a segmented table space.

- Rows may relocate on update

  Any update of the partitioning columns to a value that no longer fits in the range of the current partition operates like a delete and insert, not only in the table space but in the indexes as well. Not only does this result in additional overhead, but when this occurs, it is possible that the same row may be returned to a cursor more than once as it moves through the table. This behavior can be prevented by setting DSNZPARM PARTKEYU to a value of NO or SAME.

- More data sets may cause system level overhead

  The operating system has a finite limit to the number of data sets that can be open concurrently. For z/OS Version 1 Release 6 and DB2 Version 8, the verified limit is approximately 65,000. For a particular DB2 subsystem, the limit on the number of open data sets is controlled by DSNZPARM DSMAX. This DSNZPARM value can now be set up to 65,000 with APAR PQ96189.

  As the DB2 subsystem approaches the maximum number of open data sets, it must begin closing data sets. A high amount of data set open and close activity can degrade DB2 system performance.

1.3.3 Identifying and evaluating candidate partitioning key

Choosing a partitioning key is much like choosing a primary key for a table. There are typically multiple columns and groups of columns that are potential partitioning keys. With DB2 Version 8, even more columns are potential candidates since the partitioning key no longer must be indexed. Each of these candidate partitioning keys should be evaluated using the following criteria to determine if it achieves the desired results:

- Will the column value be known by the applications when accessing the table?

  It is not necessary to know the value of the partitioning column when accessing the table using a nonpartitioned index, but it is critical to optimum performance when accessing the table using a partitioned index. See “Indexing partitioned tables” on page 98 for a detailed discussion of indexing partitioned tables.

- Does the column value change?

  The ability to change the value in a partitioning key is controlled by the PARTKEYU DSNZPARM. Table 1-9 on page 46 contains the possible values of the PARTKEYU DSNZPARM and how these values affect the ability to update partitioning keys.

Table 1-9 PARTKEYU values and partitioning key update behavior

PARTKEYU value	Partitioning key update behavior
NO	No updates to any partitioning keys are allowed.
SAME	Updates to partitioning keys are allowed, but only if the updated partitioning key falls within the range of the current partition.
YES	All updates to partitioning keys are allowed.

In Version 8, DB2 no longer has to drain the partitions in order to move a row from one partition to another. The behavior of the update of a partitioning key has been changed to act like a delete and insert. Because of this, if the partitioning key of a row is updated while another application process is reading through the table using a cursor, it is possible that the updated row appears in the cursor twice, once with the old partitioning key value and once with the new partitioning key value. The delete and update process also causes an S lock to be taken and held until commit on the parent row or page, if this table is a dependent in a database-enforced RI relationship.

Tip: If the PARTKEYU DSNZPARM was set to NO or SAME to prevent the drain that occurred prior to Version 8, you can now safely change this DSNZPARM to YES.

If your application needs the ability to update the partitioning key, you need to be sure that this DSNZPARM does not prevent your transaction from succeeding. You can talk to your system DBA to ask if the DSNZPARM can be changed, but, keep in mind that DSNZPARMs control behavior subsystem-wide, and you may not have success in getting them changed in a timely manner.

If your DSNZPARM settings prevent you from updating the partitioning key and you are considering an updatable column for partitioning, it is not automatically excluded from consideration. You may still consider partitioning on the column and coding the application to delete and insert the row in the event of an update. However, if this table is a referential parent, this would require you to also delete and reinsert all of the children as well.

- What is the cardinality or number of possible unique values of this column?

  If the number of distinct values in the column is fewer than the desired number of partitions, it is necessary to add another column to the partition key to achieve the desired result.

- Are the number of rows with each value in the partition key fairly consistent, or do some values in the partition key appear much more frequently?

  If the number of values for each partition key are significantly different, it becomes difficult to keep the partition sizes uniform. There is no requirement that partition sizes must be equal, but the benefits of parallelism are reduced if utilities and SQL, that run on the largest partitions, take significantly longer than those run on the smaller partitions.

  In addition, having one partition that grows unexpectedly large may result in that partition reaching the maximum size. DB2 V8 allows many more online changes to partitions, but it is safer to consider the growth of all partitions when choosing a DSSIZE.

- What is the ratio of the cardinality to the domain of the column?

  In other words, are all possible values of the column represented in the table at the time of the initial load, or are new values of the column expected to be added over time?

  A column for which all values are represented should provide a more stable partitioning scheme.

Partitioning on a column for which new values continue to be added, such as a date column, means that the table probably requires periodic maintenance to add, resize, or rotate partitions. This does not disqualify a column from partitioning, since DB2 provides the ability to add, drop, and resize partitions while the table is available, but it should be considered if there are other partitioning alternatives.

- Will partitioning on the column spread workload across the entire table space?

  Depending upon your particular partitioning goals, you may want to spread new rows in the table across the entire range of partitions evenly. Not only does this allow your partitions to grow evenly, but it may provide greater insert throughput than you can achieve if all new rows fall into a small number of partitions.

- Do the column's values provide some logical grouping of application work?

  Partitioning on a column whose values can be easily used to divide large batch tasks supports application parallelism where multiple processes can run against different partitions simultaneously.

  In many cases, the partitioning scheme can be used to solve difficult application design problems. For example, if the application must add and delete a massive number of rows on a frequent basis, would a specific partitioning scheme be able to support this requirement through the use of rolling partitions?

  The partitioning scheme may also be used to support multiple batch jobs accessing separate partitions.

An ideal partitioning scenario: Multi-client table

An example of a table that would be ideal for partitioning would be a single table that contains rows for multiple clients. By partitioning on the CLIENT_ID, which is not updatable, each client's data is separated into different partitions. This provides the following benefits:

- Each client's activity is limited to their own partition.

  Because the CLIENT_ID is specified on all queries, partition pruning would result in only the specific partitions containing that CLIENT_ID being accessed, regardless of the access path or the type of index used.

- Workload is spread across all partitions of the table.
- No lock contention occurs between different client's transactions.

  Because each client's transactions only accesses their partition, there is never be a lock contention between a transaction from CLIENT A and a transaction from CLIENT B.

- Partition level utilities can run according to the scheduling needs of each client.

If you have tables that contain such naturally occurring partitioning keys, consider yourself lucky. Otherwise, you can consider one of the techniques discussed in the next section to create a partitioning key.

What if partitioning values are good enough?

If none of your candidate partitioning keys or columns appear to satisfy your partitioning requirements, consider creating an artificial partitioning key. While it would be possible to just randomly assign a number between 1 and your maximum number of partitions, a better choice is to use a partitioning key that can be derived from some other value that is typically known when accessing the table.

For example, if your table contains account information, and it is typically accessed using an equal predicate on the ACCT_NR column, consider using the last three digits of the ACCT_NR column as a partition key. With 1,000 different values, you can easily support

partitioning schemes of 1,000, 500, 250, 200, 100, 50, 25, 20, 10, 5, or 2 equal partitions, assuming that the account number assignment process is fundamentally sequential.

This approach requires the addition of a new column, since DB2 cannot partition on part of a column. In this case, a SMALLINT column called PART_ID t works nicely. All applications that access the table with ACCT_NR = :HV-ACCT-NR predicates will need to derive the PART_ID value from the account number and add a predicate on PART_ID = :HV-PART-ID so that only the partition containing the requested account number is accessed.

Managing partitions using a hash table: Case study

Another approach to an artificial partitioning key is to use a hash table. This is a separate table that is used to translate a key into a partition number.

Suppose you need to partition the TRAN table shown in Figure 1-16. This table is a repository for daily transaction data that must be retained for at least one year. You would like to define 365 partitions and use the TRAN_DT as the partitioning key. The ROTATE PARTITION option can then be used to rotate the oldest partition to hold the new day's data and a Load utility can be run on the new partition while the rest of the table is available. However, there is a catch. Individual transactions can be flagged for longer retention through the use of the OVERRIDE_DT column. This column is null when the row is inserted, but if it is updated to a not null value, the TRAN row must be retained at least until that date. Typically, less than 1% of the rows are flagged for retention, but this business requirement must be supported.

TRAN			
ACCT_NR	DECIMAL(21,0)	<pk>	not null
TRAN_NR	DECIMAL(21,0)	<pk>	not null
TRAN_DT	DATE		not null
TRAN_TYPE_CD	CHAR(1)		not null
TRAN_AMT	DECIMAL(15,2)		not null
OVERRIDE_DT	DATE		null

Figure 1-16 TRAN table layout

One possible solution is to check the oldest partition to see if there are any rows with OVERRIDE_DT > CURRENT DATE. If so, instead of rotating the partition, you can just add a new partition. The result of this approach is a steady increase in the number of partitions and the retention of more data than necessary.

Deleting just the non-retained rows out of the old partitions requires an expensive application process, and would only result in space savings if the partitions were reorganized. The number of partitions would still increase, since no partitions could be rotated until the oldest partition had no transactions that need to be retained, and the oldest partitions would be very small relative to the full partitions being loaded every day. Partitioning by TRAN_DT does not seem to work.

What if instead of partitioning by the TRAN_DT column, we add a new column called PART_NR that contains a value from 1 to 365. A separate hash table can then be used to assign a PART_NR value for each TRAN_DT. One TRAN_DT would only map to one partition, but a PART_NR may contain transactions for more than one TRAN_DT. Figure 1-17 on page 49 shows the new design with the hash table.

HASH			
TRAN_DT	DATE	<pk>	not null
PART_NR	SMALLINT		not null

TRAN			
PART_NR	SMALLINT	<pk>	not null
ACCT_NR	DECIMAL(21,0)	<pk>	not null
TRAN_NR	DECIMAL(21,0)	<pk>	not null
TRAN_DT	DATE		not null
TRAN_TYPE_CD	CHAR(1)		not null
TRAN_AMT	DECIMAL(15,2)		not null
OVERRIDE_DT	DATE		null

Figure 1-17 Redesign of TRAN table with hash table to assign PART_NR

This design requires a process to periodically insert new rows into the HASH table to assign partition numbers in advance. Example 1-11 shows simple pseudocode to perform this process monthly.

Example 1-11 Populating the HASH table

```
SET MAX-PARTS TO 365

--GET THE MAXIMUM DATE AND THE PARTITION NUMBER IT IS USING FROM THE HASH TABLE

SELECT TRAN_DT + 1 DAY,
       PART_NR + 1
  INTO :HV-TRAN-DT,
       :HV-PART-NR
  FROM HASH
 ORDER BY TRAN_DT DESC
 FETCH FIRST 1 ROW ONLY

-- IF THE TABLE IS EMPTY, START WITH THE CURRENT DATE AND PARTITION 1

IF SQLCODE = +100
   SET HV-TRAN-DT TO CURRENT DATE
   SET HV-PART-NR TO 1
END-IF

PERFORM 31 TIMES

   IF HV-PART-NR > MAX-PARTS
      SET HV-PART-NR TO 1
   END-IF

   INSERT
     INTO HASH
   VALUES (:HV-TRAN-DT,
           :HV-PART-NR)

   SET HV-TRAN-DT = DATE(HV-TRAN-DT) + 1 DAY
   SET HV-PART-NR = HV-PART-NR + 1

END-PERFORM
```

Example 1-12 on page 50 contains the pseudocode of the daily process to load the next partition of the TRAN table.

Archived

Example 1-12 Daily process to populate the next TRAN table partition

```
SET HV-TRAN-DT TO THE TRANSACTION DATE OF THE ROWS TO BE LOADED

SELECT PART_NR
  INTO :HV-PART-NR
  FROM HASH
 WHERE TRAN_DT = :HV-TRAN-DT

UNLOAD ALL ROWS FROM TRAN TABLE
 WHERE PART_NR = :HV-PART-NR
   AND OVERRIDE_DT > CURRENT DATE

IF NO ROWS ARE UNLOADED
   DELETE FROM HASH
    WHERE PART_NR = :HV-PART-NR
      AND TRAN_DT <> :HV-TRAN-DT
END-IF

MERGE UNLOADED ROWS WITH TODAY'S LOAD FILE

LOAD REPLACE PARTITION NUMBER HV-PART-NR WITH MERGED FILE
```

If the retention requirements change in the future, you can merely alter the table space to add new partitions and change the MAX-PARTS parameter used by the process in Example 1-11 on page 49 to the new number of partitions. The process automatically begins populating the new partitions.

1.3.4 Determining the number of partitions

A basic guideline for partitioning is to use as few partitions as possible while still achieving the maintainability, availability, and performance goals. Having said this, your options may be limited by your choice of partitioning key.

If each value in the domain of the columns used to partition the table provides a natural and reasonable number of balanced partitions, your decision is an easy one. If the cardinality of the column is too high, or the anticipated number of rows for each value varies widely, then you should consider combining ranges of values into partitions that are roughly the same size. It is not necessary that the partition sizes are equal, but to get the maximum benefit from partitioning and parallel operations in particular, a good rule of thumb is that the largest partition should be no more than twice the size of the smallest partition.

The DSSIZE parameter

The DSSIZE parameter of a partitioned table space allows you to specify the maximum size of each partition. The DSSIZE parameter is specified in gigabytes, with valid values ranging from 1 GB to 64 GB.

When creating a partitioned table space, estimate the size of each partition, and make sure that the table space is defined with a DSSIZE large enough to accommodate the largest partition. Remember that a larger DSSIZE reduces the number of allowed partitions, and it also affects the size of the RID. Table spaces with a DSSIZE smaller than 4 GB use a 4 byte RID, and table spaces, with a DSSIZE of 4 GB or greater, require a 5 byte RID. Keep this in mind when calculating the size of your indexes.

In order to specify a DSSIZE value greater than 4 GB, the following conditions must be true:

- DB2 is running with DFSMS Version 1 Release 5.
- The data sets for the table space are associated with a DFSMS data class that has been specified with extended format and extended addressability.

The default value for DSSIZE varies depending upon the number of partitions and page size of the table space. Table 1-10 provides the default DSSIZE for each possible combination.

Table 1-10 Default DSSIZE values

NUMPARTS	Page size	Default DSSIZE
1 to 16	4 KB, 8 KB, 16 KB, 32 KB	4 GB
17 to 32	4 KB, 8 KB, 16 KB, 32 KB	2 GB
33 to 64	4 KB, 8 KB, 16 KB, 32 KB	1 GB
65 to 254	4 KB, 8 KB, 16 KB, 32 KB	4 GB
254 to 4096	4 KB	4 GB
254 to 4096	8 KB	8 GB
254 to 4096	16 KB	16 GB
254 to 4096	32 KB	32 GB

1.4 Using DB2 compression

Any application performance benefit from DB2 compression comes in the form of reduced I/O. In order to achieve reduced I/O with compression, you must reduce the disk occupancy of the table space. No matter what else happens, you always pay an increased CPU cost when accessing compressed rows using SQL. If DB2 compression does not reduce the size of your table space because the data does not compress well or because you have added free space to avoid relocated rows, you have achieved no savings. To the contrary, you have actually increased your CPU consumption and probably your elapsed time as well.

As is the case with variable length columns, a reduced row length increases the number of rows that fits on a page, which in turn reduces getpages. Fewer getpages translate into fewer locks and less CPU. However, the CPU overhead of compressing and uncompressing the row more than offsets this CPU savings. By increasing the number of rows per page, compression can improve buffer pool density and efficiency, and this savings translates back into reduced I/O.

1.4.1 When should you use DB2 compression?

The use of DB2 compression does not guarantee that the size of your table space is reduced. To begin with, DB2 compression is performed using a static compression dictionary that is stored at the beginning of each partition of a table space. The size of this dictionary varies depending upon the number of entries, but occupies at least one segment of a segmented table space and up to 16 pages per partition for a partitioned table space.

Additionally, not every row in a compressed table space is always compressed. If the length of the row is not reduced by the compression algorithm, it is stored uncompressed.

Before compressing a table space, the DSN1COMP utility can be used to estimate the amount of savings that would be achieved using DB2 compression. Example 1-13 on page 52

shows the output from the DSN1COMP utility. The *DB2 UDB for z/OS Version 8 Utility Guide and Reference,* SC18-7427-02, describes the use of the DSN1COMP utility.

Example 1-13 Sample output from the DSN1COMP utility

```
DSN1COMP INPUT PARAMETERS
  4,096  DICTIONARY SIZE USED
     15  FREEPAGE VALUE USED
     20  PCTFREE VALUE USED
 50,000  ROWLIMIT REQUESTED
         ESTIMATE BASED ON DB2 REORG METHOD
    255  MAXROWS VALUE USED

DSN1COMP COMPRESSION REPORT
  5,929  KB WITHOUT COMPRESSION
  3,305  KB WITH COMPRESSION
     44  PERCENT OF THE BYTES WOULD BE SAVED

  1,326  ROWS SCANNED TO BUILD DICTIONARY
 50,000  ROWS SCANNED TO PROVIDE COMPRESSION ESTIMATE
  4,096  DICTIONARY ENTRIES

    124  BYTES FOR AVERAGE UNCOMPRESSED ROW LENGTH
     70  BYTES FOR AVERAGE COMPRESSED ROW LENGTH

     16  DICTIONARY PAGES REQUIRED
  2,052  PAGES REQUIRED WITHOUT COMPRESSION
  1,175  PAGES REQUIRED WITH COMPRESSION
     42  PERCENT OF THE DB2 DATA PAGES WOULD BE SAVED
```

Assuming that you have run the DSN1COMP utility and it showed that you would save disk space by using DB2 compression, the estimated savings must be weighed against the costs of compression.

The cost of updates on compressed rows

An UPDATE statement is the most expensive SQL statement when working with compressed rows. When an update is performed, the row must be read, uncompressed, updated, and then recompressed. In addition, because all compressed rows become variable in length, an update introduces the possibility of row relocation.

As shown in Figure 1-1 on page 8 for variable length columns, whenever a variable length row is updated, the new row length could be longer than the original row length, and the row may no longer fit on the page. When this happens, DB2 relocates the row on another page and places a pointer to it on the original page.

A particularly costly example is a table in which rows transition through multiple states, beginning with a “default” state. In this type of table, the business process is to create a new row with a unique key, but with most other columns set to default values. This row is subsequently updated by other processes that replace the default values in the columns. When this is the case, it is very likely that the “default” row compresses very well, and as the columns are updated with more distinct values, the compressed row length increases, causing the row to relocate.

When the CRUD information shows that the table is updated heavily with a high risk of relocated rows, it is probably best to not use DB2 compression.

In all other normal cases, the general recommendation is to use DB2 compression if the estimated percentage of pages saved is over 20%.

1.4.2 When does the compression dictionary get built?

After the table space has been altered to COMPRESS YES, there are two ways to build a compression dictionary and activate data compression in a DB2 table space:

- A LOAD utility with the REPLACE option specified
- A REORG utility without the KEEPDICTIONARY option specified

Subsequently inserted, loaded, or updated rows are automatically compressed using the compression dictionary.

The REORG utility is the recommended method of building the compression dictionary, because it creates a better dictionary and it compresses all of the rows.

The LOAD utility with the REPLACE option uses the first *n* rows to build the compression dictionary. These rows are not compressed. The rest of the data is compressed using the dictionary that was built from these first *n* rows. The value of *n* is specified in the output of the LOAD utility. Example 1-14 shows the output from a LOAD utility that built a compression dictionary.

Example 1-14 Sample output from the LOAD utility showing compression statistics

```
DSNU241I : DSNURBDC - DICTIONARY WITH 4096 ENTRIES HAS BEEN
                      SUCCESSFULLY BUILT FROM 666 ROWS FOR TABLE SPACE
                      DB2COMP.TERA39C
DSNU244I : DSNURWT  - COMPRESSION REPORT FOR TABLE SPACE
                      DB2COMP.TERA39C

        1338 KB WITHOUT COMPRESSION
        725 KB WITH COMPRESSION
          45 PERCENT OF THE BYTES SAVED FROM COMPRESSED DATA ROWS

          93 PERCENT OF THE LOADED ROWS WERE COMPRESSED

         140 BYTES FOR AVERAGE UNCOMPRESSED ROW LENGTH
          77 BYTES FOR AVERAGE COMPRESSED ROW LENGTH

         436 PAGES REQUIRED WITHOUT COMPRESSION
         254 PAGES REQUIRED WITH COMPRESSION
          41 PERCENT OF THE DB2 DATA PAGES SAVED USING COMPRESSED DATA
```

The REORG utility builds the compression dictionary while the data is being unloaded. It also builds the initial dictionary using the first *n* rows, but this dictionary is then refined as repeated patterns are found in subsequent rows. Then, all of the rows are compressed using this compression dictionary during the reload phase. Example 1-15 shows the output from a REORG utility that built a compression dictionary. Notice that for the same table space we obtain a better compression value than with the LOAD.

Example 1-15 Sample output from the REORG utility showing compression statistics

```
DSNU050I   DSNUGUTC - REORG TABLESPACE DB2COMP.TERA39C LOG NO
DSNU251I   DSNURULD - UNLOAD PHASE STATISTICS - NUMBER OF RECORDS=9965
                      FOR DB2COMP.TERA39C
DSNU250I   DSNURULD - UNLOAD PHASE COMPLETE, ELAPSED TIME=00:00:05
DSNU241I : DSNURBDC - DICTIONARY WITH 4096 ENTRIES HAS BEEN SUCCESSFULLY
                      BUILT FROM 1337 ROWS FOR TABLE SPACE
                      DB2COMP.TERA39C
DSNU244I : DSNURWT  - COMPRESSION REPORT FOR TABLE SPACE DB2COMP.TERA39C

        1338 KB WITHOUT COMPRESSION
```

Archived

```
643 KB WITH COMPRESSION
 51 PERCENT OF THE BYTES SAVED FROM COMPRESSED DATA ROWS

100 PERCENT OF THE LOADED ROWS WERE COMPRESSED

140 BYTES FOR AVERAGE UNCOMPRESSED ROW LENGTH
 69 BYTES FOR AVERAGE COMPRESSED ROW LENGTH

436 PAGES REQUIRED WITHOUT COMPRESSION
226 PAGES REQUIRED WITH COMPRESSION
 48 PERCENT OF THE DB2 DATA PAGES SAVED USING COMPRESSED DATA
```

A load of an empty table space with the RESUME YES option specified does not build a compression dictionary, and only compresses the loaded data if a compression dictionary already exists.

1.4.3 When should the compression dictionary be rebuilt?

Once a compression dictionary has been built, the KEEPDICTIONARY option of the REORG should be used to prevent rebuilding the dictionary repeatedly.

The initial percentage of pages saved is reported in the output of the LOAD or REORG utility that built the compression dictionary. This does not update the value of the PAGESAVE column on the SYSIBM.SYSTABLEPART catalog table. The catalog value is only updated by the RUNSTATS utility.

Over time, the PAGESAVE value may change as rows are inserted, updated, and deleted. Rows with new patterns of repeating characters may be inserted into the table, and rows whose repeating patterns were used to build the original dictionary may have been updated to different values or deleted from the table.

As long as the PAGESAVE value stays fairly consistent or increases, there is no need to rebuild the dictionary. If the PAGESAVE value has decreased significantly from the original build of the dictionary, you should consider removing the KEEPDICTIONARY option from the REORG to build a new dictionary. Once a table space is compressed, there is no way to estimate the amount of compression that might be achieved with a new dictionary because the DSN1COMP utility only works for uncompressed data.

Another indicator of compression efficiency is the PCTROWCOMP column on the SYSTABLES catalog table, which identifies the percentage of rows in the table that is compressed. Like PAGESAVE, this value is only updated by the RUNSTATS utility.

There are two possible reasons for a decrease of the value of PCTROWCOMP:

- For INSERT or LOAD of new rows into the table, the length of the row is not being reduced by the compression algorithm.
- For UPDATE of existing rows in the table, the rows assume values that are not being found in the compression dictionary.

This is another indication that the compression dictionary should be rebuilt.

If the PAGESAVE value for a table space drops too low, or any of the other conditions that previously favored compression change, you may have to reconsider the decision to compress.

Like PAGESAVE, this value is only updated by the RUNSTATS utility. A decrease in this value indicates that rows are being inserted or loaded into the table whose length is not being

reduced by the compression algorithm. This may also be an indicator that the compression dictionary should be rebuilt.

> **Important:** Be careful not to rebuild the compression dictionary when data propagation is in progress because the propagation tool may be using the compression dictionary to uncompress log data. Allow the propagation to “catch up” before reorganizing the table space.

For more information on the performance and use of DB2 compression, refer to the redbook, *DB2 for OS/390 and Data Compression,* SG24-5261, and *DB2 for z/OS and OS/390 Version 7 Performance Topics,* SG24-612.

1.5 Choosing a page size

The page size of a table is determined by the page size of the buffer pool in which its table space is defined. In most cases, you should use a page size of 4 KB. However, there are instances when a larger page size should be considered.

1.5.1 Maximum record size

In order to determine the optimum page size, you must first calculate the maximum record size of your table. Using the values from Table 1-2 on page 5, the maximum record size of a table can be calculated using the formula shown in Example 1-16:

Example 1-16 Calculating the maximum record size for a table

```
Max table record size = SUM(byte count of each column) + number of nullable columns + 8
```

Table 1-11 shows the maximum record size that is accommodated by each different page size. Note that the maximum record size is reduced by 10 bytes if an EDITPROC is used.

Table 1-11 Maximum record size supported by each page size, in bytes

EDITPROC	Page Size = 4 KB	Page Size = 8 KB	Page Size = 16 KB	Page Size = 32 KB
NO	4056	8138	16330	32714
YES	4046	8128	16320	32704

If the maximum table record size is greater than the maximum record size allowed for a specific page size, then a larger page size must be used.

1.5.2 Rows per page

After determining the minimum page size that can be used to support the maximum record size of your table, you should calculate the number of rows that fits on each page using the average record size from your table.

The average record size for a table can be estimated as follows:

- For a fixed length, uncompressed row, the average record size is equal to the maximum record size calculated above.

- For a row that is variable length because of compression, use the average compressed row length from the DSN1COMP output, if it is available, or estimate the average by multiplying the maximum record size by the estimated compression percentage.
- For a row that is variable length because it contains variable length columns, calculate the average record length using the formula in Example 1-16 on page 55, but using the estimated average length for each variable column instead of the maximum length.

Once the average record size of the table has been calculated, you can estimate the number of rows that fits on each page as follows:

```
Rows per page = FLOOR(max. record size supported by page / avg. record size of table)
```

The largest number of rows per page supported in DB2 is 255. If your calculation yields a higher number, replace it with 255.

Assuming that your estimates are accurate, you can now estimate the size of your table in KB as follows:

```
Table size (KB) = CEILING(number of rows / rows per page) * page size
```

1.5.3 Using page size to optimize storage

In some cases, using a larger page size than the minimum can reduce the total size of a table. This is due to the fact that a row cannot be split across two pages. Therefore, if all rows in a page are equal to the average record size calculated above, the amount of unused space in the page can be calculated using the following formula:

```
Unused space = max. record size supported by page - (rows per page * avg. record size)
```

Using the results of all of these calculations, we can now apply the following rules to determine the optimum page size for storage:

- When the amount of unused space per page is over half of the average record length of the table, consider using the next larger page size since it reduces the total table size.
- When the average record size of the table is more than half of the maximum record size supported by the page, consider the next larger page size and recalculate.

The following examples demonstrate this technique.

In Example 1-17, table T1 has an average record length of 230 bytes and contains 70 million rows. By changing the page size from 4 KB to 8 KB, the table size is reduced by 470,588K, or approximately 2.85%. Before implementing a page size of 8 KB for this table, consider the accuracy of your estimate. If T1 has a fixed record length of 230 bytes, then this savings estimate is accurate. However, if T1 has a variable record length, it is very likely that your margin of error when estimating the average record length is greater than the estimated 2.85% savings. In either case, be sure to read “Other page size considerations” on page 58 before making a final decision.

Example 1-17 Optimum page size for T1

```
T1 average record length: 230 bytes
T1 estimated rows: 70,000,000

Using minimum page size (4 KB):
   Rows per page         = FLOOR(4056 / 230) = 17
   Table size            = CEILING(70,000,000 / 17) * 4 KB = 16,470,588K
   Unused space per page = 4056 - (17 * 230) = 146

Unused space per page > average record length/2        ? YES
```

```
Average record length > maximum record size for page/2 ? NO

Recalculating using next page size (8 KB):
   Rows per page         = FLOOR(8138 / 230) = 35
   Table size            = CEILING(70,000,000 / 35) * 8 KB = 16,000,000K
   Unused space per page = 8138 - (35 * 230) = 88

Unused space per page > average record length/2       ? NO
Average record length > maximum record size for page/2? NO

Optimum page size is 8 KB
```

In Example 1-18, table T2 has an average record length of 3,000 bytes and contains 8 million rows. By changing the page size from 4 KB to 16 KB, the table size is reduced by 6,400,000K, or 20%.

Example 1-18 Optimum page size for T2

```
T1 average record length: 3000 bytes
T1 estimated rows: 8,000,000

Using minimum page size (4 KB):
   Rows per page         = FLOOR(4056 / 3000) = 1
   Table size            = CEILING(8,000,000 / 1) * 4 KB = 32,000,000K
   Unused space per page = 4056 - (1 * 3000) = 1056

Unused space per page > average record length/2        ? NO
Average record length > maximum record size for page/2 ? YES

Recalculating using next page size (8 KB):
   Rows per page         = FLOOR(8138 / 3000) = 2
   Table size            = CEILING(8,000,000 / 2) * 8 KB = 32,000,000K
   Unused space per page = 8138 - (2 * 3000) = 2138

Unused space per page > average record length/2        ? YES
Average record length > maximum record size for page/2 ? NO

Recalculating using next page size (16 KB):
   Rows per page         = FLOOR(16330 / 3000) = 5
   Table size            = CEILING(8,000,000 / 5) * 16 KB = 25,600,000K
   Unused space per page = 16330 - (5 * 3000) = 1330

Unused space per page > average record length/2        ? NO
Average record length > maximum record size for page/2 ? NO

Optimum page size is 16 KB
```

The sum of the byte counts of the columns must not exceed the maximum row size of the table. The maximum row size is eight fewer than the maximum record size. For columns that do not allow null values, Figure 1-2 on page 5 gives the byte counts of columns by data type. For columns that allow null values, the byte count is one more than shown in the table.

The page size of a table space is determined by the page size of the buffer pool in which the table space is defined.

The maximum record size of a table depends on the page size of the table space, and whether or not the EDITPROC clause is specified, as shown in Table 1-11 on page 55. The page size of the table space is the size of its buffer, which is determined by the BUFFERPOOL clause that was explicitly or implicitly specified when the table space was created.

1.5.4 Other page size considerations

There are other consequences when using page sizes larger than 4 KB, so consider the following before using a page size other than the minimum required to support your table's maximum record length, particularly if your table is accessed in a random fashion.

Page size and I/O cost

Before DB2 V8, even with page sizes larger than 4 KB, the data in table spaces was still stored in 4 KB control intervals ((CI). Therefore, an 8 KB page size meant two 4 KB CIs, and a 32 KB page size meant eight 4 KB CIs. An I/O using a larger page size consumes more resources than an I/O of a smaller page size. With DB2 V8, DB2 uses the option (via DSVCI DSNZPARM) to have, by default, a VSAM CI size that matches the page size. This eliminates some integrity exposures and introduces performance benefits for sequential processing.

Page size and buffer pool efficiency

Random access using larger page sizes also affects the efficiency of the buffer pool. A random access to a row on a 16 KB page requires that four 4 KB pages or one 16 KB page be read into the buffer pool. In theory, the buffer pool supporting the 16 KB page may require more pages to achieve the same hit ratio for random access than you would get with a 4 KB page. But, at the same time, the 16 KB page could contain more rows than four 4 KB pages, so the overall density in the buffer pool should be higher with a larger page size, unless the row is very large. "Your actual mileage may vary."

The cost of locking

The use of a larger page size typically reduces the locking costs of page level locking due to the lower number of pages in the table, unless all accesses to the table are completely random. This also translates into lower data sharing overhead, because of reduced accesses, particularly when using sequential processing.

The other impact of a larger page size on locking is the granularity of the page lock. With a larger page size, a single lock locks more rows than with a smaller page size. This may have an impact on concurrency, and increases the possibility of a hot page. See "Hot data page problem" on page 354 for more information.

Do not forget about MAXROWS

The maximum number of rows on a table space page is controlled by the MAXROWS parameter. The maximum (and default) value of this parameter is 255. If your calculated rows per page is greater than the MAXROWS parameter on your table space, replace your rows per page value with the MAXROWS value. This is particularly important if the MAXROWS parameter is used to reduce the number of rows per page to improve concurrency. For more information on the use of MAXROWS and concurrency, see "The MAXROWS parameter" on page 335.

2

Index design

In this chapter, we review the fundamental concepts of index design and examine the associated performance implications. We discuss how to choose the appropriate data type for each attribute, including the use of VARCHAR, large objects, and storing XML data.

This chapter contains the following topics:

- What is an index?
- Why build indexes?
- Cost of an index
- How does DB2 use the index?
- Will DB2 use the index?
- Which indexes to build
- Clustering index
- Indexing partitioned tables

© Copyright IBM Corp. 2005. All rights reserved.

2.1 What is an index?

An index is an ordered set of pointers to rows of a base table or an auxiliary table contained in 4 KB pages. Conceptually, you can think of an index to the rows of a DB2 table like you think of an index to the pages of a book. Each index is based on the values of data in one or more columns of a table.

An index is an object that is separate from the data in the table. You define an index using the CREATE INDEX statement. DB2 builds the index B-tree (balanced tree) structure and maintains it automatically. See Figure 2-1.

The number of index levels increases as the number of index entries increases, but an index always has at least two index levels.

Pages of indexes that point directly to the data in tables are called leaf pages (level 0). Besides the pointer to data, leaf pages contain the key and record-ID (RID). If an index has more than one leaf page at level 0, then it needs the next level up in the hierarchy, with at least one nonleaf page that contains entries that point to the leaf pages underneath. If the index has more than one nonleaf page, then the index has a level 1. Depending on the number of entries, other levels will be present. The highest level of an index contains a single page, which is called the *root page*.

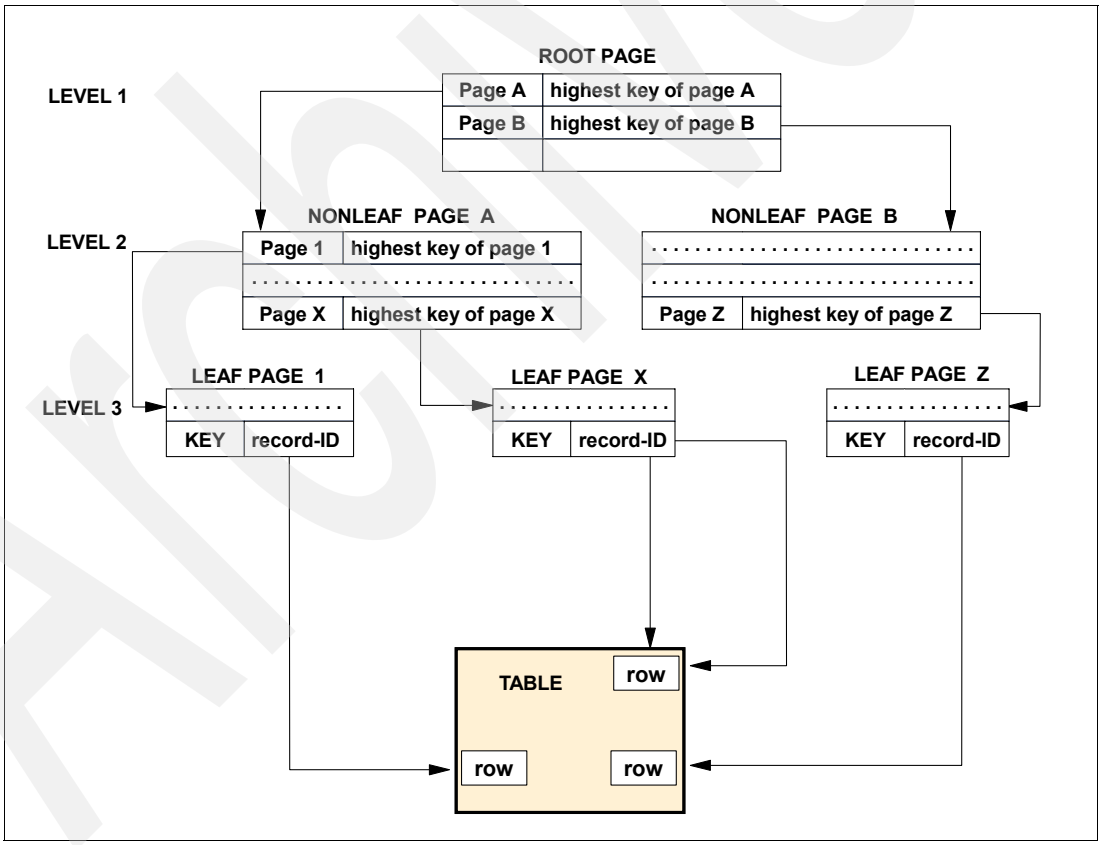


Figure 2-1 Sample three-level index structure

The number of index levels is important because an extra index level means an extra getpage for each (matching) index access (unless index look-aside can be used). For more information on index look-aside, see Appendix A.4, “Index look-aside” on page 426.

A DB2 index is a Type 2 index. In the past, there was a Type 1 index that had different characteristics, but starting with Version 8, all indexes are Type 2 indexes. Type 2 indexes require no locks on their pages. A lock on the data page or row locks the index key. They promote high concurrency because locks are not acquired on index pages during insert and delete processing or structure modification. Lock suspensions, timeouts, and, possibly, deadlocks are reduced, thereby increasing concurrency.

2.2 Why build indexes?

DB2 can use indexes to:

- Enforce uniqueness of primary keys and unique constraints, that is, ensure data integrity in an entity
- Improve performance in data access
- Improve concurrency

In most cases, access to data is faster with an index than with a scan of all the data. For example, you can create an index on the DEPTNO column of the DEPT table to easily locate a specific department and avoid reading through each row of, or scanning, the table.

Before Version 8, there was an additional reason to use an index. When using index-based partitioning, a partitioning index was required to indicate which rows go into which partition. Version 8 introduced table-based partitioning. When using table-based partitioning, a partitioning index is no longer required. We recommend table-based partitioning, although index-based partitioning is still fully supported in Version 8.

2.2.1 Uniqueness

DB2 uses a *unique index* to ensure that data values are unique. In a unique index, each value must be different. Each value can occur only once. To create a unique index, you use the UNIQUE clause of the CREATE INDEX statement. For example, the DEPT table does not allow duplicate department IDs. Creating a unique index prevents duplicate values. A unique index on DEPTNO can be created as follows:

```
CREATE UNIQUE INDEX X0DEPTNO ON DEPT (DEPTNO);
```

The index name is X0DEPTNO, and the indexed column is DEPTNO.

If a relational entity has a primary key defined (such as the DEPT table), its entries must be unique. The only way to enforce this uniqueness in DB2 is by defining a unique index on the primary key columns of the entity, with the index columns in the same order as the primary key columns.

A DB2 index can also be *nonunique*. The index allows duplicate entries for the same key value, or an index can be defined as *UNIQUE WHERE NOT NULL*. In this case, multiple null values are allowed, but any not null value must be unique.

2.2.2 Improve performance

Using an index can improve the access path in several ways.

- Indexes can be used to reduce the number of getpages.
  - Applying a predicate to the index may be more efficient than applying the same predicate to the data page. Using an index (by changing a table space scan into a matching or nonmatching index scan, for example) can reduce the number of data

pages and rows scanned. If a row does not qualify the predicates in the index, DB2 does not need to retrieve the data page. This applies to SQL statements that only access a few rows of data in a table, but also to processes that (sequentially) access a large quantity of data in the table, and to single table, as well as multi-table, join statements.

  - Reducing the number of data pages accessed is particularly interesting if the access is via an index that has a low cluster ratio. A low cluster ratio increases the probability of synchronous I/O and therefore the cost of the statement.
  - The best option is to make the predicate (matching) indexable. However, if this is not possible, you may still benefit from index screening.
  - Using an index can completely eliminate access to data pages when index-only access is possible.
- Because data access is completely eliminated when using index-only access, such an index creates an “alternative” clustering sequence for those queries.
- An index can also eliminate a sort, if the requested order matches the index (or is the exact opposite, in which case, you can use a backward index scan).

In these cases, the index may improve application performance. However, you should always evaluate the access path improvement that the index provides against the cost of maintaining the index.

2.2.3 Index design for concurrency

DB2 uses *data-only locking*. This means that no locks are taken on the index pages (the only exception is the end-of-file lock for repeatable read (RR) isolation), providing high concurrency and availability.

DB2 uses latches (and page P-locks in data sharing) to serialize updates to the index. When evaluating a predicate (matching or screening) in the index, DB2 can do most filtering in the index without taking a lock on the underlying data pages. Also, when updating index entries, DB2 does not keep a lock on the index page until commit, which is the case for a data page, but DB2 only latches the page for the duration to make the actual update to the page.

Therefore, data-only locking is great when most of the filtering for a query is done at the index level. As long as no data page/row lock needs to be acquired, multiple transactions using the index do not have to compete for potentially incompatible locks.

In a high concurrency environment, applications often have to rely on index usage to avoid locking conflicts and achieve the required throughput. These applications may benefit from using the VOLATILE attribute on tables with high concurrency requirements, since DB2 will highly favor an access path using a (matching) index. This may lead to the least optimal access path from a performance point of view, but it can improve concurrency. For more information about the use of volatile tables, see “Volatile tables” on page 23.

2.3 Cost of an index

An index can improve performance by reducing the number of rows and pages that have to be accessed, which provides more efficient access. However, overall performance must also be evaluated. Indexes also have an associated cost due to:

- The additional cost to maintain the index. The cost can be divided into:
  - CPU cost to update the index during an INSERT, UPDATE that references a column in the index, or DELETE operation.
  - Additional I/Os that may be required to make the required changes to the index. The I/O impact can be very different depending on the *cluster ratio* of the index. If the processing sequence is the same as the clustering index:
    - Indexes with a high cluster ratio will incur sequential data I/O and are likely to benefit from asynchronous I/Os, and the application will probably not have to wait.
    - Indexes with a low cluster ratio will incur random data I/O.

  So, depending on the index columns and their correlation with the table's clustering sequence, adding an index can have a different impact on the application's performance. Adding an index is not at a linear cost.
- Higher CPU usage and longer elapsed times for utilities, such as RECOVER, LOAD, and REORG that need to maintain the index.
- Disk space used by the index.

Maintaining the index

Indexes are great for SELECT operations, since they normally speed up the data retrieval considerably, but you always have to keep in mind that other operations that need to maintain the index, such as an INSERT/DELETE statement, require extra work when an index is involved. In case of an INSERT/DELETE, not only does DB2 have to insert/delete the row into the table, but it also has to put/remove the key columns and the corresponding RID of the row in all the indexes that are defined on the table.

The costs of maintaining indexes are often higher than the costs of maintaining the data. In a data sharing environment, inserting, updating, or deleting index entries comes at an even higher cost, because of the extra P-lock and cross-invalidation overhead that may be necessary to guarantee data integrity.

INSERT, UPDATE, and DELETE frequency

An INSERT, DELETE, or UPDATE operation modifying columns included in an index requires extra work when indexes are involved. This is because DB2 has to maintain those indexes, not just only the data. Therefore, when deciding which indexes to build, it is important to consider the amount of update activity (INSERT, UPDATE, and DELETE) against the table. With very little update activity, you can afford more indexes compared to a case where keys need to be updated all of the time.

Also, do not forget to take into consideration the update activity caused by referential integrity if the table has an (indexed) foreign key defined that is defined as ON DELETE SET NULL, or ON DELETE CASCADE. If there is an index defined on that foreign key, frequent deletes of rows in the parent table can cause heavy activity on the dependent table and its indexes.

Pseudo delete

If an indexed column in a table is updated, the key (and RID) is deleted from its existing position in the index and inserted with the new value in the correct (new) position. To make sure the deleted key does not have to be reinserted in case a rollback is performed, the

deleted key entry is marked as *pseudo-deleted*. Other processes retrieving the key entry will check the corresponding data row to find out if the deleter has committed in the meantime.

Page splitting

When an index page is full, a new page is acquired (either by using an existing empty page (free page) or allocating a new one). DB2 uses either of two techniques:

- The rows of the full page are split in half, leaving the old page with half the entries and the new index page with the other half; this is a so-called 50-50 split.
- The new key is inserted into the new (empty) page. This is a 100-0 split.

  DB2 uses a 100-0 split at the full end of the index; that is if the key that needs to be added is higher than any key currently in the index. A 50-50 split would waste half of the pages, because, since we are always inserting a higher key, we will never add any new keys to the (full) page that we had to split.

> **Note:** Be careful with null values in combination with an ever ascending key in an ascending index. A null value is x'FF' and will always sort at the end of the index B-tree. The new key that triggers the page split will be lower than x'FF', hence, a 50-50 split.
>
> The same applies when using a high value like 9999999 as a default value, when the actual column value is unknown. If 9999999 is in already in the index, and is higher than the key you try to insert, the result will be a 50-50 split.

Free space

Specifying appropriate free space is not only important for the data, but also for the index. The same parameters, PCTFREE and FREEPAGE, exist for indexes to specify the initial percentage of free space to keep per page, and to specify after how many pages to provide an empty page.

As for data free space allocation, index free space allocation is determined by the insert patterns used by the processes accessing the table.

If the insert and update pattern does not match the index key sequence, you must allocate free space in all pages in the index. In this case, you must select the PCTFREE value according to your table growth rate predictions and REORG frequency.

See also "Free space" on page 399.

REORG the index

When you REORG the index, previously pseudo-deleted entries are physically removed, and free space is reinstated.

REORG INDEX reads the index keys and associated RIDs from the index, which is normally less work than scanning the entire table space or partition (like a REBUILD INDEX does). REORG INDEX can also be performed using SHRLEVEL CHANGE. In that case, the index is available for read and write activity during most of the time while the index is being reorganized. Only during the last log iteration phase are writers drained, and only during the short switch phase is the index unavailable.

When you REORG the entire table space, you rebuild the index from scratch.

REBUILD the index

Instead of reorganizing the index, you can also rebuild it from the data in the table space. REBUILD INDEX is more expensive and more disruptive than REORG INDEX.

REBUILD INDEX has to read the entire table space, extract the key columns for the index, and build the index. When rebuilding an index, the table space or partition is in UTRO status and all writers are drained. This means that no update activity is allowed during the rebuild process.

CREATE INDEX DEFER YES

When creating an additional index on a table that already contains data, we recommend that you create the index with the DEFER YES attribute. This way the index definition is put into the DB2 catalog, but the index itself is not built at CREATE INDEX time. The newly created index is put into rebuild pending (RBDP) state, and you must use the REBUILD INDEX utility to actually populate the index. The reasons for using DEFER YES are:

- If you are adding multiple indexes to the table, using DEFER YES, REBUILD INDEX only has to scan the table once to extract the keys for all newly defined indexes. In addition, those indexes can be built in parallel.
- When using CREATE INDEX DEFER NO, the index is immediately populated at CREATE INDEX time. To get the keys in the correct (index key) order, they are sorted using the DB2 workfile database, usually DSNDB07, using the DB2 internal sort. When using REBUILD INDEX, DFSORT™ is used instead of the DB2 sort, so there is less interference with other sort activity in the system compared to using DEFER NO.

2.4 How does DB2 use the index?

There are different ways for DB2 to exploit an index. We discuss these methods in this section.

2.4.1 (Matching) indexable predicates

In DB2, there are two types of predicates that can use an index, or, in other words, that can be processed by DB2. They are:

- Indexable predicates
- Index screening predicates

DB2 uses the term *indexable predicate* when it can match index entries. So the DB2 term indexable predicate is more restrictive than just a predicate that is able to use an index. A predicate has to be able to use a matching index access path in order to be called an indexable predicate.

However, a predicate, that is indexable, may not become a matching predicate of an index at execution time. That also depends on:

- The different indexes that have been defined on the table
- The access path selected by the DB2 optimizer

Indexable predicates are good because they are processed early on during query execution. The more filtering that can be done by index manager earlier in stage 1, the less work DB2 has to perform at stage 2 or later.

For example, if the employee table (EMP) has an index defined on the LASTNAME column, the following predicate can be a matching predicate:

```
SELECT LASTNAME, FIRSTNME, JOB, SALARY FROM DSN8810.EMP WHERE LASTNAME = 'SMITH';
```

The following predicate cannot be a matching predicate, because it is not indexable:

```
SELECT LASTNAME, FIRSTNME, JOB, SALARY FROM DSN8810.EMP WHERE SEX <> 'F';
```

Recommendation: To make your queries as efficient as possible, use indexable predicates in your queries and create suitable indexes on your tables. Indexable predicates allow the possible use of a matching index scan, which is often a very efficient access path.

You can find a list of indexable predicates in the “Summary of predicate processing” section of *DB2 UDB for z/OS Version 8 Administration Guide*, SC18-7413-02.

Another way to find out if a predicate is indexable is to use DB2 Visual Explain. Since the table in the *DB2 Administration Guide* is not very easy to interpret, and DB2 maintenance can change a predicate’s indexability, using Visual Explain is the easier and most accurate way to determine if a predicate is really indexable or not. See Figure 2-2.

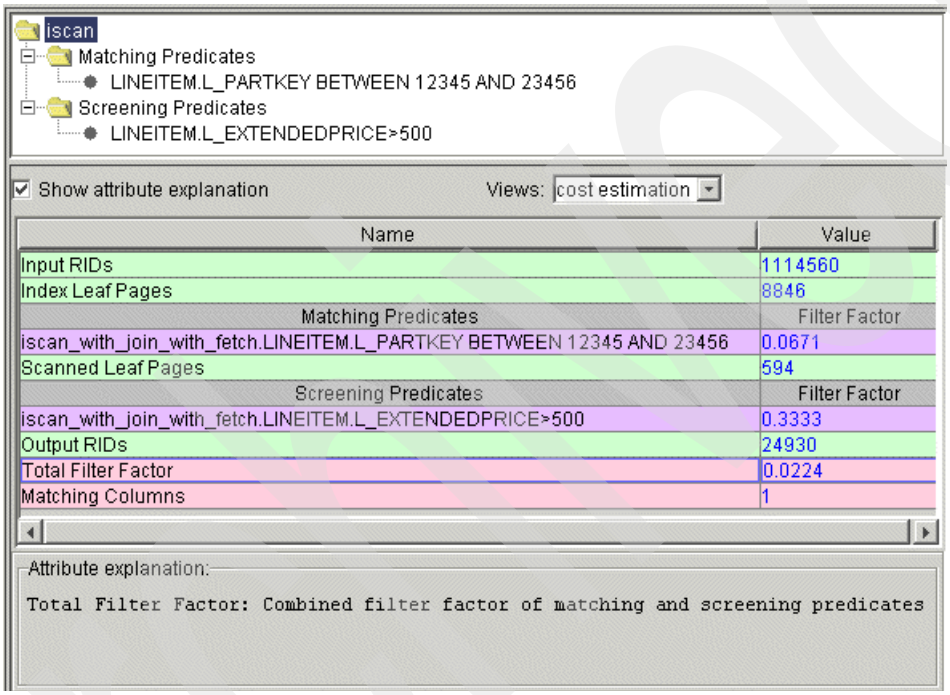


Figure 2-2 Index matching and screening predicates

When an index exists on multiple columns, it is generally true that the more predicates on which DB2 can do matching, the more filtering can be done in the index, and the more chances you have that the query will have good performance.

Note: The amount of filtering that can be done is more important than the number of matching columns. See “Selectivity is more important than the number of matching columns” on page 72 for more details.

2.4.2 Index screening predicates

The next best thing to being a (matching) indexable predicate is an *index screening predicate*. An index screening predicate can also be evaluated by DB2, but DB2 cannot use the index tree structure. In other words, index screening predicates are predicates that are referencing columns in the index, but are not part of the matching index columns.

Index screening predicates do not limit the index I/O, but can reduce data page I/O. Only index matching (or page range screening) predicates can reduce index I/O.

Index screening can occur when using:

- A nonmatching index access (ACCESSTYPE=I and MATCHCOLS=0)
- Additional filtering on columns in the index after the matching predicates have been applied, including access paths where list prefetch is used (ACCESSTYPE=I and MATCHCOLS=*n*, where *n* < number of columns in the index).

An example to illustrate this:

```
SELECT my-cols
  FROM T
 WHERE
       C1 = 1
   AND C3 > 4
   AND C4 = 6;
```

With an index on T(C1,C2,C3), C3 > 4 is an index screening predicate. It can be applied when accessing the index, but it is not a matching index predicate like C1= 1. The value of MATCHCOLS in the PLAN_TABLE is 1. C4 = 6 is not an index screening predicate.

Note: A predicate (C3 > 4 OR 0=1) cannot be an index screening predicate, because it is a stage 2 predicate.

Any column in the index that is not one of the matching columns for the query will be considered for index screening. For example, if the EMP table has an index on LASTNAME, JOB, SALARY and you have a predicate:

```
WHERE LASTNAME='HAAS' and SALARY > 20000
```

LASTNAME='HAAS' can be a (matching) indexable predicate, whereas SALARY > 20000 can be an index screening predicate. Since there is no predicate on JOB, SALARY > 20000 is not an indexable predicate in this query.

The fact that a predicate is an index screening predicate does not show up in the EXPLAIN output, but Visual Explain lists both matching indexable and index screening predicates (see Figure 2-2 on page 66).

2.4.3 List prefetch

List prefetch is a data prefetch technique that only applies when index access is involved. List prefetch is used to read a set (list) of data pages, determined by a list of RIDs taken from an index. The data pages do not need to be contiguous. The maximum number of pages that can be retrieved in a single list prefetch is 32 (64 for utilities). Prior to DB2 Version 8, the pages within a single list prefetch I/O request had to be within a range of 180 VSAM control intervals. This restriction was in place to avoid contention on long I/O response times. With advances in disk technology, especially Parallel Access Volume (PAV) support, this restriction was lifted in Version 8 for list prefetch (and castout I/O).

List prefetch can be used in conjunction with either single or multiple index access.

List prefetch uses the following three steps:

1. RID retrieval: A list of RIDs for needed data pages is found by index access of one or more indexes.
2. RID sort: The list of RIDs is sorted in ascending order by page number.
3. Data retrieval: The needed data pages are prefetched in page order using the sorted RID list.

List prefetch does not preserve the data ordering given by the index. Because the RIDs are sorted in page number order before accessing the data, the data is not retrieved in order by any column. If the data must be ordered, by an ORDER BY clause or any other reason, an additional sort will be performed.

In a hybrid join, if the index is highly clustered, the optimizer does not sort the page numbers before accessing the data. List prefetch can be used with most matching predicates for an index scan. IN-list predicates are the exception; they cannot be the matching predicates when list prefetch is used.

List prefetch is mostly used:

- With a single index if its cluster ratio is lower than 80%.
- With multiple index access, list prefetch is always used to access data.
- Sometimes on indexes with a high cluster ratio. That is, if the estimated amount of data to be accessed is too small to make sequential prefetch efficient, but large enough to require more than one synchronous read.
- Always to access data from the inner table during a hybrid join.
- Usually for updatable cursors when the index contains columns that might be updated.

List prefetch and multiple index access

Most of the time, DB2 picks a single index to access a table. However, when multiple indexes are available, DB2 may decide to use more than one index to access a table. This access path is called *multiple index access*.

DB2 constructs a list with the qualifying RIDs for each of the indexes involved. If the predicates are ORed together, DB2 unions the RID lists. When ANDed together, DB2 intersects the RID lists to produce a final list of qualified RIDs. That list is used to retrieve the result rows, using list prefetch. You can consider multiple index access as an extension to list prefetch with more complex RID retrieval operations in its first (RID retrieval) phase. The complex operators are union and intersection.

List prefetch and index screening

DB2 can use index screening during RID list processing to filter out additional rows at index access time. This has the following positive effects on query processing time as well as more concurrency:

- RIDs that do not qualify, after evaluating the index screening predicates, no longer need to be passed on for data page access and evaluation of the predicate. In other words, DB2 only needs to do a getpage request for those pages that qualify both the matching index predicates as well as the index screening predicates. Depending on the amount of rows that are discarded by the index screening predicates, this can reduce the amount of data getpages significantly.
- Because more rows can be filtered out at index (screening) access time, the number of RIDs that require sorting, or need to be processed by multiple index access AND and OR operations, also decreases by this enhancement. This can have a significant impact on the RID pool size that your installation requires.
- Index screening during RID-list processing can also have a positive effect on concurrency. Since DB2 can apply index screening predicates at the index level when using RID-list processing, the number of rows to be evaluated at the data page level is much smaller, and, therefore, it reduces the chances of running into a lock suspension, therefore, increasing concurrency.

The optimizer filter factor calculations take index screening during list prefetch into consideration.

List prefetch thresholds

There are two types of thresholds when dealing with list prefetch; bind time and run time thresholds. We describe our understanding of DB2 V8 settings, they are subject to change with any maintenance.

Bind time thresholds

DB2 does not consider list prefetch (and, as a consequence, also disable multiple index access) if the estimated number of RIDs to be processed would take more than 50% of the RID pool when the query is executed, or when DB2 estimates that more than 25% of the rows of the table will qualify.

Run time thresholds

The maximum number of RIDs that can be addressed from a single RID map is approximately 26 million RIDs.

When RID list processing is requested, you will always get at least one RID list, which can store about 6,530 RIDs. So, if fewer that 6,530 RIDs qualify, list prefetch will not fail.

During execution, DB2 ends list prefetching if more than 25% of the rows in the table (with a minimum of 6,531 (number of RIDs in a single RID list)) must be accessed. The decision is made based on the catalog statistics information available at bind time.

When DB2 forms an intersection of RID lists, if any list has 32 or fewer RIDs, the intersection stops, and the list of 32 or fewer RIDs is used to access the data.

When list prefetch ends, the query continues processing by a method that depends on the current access path.

- For access through a single index or through the union of RID lists from two or more indexes, processing continues by a table space scan.
- For index access forming an intersection of RID lists, processing continues with the next step (next index) of multiple index access. The index predicate that failed to use RID list access is flagged to be evaluated again when the data pages are accessed later.
  If no step remains and no RID list has been accumulated, processing continues by a table space scan.

List prefetch monitoring

IFCID 125, in the performance trace class 8, provides the details about whether list prefetch was used successfully or not, for each index processed by list prefetch. For more details, see “List prefetch monitoring” on page 395.

2.4.4 Sort avoidance

You can use indexes to avoid a sort in SQL statements. GROUP BY, ORDER BY, DISTINCT, and join processing can all benefit from indexes. Using indexes to guarantee uniqueness does not require any specific column sequence, as long as all the columns are present in the index. In fact, the SQL statement may even include more columns since you cannot get more unique than unique.

GROUP BY requires all columns to be present in the index and in the same sequence as the SQL statement. Whether ascending and descending corresponds is irrelevant since the GROUP BY does not imply any sequence.

Using an index to avoid a sort for ORDER BY has the most requirements. The sequence of the columns in the index must be the same as in the ORDER BY clause. In some cases, extra columns in the ORDER BY clause that do not appear in the index are allowed in the query, and DB2 will still be able to avoid having to sort. The ascending and descending preference in the index definition should also match the specification in the ORDER BY, or be the exact opposite in the index compared to the ORDER BY clause.

A few examples to illustrate this. Table EMP has an ascending index on EMPNO defined.

```
SELECT EMPNO FROM DSN8810.EMP ORDER BY EMPNO ;
SELECT EMPNO FROM DSN8810.EMP ORDER BY EMPNO DESC ;
```

In DB2 Version 8, both queries use index-only access without a sort. DB2 Version 8 introduced the *backward index scan* feature. The second statement does not require a sort in Version 8 but did in previous DB2 versions. The backward index scan feature allows DB2 to exploit the existing backward chain pointer inside the Type 2 index. Backward index scan can be used when the ORDER BY columns are the same as in the index, and have the exact opposite collating order. In the example above, EMPNO has an ascending index defined; an ORDER BY EMPNO DESC can exploit the index backward pointer, and can be performed without invoking a sort.

The same applies for the MIN and MAX functions.

```
SELECT MAX(EMPNO) FROM DSN8810.EMP ;
SELECT MIN(EMPNO) FROM DSN8810.EMP ;
```

Both queries can use a one-fetch access (ACCESSTYPE='I1') since Version 7, and no sort is required for either query. Version 7 allowed MIN and MAX to exploit the backward index pointer.

DB2 can also do sort avoidance in more complicated situations. Assume a table T1 with columns A, B, C, and D, and a multi-column ascending index (IX1) on columns A, B, and C. When executing:

```
SELECT A,B,C
  FROM T1
 ORDER BY C DESC
```

Can this query avoid a sort? The index that exists contains more columns than what is contained in the ORDER BY columns. Can DB2 handle the extraneous columns and use backward index scan as long as the ORDER BY is opposite the built index collating sequence and contains the column in the ORDER BY clause?

The answer in this case is no. A sort is needed because the index is not in column C order, neither in forward nor backward order. If the index had been defined (C, A, B), then ORDER BY C DESC could use that index, since the index would now be in column C order, so a sort could have been avoided.

However, in certain cases, it is possible to have an index on A, B, C, and have ORDER BY C DESC avoid the sort. This is the case when you have '=' predicates on columns A and B. For example:

```
SELECT A, B, C,
  FROM T1
  WHERE A=1 AND B=2
 ORDER BY C DESC;
```

In this case, DB2 can avoid the sort because of the '=' predicates. In this case, DB2 knows that only one value will qualify for both column A and B, which means that all qualifying rows will be in column C order if index IX1 is used.

However, keep in mind that in general, the filter factor of the predicates that can be applied to the index is more important than the ability to avoid a sort, since an efficient matching index access is normally more desirable than sort avoidance.

On the other hand, sort avoidance can be very important when not all rows from the result set will be processed by the application. In other words when the application stops fetching before SQLCODE +100 is reached, or when FETCH FIRST *n* ROWS ONLY is used. When you must perform a sort of 100,000 rows, and only the first 20 are ever looked at, it is important that sort is avoided. You want to avoid sorting 100,000 rows and only looking at 20 of them.

2.4.5 Index-only access

It may make sense to add a column to the index, although the column does not influence the filtering of the index, because it can make your access path index-only. Using an index-only access path can have a significant impact on the efficiency of a non-clustering index. A low cluster ratio can reduce the efficiency of an index if the number of qualifying rows is important. Even though DB2 uses list prefetch to alleviate the cost of the synchronous I/Os on the data pages, you still incur the cost of sorting the RIDs. By making the access path index-only, access to the data pages is eliminated, and the cluster ratio of the index is no longer an issue.

If the index is defined as NOT PADDED, DB2 can use index-only access even when VARCHAR columns need to be returned to the application.

The basic rule of the change/retrieval ratio applies when deciding whether to add a column or not.

2.5 Will DB2 use the index?

When you give DB2 an SQL statement to process, it is the DB2 optimizer that decides which is the best way to retrieve the requested data. This also applies to whether or not an index will be used. However, in general you do not need to understand how the decision is made. Avoid building indexes that the optimizer will never use. Indexes that are not used for data retrieval still pay the price for maintaining the indexes during update operations (I,U,D).

So when will DB2 use an index?

We have to distinguish among three ways of “using” an index:

- The first way is when DB2 must use the index because the SQL operation requires that the index is modified. This is the case for INSERT, DELETE, and UPDATE (if the updated column is present in the index) operations.
- The second way is during retrieval. In this case, the use of the index is optional, and is only used to speed up the query processing. At query optimization time, the optimizer calculates the relative cost of all access paths. When the optimizer thinks that the cost of using the index is higher than the cost without the index, the index is not used.
- The third way is usage of the clustering index to find the preferred location for a row to be inserted.

So let us look at a few cases where the optimizer is not likely to use an index:

- If the filtering that can be achieved by applying predicates (matching and screening) to the index is very limited, the index is unlikely to be used.

- If the access path is index + data, the filtering is not significant, and the cluster ratio of the index is very low, the optimizer is not likely to use that index. List prefetch can be used, however, to reduce the I/O requirements.
- If the number of levels in an index is higher than another index, the one with more levels is not likely to be used unless it provides greater filtering. More index levels mean more getpages, so more CPU and potentially more I/O will be required.

While the filtering plays a major role in index selection, I/O reduction is another major factor. Better clustering can compensate for lower filtering, as more rows may be retrieved, but these may occur on fewer pages, when compared with better filtering, but poor clustering. Another exception to the above cases is index selection encouraged by sort avoidance, or a requirement to retrieve a subset of rows using FETCH FIRST or OPTIMIZE clauses.

2.6 Which indexes to build

Index design is a form of art. This means that it is difficult to describe a process that is simple and easy to understand, but still encompasses all the things you need to consider at the same time. In addition, there is always a trade-off depending on a specific application or environment.

There is much more to be said. In this section, we merely intend to give you a framework to use when performing index design.

2.6.1 General index design guidelines

The idea behind using an index is to provide faster access to the data in the table. Instead of scanning all rows, we have an index tree that we can traverse very efficiently, get the qualifying RIDs, and retrieve the data. The efficiency of the index usage depends on a number of things, such as:

- The number of matching index columns

  This depends on the predicates that are used on the query and whether they are the leading columns in any available index.

- The availability of index screening columns

  This depends on the predicates that are used on the query and whether the columns used in the predicate exist in the index.

- The amount of filtering that can be done in the index

  Even if DB2 can match on a number of columns, if the indexed columns have low cardinality, or the index filtering is not significant; then a lot of index keys will qualify, and many data rows have to be retrieved to apply the additional predicates.

- The number of levels in the index

  This depends on the number of rows in the table and the size (number and length of the columns) in the index. As a general rule of thumb, more than three levels are undesirable.

Selectivity is more important than the number of matching columns

As mentioned above, when the EXPLAIN output shows that an index is used, for example, with MATCHCOLS=1, this does not necessarily mean that the access path is a good access path. It is probably better than a table space scan, but you also need to take into consideration the amount of filtering that is done by the index.
If the indexed column that you are matching only contains a few different values, a low

cardinality column, a predicate on that column is unlikely to provide good filtering. We assume a uniform distribution here.

Generally speaking, low cardinality columns, such as year, state, or gender, are less likely to be selective columns. Therefore they are not a good choice for single column indexes, unless the values are not uniformly distributed, and you are frequently searching for one of the low frequency values. For example, WHERE STATUS='NEW' and only 1% of the values in the status column is a new order (STATUS=NEW).

On the other hand, low cardinality columns can be good candidates for a partitioning or clustering column.

Therefore, adding additional columns to an index is usually a good idea, assuming those columns are also used as predicates that provide additional filtering.

High cardinality columns, such as account number, are good candidates for a single column index. They usually provide good filtering in the index, so only a limited number of rows need to be retrieved from the table for additional filtering.

Tip: High cardinality columns are viable candidates for single column indexes.

Note that the cardinality is certainly not the only consideration to index a column or not. Things such as sort avoidance, clustering, and index-only access also influence index design.

Column correlation

When looking at filtering, not only the cardinality of a column is important, but also, whether or not a column is correlated to other columns.

If column correlation is not known to the optimizer, then it can overestimate the filtering of applying multiple predicates, and the optimzer can pick a bad access path.

Good examples of column correlation are:

- Zip code and city.
- City and state.
- Automobile manufacturer and model.
- Product and supplier, since most companies normally buy a certain product from a limited set of suppliers.
- Customers and stores. Most customers return to the same or only a few stores.

For instance, examine the following query:

```
SELECT a-bunch-of-columns
  FROM CUSTOMER
  WHERE ZIPCODE = '95123'
    AND CITY = 'SAN JOSE'
```

After you filter on zip code (ZIPCODE='95123'), the CITY='SAN JOSE' is not going to do any additional filtering, since zip code "95123" only occurs in San Jose. However, unless the optimizer knows about this, it will assume that both columns are independent (not correlated), and the combined filtering is (FF of zipcode column)* (FF of city column), which is clearly a wrong assumption. If there is an index IX1 on (ZIPCODE, CITY, LASTNAME), the previous query would use an index access path with MATCHCOLS = 2, but the second matching column on CITY would not do any filtering.

To get an idea of column correlation, you need to collect proper statistics, more precisely multi-column cardinality statistics. If you suspect columns are correlated, it is important that the optimizer knows about it, so the optimizer can take this information into account during access path selection. Therefore:

- For leading indexed columns, run the RUNSTATS utility with the KEYCARD option to determine the column correlation.
- For all other column groups, run the RUNSTATS utility with the COLGROUP option.

2.6.2 Must build indexes

Since the only way to guarantee uniqueness in DB2 is by creating a unique index on a column or a group of columns, you must create an index:

- On each primary key
- On columns or a group of columns that have uniqueness requirements (without being a primary key)

Tip: When using index-based partitioning, you must build the partitioning index. It was required to indicate which rows went into which partition. Version 8 introduced table-based partitioning where a partitioning index is no longer required.

If you have a partitioning index that dates back from pre-Version 8 that is not used by any SQL statement, that index can be dropped. When the index is dropped, DB2 will automatically convert the table from index-controlled to table-controlled. This conversion does not require any changes to the existing data; it is merely a few changes to the DB2 catalog.

Restriction: You cannot add additional columns to an index when enforcing uniqueness. An index that enforces uniqueness can only be defined on the set of combined columns that have to be unique. For example, if you determine that the combination of COLA, COLB is a primary key (and, therefore, unique), you cannot define an index in DB2 for z/OS on COLA, COLB, and COLC, and then indicate that only the combination of COLA, COLB values have to be unique, and disregard COLC values when enforcing uniqueness.

2.6.3 Highly recommended index candidates

An index should be defined on each foreign key to speed up performance during CHECK DATA processing and when deleting or updating parent rows. If no index is available on the foreign key, checking the foreign key to see whether a value exists, or finding that corresponding foreign keys to delete or set to null, would require a table space scan. This can seriously affect performance.

Foreign keys also often participate in join operations. Many joins are between the primary and the foreign key columns. Again, an index on the (foreign key) join column will considerably improve join performance. Generally speaking, we recommend that you create an index on columns that are frequently joined together. This is certainly true when the table is the inner table of a nested loop join. We discuss indexing for join columns in “Index considerations for join operations” on page 79.

Another index that you should create is the clustering index. It is important to have an explicit clustering index defined. See “What is a clustering index?” on page 88.

2.6.4 Indexes that require analysis

Similar to other physical design decisions, a CRUD (create, retrieve, update, and delete) matrix is very useful to understand how the data will be processed by different processes in the system, and which indexes can be built to speed up those processes. Whether or not you use a CRUD matrix is irrelevant. The important thing is that you do a detailed analysis of how different processes access your tables.

Typically, the following columns are good candidates to include in an index:

- Columns that are frequently used in column functions, for example, MIN, MAX, COUNT, SUM, and AVG.
- Columns that are used for existence checking.
- Columns frequently used in an ORDER BY, GROUP BY, or DISTINCT clause where an index can avoid the sorts that are normally required to process these clauses.
- Columns that are used during browsing operations, especially in traditional transactional environments such as IMS and CICS, where each scroll operation requires the program to reposition where it left off the last time.
- Columns, which are searched or joined over a fairly large percentage of the total number of rows, are good candidates to become the clustering index. (Clustering indexes are discussed in “Clustering index” on page 88.)
- Columns, which are searched or joined over less than 5 to 10% of the rows, are usually good candidates to become a non-clustering index.
- Columns, frequently used together in a WHERE clause, can benefit from a composite index to avoid maintaining multiple indexes.
- Consider adding additional columns to the index to be able to use index-only access.

General indexing strategy

Although it is difficult to provide an all encompassing but simple index design strategy, we give it a try in any case:

1. Perform physical table design, without considering indexes.

   Make sure that the table “attributes“ are addressed during the table design. This does not mean they have to be 100% correct at this point. A design usually goes through multiple iterations.

   Note that at this point, partitioning has only been considered when you already know that you will need to partition. For example, because you are going to exceed 64 GB, which is the maximum table size for a nonpartitioned table.

2. Create “no brainer indexes”. These are the unique indexes, indexes on primary and foreign keys, as discussed in “Must build indexes” on page 74, and “Highly recommended index candidates” on page 74.

3. Create a CRUD matrix for all the processes accessing the table. The following items need to be considered:

   – The type of operations; SELECT, INSERT, UPDATE, or DELETE.

   – The ratio of the number of rows returned by an SQL statement and the total number of rows in the table.

   – The type of predicates being used; equal, range, join.

   – The columns that are retrieved from the table (in the SELECT list).

   – The importance of the statement/process and the frequency at which the statement/process will be executed.

If there are a lot of processes/tables involved, use the 80/20 rule, and focus on the 20% that represents 80% of the work.

– Note that an individual table indexing approach may not work. Some tables are processed together and the data must be in the same clustering sequence, and, therefore, indexes should be designed to support these processes.

This analysis should lead to a list of potential index columns and indexes; the so-called Version zero of the IX design.

4. Now that you have a list of potential indexes, see if one them is a good candidate to become the clustering index for the table. If not, an additional index to support the clustering may be required. See “Clustering index” on page 88 for a more detailed discussion about clustering.

2.6.5 A multi-column index or multiple indexes

During index design, you have to decide whether you combine all columns into one large multi-column index or cluster the columns into multiple indexes and use multiple index access. Five criteria are important when making the decision:

- Predicate use
- Filtering of the index columns
- Enforcing uniqueness
- Efficiency of the access path
- Index maintenance

Predicate use

The way in which processes use predicates is the main factor in deciding which option to use. The combination of the predicates, the criticality of the process, and the logical operator combining the predicates determine your design.

- When certain columns are present in most queries and the predicates are connected via an AND, they can really benefit from multi-column index.
- Predicates connected via an OR cannot use a multi-column index unless the OR is on the first column in the index or a non-matching index access is used.

Filtering of the index columns

The cardinality of the index columns is an important factor when deciding to cluster a set of columns in a separate index. You could decide to create a separate index on a set of columns assuming that additional predicates can be applied to another index using a multiple index access path. The thresholds in multiple index access path processing can make your access path switch to a table space scan. To avoid reaching the thresholds, the number of qualifying RIDs must be limited, which means that the filtering of the index predicates must be sufficient.

Enforcing uniqueness

If you want DB2 to enforce the uniqueness in an entity and to make it unique, you need multiple columns. You must define a unique (multi-column) index over those columns. You cannot add additional columns, or split the column set into multiple indexes when enforcing uniqueness. The only design option you have for a unique index is the sequence of the columns in the index. The column sequence in the index, that is used to enforce the primary key constraint, must be the same as the column sequence of the primary key.

Efficiency of the access path

The efficiency of the access path depends on a number of factors. Given the design options that you have, there are two main access path possibilities:

- Multiple index access
- Single index access

Multiple index access path

You can effectively use multiple indexes in one SQL statement if the optimizer decides to use a multiple index access path. List prefetch is used to retrieve a set of RIDs and not actual column values.

The use of list prefetch has two side effects:

- You will never have an index-only access path.
- All qualifying rows must be retrieved from the index before the first row is passed to the program.

The fact that all qualifying rows must be retrieved from the indexes can be an issue if your program does not have the intention to process the entire result set. The efficiency of a browsing application could be affected by this behavior. List prefetch can also cause an additional sort, which is not a problem if the number of qualifying rows is limited.

Single index access path

A single index can provide an efficient access path provided the filtering of the matching and screening predicates is sufficient. It is difficult to design one efficient index if half of the SQL statements provide a predicate for COL1 only, and the other half provide a predicate with only COL2. In half of the cases, you would only get a non-matching index scan, if not a table space scan. To get efficient SQL using a single index, the majority of your SQL statements must use the same matching predicates.

A single index access path can also exploit index screening, and, therefore, have a positive impact on the number of accesses to the data pages.

Using an index-only access path can also enhance the efficiency of those SQL statements or programs processing a large number of rows using a non-clustering index.

A multi-column index can also be used to avoid a sort. A sort is always required if multiple indexes are used and an ORDER BY clause is present.

Compared to a single-column index, a multi-column index needs more space, and is, therefore, likely to have more index levels, which makes it less appealing to the optimizer. Because having more index levels requires more getpages, especially if index look-aside cannot be used.

However, access via multiple indexes can require the retrieval of more RIDs than access via a single multi-column index. For instance, examine the following query:

```
SELECT my-cols
  FROM T
 WHERE C1 = 1
   AND C2 = 2;
```

Assume an index on C1 qualifies 1000 rows, an index on C2 qualifies 500 rows, and C1 and C2 combined qualifies 100 rows. Multi-index access must first retrieve 1000 RIDs from the index on C1, 500 RIDs from the index on C2, and finally intersect those rows existing in both RID lists to retrieve a final 100 qualifying RIDs. With a single index on C1, C2 is able to match both columns and retrieve only the 100 RIDs that qualify both predicates combined.

Index maintenance

If a lot of INSERTs, DELETEs, and UPDATEs are performed, keep in mind that maintaining multiple indexes is more expensive than maintaining a single multi-column index.

Column sequence in a multi-column index

After the decision had been made that a multi-column index is the best option, you need to decide on the column order inside the multi-column index. The following things need to be considered:

Maximizing the number of matching columns

The number of matching predicates is maximized if you choose all the columns that have the ' = ' (or single 'IN') operator as the first columns. When several columns are always used with an ' = ' (or single IN) operand, then choosing the column sequence depends on other factors such as which sequence aligns more closely with the clustering, ensuring that data access will be more sequential (and dense) rather than random (or scattered).

It is not required that the leading index column has the highest leading cardinality. This was recommended prior to DB2 V5 due to the lack of correlation statistics available to the optimizer. The additional statistics that can be obtained by the KEYCARD option of RUNSTATS, which are now available in DB2, provide you with greater design flexibility.

Remember that range predicate columns end the matching, and should therefore be put after '=' columns. For 'IN' predicates, DB2 can match through the first 'IN' predicate, but will stop matching immediately before the second 'IN' predicate. Therefore, '=' predicates should take priority for index column sequence, followed by the most filtering 'IN' predicate, then the most filtering range predicate. For filtering, the sequence of remaining 'IN' or range predicates is not important.

However, the index column sequence of any range predicates, and nonmatching 'IN' predicates, can be dictated by the requirement for sort avoidance (see below) or better alignment with clustering.

In addition, after the matching stops, additional predicates can still be evaluated during index screening if their columns occur after the matching columns in the index.

Sort avoidance

Using an index to avoid a sort enforces the column sequence since the index has to match whatever the SQL statement is using. Matching the sort requirement (such as from ORDER BY or GROUP BY) to the index will take into consideration '=' predicates that guarantee a constant value, and, therefore, can be ignored.

After taking into consideration '=' predicates, you do not have any further design options unless you change the SQL statement to match the index sequence, which is possible for GROUP BY, but generally not for ORDER BY. Therefore, you may have to decide which one is more important, avoiding the sort, or increasing the matching index filtering.

Decreasing the reorganization frequency

The column sequence and the insert pattern determine the degree of disorganization you experience in your table. If you want the clustering index to match the insert pattern, you may be required to use a specific column sequence. There might be a conflict of interest between the insert sequence and the index design to support the predicate filtering. The impact on access path efficiency is probably the most important factor and therefore clustering to support data retrieval is generally more important than insert sequence. As a result, you may be required to increase the reorganization frequency or adapt the free space in your objects.

Partitioning can also reduce the insert space search since data must be inserted into the partition which qualifies against the limit keys. Restricting the inserts to a smaller number of partitions can reduce the scope of data that must be reorganized.

2.6.6 Index considerations for join operations

All the considerations for single table access, such as picking those predicates that provide good filtering, also apply for joins. However, there are additional considerations when dealing with joins that we discuss in this section.

First table index design

Besides the single table index design options, when dealing with the outer table of a join, you also need to consider the following when deciding on the index column sequence:

- When accessing the first table in a join, only local predicates are applied. To make the applying of those local predicates as efficient as possible, make the filtering local predicates the leading columns in your index so they can be used as matching index predicates.
- Choose columns "=" predicates and/or the most filtering "IN" predicates as leading index columns over range predicates. Range predicates will stop index matching, and so will the second 'IN' predicate.
- Join predicates are not relevant for index matching. However, they may be important for index-only access, the cluster ratio, guaranteeing order for the join, or for the ORDER BY.

Index design after the first table

When evaluating predicates after the first table has been accessed, local and join predicates are treated the same. Both local and join predicates can be matching index predicates on an inner table of a join. Make sure that the local or filtering predicates (local or join) are index matching predicates, and not index screening. The faster a row can be discarded during the join the better.

Index column sequence

When you have a join predicate and a local predicate on the inner table, which column should go first in the index? For example, if both predicates, local and join are "=" predicates, both explains would show MATCHCOLS=2 for the index, but that does not mean they will perform the same in all cases. Here are some considerations:

- Local predicate column first
  - Putting the local predicate first is a good choice when the (inner) table can also be the outer table in other queries. (When accessing the outer table, only local predicates can be matching, as mentioned before).
  - A leading local predicate in the index can be exploited by a nested loop join, merge scan join, or hybrid join.
  - If the local predicate is the first index column, all qualifying index entries (after evaluation of the local predicate) are clustered together. This limits the number of I/Os in the index, and may allow DB2 to exploit index look-aside and reduce the number of getpages in the index as well.
  - Putting the local predicate first requires that the local predicate is present in most queries.
- Join predicate column first
  - Making the join predicate column the first column in the index certainly encourages this table to become the inner table of the join

- It encourages nested loop or hybrid join, but not merge scan join, because there are no local predicates that can be applied in the index before the join operation.
- Even though we can do a quick match on the join column, it is likely that the values of the local predicates are spread around within the index and can potentially increase the number of index I/Os.
- This is a good option when the local predicate does not exist in most queries, as it allows for (matching) index support during the join.

Summary

When designing indexes for joining operations you must consider that:

- Local predicates make better leading index predicates.
  - Provided they exist in the majority of the queries.
  - Because they support all three major join methods.
  - They support the table as either the inner or outer table of a join.
- Join predicates are only matching predicates on non-leading tables.
  - They should be used as leading index columns if local predicates are not always present.
  - You can add them after local predicates to guarantee subsequent join order.
  - Make sure that join predicates are index matching, not index screening.
- While clustering within a single table is important as more rows are retrieved, even more important for joins, is the clustering between tables.
  - If the inner table sequence from the outer table is guaranteed by the outer index, then it is less important to cluster both tables in the same sequence.

2.6.7 Less attractive index candidates

In this section, we examine situations where indexes are not recommended or produce strange side effects.

Short-term data

Short-term (volatile) data should have fewer indexes than static data. This is the case for data collection tables, where data is added and deleted the same day or a few days later. In these cases, the efficiency in a few queries may not compensate for the insert and delete overhead, unless you need the index for filtering and the processing sequence.

Nonunique low cardinality indexes

Nonunique indexes keep duplicate entries in RID ascending order. A binary search method is used to find the required RID or RID location, so inserts, updates, and deletes of an occurrence are not required to scan the entire list of duplicate RIDs.

For query performance, low cardinality columns do not provide significant filtering. For this reason, low cardinality columns are generally not a good choice for indexes.

There are two exceptions, however, when you are searching for a very low frequency value, and/or the low cardinality column is used for clustering. Since the goal of the index is to reduce data page access, then the low frequency value should reside on fewer data pages, and clustering ensures that data with the same values is densely packed together on consecutive data pages. Therefore, even moderate filtering from the clustering columns reduces data page access.

Consider the case of an index on columns with a highly skewed distribution of data, where one entry has many duplicates (typically, the default, null, zero, or blank values), and the rest have one or few duplicates on average. For example, codes are used to specify the status of occurrences in their life cycle, and a particular status has the most occurrences. In the Invoice table, for example, the typical value of the Invoice_Status column is “paid”. In general, you need an index on this code to retrieve the entries in the atypical status, for example, “unpaid”. Whenever any occurrence changes its status to or from the usual status (when you pay the invoice or reclaim a paid invoice), DB2 inserts or deletes a duplicate in the index entry.

When inserting a new duplicate, DB2 locates the target leaf page by analyzing the RID values in the nonleaf pages. Subsequently, DB2 positions the RID in the leaf page by a binary search of the key map and adds the new RID.

When deleting a duplicate, DB2 locates the leaf page containing the effected RID by analyzing the RID values in the nonleaf pages. Subsequently, DB2 locates the RID in the leaf page by a binary search of the key map and pseudo-deletes the effected RID. We show an example in Figure 2-3.

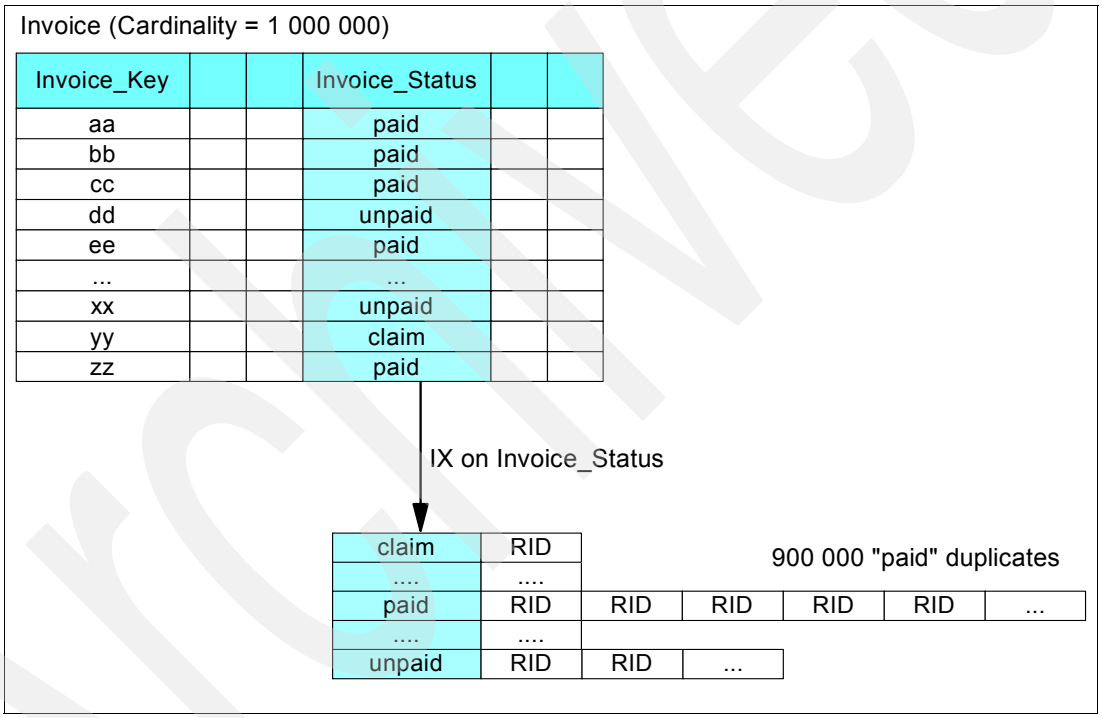


Figure 2-3 Nonunique index

Even though DB2 uses an efficient binary search to find the matching RIDs, if you have many duplicates, you may still want to consider reducing the number of duplicate index entries by adding additional columns to the index. An excellent candidate is the primary key. Usually, it is short and very stable. However, it is not a good idea to do this if the primary key is a hot spot in the index and the nonunique index has a low cardinality, because you would create a second hot spot, for example, adding the Invoice_Key to the index (because all new invoices have Invoice_Status “unpaid”). You can even define new columns for this purpose, such as a sequential number or a timestamp used as the last index column.

In practice, if you need to retrieve the rows with the typical value as the only condition, you need to add columns to the index in case your application uses a cursor that needs to be able to reposition itself.

Application-maintained sparse index

In cases where the number of duplicates of the atypical values is low, you can avoid the index on the Invoice_Status column if you create a new table that contains the Invoice_Status and the primary key (Invoice_Key) of the atypical occurrences. Figure 2-4 shows this technique for the Invoice example.

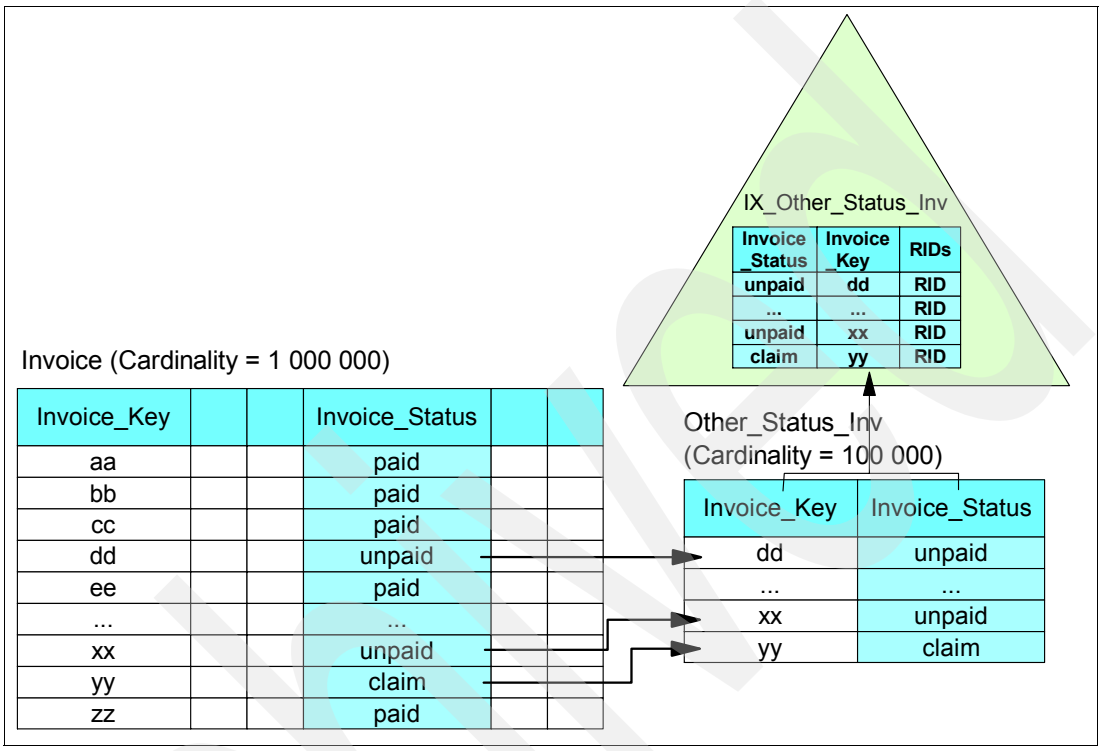


Figure 2-4 Application-managed sparse index

To retrieve the rows with a status different than “paid,” you have to join both tables on the primary key (Invoice_Key) and Invoice_Status columns. This implies that the index on the new table must include both columns.

The advantages of using this technique are:

- You avoid the nonunique index on Invoice. It will not be picked by the optimizer when a query is looking for “paid” invoices because a table space scan will perform better.
- Depending on the length of primary key plus status columns, you save disk space.

The index on the new table usually has fewer levels than the previous Invoice index.

However, there are disadvantages:

- Retrieving the atypical status requires a join (unless you are only interested in the Invoice_Key).
- The application must deal with the consistency between data in both tables when the status changes, or you can use a trigger to keep both tables in sync.

Another technique to avoid the index is to define one table per atypical status, including only the Invoice_Key column as shown in Figure 2-5 on page 83.

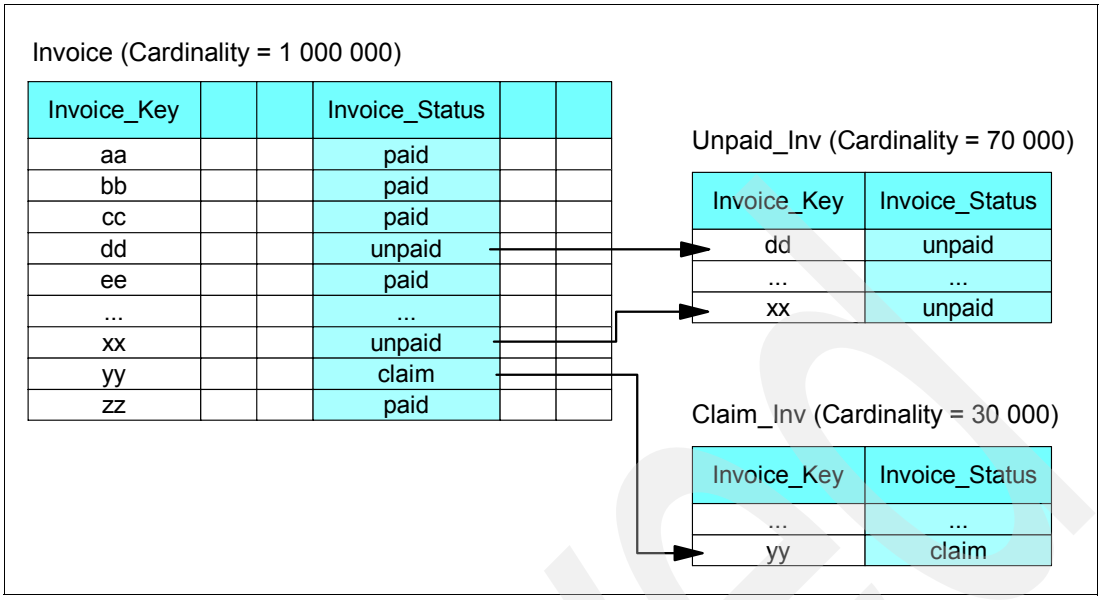


Figure 2-5 Application-managed sparse index: One table per status

You can only implement this technique if there are only a few atypical values. The advantages of implementing this technique are:

- Status tables only need to have the Invoice_Key column and an index on this column.
- You do not have to apply predicates on the status tables different from the join condition.
- Disk space is saved since you do not have to replicate the status column.
- Indexes in the new tables have fewer pages and levels.

However, you need to create or drop a table whenever you define or remove a status for an Invoice. You also need to be careful that a status update may require the row to move from one table to another. Again, the different tables can be kept in sync by using a trigger.

2.6.8 UNIQUE WHERE NOT NULL

You can define indexes that enforce uniqueness for all values, excluding nulls, using the UNIQUE WHERE NOT NULL clause. In a single-column index created using this clause, each not null value must be unique, but it allows many null occurrences. If this index includes several columns, you have a null entry if any of the columns contain the null value.

> **Note:** The optimizer will exclude NULLs from filter factor estimation for predicates that cannot be null, such as WHERE C1 = :HV, assuming that non-uniform distribution statistics have been collected on the column and nulls are within the top “n” most frequently occurring values.

2.6.9 NOT PADDED and PADDED indexes

When you include a VARCHAR type column in a PADDED index, DB2 stores that column in the index pages as a fixed length column with its maximum defined length. Both memory and disk space can be wasted if the average length of the data in that column is much smaller than the maximum length.

When you use the NOT PADDED keyword when you create (or alter) the index, VARCHAR columns are stored as variable length columns in the index. In this case, the VARCHAR

column in the index looks exactly the same as it looks in the table, a two byte length field followed by the actual data.

Using NOT PADDED indexes have a number of *advantages*:

- NOT PADDED indexes allow for true index-only access. If you have applications that today cannot take advantage of index-only access because they contain VARCHAR columns in the index, NOT PADDED indexes can allow for index-only access and may improve performance for those queries.
- NOT PADDED indexes may be smaller in size than their PADDED counterparts. Smaller indexes can mean fewer index pages to scan, and potentially fewer index levels.

The *disadvantage* of using NOT PADDED indexes is that they are more complex to process. DB2 has to read the length of the NOT PADDED column in the index in order to determine where the next column starts. With fixed length columns, this is much easier. Processing a NOT PADDED index with VARCHAR keys can require a significant amount of additional CPU time, just like VARCHAR columns in a data record.

NOT PADDED index performance is heavily dependent on the number and size of VARCHAR columns in the index key, because column comparisons are more expensive, and, also, the cardinality of the VARCHAR column, since lower cardinality columns contain fewer variable length keys.

Tip: A general rule is to use NOT PADDED indexes when the length is at least 18 bytes or when index-only access is required to avoid random I/O from a poorly clustered index.

So, the balancing act that you must perform is how many access paths can become index-only and how many index getpages can be saved, compared to the extra CPU required to evaluate variable length predicates in the index. See *DB2 UDB for z/OS Version 8 Performance Topics,* SG24-6465, for the performance implications.

2.6.10 Indexing long columns

In this section, we discuss a few techniques that you can use when dealing with long columns, and when indexing or retrieving information from those columns.

Indexing long VARCHAR columns

Indexing long VARCHAR columns usually presents a few challenges.

A VARCHAR column with a length greater than 255 (or greater than 127 bytes for a VARGRAPHIC) used to be treated as a long string, and prior to DB2 Version 8 a lot of restrictions applied to long strings and long string columns. Starting in DB2 Version 8, you can use these long strings in:

- An ORDER BY clause
- A GROUP BY clause
- A SELECT DISTINCT statement
- A UNION (without the ALL keyword - requiring a sort)
  provided that the sort key does not exceed 16,000 bytes
- A CREATE INDEX statement
  provided that the sum of the indexed columns does not exceed 2,000 bytes

Prior to DB2 Version 8, the maximum length for predicate operands is 255 bytes and for graphic strings is 254 bytes. In DB2 Version 8, the maximum length for predicates is increased to 32,704 bytes, matching the maximum defined size of a VARCHAR column.

Note that:

- The maximum length for a string to match on a LIKE predicate is still 4,000 bytes.
- The maximum length of the sum of the column lengths in the partitioning key is still 255 bytes.
- A string literal can be 32,704 bytes.

Indexing only part of a column

Even though DB2 has become a lot more flexible in indexing long fields, you still have to store these long fields in the index, and they take up a lot of extra space, or the field may still be too long for DB2 to index.

A technique to circumvent this situation consists of including an additional fixed length column in the table that contains the truncated contents of the variable length column, and then define the index on this new column.

The new column should be as short as possible to minimize the additional disk space required to store it in both the index and data pages. But on the other hand, the new column cardinality must be sufficiently high to reduce the number of duplicates in the index, and effectively represent the contents of the VARCHAR column.

Both requirements are in conflict, and you must balance them. You can analyze the substring uniqueness of the contents of the VARCHAR column to decide where to truncate the data.

To give you an idea about the number of characters to include in that extra column, you can repeatedly execute the SQL statement in Example 2-1 by varying n. The query returns the minimum, average, and maximum number of duplicates the index should have for certain length "*n*". Choose what is appropriate for your needs.

Example 2-1 Determining the number of duplicates per truncation length

```
SELECT MIN(NUM) AS MIN, AVG(NUM) AS AVG, MAX(NUM) AS MAX , 'LENGTH n'
  FROM ( SELECT SUBSTR(LONG_ALPHA_COL,1,n), COUNT(*) AS NUM
           FROM TABLE.EMP
          GROUP BY SUBSTR(LONG_ALPHA_COL,1,n)
       ) AS T1
```

By indexing only part of the column, the application, or user at the screen, must further eliminate rows that do not really apply, but just happen to start with the same character string, which may or may not be acceptable. Also applications that make updates to the table must make sure to maintain the extra column, or you can use a trigger for this purpose.

Matching information in alphanumeric columns

Consider a table that stores customer-related information (see Figure 2-6 on page 86). It has columns containing the customer's last name, first name, and the address for each customer. In general, these columns store lengthy alphanumeric strings, even if the columns are defined with a fixed length. Usually you have to specify in the SQL statements matching conditions on these columns to retrieve data from this table.

Customer

Cust_ID	Last_Name	First_Name	Address	...	...
11					
22	Steeghans	Jasper			
33	Steegmans	Andreas			
44	Steegmans	Jasper			
55					
66	Steens	Amélie			
77	STEEGMANS	ELINE			
88					
99					

IX on: Last_Name CHAR(40)
First_Name CHAR(20)

Figure 2-6 Index on Last_Name, First_Name

Even though you can define an index on these columns to access the data efficiently, you still have to be aware that:

- It is difficult to find the logically correct answer set, because of misspellings, upper case letters, unwanted blanks, and abbreviations in the data entry.
  In the Customer table, condition Last_Name = 'Steegmans' retrieves only customer 33 and 44, but the rows you are looking for are 22, 33, 44, and 77.
- In the condition Last_Name = 'Steegmans', you have to replace ' = ' by LIKE with wild card conditions to retrieve the rows. But then, the answer set using LIKE can be larger than the desired answer set, and you must include logic in your program to identify the rows you want to retrieve.
  Condition Last_Name LIKE 'Stee%' gets rows 22, 33, 44, and 66. Logic code must exclude Customer 66.
- Even with LIKE predicates, you can miss logically correct rows, such as Customer 77 in the example.

Furthermore, it is likely the index has a skewed distribution of duplicates and some values with an extremely high number of duplicates, for example, the value SMITH or JONES in Last_Name.

A technique to improve the usability of these types of indexes is to include a new column in the table that represents in some way the contents of the lengthy columns, typically an anagram, and then define the index on this new column. Other typical solutions use techniques like hashing or *soundex* (search of names and phrases based on phonetic similarity). The Text Extenders also use these types of techniques.

Figure 2-7 on page 87 shows how to implement this solution on the Last_Name, First_Name columns.

Customer

Cust_ID	Anagram	Last_Name	First_Name	Address	...	...
11						
22	STEEGHJS	Steeghans	Jasper			
33	STEEGMND	Steegmans	Andreas			
44	STEEGMJS	Steegmans	Jasper			
55						
66	STEENSML	Steens	Amélie			
77	STEEGMLN	STEEGMANS	ELINE			
88						
99						

IX on: Anagram CHAR(8)

Figure 2-7 Index on the anagram

Each anagram must uniquely identify each string, and each different string must be represented by a different anagram. You can convert a long alphanumeric string in an anagram by several methods, depending on your requirements and on the type of data:

- Vowel elimination
- Phonetic transformation
- Numeric code assignment

The routine that transforms the lengthy string in an anagram must be applied before you access the table. Now you can include matching predicates on the anagram column in the SQL statement. The transformation routine should also deal with misspellings and similar problems.

Other more typical techniques involve the use of a hash or a technique such as soundex. There are other techniques noted in the Text Extender.

Documents stored in long fields

When taking the previous discussion to the next level, how do we search for information stored in fields that store entire documents, for example, text or XML documents in a VARCHAR, CLOB, or DBCLOB column? Using a LIKE '%search-string%' is usually not a good solution.

For these situations, you may want to look at tools (or programs) that process these long strings, and create an index (often not a DB2 index) that is then used at retrieval time to extract the correct documents. The indexing of the documents can be done at INSERT time or afterwards. Doing the indexing at insert time has the advantage that the index information is immediately available for retrieval, but doing the indexing at insert time may affect your system throughput, because the indexing is now a synchronous process. The alternative is to flag changed rows (inserted, updated, or deleted rows) when the update operation takes place, and do the indexing asynchronously. This way, the update operation only experiences minimal impact (just flagging the row for later processing), but the data is not immediately available for retrieval. Pick the best strategy depending on the requirements of your environment.

Since we assume here that the data is stored in a column in a DB2 table, you can use tools, such as DB2 UDB for OS/390 and z/OS Net Search Extender, DB2 UDB Text Extender for

z/OS, or DB2 XML Extender for OS/390 and z/OS to index parts of the information in that "long column". These products also provide user-defined functions and/or stored procedures that can subsequently be used to retrieve information from the indexed columns. For more information about the DB2 Extenders™, see:

http://www.ibm.com/software/data/db2/extenders/

For more information about storing and retrieving XML data with DB2, see Chapter 7, "XML support" on page 177.

If you need to index data that is not just in DB2 tables, you may want to use a more general "indexing product", like WebSphere® Information Integrator OmniFind™ Edition. For more information, see:

http://www.ibm.com/software/data/integration/db2ii/editions_womnifind.html

2.7 Clustering index

One of the most important index-related decisions is which index is going to be the clustering index. In many designs, the primary key is often "automatically" being picked up as the clustering index. However, this is rarely the most appropriate index for clustering.

2.7.1 What is a clustering index?

The index that is defined as the clustering index defines the order in which the rows are stored (inserted) in the table. You define a clustering index by specifying the CLUSTER keyword on the CREATE/ALTER INDEX statement. DB2 tries to store the data rows in the data pages in the sequence of the clustering index. Since rows can be stored only in one sequence, you can define only one clustering index per table. However, it is not required to be unique, and duplicates are allowed.

Figure 2-8 on page 89 compares a clustering index and a non-clustering index.

Archived

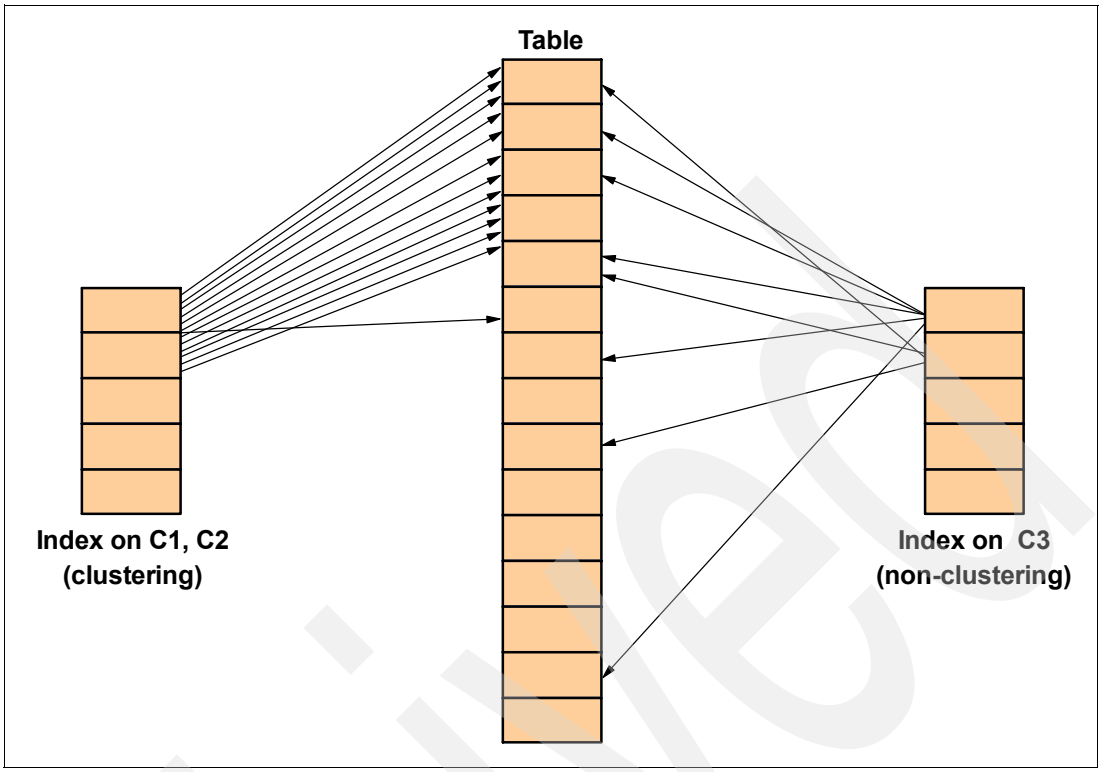


Figure 2-8 A clustering versus a non-clustering index

The more closely the ordering of the index entries matches the actual ordering of the rows on the data pages, the more clustered an index is. The extent to which this is the case is expressed by the *cluster ratio* (column CLUSTERRATIOF in the catalog tables SYSIBM.SYSINDEXES and SYSIBM.SYSINDEXSTATS). The information can be gathered by running RUNSTATS. The closer CLUSTERRATIOF is to 100%, the more closely the ordering of the index entries matches the actual ordering of the rows on the data pages. In the figure above, the clustering index on C1, C2 will have a very high cluster ratio. Only one row is not in its "optimal" position. The index on C3, on the other hand, will not have a high cluster ratio.

If the table is in an index-controlled partitioned table space, the partitioning index is also the clustering index. In a table-controlled partitioned table space, any index (a partitioning index or a secondary index) can be defined as the clustering index.

When you define an index using the CLUSTER keyword, it becomes the *explicit clustering index*. When no explicit clustering is defined, DB2 uses an *implicit clustering index*. The implicit clustering index is the first index on the index chain for the table in the DBD. Since defining the correct clustering sequence for your data is critical to the success of many applications, you do not want to rely on an implicit clustering index. When you need to do some maintenance on the table or index, there is no guarantee that the same index will end up as the clustering index, unless you explicitly define one with the CLUSTER keyword.

Tip: Define an explicit clustering index. Clustering is very important, so you want to make sure that you control which index is the clustering index, not DB2.

2.7.2 Choosing the clustering index

When retrieving only a single row from a table using a predicate on which a unique index is defined, clustering is irrelevant, unless the single SQL is repeated many times from another input, such as an input file, or another cursor.

In general, a clustering index optimizes the performance of an SQL statement that accesses subsets of a table in the sequence of that clustering index, such as:

- Range predicates (BETWEEN, > , <) on the index columns
- Predicates against low to medium cardinality columns, therefore, many rows qualify
- Enables prefetch (both sequential and dynamic)

When defining the clustering index, the most important criteria are:

- The clustering index must reflect the sequence needed for sequential (batch) processes that access the table, especially when they scan most data for medium or large size tables, in order to reduce costs and elapsed time of the process.
- Most online applications are not affected by the choice of clustering index, because they retrieve small sets of data, except for applications that access a range of rows in a table. An example of an application that accesses a range of rows in a table is an online browsing transaction that needs to avoid sorting or list prefetch I/O.
- However, always consider the impact on the other indexes when selecting one as the clustering index. Selecting one to become the clustering index is likely to make the other indexes random indexes. This can be minimized if all indexes contain a common key column that matches the clustering.

If your processes require a lot of join activity:

- Choosing a foreign key as the clustering index facilitates efficient join processing.
- Choosing the join columns as the clustering index columns facilitates efficient join processing.

  See “Clustering and the impact on join access” on page 95 for more details.

Clustering index selection strategy

As for designing general indexes, it is difficult to come up with an all encompassing clustering strategy, but this section provides you with a good basis for making a sound decision about defining the clustering sequence, and, therefore, the clustering index for a table.

1. Build CRUD for each table/process, and perform general index design as described in “General indexing strategy” on page 75.
   This should result in a list of potential indexes and clustering candidates.
2. Analyze each process (or use an 80/20 rule) and ask yourself:

   “*Can I afford this process to be a random process?*”

   In other words:

   – Do I have enough CPU and I/O bandwidth to handle this process as a random process? To get a rough estimate of the cost of a process, see “Determining the duration of your process” on page 113.

   – When this process does random processing throughout my data, will I need to recluster my data later? And if yes, can I afford to recluster my data, for example, by running REORG? For a more elaborate discussion on the impact of reorganizing your data, see “Can I recluster afterwards?” on page 92.

If the answer to the question is:

a. Yes. In that case, you do not need to worry about that process too much, and there is no real need to have that process dictate the data's clustering sequence.

   If the answer is yes for all processes, pick that index as the clustering index where applications benefit the *most* from having that index as the clustering index.

b. No. In that case, you need to ask yourself:

   ** "Can I make it work if I could make the process work sequentially?"*

   i. Yes. In that case, you have a good candidate process to determine the clustering order.

      - If you only have one clustering candidate after evaluating all processes, you can use that one as the clustering sequence.
      - If you have multiple clustering candidates, continue.

   ii. No. In that case, can you force the process to cluster its work? This is described in step 3.

   Other questions you need to ask are:

   * "*Can I make it work if I partition?*" (See "Can partitioning help?" on page 92.)

   * "*Can I make it work if I use striping?*" (See "Can striping help?" on page 94.)

   * "*Can I make it work if I use larger buffer pools?*"

3. Can I force the process to cluster its work?

   Since clustering the data in the order of the process is not working for this process, we can try to "force" the process to cluster its work. This discussion is also important once you have decided on the clustering sequence, and you are left with a number of processes that do not fit the clustering sequence that you have picked.

   a. Can I sort the input in another way so it can execute in a different order?

      If the data is the result from another process, it can be sufficient to write the data to a sequential file, sort it in the clustering sequence of the table that will be processed next, and have the next process use the sorted (in the clustering sequence of the new process) flat file. Using this intermediate step can save a lot of time because the data can now be processed sequentially instead of randomly.

   b. Is the process a SELECT process?

      If yes, you can "force clustering" by making the process index-only access. You can add additional columns to the index so the process becomes index-only. Because the entries in the index are always perfectly clustered, you can use a sequential scan of the index to process everything in index key order.

      The question then becomes, can I afford adding columns to the index to make it index-only. It will make the index much bigger, and will impact more UPDATE statements, since any column in the index needs to be maintained when the corresponding data column in the table is updated.

   c. Is the process an INSERT process?

      If yes, you can "force clustering" of newly inserted rows at the end of the object by using zero free space, and REORG later to recluster (see also "Can I recluster afterwards?" on page 92). Inserting at the end only works for the data. You cannot force index keys at the end, unless the index keys are ever increasing keys.

      Another alternative may be a LOAD RESUME SHRLEVEL(CHANGE).

Archived

d. Is the process a DELETE process?

   If yes, there is not really a lot you can do, other than try to stage the deletes, and not do them immediately.

   Another option may be to do the delete with a REORG DISCARD SHRLEVEL(CHANGE). This allows for a true online REORG, and in the process of reorganizing, rows can be deleted (and written to a discard file, if requested). Since you often need to REORG after deleting a large number of rows, for example, to reclaim some space, it is a good idea to combine the delete (discard) and REORG.

e. Is the process an UPDATE process?

   If yes, then you really do not have a lot of flexibility. You may have to go back to the users to see if you can "stage" the updates.

For more information about techniques to do massive change operations, see "Massive insert, update, and delete" on page 264.

Can I recluster afterwards?

If a process makes a large number of random (meaning not according to the clustering sequence of the table) updates (I, U, and D) to a table, you may have to REORG the table and/or the indexes.

To see if reclustering is an achievable goal, you need to consider:

- The *REORG frequency* that will be required. Do we need to REORG every week or twice a year, for example?
- What is the *scope* of the REORG? Will we need to REORG:
  - The entire table space (and all its indexes as a consequence)?
  - A partition, or a range of partitions, and what will be the impact on any of the nonpartitioned indexes if the object is a partitioned table?
  - Is a REORG of the indexes sufficient? Maybe we do not need to REORG the table space. Note that even if the data is processed in the table's clustering sequence, you still may have to reorganize all the nonclustered indexes. If you process the data in clustering order, that is likely to reduce the amount of disorganization that is created in the table and the clustering index, but is very likely to leave your nonclustered indexes in a more disorganized state than before the process was run, and a reorganization of those indexes may be required. Make sure to provide sufficient free space to accommodate for future inserts when you run REORG.
  - Maybe a REBUILD of some nonpartitioned indexes (NPIs) is required? Sometimes the NPIs are dropped before a process is run, because the updates in the NPI are random and would only make the process run longer, and an REORG of the index would have been required in any case after the process was run.

Can partitioning help?

When you are dealing with a lot of rows, a massive update, or retrieval operations, you should always ask yourself, will things work better if I partition the table? Let us first look at some positive aspects of partitioning.

- For INSERT processes
  - Can I partition the data in such a way that INSERT operations are concentrated in a single partition? For example, if you need to insert last month's data into your warehouse every month, partitioning by month allows you to always insert into the last (new) partition. When there is a requirement to maintain historical data, such as with a data warehouse, then partitioning by date ensures that the most commonly accessed

data is contained within the most recent partitions, which benefits query performance and minimizes maintenance operations on older partitions.

  - Can I partition in such a way that INSERTs are spread across multiple parts? This may be an attractive option if the inserts need to be spread out because a single object or partition may not be able to handle them.
  - If spreading data across partitions, then ensure either clustering within a partition is maintained, or insert at end processing is employed. Inserts that use a purely random key benefit neither insert nor retrieval performance.

- For DELETE processes
  - Is it possible to use LOAD REPLACE of a partition to delete old data? LOAD REPLACE of a partition of a partitioned table space can be very fast (especially when no nonpartitioned indexes are present).
  - Is it possible to run REORG SHRLEVEL(CHANGE) with the DISCARD option?
- For UPDATE processes
  - An UPDATE of a partitioning key, when the key moves to a different partition, is processed as a DELETE and INSERT operation.
    Note that DB2 no longer drains partitions when moving a row from one partition to another.
- When you partition a table, you reduce the number of index levels in all partitioned indexes, since each partition of a partitioned index has its own B-tree index. Reducing the number of index levels can reduce the number of getpages in the index. This can increase the probability of the index being chosen by the optimizer.
- Can partitioning make things worse?

Not only should you ask yourself, “How much better will partitioning make things?” but also “Which things might get worse if I partition the table?” Some things to consider are:

- Nonpartitioned indexes
  - An NPI does not really make things worse, because if you do not partition, an index that looks identical to an NPI, will still be there.
  - You can also still opt to reorganize the entire table space. So it would behave in a similar way to a nonpartitioned table.
  - However, you should keep in mind that if you need to create an NPI, you lose some of the advantages related to partitioning, such as full partition independence.
- Data-partitioned secondary indexes (DPSIs)
  - DPSIs work really well with utilities and affinity routing in data sharing.
  - Always make sure that DB2 can do partition pruning when DPSIs are involved.
  - Remember that a DPSI cannot be unique. Therefore, if uniqueness is a requirement, a DPSI is not an option.
- The space management in a segmented table space is better than in a partitioned table.
  - A segmented table space has better free space management. There are more bits in the space map page that indicate where and how much free space is available.
  - Mass delete is only available for segmented table spaces.
- Updating the partitioning key is done using a DELETE/INSERT operation.

- More partitions mean more open data sets. This does not just apply to the data partitions, but also to all partitioned indexes. Each data partition and each part of a partitioned index have their own VSAM data set.
  - When using partitioned tables, make sure that the maximum number of DB2 open data sets that is allowed in the system (DSMAX DSNZPARM) is not exceeded.
  - More partitions mean more data sets, and this also means that DB2 needs to switch more data sets when using online REORG. Having to switch more partitions and partitioned indexes elongates the elapsed time of your REORG job's switch phase.

Can striping help?

When trying to eliminate I/O bottlenecks, it is always good to see whether or not striping can help.

From an implementation point of view, striping is totally transparent to your applications and DB2, maybe even somewhat too transparent.

Striping is easier to implement than partitioning, especially when an existing application needs to be changed. When using partitioning, the table needs to be unloaded, dropped, and recreated as a partitioned table, and the data reloaded. Implementing striping does not require any changes in DB2.

VSAM I/O striping is defined when the VSAM data set is created. Data sets need to be SMS-managed and must be defined with the Extended Format (EF) attribute (and your disks must also support EF data sets). VSAM I/O striping can be used in a DB2 environment for active logs, as well as any type of table space, or index.

If a data set can use three stripes, you have the potential to triple your I/O throughput, which, of course, is not the case in real life, because the work is rarely completely independent, so it cannot fully exploit the stripes to the maximum of its abilities, but you still get a great I/O bandwidth boost.

VSAM data striping can be a good solution for nonpartitioned table spaces or for table spaces where partitioning cannot be implemented (for instance, simple or segmented table spaces); also nonpartitioning indexes can benefit from striping. Less obvious candidates are the workfile table spaces; during the merge phase of a sort, all sort work files are merged to one logical workfile, which can result in contention for that merged workfile data set.

One difference between VSAM data striping and partitioning is that the sequential prefetch I/O quantity (number of pages returned by one prefetch I/O) is reduced by a factor equal to the degree of striping; this is also true for deferred writes.

The DB2 prefetch is still 32 pages, but, internally the I/O will be reduced in scope to account for the allocated stripe. Whether this is an advantage or a disadvantage depends on the workload; a table space scan on a 16-stripe table space data set could definitely be slower than the same scan using parallelism on a 16-partition table space, due to the internal reduction of the prefetch quantity I/O from 32 to 2. The same query on a nonpartitioned table space will run faster for the striped table space data set than for the non-striped one, because the 16 parallel I/O streams only prefetch two pages each.

When using striping on an NPI, make sure to define the index using an appropriate PIECESIZE. Each piece is a separate VSAM data set that can be striped.

Also, be aware that DB2 query parallelism is decided by the optimizer, and VSAM I/O striping is independent of the optimizer and/or workload. A lot of the DB2 parallelism is driven by partitioned tables. Therefore, you may see parallelism with a partitioned table, but no DB2 parallelism when using striping.

2.7.3 Clustering and the impact on asynchronous I/O

DB2 uses asynchronous I/O to read pages from disk whenever it detects a sequential pattern in data access, that is, you are scanning a range of data in the sequence that is stored on the pages.

DB2 tries to keep data stored in the clustering sequence. All accesses that follow the clustering sequence (that is, enforced by the clustering index) may benefit from asynchronous I/O.

Usually, batch processes deal with massive sequential scans of data. If you choose the clustering index in a way that reflects these sequential accesses, your batch processes will profit from asynchronous I/O and gain major performance improvements by drastically reducing elapsed time.

2.7.4 Clustering and the impact on other indexes

The index that you pick as the clustering index not only affects the way the data will be inserted into the table, but also the way the index key entries will be inserted into the other indexes.

For example, table TAB01 has a number of columns, including COLA and COLB. If index IX1 on column COLA is the clustering index, and we always insert higher COLA values into the TAB01, new entries will be inserted at the end of IX1, and at the end of the table, but it is possible that the new key entries for IX2 on COLB will be in a random order. So if TAB01 is expected to have a heavy insert rate, inserting at the end may alleviate the insert problem in the table, but may very well move the problem to index IX2 that is now suffering from massive synchronous I/O because of all the randomly inserted key entries.

Making an index the clustering index also determines the access pattern (sequential or random) for the other indexes. For this reason, it can be beneficial to cluster by low cardinality columns, which can be shared among many indexes, provided these columns are used in query predicates. This has the effect of minimizing the random nature of the non-clustering indexes.

2.7.5 Clustering and the impact on synchronous I/O

When you access data via a nonclustered index, the data is likely to be random in nature, either scattered over many data pages, or not retrieved in sequence, or both. To reduce random I/O, avoid repeated access to the same data pages, and retrieve the data in a sequential manner, DB2 is likely to use list prefetch. Another advantage of accessing data in a consistently sequential manner is that deadlocks do not occur.

Clustering is the most powerful weapon in reducing random I/O, since data retrieved via the clustering index will be densely stored on consecutive data pages. While retrieval of 100 nonclustered data rows may require as many as 100 I/Os, retrieving 100 clustered rows may require as few as one I/O. Note, it is the clustering which determines where the data is stored. Therefore, accessing the same 100 clustered rows via a nonclustered index will still find the rows on the one data page.

2.7.6 Clustering and the impact on join access

Clustering joined tables in the same order can have a significant impact on performance. This, by the way, also applies to RI enforcement, especially for cascading deletes or SET NULL operations. For parent-child table relationships, it is easy to cluster tables by the same columns because the same leading key column is shared from the parent to child table.

What if each joined table has a different primary key? In the following example, we join three tables. In all tables, the low cardinality column C1 is present. In Figure 2-9, only the first table is clustered according to C1. The other tables are clustered and joined according to their unique indexes CX and CY. The first table is accessed using the clustering C1 index, and, even though the access to the other tables is using a 100% clustered index, it results in a significant number of random I/O operations on the inner tables of the join.

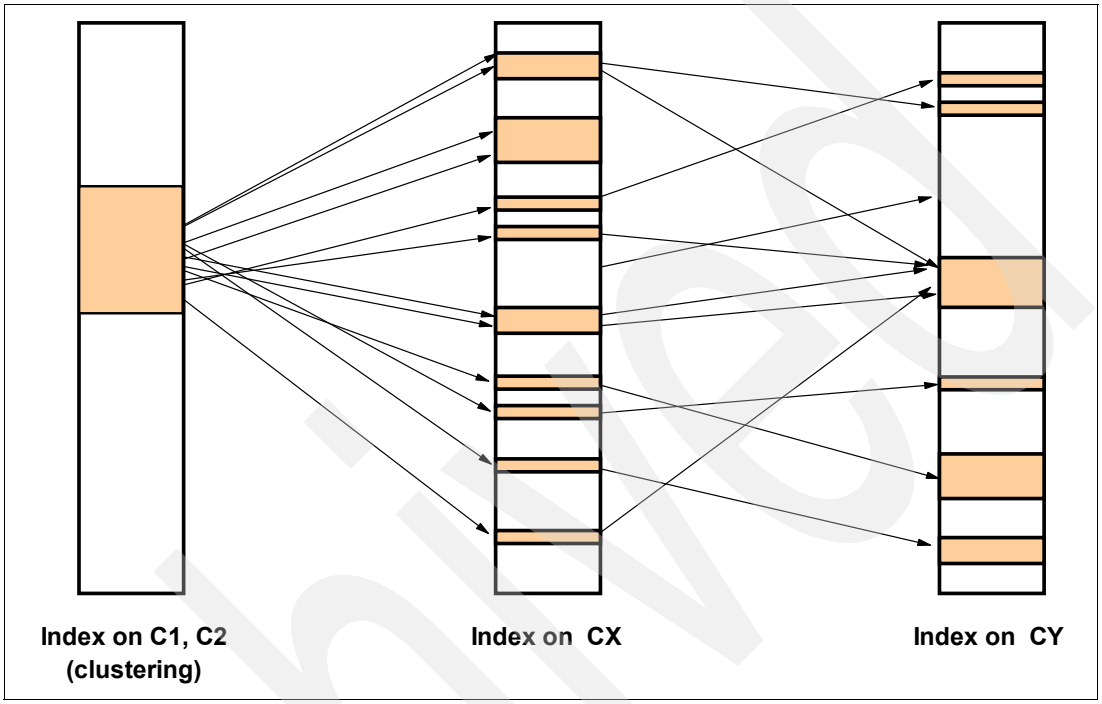


Figure 2-9 Joined tables in a different clustering sequence

Figure 2-10 on page 97 shows the situation after all tables were changed to cluster the data based on column C1. The original (unique) columns CX and CY were appended to the leading column C1. This results in a much better concentration of the access inside the inner tables of the join, a better exploitation of sequential data access, an improved buffer hit ratio, and allows for the use of index look-aside.

Archived

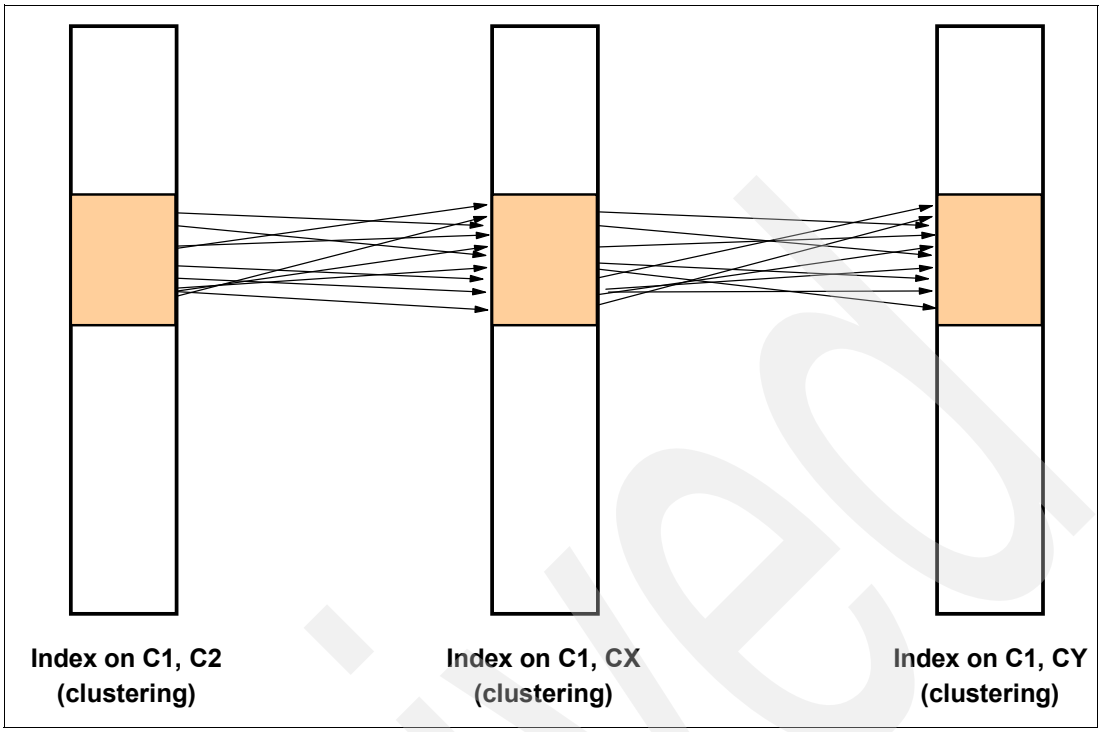


Figure 2-10 All joined tables clustered the same

This example demonstrates how to minimize the impact of random I/O for the join to the inner table. Another alternative or complementary solution is to create indexes that include the join column after local predicates. For example, on the first table, an index on C1, CX (assuming from this example that there is an equal predicate on C1 for the first table) would allow filtering to occur on C1, and would guarantee that the join to the second table would be in CX order. Since CX is unique on table two in this example, it is not possible to provide ordering from the join from table two to table three, and, therefore, the solution provided in this section will minimize random I/O.

2.7.7 Changing a clustering index

Since DB2 Version 8, you have the capability to change the clustering index with an ALTER INDEX statement. Since only one index can be explicitly defined as a clustering index, you must follow these steps to change the clustering:

1. ALTER INDEX ixname1 NOT CLUSTER on the current clustering index.

   Clustering continues to be done according to this index (ixname1) until a new clustering index is explicitly defined.

2. ALTER INDEX ixname2 CLUSTER on the index you wish to be the new clustering index.

   New rows will be clustered according to the new clustering index. Old rows will remain in their current location.

3. REORG the table space to rearrange the rows in the new clustering index order.

   All existing rows will be rearranged in the new clustering sequence. Any new rows will be inserted with the new clustering sequence.

If no explicit clustering index is specified for a table, the DB2 Version 8 REORG utility recognizes the first index created on each table as the implicit clustering index when ordering

data rows. If the explicit clustering for a table is removed (changed to NOT CLUSTER), that index is still used as the implicit clustering index until a new explicit clustering index is chosen.

2.8 Indexing partitioned tables

In this section, we discuss issues that are specific to indexes on partitioned tables. The following characteristics apply to indexes on partitioned table spaces.

- *Partitioning*
  The columns in the index are the same as (are in the same order and have the same collating sequence) or start with the same columns as those specified in the PARTITION BY clause of the CREATE TABLE statement for table-controlled partitioned tables, or on the CREATE INDEX statement of the partitioning and clustering index for index-controlled partitioned tables. A partitioning index can have a superset of the partitioning columns, that is, it can contain all the partitioning columns, plus additional columns.
- *Nonpartitioning or secondary index*
  Any index where the columns do not coincide with the partitioning columns of the table.
- *Partitioned*
  The index is made up of multiple physical partitions (one per data partition), not just index pieces.
- *Nonpartitioned*
  The index is a single physical data set, or multiple pieces (but not a one to one match with the number of data partitions).
- *Logical index partition*
  The set of all keys in a nonpartitioned index that references the same data partition.
- *Physical index partition*
  The set of keys in a partitioned index that references a single data partition.

2.8.1 Partitioned partitioning index

Before DB2 Version 8, all partitioned table spaces are so-called index-controlled partitioned table spaces. This means that the CREATE INDEX statement controls which rows go into which data partition. When using index-controlled partitioning, a (partitioned) partitioning index is mandatory, because the index defines the boundaries of each partition by establishing the ranges of data each partition stores. On an index-controlled partitioned table, the partitioned partitioning index must also be defined as the clustering index.

All of this changes when using table-controlled partitioned tables. In that case, the CREATE TABLE statement controls the partition boundaries, and a partitioned partitioning index is no longer mandatory. In addition, any index on a table-controlled partitioned table can be defined as the clustering index: partitioned, nonpartitioned, partitioning, and secondary index.

Another difference between index-controlled and table-controlled partitioned tables is that the boundary of the last partition is always enforced for table-controlled partitioned tables, irrespective of whether they are defined as DSSIZE or LARGE.

2.8.2 Nonpartitioned indexes

When using index-controlled partitioned tables, only one index (the partitioning index) can be partitioned. Any other index is a nonpartitioned index (NPI). It consists of a single data set or optionally by a number of index piece data sets (determined by the PIECESIZE) parameter.

NPI advantages

- All RIDs for a single key are together. An index entry consists of the key value, followed by one (for a unique index), or more RIDs (for a nonunique index). In an NPI, any index value can only occur once in the index. This is not the case for Data-Partitioned Secondary Indexes. When using a data-partitioned secondary index (DPSI), a key value can occur once for every partition, because each data partition has its own DPSI partition that contains a complete B-tree index structure with all the keys from that partition. Therefore, if you need to retrieve all rows with a matching key value, you only have to perform a lookup into the NPI index structure once to find the key and the RIDs, where with a DPSI, you potentially have to perform an index lookup for each DPSI index partition.
- As mentioned before, a nonpartitioned index can consist of many “pieces”. Each piece is a VSAM data set. The PIECESIZE option on the CREATE or ALTER INDEX statement allows you to specify the maximum size of each nonpartitioned index data set piece. The lower this value, the greater the number of data sets you get for your nonpartitioned index. You can spread the nonpartitioned index data sets across multiple I/O paths, improving the access performance by reducing disk contention. Processes running in parallel accessing the nonpartitioned index may benefit from improved I/O performance.

  The optimizer understands partitioning, but not PIECESIZE, PAV, and I/O striping. With the advent of logical volumes and PAV, these disk functions are preferable to defining and distributing pieces.

NPI disadvantages

NPIs are usually a problem when dealing with utilities. Partition independence works really well, except when an NPI is involved.

- The sheer size of NPIs over very large partitioned tables makes their management as a single large object difficult. RUNSTATS, REORGs, REBUILDs, and so on take longer clock time than if they can be run on smaller object parts in parallel.
- An NPI has more index levels than a partitioned index because the NPI contains the keys of all partitions, where an index part of a partitioned index only contains the keys of the corresponding data partition.
- Although you can take image copies at the piece size level of an index, you always need to recover the entire NPI. Recovery from a media failure on a nonpartitioned index can only be done at the entire index level. No piece-level rebuild or recovery can be done for a nonpartitioned index.
- Partition-level operations become less clean if there are nonpartitioned indexes present. For example, to erase the data of a partition, you normally LOAD that partition with an empty input file. This operation quickly “resets” the data partition as well as the partitioned index, but also entails removing key entries from the NPIs that reference the partition being erased. For each row being removed from that partition, DB2 has to look up the key entry for that row in the nonpartitioned index and delete it.
- When nonpartitioned indexes exist on the table space, a BUILD2 phase is performed during online REORG of a partition. This phase uses the shadow index for the index's logical partition to correct RID values in the NPI. During this phase, the utility takes exclusive control of the logical partition. This blocks queries that are not partition-restrictive from operating.
- You can do a LOAD REPLACE for a partition in a partitioned table and rebuild the corresponding part of the partitioned indexes, but when you get to the NPI, things become more complicated. Because the NPI contains keys from all partitions, a LOAD REPLACE of one means that you need to find all the index entries that belong to that logical partition that you replaced, and replace those index entries with the new ones based on the rows that were just loaded. Finding the individual entries is a CPU-intensive and time-intensive

process, much slower than building a part of a partitioned index. It may get worse when multiple LOAD PART jobs run concurrently. In that case, they contend on nonpartitioned indexes because keys of all parts are interleaved. In addition, during a LOAD PART job, key processing against nonpartitioned indexes follows insert logic (a row at a time), which is slower than append logic (a page at a time).

- In a data sharing environment, some customers find benefit from isolating the work on certain members to certain partitions. Such affinity-routing eliminates intersystem read-write interest on physical partitions, thereby reducing data sharing overhead. Affinity routing does not alleviate contention on nonpartitioned indexes, since keys that belong to different data partitions are spread throughout the nonpartitioned index.

2.8.3 Data-partitioned secondary indexes

A Data-Partitioned Secondary Index (DPSI) is a secondary index that is made up of multiple physical partitions (that match the partitioning scheme of the table). It is an index which is partitioned based on data rows. However, the index contains different columns than the partitioning columns or in a different order or collating sequence than those which partition the table. Each part of a DPSI is a complete index B-tree structure. DPSIs must allow duplicates, and, therefore, must not be unique. DPSIs were introduced in DB2 Version 8.

DB2 now has the ability to physically partition secondary indexes. There are as many index partitions in the data-partitioned secondary index as table space partitions. The index keys in partition 'n' of the DPSI reference only the corresponding data rows in partition 'n' of the table space.

In Figure 2-11 on page 101, we see a partitioned table. The table is partitioned on account_num, in ascending order. The index at the bottom, PI_1, is a partitioning index, because it has the same leading columns and collating sequence as the partitioning column. The second index, DPSI_1, is a data-partitioned secondary index. It is a secondary index because its leading column is not the same as the partitioning column, and it is partitioned, because it contains as many physical partitions as there are data partitions.

Also, notice that the data is clustered on the date of the last activity (last_activity_dt). When using table-controlled partitioning, you can cluster the data within each partition in any column order. It is no longer mandatory that the partitioning columns are also the clustering columns, which is the case for index-controlled partitioning.

Archived

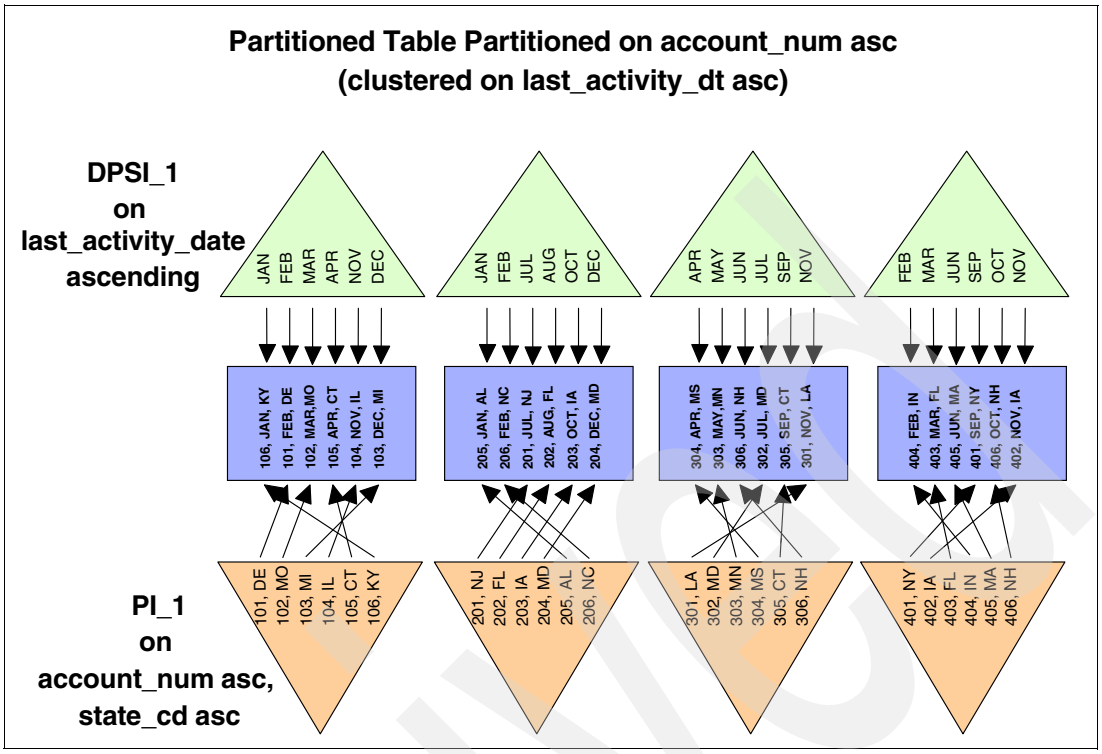


Figure 2-11 Data-partitioned secondary index

DPSI advantages

- DPSIs solve *all* the utility issues with NPIs that were discussed earlier. From a utility point of view, DPSIs behave pretty much in the same way as a partitioning index, and allow you to achieve true partition independence.
- DPSIs also allow for better affinity routing in a data sharing environment. If only rows within a single or a set of partitions are accessed by a certain member, the DPSI will not need to become GBP-dependent.

DPSI disadvantages

If your query does not have a predicate on the partitioning column, DB2 needs to look into each part of the DPSI to check for qualifying index entries. However, if your query also specifies partitioning columns, DB2 can use this information to limit the number of DPSI parts that need to be probed.

2.8.4 Page range screening

Although it is really not just related to indexing, page range screening plays a very important role in determining the way to design DPSI indexes, and is for that reason discussed here.

Page range screening or *limited partition scan* is the removal from consideration of inapplicable partitions. A limited partition scan can be determined at bind time or at run time. Sometimes, the term *partition pruning* is also used, especially when related to DPSIs.

Page range screening does not only apply to DPSIs.

A query must provide a predicate on the first key column of the partitioning index. Since DB2 Version 8, DB2 uses all leading columns of the partitioning key to determine the screening. The rules for using a partitioning column to limit partitions are the same as the rules for

determining a matching column on an index. If a partitioning column that is specified in a predicate is a candidate to be a matching column, that column can limit partitions. A limited partition scan can be combined with other access methods.

As mentioned above, DB2 does page range screening at bind time when literals are used, or at run time when host variables or parameter markers are used. Unlike in Version 7, the partition pruning at run time does NOT require the use of REOPT(VARS).

Page range screening shows up as PAGE_RANGE=Y in the PLAN_TABLE.

Page range screening for NPIs

When using an NPI, and a predicate exists on the leading partitioning columns, DB2 uses that information to limit the number of data partitions that need to be accessed. See Figure 2-12.

Page range screening can be thought of as an additional index screening predicate, without the column having to be in the index.

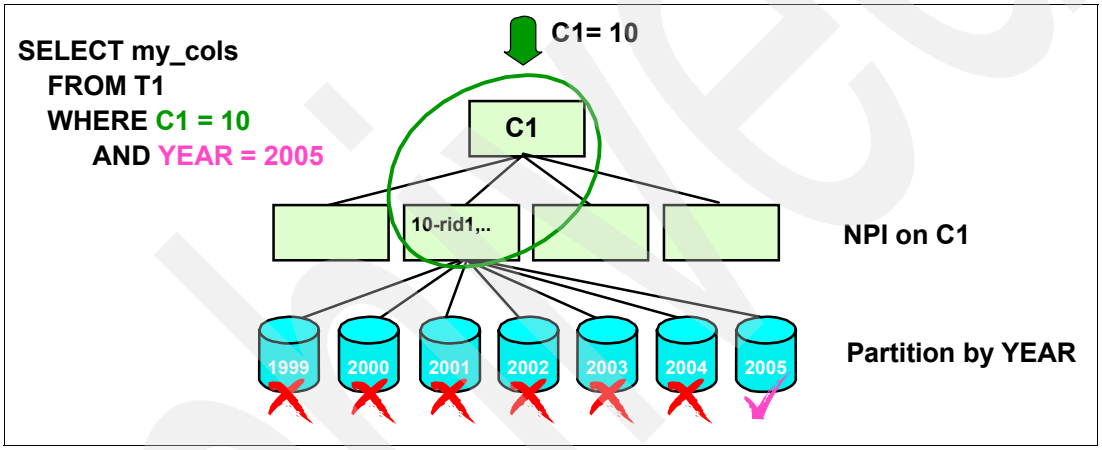


Figure 2-12 Page range screening using an NPI

Page range screening for DPSIs

The same applies to DPSIs. Page range screening in DPSIs is extremely important because it limits the number of parts of the DPSI that DB2 needs to probe to find qualifying keys.

An example is shown in Figure 2-13 on page 103. Page range screening can be applied when DPSI index access occurs, and a predicate on the leading partitioning columns is present in the query. Again, the predicates on the partitioning columns that do the page range screening (YEAR=2005 in the example below) do not have to be part of the index. Page range screening in a DPSI has the same effect as an extra matching index column without being part of the index; one from the page range screening column YEAR, and one matching column from the index column C1.

In this example, only a single DPSI part needs to be probed for C=10 entries instead of all seven. This illustrates the importance of having predicates on leading partitioning key columns to boost the performance of queries using DPSI.

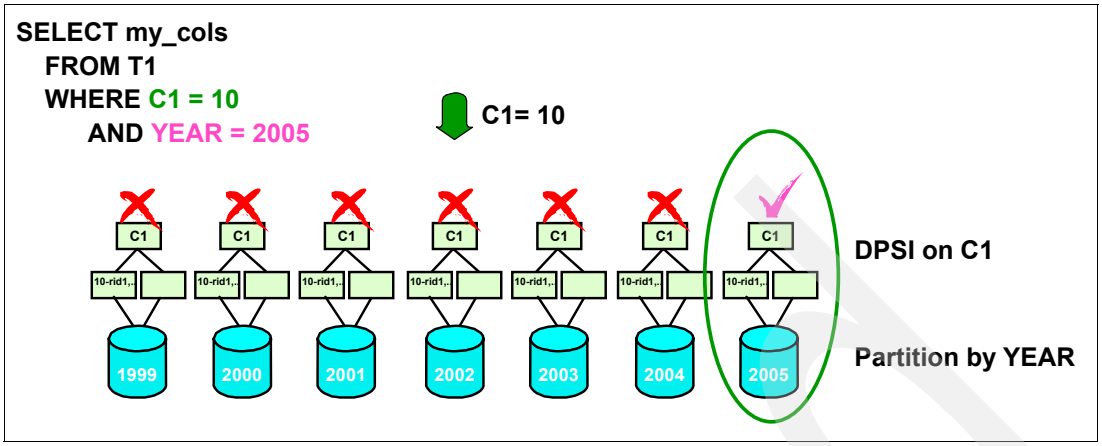


Figure 2-13 Page range screening using a DPSI

Page range screening for join predicates

Join predicates are not eligible for page range screening. In the following example:

```
SELECT my_cols
  FROM T1 INNER JOIN T2
    ON    T1.C1 = T2.PARTIT_COL
      AND T2.C2 = T2.DPSI_COL
```

"T1.C1 = T2.PART_COL" is not eligible for page range screening.

Page range screening and predicate transitive closure

On the other hand, predicates, generated by predicate transitive closure, are eligible for page range screening. For example:

```
SELECT my_cols
  FROM T1 INNER JOIN T2
    ON    T1.C1 = T2.PARTIT_COL
      AND T2.C2 = T2.DPSI_COL
  WHERE T1.C1 = ?
```

For this query, predicate transitive closure will generate an additional predicate "T2.PARTIT_COL=?". DB2 can derive this extra predicate, because if T1.C1=? and T1.C1=T2.PARTIT_COL, then T2.PARTIT_COL=?. So, the query is rewritten by DB2 as:

```
SELECT my_cols
  FROM T1 INNER JOIN T2
    ON    T1.C1 = T2.PARTIT_COL
      AND T2.C2 = T2.DPSI_COL
  WHERE T1.C1 = ?
    AND T2.PARTIT_COL = ?
```

When DB2 is now accessing table T2, this additional predicate can be used to filter the number of partitions to look at, for example, to eliminate parts of the DPSI when evaluating predicates on T2.

2.8.5 Data-partitioned secondary indexes at work

Let us now look at an example where DPSIs can make a difference. An application needs to handle banking transactions (deposits into, withdrawals from, and transfers) related to all accounts. There are a large number of transactions per day (inserts into the table) and

inquiries where people want to see the transactions they did with the bank for their account for a certain time period (retrieval by account number and timestamp).

Let us look at a number of design options.

Partitioning or not

Partitioning provides us with the following advantages in this case:

- We spread the inserts across multiple (97) partitions, which should allow us to sustain a much higher insert rate.
- We use partitioned indexes; they will be smaller. Each index tree will have fewer leaf pages, and even more important, fewer index levels.
- Since the indexes that we use are partitioned, we can take maximum advantage of partition independence and run utilities in parallel against different partitions. We will also avoid the BUILD2 phase for REORGs at the partition level, as well as the slow process of updating the NPI during LOAD or REBUILD of a partition.

Partitioning column

To spread the number of inserts, we use a partitioned table. Account numbers have the following structure xxx-xxxxxxx-yy, where yy are check digits (based on a modulo 97 division to verify that the account number is correct). We can use the yy check digit as the partition number. This means that we will have 97 partitions. Using the check digits, accounts will be more or less equally spread over the 97 partitions.

```
CREATE TABLE ACCT_TRANS
   ...
    PARTITION BY (CHECK_DIGITS)
```

Partitioning index

Since this is a table-based partitioning table, a partitioning index is not required. Since we are not interested in querying on CHECK_DIGITS, there is no need to build this index.

Partitioned indexes

Since we are very much aware of the impact of using an NPI on our ability to run utilities at the partition level, we try to design without having to use an NPI, and use only partitioned indexes.

In order to get good query performance when using a DPSI, we need to make sure that the partitioning columns or at least the leading partitioning columns are specified as a predicate in each SELECT statement. In our design, the partitioning is on CHECK_DIGITS. Therefore, each query should contain a CHECK_DIGITS = :hv-check-digits predicate. The application code can easily deliver the value for the :hv-check-digits, because CHECK_DIGITS are the last two digits of the ACCOUNT_NUM column, which is provided as an input field for every retrieval request. So each query will contain at least:

```
WHERE ACCOUNT_NUM = :hv-account-num
  AND CHECK_DIGITS = :hv-check-digits
```

Since all access is via account number, and since we can always derive the check digits from the account number, DB2 is always able to do page range screening (partition pruning) and limit the access to a single part of the DPSI (provided the CHECK_DIGITS = :hv-check-digits predicate is present).

Note that there is no need for an index on the CHECK_DIGITS column to allow for page range screening to occur.

Tip: To enable page range screening, use partitioning columns that are either naturally occurring columns in the table, or use a column whose value can be derived from an existing column of which the value is always known at retrieval time.

For example, do not use last the two digits of a timestamp to insert randomly into your different partitions to spread the inserts. It will be impossible to use that value at retrieval time to enable partition pruning.

Clustering order

Now we evaluate different alternatives. Remember that with a table-based partitioning table, any index can be the clustering index.

Option one: Cluster by account number and transaction timestamp

Since we want to retrieve data by account number and timestamp, this is a good candidate for the clustering index.

Creating a DPSI on ACCOUNT_NUM and TRAN_TS and defining it as the clustering index will work fine for retrieval, since all the data you need is nicely grouped together.

To be able to maintain the clustering sequence, INSERTs need to have enough free space available so newly inserted transactions go into the correct page. To maintain enough free space for newly inserted transactions, regular REORGs may be required. Since the table is partitioned, and does not have any NPIs, running parallel REORGs at the partition level should not be a problem. If a single partition REORG still takes too long, you may want to consider using more partitions and making individual partitions smaller.

This design is shown in Figure 2-14.

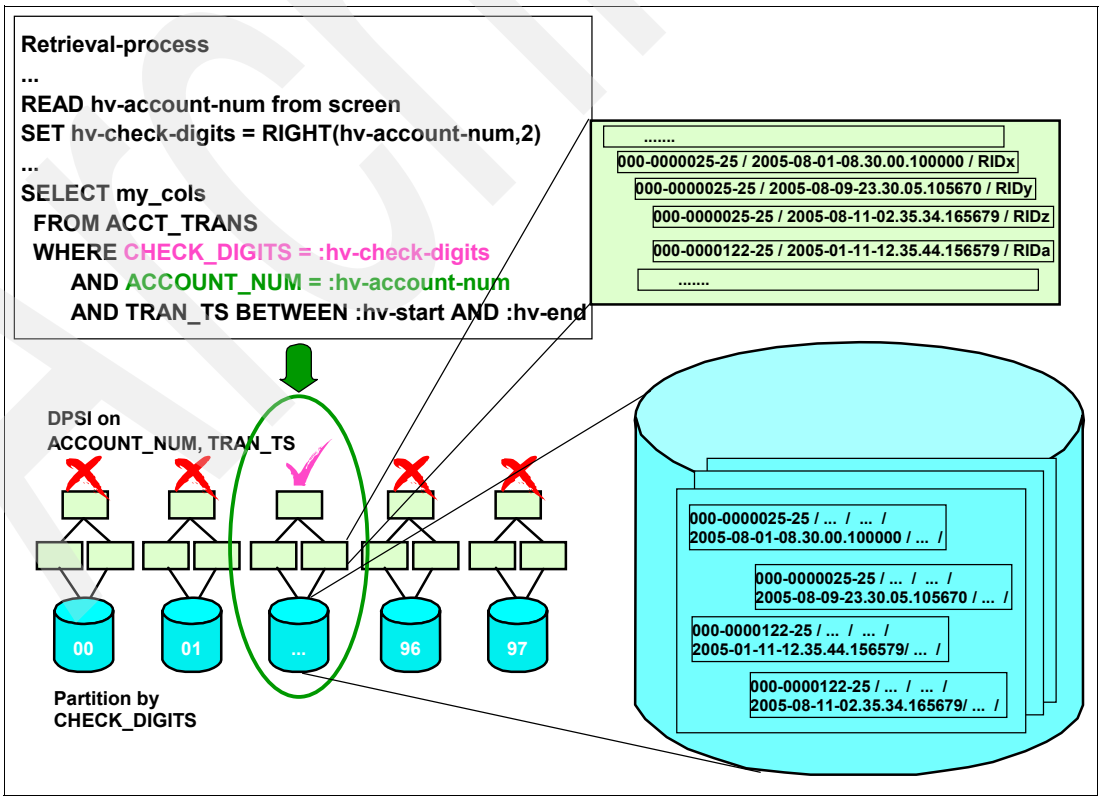


Figure 2-14 DPSI design

Option two: Cluster by transaction timestamp and account number

If for some reason, REORGs cannot be run, or providing appropriate free space for customers is not possible. For example, because of data distribution, some customers have an extremely large number of transactions and others have only a few transactions; therefore, providing a single free space number to accommodate both situations is not possible. You may want to consider clustering on the transaction timestamp (and account number).

This makes the inserts go to the end of each partition, and improves insert performance, because fewer getpages and sync I/Os will be required to insert the new transaction into the correct page (according to the clustering sequence).

This design will still allow for partition pruning, but since we now get into the index using the TRAN_TS range predicate, TRAN_TS >= :hv-tran-ts, more index pages will need to be accessed (touched), since it will require a lot of index page screening on the ACCOUNT_NUM = :hv-account-num predicate.

Option three: Cluster like option one, but without free space

As in option one, we use ACCOUNT_NUM and TRAN_TS for the clustering index, but we define the table space partitions with no free space, PCTFREE 0 and FREEPAGE 0. This allows DB2 to do more efficient insert processing, because all new bank transactions will be inserted at the end of each partition. There may be extra work, looking at the space map, and searching for space, but there will be fewer random getpages and synchronous I/O compared to option one.

Option four: Like option three, but defining the object as MEMBER CLUSTER

Table spaces that are defined as MEMBER CLUSTER (V7 and V8) and also use no freespace (PCTFREE=0 and FREEPAGE=0) can benefit from APARs PQ86037 and PQ87381. They enhance the space search algorithm for MEMBER CLUSTER table spaces.

Using PCTFREE=0 and FREEPAGE=0 indicates to DB2 that no free space needs to be provided at LOAD or REORG time. It is the appropriate value if the application always appends data at the end of the table space.

Prior to this maintenance, an exhaustive space search from the beginning of the table space is performed prior to issuing an extend request. Because a MEMBER CLUSTER table space only covered 199 pages, it usually meant that a lot of space map pages needed to be checked, which could impact the insert rate of heavy insert applications.

The APARs mentioned above implement an append logic, always adding data at the end of the page set. This could lead to a data set that grows very quickly. Even if rows in the beginning of the object would get deleted, the space is never reused by DB2, because it is always appending data at the end. To avoid this, DB2 also tracks the lowest space map that has available space, so deleted space can be reused without having to go through an exhaustive scan of all space map pages.

Still such objects are likely to grow faster than others and may need more frequent REORG to reclaim space, but it will speed up insert because they all occur at the end of the extent.

As mentioned, the append action is triggered by specifying PCTFREE=0 and FREEPAGE=0 on the definition of a MEMBER CLUSTER table space. If you need to avoid this behavior, use PCTFREE=1. It will negate the append action and give the same relative function for LOAD and REORG as PCTFREE 0.

Member Cluster table spaces with 0 PCTFREE and FREEPAGE are a good solution when looking for the best insert performance and *not* trying to insert in a clustering index key sequence.

Considerations

However, do not forget to provide adequate free space in the clustering index. In there, DB2 will still have to insert the keys in the correct place in the index. When not enough free space is available, page splits will occur, and when the free pages (“set aside” by the FREEPAGE parameter during LOAD or REORG) are used, the newly allocated page with half of the keys will go to the end of the index page set, and will cause additional synchronous I/Os (since it is unlikely that the page will stay in the buffer pool, unless the account is very active).

When data needs to be retrieved, it is likely that option three will not perform as well as option one. How big the difference is depends on the type of data that you are retrieving. If the data that you are retrieving is “old data”, that is data that is in the correct clustering sequence, option three performs the same as option one. However, if the data that you retrieve is mostly “new data”, that is data that was inserted at the end of each partition, that is not clustered properly, and will cause additional I/Os. So, if most data that is retrieved is “old data” that is clustered properly, option three is a good solution, because it improves the performance of all critical inserts, but only impacts a limited number of retrievals (only those searching for newly added rows).

So, the trade-off here is to improve insert performance, and sacrifice select performance for newly added, retrieved rows. The difference for the insert performance between option one and three critically depends on what kind of inserts is being done and can vary greatly.

To reduce the impact on SELECT statements, regular REORGs need to be scheduled to put the data back in clustering sequence.

Note that if this table (ACCT_TRANS) is a dependent table in a foreign key relationship, deleting the parent requires probing all parts of the DPSI to enforce the RI. Since we are dealing with account numbers, deleting or updating the account number will not happen very often, so this is probably not something to about which to worry.

More generally speaking, if a DPSI is used as a foreign key, this may be a case where you want to use application-enforced RI, because the application can derive the partition number and run an SQL statement that includes the partitioning columns and allow for partition pruning. DB2-enforced RI cannot implement this, since it does not understand how the partition number (CHECK_DIGITS) is derived from the account number (ACCOUNT_NUM) column. See also “RI enforced by DB2 or enforced by application” on page 132 for more details about application versus DB2 RI.

Archived

3

Model validation

The model generated during the design phase is validated against the original assumptions and additional conditions that were not used for generating the model. The error boundary of the model is also established during this phase. The validated model is also interpreted by what-if considerations.

Before investing a lot of time and effort in the implementation of the physical design of the model, it is a good idea to validate your model with the user. The purpose of the review is to confirm that the model meets the user's requirements, and the user understands the model well enough to develop applications on it.

The model might not meet all of the user's requirements. This does not always mean going back to the beginning. It is often an iterative process where price/performance is the guiding element. Capitalize on the validated part of the model and start working on the design. The remainder should go back to the requirements gathering stage. Either the requirements need to be better understood or they have changed and need to be redefined. If this leads to additions, and possibly changes, to the model already created, you might need to agree on a piecemeal approach, and delimit what can be provided sooner with benefits to the user.

The iteration of development and the continued creation of partially complete models are the key elements that provide the ability to rapidly develop databases.

In this chapter, we discuss the existing techniques to estimate the size of your objects and the duration of your processes.

We also show with an example the added value of using estimation as a technique to support your physical design process.

This chapter contains the following topics:

- Defining process requirements and service level agreements
- Determining the size of your objects
- Determining the duration of your process
- Impact of the access path and table design on cost

© Copyright IBM Corp. 2005. All rights reserved.

3.1 Defining process requirements and service level agreements

A first and very important step of any database design is the gathering of information. Any design starts with a phase during which you assess the application requirements. Since there is no such thing as the best implementation, we can only state that a certain implementation supports a specific workload efficiently. Another workload might perform very badly when using the same design. It is clear that the applicability of the solutions put forward in this redbook depend on your requirements.

Your ultimate goal is the achievement of the Service Level Agreement (SLA). To achieve these goals, you might have to choose an implementation that consumes more resources but gets the job done.

You must gather the following information to get a clear picture of the application requirements:

- Size of the DB2 objects
- Growth rate of the DB2 objects
- Throughput of the application
- The source of the attributes
- The average and peak volume of each application process that will interact with DB2 objects
- SLAs

Failing to provide this information in an early stage of the life cycle of the application often leads to inefficient applications or even project failures.

3.1.1 Reasons to estimate

Why do we need to determine the duration of the SQL or calculate the CPU consumption of the program?

- Compare design alternatives.
  - Which of two table partitioning schemes is the best?
  - Validate index design options.
- Build a capacity model.
  - When a new application is being put into production, your capacity planner might want to have an idea on the number of MIPS that are required. As more and more people move into usage-based pricing, this angle becomes more of an issue. It is not just the question of the number of processors that need to be installed, but also the impact of the new application on the four hour rolling average.
  - You might also need to build a model for the disk space allocation of your physical design. This model should cover all aspects of the space consumption which means that you need to consider the number of physical devices that are required and of course the space needed to be able to perform ONLINE REORG.
- Determine the elapsed time of an application process.
  - When twenty million rows need to be updated in a few minutes, it might be worthwhile to validate the estimated elapsed time before writing the application code. You should at least consider the cost and availability aspects, before you decide that an UNLOAD to a file with a subsequent reload is faster.

- Determine the maximum throughput of an object.
  - If you come up with a design that keeps track of the last allocated number in a DB2 table and you need to guarantee that all numbers are used, locking could become a bottleneck for your application. The number of times NEXT VALUE FOR sequence can be performed per second has limitations that need to be taken into consideration when designing the applications.
- Verify that the current measurements are reasonable.
  - Your application is currently consuming 10 seconds of CPU. Is this reasonable given the number and complexity of SQL statements?

Depending on the reason for the estimate, different approaches can and should be used. The speed of the processors has evolved significantly, which makes the estimation of the CPU cost more an overall capacity question than a determining factor when evaluating individual statements. The only exception to that rule are business intelligence (BI) type of queries since they tend to have a different getpage to fetch ratio.

3.2 Determining the size of your objects

Do you have a large database? This is an interesting question, but an even more interesting question would be “At what point do I start to worry about size?” Some people consider a table large when it contains more than 100 million rows, but do not bother mentioning that the length of the row is only 10 bytes.

What are the parameters that determine if size becomes an issue?

A basic red flag would be the sheer number of bytes that are required to store the data. DB2 is using VSAM to make the rows persistent, and VSAM introduces some limitations. Your system programmer will be an interested party, if you are going to exceed the 4 GB in a page set limit. The system can handle this, but it has to exploit extended address ability (EA) data sets which require some additional system setup. Nonpartitioning indexes (NPIs) are more likely to hit this threshold, because you do not have compression in the index, and, since the NPI, by definition, spans all the partitions of a partitioned table.

The cost of the SQL execution is also related to the size of the object, since an index probe in a five level index is likely to be more expensive than an index probe in a two level index. If your application is in the right context, DB2 is able to mitigate the difference by using index look-aside. See Appendix A, “DB2 access techniques” on page 421, for details. This assumes an optimal buffer pool hit ratio. If the object is small enough to fit into the buffer pool and is read-only, then issues such as random versus sequential access are not major concerns. If the object becomes larger and it suddenly does not fit in memory any more, then random versus sequential can become a very important parameter.

The index size is determined by the number of rows in the table (or partition, if the index is partitioned) and the size of the index columns. This gives us an indication of a formula for calculating the number of leaf pages that are needed to support the index. This is the key size plus the number of RIDs. This is simple multiplication in the case of a unique index since every key entry will only have a single RID. For indexes with duplicate keys, the calculation is more complicated, because the key entry will have multiple RID entries, but every leaf page needs to repeat the key entry.

You can use comprehensive tools, such as your logical and physical modelling tool, DB2 Estimator, or you can use the quick sizing spreadsheet referenced in Appendix C, “Additional material” on page 439 under QuickPhyssizerV1R1.zip Microsoft® Excel® Spreadsheet).

The spreadsheet has two worksheets, one called Table Stats and a second one called Index Stats. The Table Stats allows you to do a quick estimate of the number of pages per row, the number of pages in total, and you can simulate the impact of compression, free space, and partitioning. See Figure 3-1 for an extract of the table worksheet.

	Sizing Information							Tablespace Results					
Table	**Row Len**	**Cmprs Ratio**	**pct free**	**Total rows**	**# parts**	**DS size (in Gig)**	**DS type**	**rows per page**	**rows per part**	**pages per Part**	**cyls per part**	**pqty per part**	**Tot cyls**
prod1	250	10%	0	100.000.000	12		REG	17	8.333.333	490.197	2.724	1.961.280	32.688
prod2	250	10%	0	100.000.000	120		LARGE	17	833.333	49.020	273	196.560	32.760
prod3	250	10%	0	100.000.000	1.000		LARGE	17	100.000	5.883	33	23.760	33.000
prod4	250	10%	0	100.000.000	120		LARGE	17	833.333	49.020	273	196.560	32.760

Figure 3-1 Definition of table sizes and attributes

Once you have filled in the table worksheet, you can refer to the table information from the Index Stats worksheet. The are two parts in this sheet, the first section, shown in Figure 3-2, on which you can enter index attributes such as uniqueness, the number of different values in the key, the length of the key, and the free space. The spreadsheet allows you to simulate the impact of making the index partitioned by changing the partitioned cell for the index.

INDEX NAME				Index Entry				Data from Table			
Index Name	**Unique**	**Partitioned**	**Fullkeycard**	**Piece Size (in Gig)**	**Key Len**	**ix pct free**	**ix freepage**	**Table Name**	**DS type**	**# parts**	**Total rows**
prod1	N	N	100,000,000	4.00	4	25	0	prod1	REG	1	100,000,000
prod2	Y	N	100,000,000	4.00	6	10	0	prod1	REG	1	100,000,000
prod3	Y	Y	100,000,000	4.00	2	10	0	prod3	LARGE	1000	100,000,000
prod4	Y	N	100,000,000	4.00	6	10	0	prod4	LARGE	1	100,000,000

Figure 3-2 Definition of index sizes and attributes

See Figure 3-3 on page 113 for the results of the values that were entered in the first part. The spreadsheet calculates the space required per key, the number of index levels, and the number of leaf and nonleaf pages. It does not take into consideration the free page value yet when calculating the total number of pages.

INDEX NAME				Index part Results						All Parts
Index Name	**Unique**	**Partitioned**	**Fullkeycard**	**Space per key**	**ix levels**	**ix entry / pg**	**ix leaf pgs**	**ix non-leaf pgs**	**ix total pages**	Total Pages
prod1	N	N	100,000,000	13	4	234	427,491	1,769	429,260	429,260
prod2	Y	N	100,000,000	13	4	279	358,423	1,282	359,705	359,705
prod3	Y	Y	100,000,000	10	2	363	276	1	277	277,000
prod4	Y	N	100,000,000	14	4	259	386,101	1,381	387,482	387,482

Figure 3-3 Quick estimations of the number of pages and levels

The numbers you see in this example refer to the case study described in “Impact of the access path and table design on cost” on page 121.

3.3 Determining the duration of your process

Measuring is knowing, but how do we know when we have not built yet? Very often there is more than one way to reach a goal and the task of the designer is to determine which way is the best. This is not a trivial task since two of possible outcomes for any design are: a) the application fails to work because it consumes too many resources or simply does not work, and b) the application works, but it is inefficient. Experience might help you to decide based on your feeling which design is the best, but when building applications that go close to the limit, you need more than just a feeling.

If you evaluate a partitioning scheme or plan to change the clustering sequence, the design impact will be significant and you do not want to find out on the first launch of the application that you picked the wrong design.

At some point, you need to validate the efficiency of your SQL and need to estimate the elapsed time and resource consumption of your utility execution.

3.3.1 Elements of the estimation

There are at least two dimensions to any performance estimate:

- CPU consumption
  - There are several reasons for consuming CPU and many of them can be influenced by the behavior of the application, design of the objects involved, and tuning of the underlying system.
  - Excessive CPU consumption can therefore be related to any of the mentioned elements, which also means that it is not always going to be straightforward to solve a high CPU consumption problem. We can however establish a lower boundary of the CPU required to perform an SQL operation.
- Elapsed time
  - The elapsed time is a dragon with many heads, in fact at least as many heads as we find class 3 counters in the accounting trace. The elapsed time is always at least equal to roughly 1.2 times the CPU estimate. We add 20% to the CPU time because it is not likely for an application to get the processor 100% of the time. Even if the application

gets the processor all the time, then you would still see stretch time because of system overhead.

– The biggest issue, when dealing with elapsed time, is to determine at design time which of the potential wait times will manifest themselves at run time. The amount of wait time is very much driven by the design choices, such as index design, clustering index determination, locking options, COMMIT frequency, and so on. The efficiency of the buffer pool also has a significant impact on the elapsed time, especially when dealing with random getpages. The hit ratio is very much a Catch-22 situation as the buffer pool tuning will be driven by both the application behavior and by the buffer pool thresholds and object placement.

3.3.2 Based on previous measurements

Direct experience is a very good way to get a feel of the actual cost of a process. If you add new parts to an existing application, measuring the performance of the existing application gives you more than just a ballpark cost estimate.

Beware that you need to recalibrate your numbers when pieces of your infrastructure change. Installing a faster CPU, moving to a new generation of disks, moving to data sharing, installing a new version of DB2, or even applying maintenance might have an impact on your numbers.

You have to be careful with previous estimates if you are changing the metrics of your current environment. This does not only mean that things tend to change if you install a new processor, but things can significantly change when you add, for example, 50% of new rows in the table. The problem with adding new rows is that the extrapolation formulas can be very different depending on the circumstances. If you increase the number of rows by a factor of ten this does not mean that a one fetch access will also be 10 times more expensive, but a table space scan might.

The relative speed of processors can be found at the following URL:

http://www.ibm.com/servers/eserver/zseries/lspr/zSerieszOS16.html

3.3.3 Estimating without tools

When evaluating different design alternatives, you might just want to get a ballpark figure of the performance characteristics of the design option. This means that you will be required to estimate the CPU and elapsed time consumption of the design option. If you are dealing with very large objects or you are working on a design for which no data exists yet, a capacity planning exercise based on calculation might be the best approach. The approach that we describe in this chapter can even be useful if you do not go all the way to applying all the detailed formulas, but just come up with some rough metrics of cost. Rough metrics are often enough when comparing alternatives.

Cost elements

In order to understand the cost of SQL, we need to define the elements that determine the elapsed time and the CPU consumption.

Touch

Scanning indexes and table rows incur a CPU cost. We assume that there is a simple relationship between CPU cost and the number of entries processed. This is expressed with the idea of a *touch*. You can also think of our use of the word reads as “processes,” “visits,” “tests,” or “touches,” the most suitable word of all. The touch is directly related to the CPU component of the response time. It is derived from the number of index entries and rows that DB2 must visit in order to obtain the result.

The cost of the *touch* is based on three components:

- The execution of the GETPAGE
- The application of any predicates on the row
- The number of rows that are processed on that page

The number of applied predicates is related to the way the SQL is coded. The number of rows that are processed depends on the nature of the *touch* and the object that is accessed, if a data page is accessed randomly, then it is more than likely that only one row is being processed since it is a data probe from a matching index entry. When the data access is sequential, hence a sequential *touch*, then the number of rows on the page will most likely determine the ratio of GETPAGES per row processed. If you want to make an estimate at a higher abstraction level, then you could simply fix the number of predicates per row and fix the number of rows per page. This would then give two different costs, one for the sequential *touch* and one for the random *touch*.

Synchronous read

If a single page must be read, DB2 will do that for you. This is an expensive read, therefore you must know the number of times that this kind of I/O occurs. DB2 calls this read a *synchronous read* (SR).

Any touch against a page that is not in memory results in a true read, that is, a physical I/O operation to retrieve a page. The I/O is scheduled on behalf of the application, as soon as it needs the page (but not earlier), hence "synchronous." The number of SRs is a little more difficult to establish than the number of touches, because to do so requires some DB2 knowledge. The probability of an I/O happening upon a touch is strongly related to the access pattern in the data, the frequency of re-referencing, and the amount of the memory that can be dedicated to the object, which can be controlled by means of the buffer pool size and thresholds. See "Buffer pool efficiency" on page 385.

An SQL statement that processes the pages in the sequence the pages are stored is going to benefit from one of the methods DB2 has to prefetch the information into memory before the application actually *touches* the page. See Appendix A, "DB2 access techniques" on page 421, and the section, Interpreting data prefetch, in the *DB2 UDB for z/OS Version 8 Administration Guide,* SC18-7413-02, for an explanation of the different prefetch techniques that DB2 supports.

Sort

The most common reason for sorting is the ORDER BY clause of a SELECT statement. DB2 can avoid a physical sort if a suitable index is available. You may in fact design an index for this very reason. If you have such an index, you do not count sorting costs.

In the design phase, watch out for sorts if the following SQL clauses are used: ORDER BY, GROUP BY, DISTINCT, some JOIN methods, and UNION may cause sorts. UNION without ALL always requires a sort; the others may use an index. To be sure, there are other circumstances where a sort is used by DB2, but none of immediate interest.

Sorting rows has several side effects, such as materialization of the result set, impact on the locking behavior, the creation of a work file, and others. This will cause an increase in the CPU consumption of the SQL statement.

Materialization also has an impact on the number of touches. Assume that you need to retrieve the first 10 employees in descending sequence of the SALARY, which is not indexed. You would probably use the SQL statement shown in Example 3-1 on page 116 to achieve this task.

Example 3-1 Retrieving the employees with the top 10 wages

```
SELECT EMPNO, FIRSTNME ,MIDINIT ,LASTNAME, WORKDEPT, HIREDATE, JOB, EDLEVEL, SEX,
BIRTHDATE, SALARY, BONUS, COMM
FROM EMP
ORDER BY SALARY DESC
FETCH FIRST 10 ROWS ONLY
```

How many rows do we need to read? Just 10 or the entire table? How many rows need to be sorted? The answers to all above questions are going to be “all the rows”. Materialization in general and potential subsequent sort overhead have a very large impact on the number of touched pages and the CPU consumption of SQL.

Referential integrity

When changing rows that are involved in RI structures, one should add the cost of the RI check. This is described in more detail in “Referential constraints” on page 130, but we can state, in general, that you can consider this cost to be equal to having the check in the application using SQL, but without the overhead of getting in and out of DB2.

Check constraints

The cost of a check constraint also needs to be taken into consideration when making changes to the content of columns which are subject to the table check constraints. The cost is 4 microseconds for each Insert, Update, or Delete SQL statement with constraint plus another 4 microseconds per effected row.

Trigger

The cost that a trigger adds to the base SQL has two different components:

- The cost of managing the trigger
- The cost of the SQL in the trigger body

The trigger management involves the allocation of the trigger package, the actual invocation of the trigger, and the potential management of the work files or memory when using transition tables and transition variables. The actual cost of this management is dependent on the create options and is explained in more detail in “Using triggers” on page 168.

The cost of the SQL in the trigger body, the actual trigger package, can be estimated because it is normal SQL outside the trigger. We just need to take into consideration that we are running the SQL inside the DB2 engine which means that no address space boundaries need to be crossed which tends to make the SQL cheaper.

Stored procedure

Stored procedures provide several advantages but the infrastructure overhead is not insignificant. DB2 uses the WLM application environment concept to schedule the stored procedure program. The cost of calling a stored procedure depends on the programming language that is used to code the stored procedure. The actual cost can range from 250 to 700 µsecs of CPU on a z900 machine. See “Performance expectations and tips” on page 158 for more details about stored procedure performance.

Performance concepts of SQL statements

We have already introduced a number of elements that influence the cost of an SQL statement. But there are other more general elements that determine how much CPU your SQL is consuming.

The number of columns that are participating in the SQL have a direct link to the cost. The application and DB2 are running in separate address spaces. Any information that is passed

between those two address spaces is going to require processing and that processing is more expensive than moving data around in your program. It does not come as a surprise that we do not want to deal with columns that are not actually used, hence, the tip *do not use* **SELECT ***.

The number of trips from the application to DB2 is another element that impacts the CPU cost of executing the SQL. There will be a significant difference in CPU consumption if you decide to use a cursor instead of a singleton select to retrieve a single row. Always try to minimize the number of SQL executions.

A second element is the actual execution of the SQL itself. Why ask DB2 for the timestamp if you can obtain the time from your programming language? Issuing a SET :HV = CURRENT TIMESTAMP is like putting your milk bottle in your neighbor's refrigerator.

SELECT

The cost of select is strongly dependent on the underlying access path. A one fetch access has a different cost than a table space scan although both can be the result of the same SQL statement. So we really need to build a mental picture of the pages that are involved in the process and put these numbers in the formula.

If we know the access path, it is quite straightforward to determine the cost.

Let us look at a few examples:

- Table space scan

  DB2 simply starts at the beginning of the table and processes all rows. You can assume that the number of touches will be equal to the number of rows in the table. Without any materialization, this process is driven by the fetch, as a result the number of touches may be smaller if the program decides not to fetch up to SQLCODE 100. The number of SR is considered to be 1 since we need to read the first page to start processing, then all the other pages are read using asynchronous processing.

- Access paths that use an index

  The variations on access paths using indexes are quite large because index scans can result in one scanned and qualifying index entry all the way to the other extreme, where all index entries are scanned and all are qualifying. If the index access path is not index-only, you also need to consider and count the accesses to the data pages. The accesses to the data pages are dependent on the way the index is defined and whether or not list prefetch is used as an optimization technique.

  The cost of an index-based access that is not index-only is composed of two and possibly three elements.

  - The index access part
    - Nonmatching index scan

      The number of touches are, in this case, related to the number of leaf pages in the index. So, the number of touches simply equals the number of leaf pages in the index, assuming we fetch all the way to SQLCODE 100. The number of SRs is going to be the same as for a table: 1 to get the sequential prefetch going.

    - Matching index scan

      An index scan is considered matching if not all leaf pages are read. As a consequence, the root and nonleaf pages are used to navigate among the leaf pages. We depend on the values of the predicates to estimate the number of pages that are actually read. A “>” predicate on the leading column can range from 1 to *n* touches, with *n* being the total number of leaf pages. We need to add the touches in

the nonleaf pages. In fact, the matching index scan may end up with more touches than the nonmatching index scan.

- Multiple index scan

  The multiple index scan is just a case of several index scans plus the process that puts the RIDs together and then goes after the data. An important point is that we are no longer in a fetch driven access path, but an access path that performs its tasks at the beginning of the SQL.

- One fetch access

  This is a special case of a matching index scan, since we can guarantee that we just need a single probe down the index, and that it does not involve processing the data pages.

– The RID processing used by list prefetch

  The full index processing, which includes the full search for qualifying RIDs in all indexes, the index ANDing/ORing and RID sort take place at first fetch. The RID sort loses the sequence of the original key values in the index as the sequence becomes data page number and slot number, which is not guaranteed to be in the key sequence. This means that any ORDER BY on the SQL forces a sort of the data rows after data access. The data access from the list of RIDs to the table space becomes driven by the application fetches. This means that not fetching to SQLCODE 100 does not impact the index processing cost, but only impacts the cost of table processing. The prime reason for list prefetch is to optimize the reading of the data part. This is most likely to be beneficial when reading multiple rows and where the sequence of the RIDs is not the same sequence as the sequence of the data rows.

  An index stores the RIDs for duplicates of a key value in ascending order. This means that list prefetch is most likely to happen on badly clustered indexes when we are looking for a range of key values or if we use multiple index access or hybrid join. It is also worth mentioning that the efficiency of the table I/O will be driven by the page distance. If the pages in the prefetch are too far apart, then the gain will be mainly a reduced number of I/Os more than savings in the elapsed time of the I/O. Do note that the I/O has become asynchronous to the thread so that we should not add it to the elapsed time, while some I/O may be synchronous due to the page distance.
  Once you understand the process, it becomes quite clear that estimating the elapsed time and CPU consumption of a process that uses list prefetch can be quite tricky since there are many factors that influence the end result. It is also obvious that any access path that needs list prefetch is going to be something you may want to look into when dealing with aggressive response times.

– The data access part, assuming no list prefetch is used

  The access of the data part when coming in through the index is pretty straightforward when no list prefetch is used. Whether the access should be considered a sequential touch or a random touch really depends on the cluster ratio of the index that is at the origin of the data access. It can go either way. The index cluster ratio gives you a pretty good idea on the probability of a touch to be in the neighborhood of the previous data access. A cluster ratio of 100% should have all the accesses in sequence. Beware that DB2 is not performing reverse sequential detection, which means that you still may end up with significant synchronous I/O even though you are processing in sequence, albeit reverse sequence, unless you perform a backward index scan.

INSERT

An insert consists of:

1. FIND
   - DB2 must first determine where the new row will be put. The insert algorithm is explained in great detail in the IBM licensed document *DB2 UDB for z/OS Version 8 Diagnostics Guide and Reference*, LY37-3201-02.
   - The clustering index (or first index defined) participates in the data page location determination. To do a quick estimate, we assume that a clustering index is present and that we are not using data sharing with MEMBER CLUSTER, since the latter would potentially create multiple insert points in the table.
   - DB2 tries to put the row on the spot that is indicated by the preceding key value in the index. We should estimate at least one touch and one SR. The number of touches can be much higher when looking at large indexes, since the number of levels has a significant impact on the number of pages (nonleaf and finally leaf) at which DB2 needs to look. So, when dealing with large objects, three or four touches can be a more accurate estimate. The number of SRs can be lower than the number of touches since the higher level pages of the index tree tend to stay in the buffer pool.
2. Referential integrity
   - For a foreign key, you must count an index-only access to the parent table. Again, we should estimate this requires one touch and one SR. When dealing with large objects, three or four touches can be a more accurate estimate.
3. Trigger
   - The cost of executing the trigger should be considered as part of the SQL since it is executed synchronously with the originating SQL.
4. MODIFY
   - To add the data row, you count one touch and one SR.
   - Additional indexes defined on the table perform a similar process and need to be counted separately.
   - Note that you do not include a check for uniqueness; any UNIQUE index is touched anyway in MODIFY.

DELETE

A delete consists of:

1. FIND
   - The cost of finding the rows is determined in the same way as for read operations.
2. Referential integrity
   - For a foreign key, you must count an index-only access to the dependent table. In that dependent table, you must count delete costs (if CASCADE is in effect) or update costs (if SET NULL is in effect).
3. Trigger
   - The cost of executing the trigger should be considered as part of the SQL since it is executed synchronously with the originating SQL.
4. MODIFY
   - To delete the data row, you count one touch. Next, for each index on the table, you add one touch and one SR, or when dealing with large objects, three or four touches can be a more accurate estimate.

Archived

UPDATE

An update consists of:

1. FIND
   - The cost of finding the rows is determined in the same way as for read operations.
2. Referential integrity
   - For a foreign key, you must count an index-only access to the parent table. Again, this should be estimated as one touch and one SR, or when dealing with large objects, three or four touches can be more accurate.
3. Trigger
   - The cost of executing the trigger should be considered as part of the SQL since it is executed synchronously with the originating SQL.
4. MODIFY
   - To update the data row, you count one touch. Next, for each index that is changed, you add twice the cost of an insert and the cost of delete. An index update is effectively a DELETE and an INSERT.

3.3.4 Quick upper bound estimate

Quick upper bound estimate (QUBE) provides a simple method that you can use as early as possible in the design phase to establish:

- Whether or not a requested performance can be achieved with a certain database and transaction design.
- Which of several alternatives, in database or transaction design, will provide better performance.
- The order of magnitude of computer costs and cost components (giving valuable insight in its own right and a first idea of possible chargeback costs).

The method is described in detail with many useful examples in the redbook, *DB2 - Quick Upper Bound Estimate An Application Design Methodology*, SG24-2549. The book was published in August 1995, but many of the concepts are still valid. The formulas are outdated and should be revised before being applied.

The QUBE method is simple, and therefore it:

- Does not require advanced DB2 knowledge
- Is fairly release-independent
- Can be understood and used by anyone working in application development
- Does not require input that is far beyond what is normally available in the design phase
- Assumes a more or less stable and “reasonable” DB2 production environment

By using QUBE, you get a simplified view of elapsed time, ideally, just the sum of CPU and I/O components. This simplified view is enough to make common design decisions and discuss any proposed alternative in a methodical way.

QUBE provides a mental picture of CPU and elapsed time in a DB2 environment. This picture and its terminology are shared within the project team and can be used throughout the development cycle. Therefore, the use of QUBE is not restricted to the design stage; it also can be used in programming, testing, and monitoring.

The assumptions for the QUBE formula are:

- Elapsed time is a sum of CPU time and I/O time.

- CPU time is a function of the number of rows that are processed.
- I/O time is a function of the number of pages that are read.
- All of the above timing functions are related to the required application functions and the access strategy. The latter, determined by DB2, can be described in a simplified manner so that any developer can make an easy early estimate.

The output of QUBE is a thoroughly tested physical database and transaction design that will support all known requirements from a performance point of view, and a good picture of the computer costs, per application function. These are tangible results.

The QUBE formula is designed to give a conservative, and therefore “upper bound” estimate. The idea is that you should encounter as few disappointments as possible when your application goes into production; in fact, you should get many pleasant surprises.

For those who expect a fairly accurate estimate, we have the following remarks:

- QUBE is not designed for such accuracy, nor is such accuracy needed in the early stages. What is needed though, is to catch utterly impossible database and program designs, before even one statement is written.
- QUBE can be tailored to your specific environment (for example, processors, disks, and MVS™); this tailoring is a specialist’s job. Tailoring may contain inaccuracies of its own. QUBE still has value in terms of a relative estimate.

It must be understood that QUBE is *not:*

- A method of calculating expected performance. Although QUBE produces many numbers, expressed in real-life units (milliseconds), they should not be used on their own. They merely act as traffic lights, answering the question: Are we on the right track?
- A capacity planning method. For this, the numbers must be exact, which they are not.
- A tool. QUBE provides a mental picture, a common framework. The QUBE formula can, of course, be put into a “tool”, for example, a simple spreadsheet, but QUBE itself is still a method.

3.4 Impact of the access path and table design on cost

Let us assume that you store all the sales details of the last year in a partitioned table. The table contains 100 million rows of sales data. You need to obtain the sum of the sales amounts for each product sold during the month of January. The table contains 12 months of sales data and our month is identified by a value of '01' in the MONTH column. The company sells 1,000 different products.

There are several ways you can partition the data, but we compare two options:

- Partition the table by month
- Partition the table by product

First, we a look at partitioning by month. You can implement 12 partitions that contain the sales data for each month. The row length is 250 bytes with a compression ratio of 60%. This results in 220,000 pages per partition. We implement table-based partitioning, which means that there is no need to include the MONTH column in the partitioned index we define on PRODUCT. The size of each index part is slightly over 30,000 pages per part with a free space of 25% and the PRODUCT is an integer. By defining the index on PRODUCT as partitioned, we get an implicit virtual leading MONTH column in the index. Every RID in the index has the partition number, which is, in our case, an equal predicate on the MONTH since every partition has a unique value for MONTH. This means that a nonmatching index scan

with partition elimination behaves as a matching index scan in an nonpartitioned index defined on MONTH and PRODUCT. Figure 3-4 illustrates the page accesses to process the query.

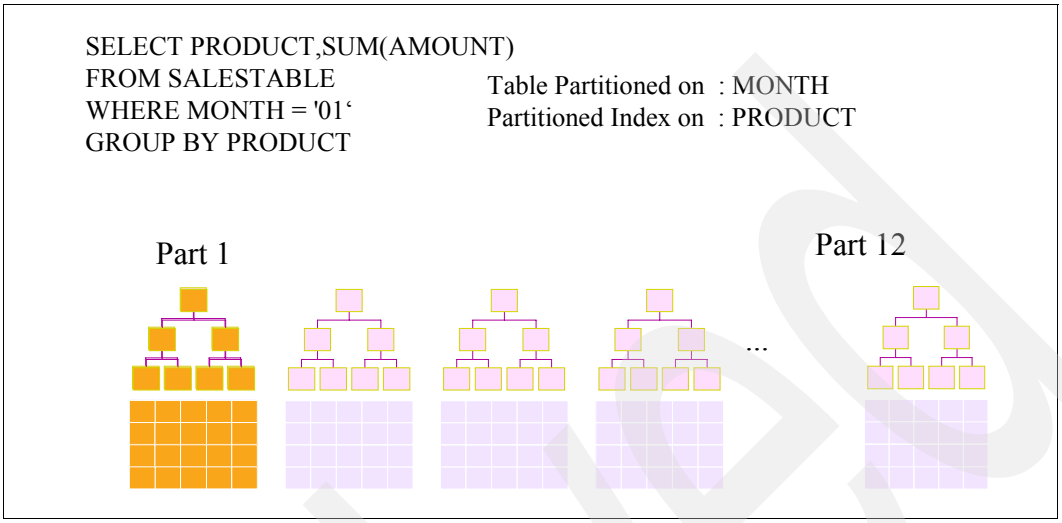


Figure 3-4 Table-based partitioning by MONTH

The access path is most likely a nonmatching index scan with elimination of all the partitions other than the first partition. The number of pages referenced is optimal since it is limited to the partition that contains the concerned month in both the partitioned index and the table partition. The sort is avoided by using the index.

The number of touches in the index is going to be at least equal to the number of leaf pages, plus the size of the tree, if we want to avoid having to sort the data rows. All these touches are sequential because we need to read all the index entries of the part. Since the partitioned PRODUCT index is the only index defined and we added the CLUSTER keyword, all the touches on the data part are going to be sequential touches and the number is again equal to the number of rows in the partition. The CPU is the largest contributor in the SQL statement cost since all the touches are sequential, which will make the required I/Os asynchronous. The total cost of the CPU may be larger than the elapsed time that is acceptable for the user. Can we reduce the elapsed time of the SQL?

Let us have a look at the cost of the query when the partitioning is changed to PRODUCT instead of MONTH. We implemented a partitioning scheme where the products are grouped in 120 partitions. Each partition contains roughly eight products. The month is no longer the leading column in the index, neither directly nor indirectly, which rules out the possibility of a matching index scan. The effect on the index is, therefore, not negligible as the number of sequential touches is now equal to the total number of leaf pages in the index, a significant increase compared to the previous option. The access to the table is also significantly changed since we are now having to skip the parts in the index that are not covering the month of January. So the number of screened rows in the index is equal to the total number of rows, but the number of touches to the table is only going to be 1/12 of the number of rows as we do index screening on the month.

These touches are still sequential, but might be less efficient since the clustering sequence of the table is product-based and month-based, which means that there are 11 months for each product that need to be skipped. This means that the table access might go in and out of qualifying the sequential detection criteria. We seem to have achieved the opposite of our objective since the number of touches has gone up and we may even see some of the sequential touches generate synchronous I/O. The only positive effect is a better probability

for DB2 to perform the SQL in parallel. There are now many more partitions involved in the SQL, and this increases the probability that DB2 will perform these touches in parallel. Is there a way to increase the degree of parallelism without increasing the number of touches or changing the nature of the touches?

See Figure 3-5 for a graphical representation of this access path.

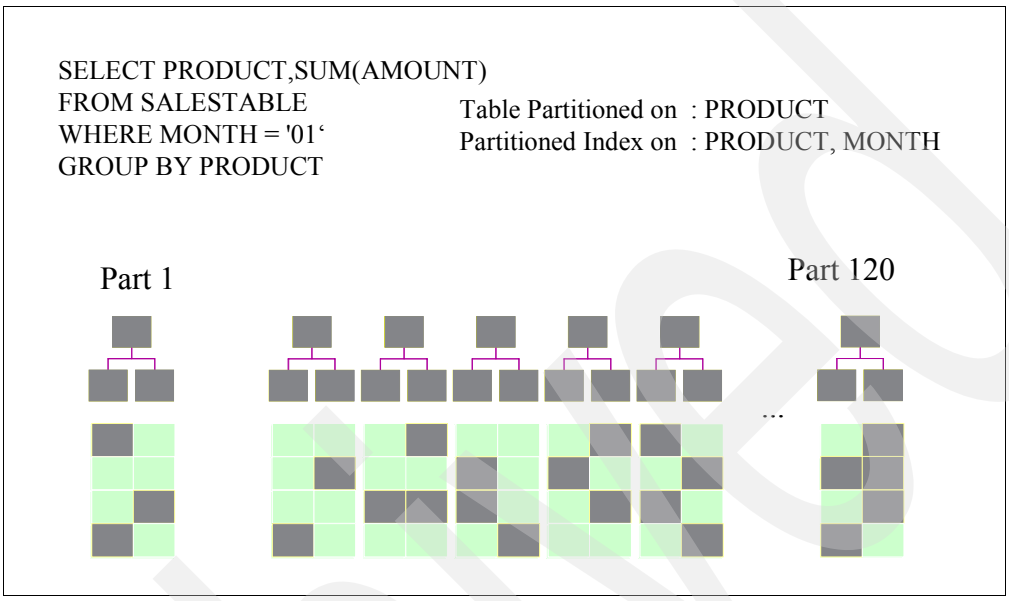


Figure 3-5 Table-based partitioning by PRODUCT with 120 partitions

We could increase the number of partitions to match the number of products. We redefine the table with new partition boundaries that are equal to the individual product number. The result on the performance is significant, since DB2 can now use the DPSI on MONTH and skip 1/12th of each index part. The 1,000 parts of the month index contain only values for one product. This creates a virtual product column that contains a single value. The number of pages that needs to be touched is now equal to the first design, since no unqualifying rows need to be touched. A potential disadvantage may be that we need 1,000 index probes, one per part. The other disadvantage is the problem of schema evolution. The number of products is not likely to be stable. Once we have a change in the product definitions, we would need to revisit the table partitioning to keep the benefits of the design. The PRODUCT column is too volatile to use this kind of design. Would it be possible to take another approach to increase the degree of parallelism without impacting the number of pages that need to be touched?

See Figure 3-6 on page 124 for a graphical representation of the access path.

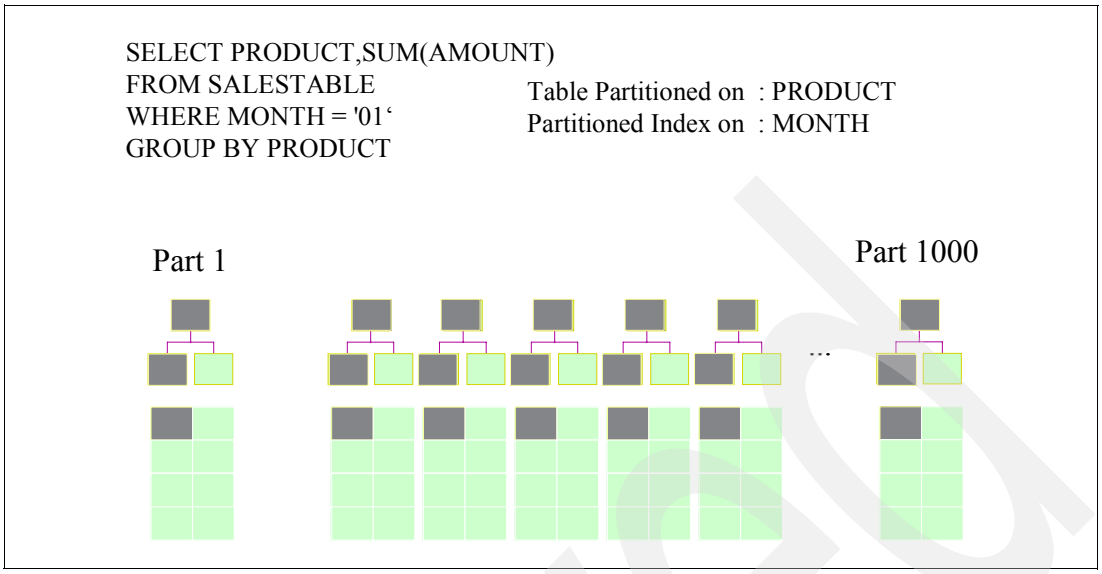


Figure 3-6 Table-based partitioning by product with as many parts as products

We can revisit our first design, but now we increase the number of partitions that are used per month. We can add the product column as part of the partitioning key. We define 120 parts by adding the PRODUCT column in the partition boundary definition. Roughly 100 products are grouped by month. The number of parts that need to be processed is going to be 10, since that is the entire month of January. So the number of getpages is equal to the first design, but the degree can be 10 times higher, therefore, a reduced elapsed time.

DB2 does not use parallelism when a partitioned index is providing the sort avoidance. This means that we need to remove the clause partitioned from the index on PRODUCT and add the MONTH column to the index to avoid a full scan of the index.

See Figure 3-7.

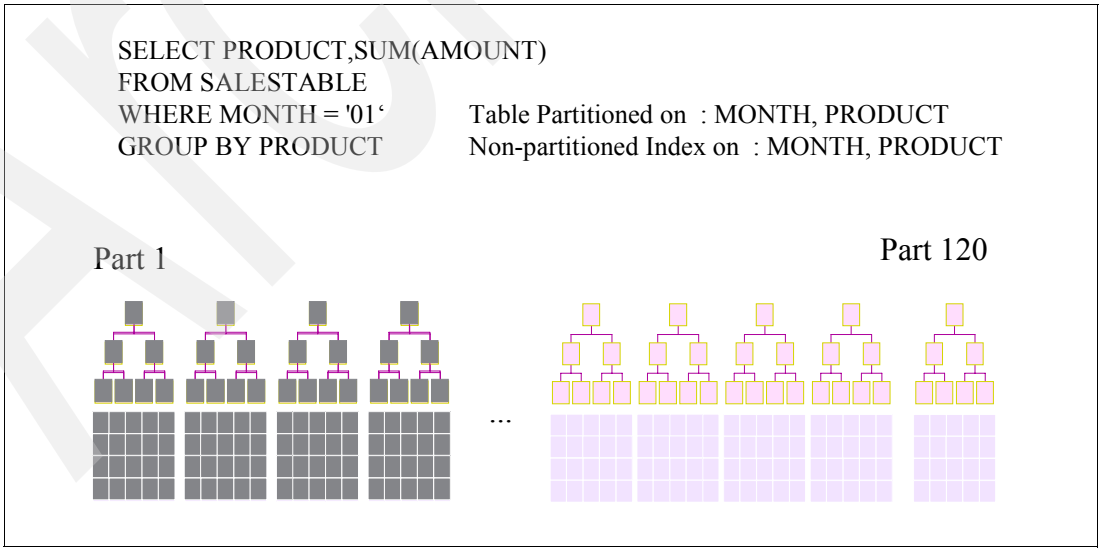


Figure 3-7 Table-based partitioning by month and group of products

Table 3-1 on page 125 now summarizes the design alternatives so far.

Table 3-1 Overview of partitioning design alternatives

Partitioning Columns	Number of parts	Index Columns	Index pages touched	Data pages touched	Number of partitions used in query
Month	12	Product	30,000	220,000	1
Product	120	Product, Month	390,000	220,000	120
Product	1000	Month	26,000	220,000	1,000
Month, Product	120	Month, Product	32,300	220,000	10

From the overview, we can deduct several things:

- The number and nature of the touches are determined by the interaction of the SQL and the table design.
- Sometimes the CPU cannot be reduced, which means that parallelism is the option to optimize the elapsed time of the SQL.

There are, of course, other options, such as the use of aggregated tables and maybe even the use of MQT to tackle the problem if the application allows them to be used. But that is just stating the obvious that the best way of solving a problem is by avoiding it.

Archived

Archived

Part 2

Application enabling components

In this part, we introduce DB2 functions which provide application enabling components. We describe referential constraints, identity columns and sequences, routines, and XML support.

This part contains the following chapters:

- Chapter 4, “Constraints” on page 129
- Chapter 5, “Techniques to generate numbers” on page 137
- Chapter 6, “Database side programming” on page 151
- Chapter 7, “XML support” on page 177

© Copyright IBM Corp. 2005. All rights reserved.

Archived

4

Constraints

DB2 provides three types of constraints that can be used to enforce business rules in the database. These constraint types are:

- Uniqueness constraints, which prevent duplicate values in one or more columns in a table
- Referential constraints, which are used to enforce *referential integrity* (RI), or the existence requirements between rows in the same or different tables
- Check constraints, which enforce domains on the attributes of the tables

In this chapter, we discuss these types of constraints, and also include special considerations for implementing code tables.

This chapter contains the following topics:

- Uniqueness constraints
- Referential constraints
- Check constraints
- Code and look-up tables

© Copyright IBM Corp. 2005. All rights reserved.

4.1 Uniqueness constraints

A uniqueness constraint is a mechanism in DB2, used to enforce that the values of a key are unique in a table. A unique constraint is established using the PRIMARY KEY or the UNIQUE clause of the CREATE or ALTER table statement. A table can have any number of uniqueness constraints, but only one of them can be a primary key.

The columns in a unique key cannot include LOB or ROWID columns, and they must be defined as NOT NULL or NOT NULL WITH DEFAULT.

Any unique constraint can be referenced by the foreign key of a referential constraint.

4.2 Referential constraints

Referential integrity (RI) is the integrity of the relationships between rows in the same or different tables. Each RI relationship has a parent table and a dependent table. The foreign key of the dependent table references a unique key of the parent table, which is called a *parent key*. Referential integrity is intact when all not null values in the foreign key of the dependent table have a matching value in the parent key of the parent table. If any column in the foreign key is null, then the entire foreign key is considered to be null.

A single table can be both the parent and dependent in the same referential constraint in order to enforce relationships between different rows in the same table. This is known as a *self-referencing* table.

A referential constraint is a DB2 mechanism that is used to enforce RI. A referential constraint enforces the following three rules:

- Insert rule

  The insert rule enforces that whenever a row is inserted into the dependent table with a not null value in the foreign key, a matching value in the parent key must exist, or the insert of the row will fail.

  The insert rule is enforced for all INSERT statements on the dependent table, as well as for all rows loaded into the dependent table using a LOAD utility with the ENFORCE CONSTRAINTS keyword specified. When loading rows, the ENFORCE phase of the utility will discard all rows from the dependent table that violate the referential constraint.

- Update rule

  The update rule enforces that whenever the foreign key in the dependent table is updated to a not null value, a matching value in the parent key must exist, or the update of the row will fail.

  The update rule is enforced for all UPDATE statements on the dependent table.

- Delete rule

  The delete rule controls the action that should be taken when a row in the parent table is deleted, and there are one or more dependent rows that reference the parent row. The action to be taken is specified using one of the following keywords on the foreign key definition:

  – RESTRICT or NO ACTION

    The delete of the parent row will fail.

– CASCADE

  The delete operation is propagated to the dependent table, and all dependent rows are deleted.

– SET NULL

  Each nullable column in the foreign key of each dependent row is set to null. This option can only be specified if one or more columns of the foreign key is defined as nullable.

The delete rule is enforced for all DELETE statements on the parent table and for any DELETE statements that propagate delete operations to the parent table. A LOAD REPLACE of the parent table will not enforce the delete rule, but it will place all dependent tables in a CHECK PENDING status.

From an elapsed time performance perspective, the CASCADE delete action will perform better than explicitly deleting the rows from each dependent table. However, a cascading delete can potentially delete a large number of rows in a single statement. In this case, the application program has no control of the commit scope, and lock contention with other processes may result.

For best concurrency, a cursor should be opened on the dependent tables and the rows deleted individually, taking commits periodically to release locks. A possible alternative would be to perform a count of the dependent rows, and only perform the cursor delete if the number of dependent rows to be deleted will exceed the desired commit scope.

There are several requirements to be able to use DB2-enforced RI:

- The parent key columns must have a uniqueness constraint supported by a unique index.
- The foreign key must have the same number of columns as the parent key.
- Each column of the foreign key must be defined the same as the corresponding column in the parent key with the exception of the column name, nullability, default value, and check constraints.
- A specific referential constraint can reference only one parent table.

  DB2 does not allow the enforcement of multiple optional parents in an exclusive/or relationship.
- The parent row must be inserted before the dependent row, unless the foreign key of the dependent row is null.

In addition to these requirements, it is highly recommended that an index be created on the foreign key columns of all dependent tables to support the enforcement of the delete rule.

4.2.1 To enforce or not

In some cases, there are relationships in your data model for which referential integrity does not need to be enforced. Typically, these are situations where the data integrity exposure is low or the cost of referential integrity enforcement is high relative to the benefit. If your data model contains any of the following types of tables or relationships, you might consider whether RI needs to be enforced at all:

- Code and look-up tables

  Because code and look-up tables are generally very small relative to their dependent tables, the cost to enforce these values through referential integrity is high. “Code and look-up tables” on page 135 discusses the options for ensuring the data integrity of these values.

- Short term data tables

  Short term data is data that is inserted and deleted over a relatively short period of time, with few accesses or updates. These tables are generally transition in nature, such as log tables, audit tables, or tables that accumulate data for later extraction.

- Derived, aggregated, or redundant data tables

  Tables that are created to store redundant, derived, or aggregated data typically do not need to have referential integrity enforced because data integrity is being maintained on the table from which the data is copied, aggregated, or derived.

- Historical data tables

  In many cases, the business rules might dictate that historical data is retained even after the data in the parent table has been deleted. In this case, obviously, enforcing referential integrity would violate the business rule.

4.2.2 RI enforced by DB2 or enforced by application

In general, DB2-enforced RI is more efficient than application-enforced RI, because it can be performed at the time of the insert or delete without an additional trip to and from DB2. It is also enforced DB2 referential integrity wherever the SQL originates from: static or dynamic applications and home grown or vendor packages.

Therefore, our general recommendation is to use DB2-enforced RI wherever possible.

However, there are some factors that might result in DB2-enforced RI being less efficient than application-enforced RI.

- When multiple dependent rows are being inserted for a single parent, application-enforced RI can perform a single check but DB2 will check for each dependent.
- DB2 RI checking cannot take advantage of index look-aside or sequential detection. If the access to the parent for RI checking is heavy and sequential, application-enforced RI, which can take advantage of both of these efficiency techniques, may be faster.
- If the foreign key of a partitioned table is indexed using a DPSI, DB2-enforced RI will require a probe into each partition looking for a dependent row when a parent row is deleted.

  If the partitioning key is derived from the foreign key so that the application can specify it in the WHERE clause, partition pruning can result in fewer partitions being probed.

There are also several circumstances under which DB2-enforced RI cannot be used, typically as a result of denormalization:

- DB2-enforced RI cannot be used if the entire primary key of the parent is not contained in the dependent table; for example, if a parent table contains a date component to allow the storage of history and the date component was omitted from the dependent tables.

  Another example of this is the denormalized code table shown in Table 1-7 on page 36. The primary key of the denormalized code table contains a type code that should not be carried on the dependent row.

- DB2-enforced RI cannot be used if the foreign key of the dependent table can reference more than one parent, in an exclusive-or relationship.

  An example of this is the table shown in Table 1-8 on page 37 in which the foreign key from the STORE or DISTRICT table is stored in a multi-valued column that will be either a STORE_ID or a DIST_ID.

Application-enforced RI recommendations

When performing application-enforced RI, either by choice or by necessity, consider the following recommendations:

- When inserting into a dependent table, be sure that the parent row is locked when the insert occurs.

  If the application reads the parent table using a singleton select and an isolation of CS, it is likely that no lock will be taken on the parent row or page. In this case, it is possible that another transaction can change or delete the parent row before the insert of the dependent row.

  Even with a cursor on the parent row, lock avoidance may leave the parent row exposed to deletion by other application processes.

  Be sure to read the parent with an isolation level that will guarantee a lock. For more information about locking and isolation levels, see “ISOLATION levels” on page 336.

- When inserting multiple dependent rows for a single parent row, read the parent once and hold the lock on the parent until all dependent inserts are complete.

  Avoiding redundant checks on the parent table is one of the ways that application-enforced RI can outperform DB2-enforced RI.

- When inserting multiple dependent rows, try to insert them in the order of the parent table.

  If application RI checks on the parent occur in a sequential manner, index look-aside and sequential detection can be used to reduce getpages and I/O.

- Even when rows are not ordered by the parent, consider the possibility that the foreign key has not changed.

  When inserting rows that have foreign keys to code or look-up tables, consider the possibility that multiple rows in sequence have the same value in the foreign key. A simple IF statement check comparing the current value to the previous value will be cheaper than a database access or even the search of an in-memory table if the value is the same.

- Use informational referential constraints to let DB2 know that you are enforcing RI.

  This topic is discussed in detail in the next section.

4.2.3 Informational referential constraints

Referential constraints between tables are important factors in determining whether a materialized query table (MQT) can be used for a query. When referential integrity is enforced in the application, DB2 may not have the information it needs to determine whether an MQT can be used to satisfy a query during rewrite.

In this case, informational referential constraints can be used to communicate to DB2 the referential integrity relationships that are being enforced by the application. DB2 will ignore the informational referential constraints during insert, update, delete, load, and check operations, but DB2 will use the informational referential constraint during automatic query rewrite to determine if an MQT can be used.

An informational referential constraint is created by specifying NOT ENFORCED on the foreign key definition of the dependent table.

> **Tip:** Define informational referential constraints for all RI relationships that are enforced in the application, so that DB2 can use this information during automatic query rewrite to use materialized query tables.

4.3 Check constraints

A check constraint is a mechanism in DB2 that can be used to enforce the domain, or the allowed values, of a column. A check constraint can be a simple check against a list of values, or it can be used to enforce transitional relationships that depend on the values contained in other columns contained in the row.

Check constraints are enforced during all insert and update operations on the table, and during the LOAD utility when the ENFORCE CONSTRAINTS keyword is specified.

When defining table check constraints:

- The syntax of a table check constraint is checked when the constraint is defined, but the meaning is not checked, so it is possible to have contradictory conditions in your check constraint.
- A table check constraint is not checked for consistency with other types of constraints or a validation routine that is applied to a table before a check constraint.

4.3.1 Check constraint benefits

Using check constraints can have the following benefits:

- Check constraints prevent inconsistent data from being put into the table using standard data manipulating language (DML).

  If you are unable to force all DML activity through a single application routine that performs data editing, check constraints provide a way to encode the editing into the database.
- Using table check constraints makes programming easier. There is no need to validate routines or program logic to enforce constraints.
- Adding or removing table check constraints does not invalidate plans or packages.

4.3.2 When to avoid check constraints

The cost of DB2 or application constraint checking is similar. However, such as with RI, there are cases where it is better to avoid the use of table check constraints:

- Redundant checking due to RI

  When a row is inserted in a dependent table, then the same table check constraint must be enforced by both primary and foreign key columns. In this situation, you can avoid the checking on the dependent table, so you can eliminate the constraint from this table. A code table is an example of this type of a dependent table.
- Repetitive checking

  This is the case for multiple inserts of the same value, where the application should check the constraint only once. Typically, massive inserts from a batch application can cause this situation.
- Log, audit, and error tables

  Sometimes DB2 tables are used to store the actual values that were entered or received in a value, even if the value failed edits on the base table.
- Derived, aggregated, or redundant data tables

  Tables that are created to store redundant, derived, or aggregated data typically do not need to have check constraints enforced because the checks would already have been performed on the table from which the data is copied, aggregated, or derived.

- Historical data tables

  Historical tables are used to store the values of data at a specific point in time. However, rules governing domains may change over time. Typically, the historical data needs to be retained as it was even if it no longer satisfies the domain requirements.

4.3.3 Check constraints as an option to implementing referential integrity

In cases where the primary key on the parent table only has a few different values, and those values rarely or never change, you can choose to enforce the domain of the foreign key on the dependent table using check constraints instead of RI. This is one option for enforcing code and lookup table relationships.

In this case, the following considerations apply:

- Inserts and updates on the dependent tables get better performance, since no access to the parent table will be needed.
- Whenever a new row is inserted into or deleted from the parent table, the check constraint on the dependent table must be altered.

  This process will place the dependent table into a CHECK PENDING status. If the only change to the check constraint was the addition of a new value, you can safely assume that all of the values in the dependent table are still valid with respect to the check constraint. However, the only recommended way to remove the CHECK PENDING status is with the use of the CHECK utility.

  If you expect frequent insert and deletion of parent table values, RI is a more flexible solution.

4.4 Code and look-up tables

A *code table* is a table where the domain of code columns is defined and the coded information is described. A *look-up table* is a type of code table that is used to replace one or more column values with a key or code that can be stored on the data table. The benefit of a look-up table is typically a reduction in space in the data table and the ability to update values in the look-up table without having to update each referencing row in the data table. In this section when we discuss “code tables”, we are referring to both variations collectively.

In many relational database implementations, the number of accesses to code tables may equal or even exceed the number of accesses to the operational data tables. When referential integrity is used to enforce relationships between code tables and data tables, a single insert into the data table may require accesses to multiple code tables. In addition, code tables are often accessed by applications performing edits or populating pick-lists or drop-down menus.

Because these tables are small and heavily accessed, they will typically stay resident in the buffer pool, which should prevent I/O costs from becoming an issue. However, there is still a cost associated with every access to these tables, so options to reduce these costs should be considered.

In any case, the cost of the accesses to the code tables is not the only headache they cause. The problems are generally caused when referential integrity to the code tables is enforced in the database.

Archived

4.4.1 RI considerations

Whenever database-enforced referential integrity is used, the recommendation is to index the foreign key columns on the dependent table. When relationships exist between code tables and operational data tables, the cardinality of the foreign key columns is typically very low relative to the cardinality of the table. Any index defined only on the foreign key, therefore, provides very little filtering, and is of almost no use from a performance perspective, unless the application has access requirements using the foreign key and additional columns that can be added to the index to make it more unique.

Indexing the foreign keys of code tables may also make the performance of insert and delete activity on the dependent table too expensive because of the additional index maintenance.

Finally, a load replace of a code table can result in multiple large operational tables being placed into CHECK PENDING status. This may present an unacceptable risk to the availability of your database.

Recommendation: Do not implement database-enforced referential integrity constraints between code tables and large operational data tables.

4.4.2 Code table alternatives

There are several alternatives for you to consider when you implement code tables:

- Check constraints can be used to enforce the domain of the foreign key in the database without accessing the code table as described in “Check constraints as an option to implementing referential integrity” on page 135.
- Code table values can be built into copybooks that can be included in programs that perform application-enforced RI checking or require look-up values.
- Batch programs can read code table values into working storage tables once and then use the values repeatedly.
- Code table values can be read from DB2 loaded into in-memory tables, such as CICS data tables, once and then used repeatedly to provide RI enforcement or look-up values.
- Denormalizing data to avoid accessing the look-up tables

  Lookup table data that rarely or never changes can be stored redundantly on the data row to eliminate the need to access the look-up table. This alternative is discussed in detail in “Denormalize look-up table into data table” on page 35.

With any of these alternatives, we recommend that the code tables still are implemented as physical DB2 tables, and that these DB2 tables serve as the master copy of all allowed values. In order to reduce the number of physical tables that must be supported, consider the denormalization described in “Combine multiple code tables into one” on page 35.

5

Techniques to generate numbers

In this chapter, we describe techniques that you can use to generate numbers using features inside the DB2 engine, ranging from normal tables to objects especially designed for the job, such as sequences.

We introduce:

- The different attributes of generated numbers
- Techniques to generate nonunique ordered numbers
- Techniques to generate unique, not ordered, nonconsecutive numbers
- Techniques to generate consecutive numbers
- IMS and CICS functionality to generate numbers

© Copyright IBM Corp. 2005. All rights reserved.

5.1 Introduction

Many online applications require the generation of numbers. These numbers are typically, but not necessarily, used as keys by the application. Number generation can be classified according to a combination of the following three characteristics:

- Do the numbers have to be unique?
- Do the numbers have to be generated in a specific order, typically ascending?
- Must the numbers be consecutive, which means no holes are allowed in the sequence?

Table 5-1 shows the different combinations for unique number generation that will be discussed in more detail later. Nonunique numbers are just a less restrictive case of unique numbers, so we will not discuss those in great detail.

Table 5-1 Characteristics of unique number generation

Ordered	Consecutive	Recommended way to handle
Not ordered	Nonconsecutive	GENERATE_UNIQUE()
	Consecutive	N/A
Ordered	Nonconsecutive	Sequences, identity columns, or GENERATE_UNIQUE() (*)
	Consecutive	Application-controlled
(*) If sequence and identity columns have to be unique, make sure to specify the NO CYCLE keyword		

There are a number of potential problems associated with number generation. For example, when the sequential number is obtained by adding 1 to a counter in a DB2 row, an update *hot spot* can occur and cause contention on that row or page.

We address these potential problems in the next sections.

5.2 Generating nonunique ordered numbers

Ordered (or sequential) numbers that do not have to be unique can be generated in a number of ways, but the most efficient way is to use an *identity column* or a *sequence*.

The identity column or sequence will guarantee ordered numbers within a single cycle. The major difference between an identity column and a sequence is that an identity column is an attribute of a (numeric) column of a table, where a sequence is a standalone object.

For more information about identity columns and sequences, their differences and similarities, see “Identity columns and sequences” on page 141, and the redbook, *DB2 UDB for z/OS Version 8: Everything You Ever Wanted to Know, ... and More,* SG24-6079.

When the generated numbers have to be ordered the majority of the time, but not always, you can allow a sequence or identity column to cycle by specifying the CYCLE keyword.

When the identity column or sequence cycles, the next generated value will be the MINVALUE. The MINVALUE is lower (or higher in case the identity column or sequence generates descending numbers) than the previous value (MAXVALUE) that was generated. If the application requirements can tolerate this, it is acceptable to allow the identity column or sequence to cycle. If this is not allowed, you define the identity column or sequence as NO CYCLE, and you receive an SQLCODE when the maximum or minimum value is reached.

Tip: The initial value of a sequence or identity column is the START WITH value, where the first value used after a cycle is the MINVALUE.

5.3 Generating unique not ordered nonconsecutive numbers

There are several ways to generate not ordered nonconsecutive numbers. In this section, we describe a number of alternatives from which you can choose.

5.3.1 GENERATE_UNIQUE()

When generating unique numbers, the cheapest way is to use the GENERATE_UNIQUE() DB2 built-in function. It is available in V8 and V7 with PTF UQ86840 for APAR PQ70901 applied. The result of an invocation of the GENERATE_UNIQUE() function is a CHAR(13) FOR BIT DATA string. The result is guaranteed to be unique, even in a data sharing environment. Example 5-1 shows an example of using the GENERATE_UNIQUE() function.

Example 5-1 Using the GENERATE_UNIQUE() function

```
CREATE TABLE TB1(GEN_COL CHAR(13) FOR BIT DATA);
INSERT INTO TB1 (GEN_COL) VALUES ( GENERATE_UNIQUE() );
SELECT GEN_COL
     , HEX(GEN_COL) AS GEN_COL_IN_HEX
     , TIMESTAMP(GEN_COL) AS TIMESTAMP_REPRES
  FROM TB1;
---------+---------+---------+---------+---------+---------+---------+
GEN_COL        GEN_COL_IN_HEX             TIMESTAMP_REPRES
---------+---------+---------+---------+---------+---------+---------+
.¨&o.$-......  00BD50962B5BCA3D0000012801  2005-07-16-00.03.51.316643
```

The actual generated string is not readable, but when converted to hexadecimal, the trained eye can see that this value is based on the TOD clock. This means that the value can be converted into a DB2 timestamp using the TIMESTAMP() function. So, even though the actual value in the table is not readable, it can be easily converted to a readable timestamp whenever required, as shown above. Because the result of the GENERATE_UNIQUE() is based on a timestamp, its values will also be ordered. Therefore, this function can also be used to generate unique ordered numbers that we discuss in “Generating unique ordered nonconsecutive numbers” on page 140.

5.3.2 ROWID

DB2 Version 6 introduced the ROWID data type. ROWID is an SQL data type, which returns a 40 byte varchar value, which is not regularly ascending or descending. It is the basis for direct row access and accessing LOB data, because the use of a LOB column requires that there is also a column defined (either explicitly specified at CREATE time, or implicitly created by DB2 under the covers in Version 8) with a ROWID data type.

However, you can also use the ROWID data type without using LOBs. You can just define a column with a ROWID data type. A ROWID is guaranteed to be unique.

When you create a ROWID column, you also specify the GENERATED attribute. You have two options:

- GENERATED ALWAYS

  This means that the application itself is not allowed to specify a value for the ROWID field, and the value always has to be generated by DB2. If you need to make a schema change that requires you to DROP/CREATE the table, the ROWID values are generated again (because of the GENERATED ALWAYS attribute) when the data is reloaded into the table, which is usually not what you want to happen. Therefore, we do not recommend the use of GENERATED ALWAYS when a ROWID column is not used together with LOBs.

- GENERATED BY DEFAULT

  This means that the application itself is allowed to specify a value for the ROWID field. When a value is not provided by the application, DB2 will generate a value. It is important to note that when you use GENERATED BY DEFAULT, the ROWID column has to have a unique index defined to guarantee uniqueness. (This is not the case when using GENERATED ALWAYS). DB2 also checks to see if the ROWID value that you provide, when using GENERATED BY DEFAULT, is a valid ROWID. If not, you receive an SQLCODE -399.

A ROWID is guaranteed to be unique, but cannot be easily converted into a timestamp such as a value generated by the GENERATE_UNIQUE() function. A ROWID value is also strange in that it has an internal and external representation. Example 5-2 shows what the external representation of a ROWID looks like.

Example 5-2 Using a ROWID data type

```
CREATE TABLE TB1(ROWID_COL ROWID NOT NULL GENERATED BY DEFAULT);
CREATE UNIQUE INDEX IX1 ON TB1(ROWID_COL)
INSERT INTO TB1 VALUES (DEFAULT);
SELECT * FROM TB1;
---------+---------+---------+---------+---------+---------+--
ROWID_COL
---------+---------+---------+---------+---------+---------+--
B2E16CAE50ABD76B2104015C56300100000000000201
```

A ROWID value generated by DB2 is guaranteed to be unique, but the values are not necessarily ordered. A ROWID value cannot be transformed into a readable timestamp.

Because of the different representations, because you need a unique index on the ROWID column when you use GENERATED BY DEFAULT, and because you cannot easily convert a ROWID value into something meaningful, such as a timestamp, a ROWID is not suitable for generating unique numbers, and, therefore, we do not recommend its usage for that purpose.

We only recommend the use of ROWID when using LOBs, because it is mandatory, or to be able to exploit direct row access. The GENERATE_UNIQUE() function is a better option to generate unique numbers.

5.4 Generating unique ordered nonconsecutive numbers

Since numbers generated as discussed in “GENERATE_UNIQUE()” on page 139, are timestamp-based, the next number will always be greater than the previous one. Therefore, GENERATE_UNIQUE() can also be used to generate ordered numbers.

Other ways to generate ordered unique values that do not need to be consecutive are:

- Inserting the CURRENT TIMESTAMP, and retry when a duplicate is encountered
- Using identity columns
- Using sequences

5.4.1 Using CURRENT TIMESTAMP

An easy way to generate unique ordered numbers is to use the CURRENT TIMESTAMP special register. In the table, define a column with a TIMESTAMP data type and define a unique index on it. When you INSERT CURRENT TIMESTAMP into a table, and the same value already exists, DB2 returns an SQLCODE -803 (duplicate key). The application can then retry the insert with a new CURRENT TIMESTAMP value.

This solution works, but requires a unique index to guarantee uniqueness, which is extra overhead to maintain the index. In addition, since we are inserting keys with an ever increasing value, frequent index page splits are required, and again require extra processing. In an environment with a high number of parallel inserts, the chances of having to retry increase, again requiring extra processing.

Therefore, the use of GENERATE_UNIQUE() is a less expensive option to achieve the objective of generating unique ordered, but nonconsecutive numbers.

Even if it satisfies the application requirements, we do not recommend that you consider using the timestamp option.

Also, note that the CURRENT TIMESTAMP is a local time, and you may get into trouble when switching back from daylight savings time. You should use the GMT time, instead of local time. In addition, if you really must use a timestamp, it is cheaper to obtain it in the application than requesting if from DB2.

5.4.2 Identity columns and sequences

Identity columns and sequences generate unique numbers for the domain, as long as the identity column or sequence is not allowed to cycle; in other words, it was defined with the NO CYCLE keyword (and provided you did not use INCREMENT BY 0, a new option in DB2 Version 8, which allows for the generation of a fixed number).

Identity columns and sequences provide a performance benefit over using a single row table with a counter that is being updated every time, especially when using the CACHE option.

> **Tip:** Use the CACHE option to minimize the number of times that DB2 needs to update the catalog. The default value is **20**, but you may want to use a higher value, especially in a high insert environment where the identity column or sequence is used to generate a (unique) number. After all the values in the cache have been handed out, the CACHE value needs to be updated, and forced to the log (in both data sharing and non-data sharing).

Identity columns

To guarantee uniqueness for an identity column, you can either use:

- The GENERATED ALWAYS attribute. This way, DB2 will always generate the number at INSERT time.
- A (single column) unique index on the column defined with the AS IDENTITY attribute.

Using GENERATED ALWAYS

When identity columns were first introduced, using GENERATED ALWAYS guaranteed that the DB2 generated number that was inserted into the table was unique. With the new identity column features that were introduced in later DB2 versions, this statement is no longer true.

- DB2 Version 7 introduced the CYCLE keyword. This means that when the MAXVALUE of the identity column is reached, DB2 cycles back to the MINVALUE. Therefore, if you need unique values in the column that is defined AS IDENTITY, make sure that the identity column is not allowed to cycle, or make sure by the time it cycles, that the values from the previous cycle have been removed.
- DB2 Version 8 allows you to change the identity column from GENERATED ALWAYS to GENERATED BY DEFAULT. After switching to GENERATED BY DEFAULT, DB2 can no longer guarantee that the values in the identity column are unique.
- DB2 Version 8 also introduces the RESTART WITH xx option. This allows you to indicate that the next value that is to be generated for this identity column will be 'xx'. Again, DB2 will no longer be able to guarantee uniqueness for the identity column's value.

Therefore, the only way to guarantee uniqueness is to define a unique index on the identity column.

Sequences

A sequence does not have a GENERATED ALWAYS attribute, but it can also be allowed to CYCLE and be RESTARTed WITH a specific value. In addition, when using a sequence, the application is in charge of when and how the sequence value is used. There is nothing inside the DB2 engine that prevents you from using the NEXT VALUE FOR seq-name only once, but then insert two rows with the same value that was retrieved from the sequence number.

Again, the only way to insure uniqueness is to define a unique index on the column that is used as a target for the retrieved sequence value.

Identity columns versus sequences

Before DB2 Version 8, identity columns were difficult to use. The major reason is that once an identity column was defined, you could not make any changes to it without dropping and recreating the table. However, these restrictions have been removed in Version 8, and you can alter any attribute related to an identity column, except for the data type.

For this reason, most pre-Version 8 customers decided to only define an identity column on a table with no real data. Basically, applications would insert a row into a dummy table with only an identity column defined, retrieve the number that was generated by DB2 using the IDENTITY_VAL_LOCAL() function, and use the returned value in the actual tables that contain data. This is a "simulation" of a sequence using an identity column. Since Version 8 allows you to change all the identity column attributes, this technique is no longer required. You can either use plain identity columns and alter the attributes when required, or use sequences in DB2 Version 8.

Important: The use of the CACHE option in data sharing may lead to nonordered number generation. If ordering is absolutely necessary, use the NO CACHE or ORDER keyword. (ORDER forces NO CACHE irrespective of the value specified at CREATE or ALTER time.)

Important: However, be aware that the use of NO CACHE or ORDER in a data sharing environment causes a lot of overhead, because we have to ping-pong the page of the catalog containing the identity column or sequence information back and forth between different members of the data sharing group, push the page to the group buffer pool, and perform cross-invalidation, each time a new value needs to be generated. Also, MAXASSIGNEDVAL in SYSIBM.SYSSEQUENCES has to be updated. In addition, each time you update MAXASSIGNEDVAL, this requires a forced write to the log.

Following are characteristics of identity columns and sequences that can influence your decision about which one to use.

Identity columns

- You can only have one column that is defined AS IDENTITY per table.
- AN IDENTITY columns cannot be defined as NULLS ALLOWED.
- The table on which the identity column is defined cannot have an EDITPROC.
- The value is generated at insert time, when defined as GENERATED ALWAYS, and you have to retrieve it, either using:
  - The IDENTITY_VAL_LOCAL() function, or
  - SELECT EMPO FROM FINAL TABLE (INSERT INTO EMP VALUES (...)), where EMPNO is defined as an identity column.

  Note: When using the SELECT FROM ... FINAL TABLE INSERT construct, there is only a single trip from the application into DB2 and back, where when using the INSERT and IDENTITY_VAL_LOCAL() function, you have to go across the DB2 API twice to achieve the same result. We recommend the FINAL TABLE construct as the preferred way to retrieve an identity column's value.

- Using GENERATED ALWAYS on an identity column has the advantage that the DBMS is in control when the value is generated, and you do not have to rely on the application to do a good job.

Sequences

Sequences are not tied to a table, and they allow for more flexibility. You can retrieve a value from the sequence separately from the actual INSERT statement, for example using:

```
SET :HV = NEXT VALUE FOR myseq
```

This gives you the opportunity to manipulate the value returned by the sequence before inserting it into a table. It also makes life easier when dealing with RI structures. For example, you can retrieve the value for your primary key from the sequence, and insert parent and child rows using that value. There is no need to insert the parent row, retrieve the value generated by DB2 for the identity column, and use that value to insert child records.

Some installations use the number that was retrieved from the sequence number as input to a randomizer. The result of the randomization is the partition number into which the row is inserted. This allows you to evenly distribute insert activity over a number of partitions. This is a valid technique to process a high volume of inserts. However, be careful when using this technique if the rows need to be retrieved afterwards. Since the partition number is

“randomly” generated, a retrieval application has no means of knowing in which partition the actual data was inserted. So, you cannot specify the partitioning key as part of the WHERE clause, and DB2 may have to look into multiple partitions to retrieve the row. We only recommend this randomization technique based on a sequence, if you are only interested in spreading inserts across multiple partitions to handle massive insert volumes. And, you retrieve the data using a sequential process later.

In other cases, use a natural key as input to the randomizer to spread the inserts. For example, if you need to retrieve an order and you know the order number, you can call the randomizer module, and use the partition number information (the output of the randomizer) as an additional WHERE predicate.

Since a sequence is a standalone object, and is not tied to a table, there is no way to know which sequence is used for which table. These dependencies need to be tracked outside of the DB2 system.

Data and database information are tied together

Do not forget that when you use sequences or identity columns, part of the information is stored in the catalog. This means that after certain operations, you need to “resynchronize” the data with the information about the sequence or identity column that is kept in the DB2 catalog, such as:

- When you LOAD additional data into a table that has an identity column (using GENERATED BY DEFAULT), or is based on values from a sequence, you must ALTER the identity column or sequence to reflect that. For example, if a sequence is generating sequential numbers, 1, 2, 3, ... and you are at 100 (MAXASSIGNEDVAL in SYSIBM.SYSSEQUENCES is 100). When you load new data with values 101 to 200, you must update the sequence to reflect this, so the next number that is retrieved from the sequence by the application is 201. To do so, you use:

  ```
  ALTER SEQUENCE myseq RESTART WITH 201;
  ```

  For an identity column, you need to use:

  ```
  ALTER TABLE mytab ALTER COLUMN ident_col_name RESTART WITH 201;
  ```

- When you do:
  - Point-in-time recovery of your table, you also need to “reset” the information to the same point to which the data is restored.
  - A schema change on a table that requires the table to be dropped and recreated. When the table is recreated, do not forget to execute an additional ALTER to RESTART the identity column with the correct value. Note that you only need to perform this action for identity columns. Sequences are not affected when a table that uses a sequence is dropped and recreated.
- Sequences have a disadvantage here since there is no easy way to find out which tables are using a certain sequence, and it is your responsibility to administer this, for example, by storing this information in the REMARKS field of SYSIBM.SYSSEQUENCES via the COMMENT ON SEQUENCE seq-name IS ‘string’ statement.

5.5 Generating consecutive numbers

This is the most difficult type of number generation, because consecutive means that we are not allowed to have any holes.

None of the standard DB2 functionality such as identity columns or sequences are able to guarantee consecutive numbers. When using sequences, after a NEXT VALUE FOR

seq-name has been executed, the value is considered consumed by DB2, irrespective of whether this value ever ends up in a table or not. The same applies for a rollback. If the application decides to roll back the unit of work, the value obtained from the sequence is NOT put back into the system so it can be reused. The same applies to identity columns. If you insert a row into a table with an identity column, for example, defined as GENERATED ALWAYS, DB2 assigns a value for that column. When the transaction is rolled back, the insert is rolled back and the assigned number is lost, and the list of generated numbers is no longer consecutive.

To solve this problem, we have to rely on application techniques. Application techniques are usually quite a bit more expensive in terms of resource usage than having DB2 implement a certain functionality. Therefore, whenever an application requests consecutive numbers, the first thing to do is to challenge the requirement. Is it absolutely necessary that the numbers are consecutive? In most cases it is not, but in other cases, for example, for legal or tax reasons, consecutive numbers are mandatory. A company is usually not allowed to have gaps in their invoice numbers. It makes the tax authorities suspicious since this usually indicates fraud.

If the application really needs to generate consecutive numbers, you can use one of the following techniques.

5.5.1 Using the MAX() function

When an application needs to derive the next value to assign, for example, to assign a new invoice number, you can select the current maximum invoice number from the table and add one to that number. You can use:

```
SELECT MAX(INVOICE_NUMBER) FROM INVOICES WITH RR
```

Although this may seem tempting, it is an extremely bad solution. It almost certainly leads to a deadlock situation, if multiple instances of this application are running simultaneously as shown in Example 5-3.

Example 5-3 Deadlock using SELECT MAX with RR

```
Tran 1                                      Tran 2

Performs SELECT MAX   -> OK

                                            Performs SELECT MAX -> OK

Tries to insert       -> Wait because of
                         RR-lock held by
                         Tran 2

                                            Tries to insert     -> Also waits

And we now have a deadlock situation
```

5.5.2 Using a single row table

Instead of using the actual table that contains the data to determine the next value, we can also create a little table that just contains the counter for the invoice numbers. An application that needs to assign a new invoice number goes to that table and adds one to the value that is currently in there.

You can obtain the sequential number by updating and retrieving a column used as a counter in a row. Assume that you have a COUNTER_TBL table, which contains only one row with a counter value in the SEQ_NBR column. You first UPDATE the row to avoid anybody else from doing the same. Then you select the value back that you just updated using a singleton select. Then you perform the application logic that uses the SEQ_NBR value, and COMMIT. Example 5-4 also shows using a single row counter table with UPDATE first.

Example 5-4 Using a single row counter table with UPDATE first

```
UPDATE COUNTER_TBL SET SEQ_NBR = SEQ_NBR + 1
SELECT SEQ_NBR INTO :SEQ_NBR
Application processing logic inserting into ORDER table for example
COMMIT (the end of the transaction)
```

Example 5-5 shows another technique that is sometimes used.

Example 5-5 Using a single row counter table with retrieval first

```
DECLARE C1 CURSOR FOR SELECT SEQ_NBR FROM COUNTER_TBL FOR UPDATE OF SEQ_NBR
OPEN C1
FETCH SEQ_NBR INTO :SEQ_NBR
UPDATE COUNTER_TBL SET SEQ_NBR = SEQ_NBR + 1 WHERE CURRENT OF C1
CLOSE C1
Application processing logic inserting into ORDER table for example
COMMIT (the end of the transaction)
```

You first fetch the row with a cursor to protect the row after the counter has been retrieved. To avoid deadlock problems, define the cursor with the FOR UPDATE OF clause. Then the update of the SEQ_NBR is performed. The big difference between Example 5-4 and Example 5-5 is the number of SQL statements necessary. Since the second example uses a cursor, it requires additional trips to DB2 for the OPEN and CLOSE of the cursor. Therefore, we recommend using the method described in Example 5-4.

When updating the counter, it is important to commit as soon as possible after updating the counter to free the X lock on the data page or row as soon as possible. Therefore, you may be tempted to split the updating of the counter, and the insert into the table that uses the counter number (table with the data) into two units of work, as we see in Example 5-6.

Example 5-6 Separating the counter update from its use

```
UPDATE COUNTER_TBL SET SEQ_NBR = SEQ_NBR + 1
SELECT SEQ_NBR INTO :SEQ_NBR
COMMIT
Application processing logic inserting into ORDER table for example
COMMIT (the end of the transaction)
```

However, doing so has the following disadvantages:

- If a transaction abends in the second unit of work (after the first commit), a hole is created in the sequence of generated numbers. You also have to be sure that no other updates get committed by the first commit, for example, updates performed earlier in another subprogram that called this program, but really belong together with the data inserted into the order table.
- Commit requires additional resources, so committing too frequently unnecessarily increases CPU usage and forces logging.

Therefore, keep the logic of updating the counter table and using the counter in your “data” tables in the same commit scope.

5.5.3 Handling multiple counters

If you are considering having more than one counter, you can have a table with multiple rows, one for each counter in a segmented table space.

Do not place the counters in different columns in the same row, because, in that case, they are always locked together. To make sure the counters are placed in different rows in different pages in the same table, define the table space with MAXROWS 1. This way there is no need for row level locking, and you do not lock rows from other counters. Using MAXROWS 1 also eliminates the page P-lock overhead of row level locking in a data sharing environment.

If the rows are forced to different pages of the same table, it is still possible that lock contention occurs.

When, for example, to access row two in page two, a table space scan can occur. The scan stops to wait at page one, if page one is locked by another transaction. The situation is the same when you use row locking. You should therefore avoid using a table space scan. To do so, you define an index on the column that identifies the counter. To “guarantee” that a (matching) index access path is used, define the table as VOLATILE.

So the design can look like Figure 5-1.

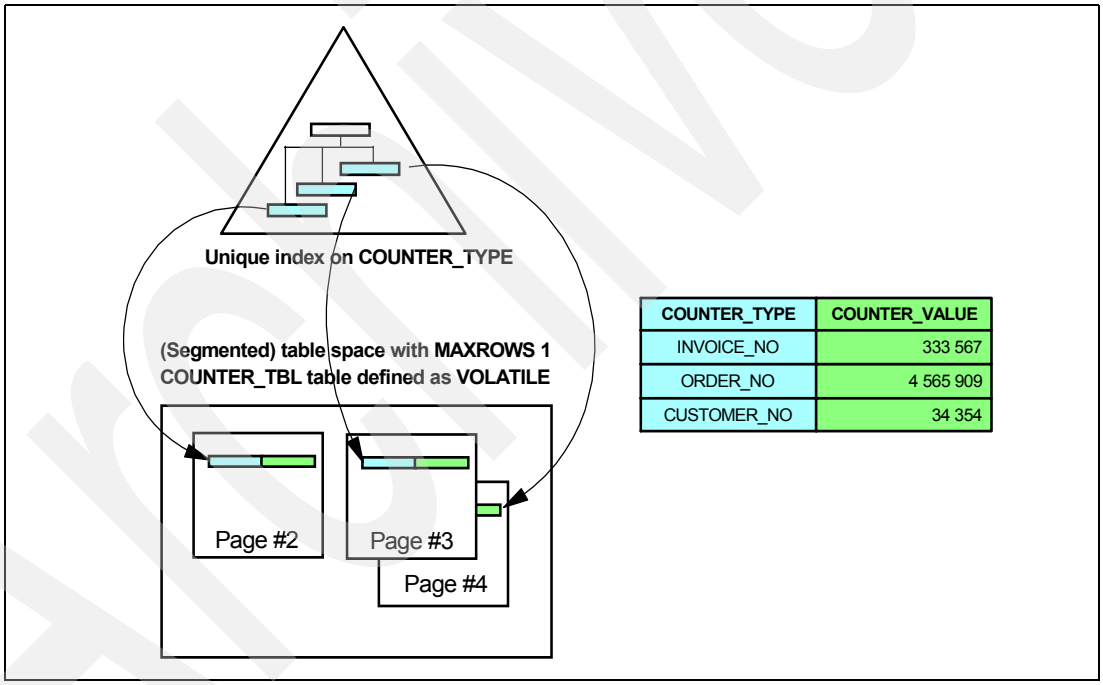


Figure 5-1 COUNTER_TBL design

You define a COUNTER_TBL table with one row per page, and each row represents a counter that needs to produce consecutive numbers. The index is defined on the COUNTER_TYPE column, so it is not being updated. The volatile attribute indicates to the optimizer to use a matching index scan whenever possible. This way we avoid locking counter types that our transaction does not update but are in the same table.

Since we have a lot of transactions that go after the same COUNTER_TYPE, for example, many simultaneous transactions that need to obtain an invoice number, we use the same techniques that we use to deal with data hot spots. We describe these techniques in more detail in “Hot spots” on page 353.

5.5.4 Using direct row access to access the counter table

To further reduce the number of getpages, you may want to add a ROWID column to the COUNTER_TBL, and “force” direct row access for the individual rows of the table. This way, we can avoid the two index getpages. To be able to use direct row access, you have to provide DB2 with the value for the ROWID of the row that you want to retrieve. Since the rows never change ROWID or location (page-rid) in the page set, you can store the values of the ROWID column in a copybook and include that in all programs that need to update any of the counters in the COUNTER_TBL. We recommend that you leave the index in place, just in case, the optimizer decides not to use direct row access, since it can still use the matching index access path instead.

Note that whenever the table space that contains the COUNTER_TBL gets reorganized (although there is no real reason to do so), it makes the ROWIDs that are stored in the copybooks invalid, and you have to redo the copybook and recompile all programs that use it.

5.5.5 Handling consecutive numbers with very high transaction rates

When you have a very high transaction rate, continuously updating a counter table has its limitations. Although we describe several variations previously, the technique usually consists of an UPDATE of the counter, SELECT from the counter, INSERT the actual data row, and a COMMIT. The lock on the counter table is only released at commit time. Assuming a commit takes 1 to 4 milliseconds (dual forced write to the log), you can only do 250 to 1,000 transactions per second at a maximum.

If you exceed that number, you have to challenge some of the requirements to be able to make this work. Some considerations may be:

- Do the numbers really have to be consecutive? If there is no legal requirement to do so, use identity columns or sequences. They can get a much higher throughput.
- Do all the numbers have to be generated right away? Let us look at the following example. Bank withdrawal numbers have to be consecutive. When you withdraw money from an account and want to take home a bank statement immediately, the ATM or teller can print the statement. The statement has to contain a withdrawal number, so one has to be assigned immediately, and the counter table is used for that. In case you do not want to take home your bank statement immediately, the system can put in a null withdrawal number at withdrawal time, and just use a system-generated confirmation number (GENERATE_UNIQUE()), that does not have to be consecutive. Later on, a batch process can go in and update all the null statement numbers with valid consecutive statement numbers. The batch program can grab 200 numbers at a time before committing, so it can achieve a much higher throughput than individual updates of the counter table can.

5.6 Generating numbers without using DB2

When you are using IMS or CICS, you can also use facilities built into those products to assist with the generation of sequential numbers.

Sequential numbers in IMS

In IMS, you can use Fast Path main storage databases (MSDBs) to generate sequential numbers. Since IMS V5, you can also use the virtual storage option (VSO), which:

- Has the same function as an MSDB
- Achieves similar performance as an MSDB
- Is operationally easier than an MSDB

- Can be shared between systems in an IMS data sharing environment, where MSDBs cannot be shared between IMS systems

VSO and MSDB databases are very efficient. However, if a single counter is updated from different IMS regions, there is still a possible contention problem.

Sequential numbers in CICS

In CICS/ESA®, you may have a counter in main storage, common work area (CWA). In this case, the transaction gets the sequential number and updates the counter before any transaction switch takes place. This method is very efficient and does not cause contention, however:

- There is no CICS/ESA backout for main storage, so holes may appear in the case of a transaction abend.
- There is no CICS/ESA recovery of main storage. The application must recover the counter if CICS/ESA region abends.
- Recovery becomes problematic in a multi-region operation (MRO) environment if you have to keep the same counter across MRO regions. You need to have the counter at a terminal owning region (TOR) and obtain the sequence number when the transaction is received. This can be done in a terminal control program (TCP) or CICS task management (KCP) user exit.

CICS 3.1 provides *named counter servers* as an efficient way of generating unique sequence numbers for use by applications in a Parallel Sysplex environment. A named counter server maintains each sequence of numbers as a named counter. Each time a number is assigned, the corresponding named counter is automatically incremented so that the next request gets the next number in sequence. Named counters are the Sysplex equivalent of COUNTER in the Common System Area (CSA) of a single region CICS system.

A named counter server provides a full set of functions to define and use named counters. Each named counter consists of:

- A 16-byte name
- A current value
- A minimum value
- A maximum value

Named counters are stored in a pool of named counters, where each pool is a small coupling facility list structure, with keys but no data. The pool name forms part of the list structure name. Each named counter is stored as a list structure entry keyed on the specified name, and each request for the next value requires only a single coupling facility access. The counters are lost if the coupling facility fails.

The values are internally stored as 8-byte (double word) binary numbers, but the user interface allows them to be treated as any length from 1 to 8 bytes, typically 4 bytes.

CICS provides a command level API for the named counter facility. A CALL interface makes the named counters available to batch applications.

For information about accessing the counters, how to create a list structure for use as a named counter pool, and on counter recovery techniques, see the *CICS Transaction Server for z/OS V3.1: Application Programming Guide*, SC34-6433.

Archived

6

Database side programming

In this chapter, we discuss the different options DB2 provides in terms of server side programming techniques. We cover the use of stored procedures, user-defined functions, and triggers. The major focus of this chapter is to provide guidance on these objects in terms of performance expectations and usability. It is not the intent to provide a full coverage of all the syntax and creation aspects of these objects.

This chapter contains the following topics:

- Overview of server side programming options
- Using stored procedures
- Using user-defined functions
- MQListener in DB2 for OS/390 and z/OS
- Using triggers

© Copyright IBM Corp. 2005. All rights reserved.

6.1 Overview of server side programming options

Relational databases are not just providing a persistent store for rows and columns. Over the years, several server side components are added that allow the program to push logic inside the database engine. The design of DB2 also positions the engine as a gateway into any kind of information that is available in the enterprise and even outside the boundaries of your company. The SQL language has been extended and allows encapsulation of just about any piece of information that you can think of. The only language the developer has to know is SQL and the complexity of the underlying data source is completely hidden.

See Figure 6-1 for an overview of all the options available to the program connected to DB2.

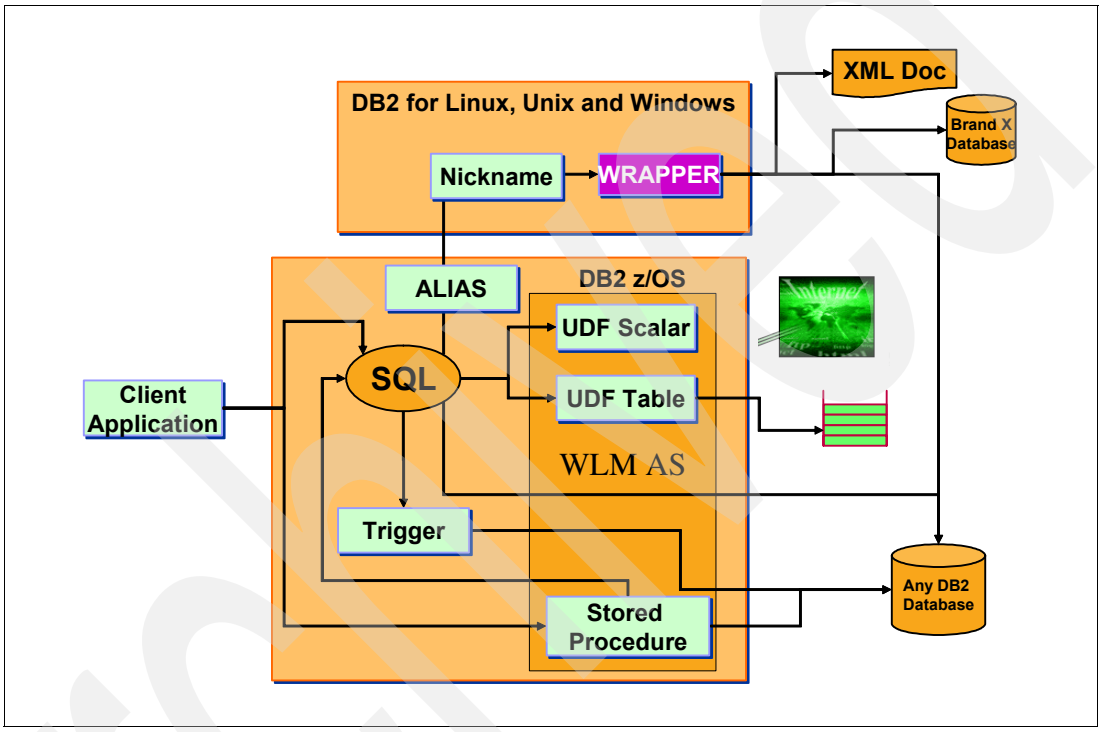


Figure 6-1 Overview of server side programming options

Expanding the reach of DB2 by using WebSphere Information Integrator

By using user-defined functions (UDF), the server side programmer can provide an SQL interface to the data source that is accessed in the UDF. Since the UDF will be running in a WLM-established address space, the data that can be accessed from within the UDF is the whole range of data sources available on z/OS. You may need to access information that is sitting on platforms other than z/OS. This kind of functionality requires the use of WebSphere Information Integrator. IBM has decided not to implement the concept of distributed request in DB2 for z/OS. That means that it is possible to access information that is on other database servers from within a single unit of work, but that using data from more than one resource manager in a single statement is not possible.

So the way to get around the problem of distributed request and the problem of including non-z/OS data sources is by adding an alias, or connect to DB2 Information Integrator (II). Once you are in DB2 II, use the concept of a nickname. The nickname is similar to an alias but with the difference that nicknames support distributed request. This means that it is possible to join between different resource managers when performing a select. The nickname communicates with the remote data source through the wrapper. The data source can be a diverse list of data sources ranging from all kinds of relational databases to flat files,

life science data, XML structures, and many more. The wrapper is providing the communication bridge with the target data source. You can think of the wrapper as a generic table UDF that deals with the communication to the data source, but it is much more than that. The huge difference between the wrapper and UDF table function is the integration with the optimizer and the ability to do query pushdown into the wrapper. The optimizer has a significant amount of information on the object referenced in the nickname. This information includes cardinality information, defined indexes, semantics of the SQL supported by the data source, network speed, and relative processor speed of the remote server. This means that the optimizer is able to perform a profound analysis of the pushdown options of the SQL.

DB2 for z/OS offers a lot of server side processing techniques, which we discuss in the rest of the chapter. Bear in mind that distributed requests or processing data source, that cannot be processed directly from within z/OS, require the deployment of DB2 II. The DB2 II product is also available in a Linux® for zSeries® package, so it is possible to do all the processing on the zSeries with a low latency when deploying HyperSockets between the z/OS partition and the zLinux partition.

6.2 Using stored procedures

DB2 and Distributed Relational Database Architecture™ (DRDA®) provide support for stored procedures to give application designers flexibility in designing their client/server applications. A stored procedure is an application program that is stored at the DB2 server and can be invoked by the client through the SQL CALL statement. Information about stored procedures can be found in the redbook, *DB2 for z/OS Stored Procedures: Through the CALL and Beyond,* SG24-7083.

6.2.1 Why use stored procedures?

You can increase the security level of your application code by encapsulating the application logic in a stored procedure. This means that the client is only aware of the call to the stored procedure and the result set. Any intermediate logic and data is hidden from the client. An example is a Web application that has interest calculation of a mortgage in an applet. The competition can take the code in the applet and determine how you calculate the cost of the mortgage, copy the code, and subtract half a percent from the value our applet calculates. Subsequently, they would run an ad in the newspaper that their bank is always half a percent cheaper. By replacing the code in the applet by a call to a stored procedure, the risk of exposing the business logic is removed.

Another case where stored procedures are useful is for client programming languages that do not support static SQL. A program written in VB .NET may call a stored procedure that is statically bound to use static SQL instead of dynamic. The network calls are significantly reduced by moving the application code to the same machine as the database. This way you just need to call the procedure and all subsequent database accesses from within the stored procedure are performed without involving the network. See Figure 6-2 on page 154 for an example of a stored procedure that saves network trips by executing four SQL statements on z/OS.

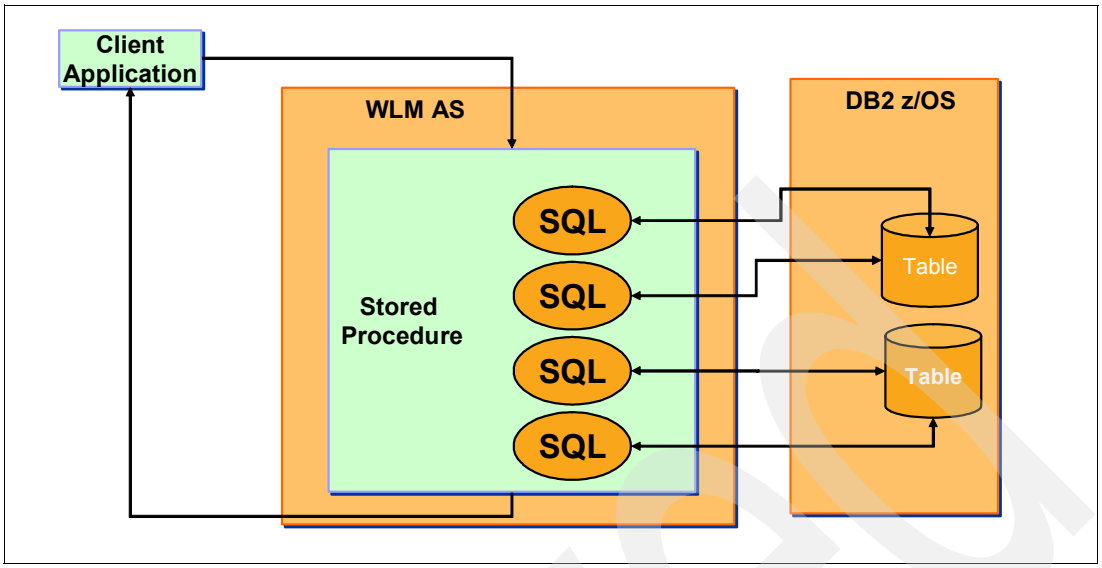


Figure 6-2 Example of a stored procedure executing four SQL statements

Abusing stored procedures

Some people migrate from other relational databases to DB2, and while doing so, they end up copying the behaviors of other products into DB2.

An example of this behavior is the implementation of a stored procedure which has only the code for a declare cursor WITH RETURN and an OPEN of that cursor. This stored procedure goes through all the overhead of being established, just to open the cursor. The reason for the original choice was that the previous database was more efficient when you added the SQL in a procedure. DB2 provides an efficient cursor implementation, even without the stored procedure.

A second reason people come up with such an implementation is because they want to use static SQL and do not know all the options DB2 is delivering. An example is a Java programmer who codes using JDBC™ and puts the cursor in the stored procedure to get static SQL. It is far more efficient to code the client Java code by means of SQLJ, which puts all the client SQL in static SQL without even calling the stored procedure.

With stored procedures, restrict your result set to a significant set. Use the power of SQL and drive around 10 statements to keep the overhead low and have good performance. Also keep in mind that a stored procedure is just like a batch program; if you access resources outside of DB2, it is the responsibility of the application (yours) to handle the consistency of the data.

Wrapping CICS and IMS transactions

New client/server workloads need to interact with strategic assets, such as information that is managed through CICS or IMS. Stored procedures are an ideal way of wrapping these assets and making them look as if they were relational data.

Interfaces exist to call CICS transactions or access IMS databases. The information retrieved from these resources can be returned as an output parameter assuming it is only a limited number of values. If the output consists of more complex data, the usage of a DTT may be a better option to return the information to the calling application. See “Performance expectations and tips” on page 158 for a comparison of the different types of temporary tables than you can use.

Synchronous versus asynchronous calls

The call to a stored procedure is always a synchronous call. This means that the calling client application is waiting while the procedure is executed.

DB2 also benefits from the WLM infrastructure for dynamic load balancing adaptation, where WLM starts up additional address spaces when the parallelism in the workload increases.

The disadvantage of the synchronous call is the tight coupling between the caller and the stored procedure infrastructure. If your stored procedure is unavailable, the caller receives an error or timeout on the call. In some cases, we just need to pass information to another process, without depending on the outcome of the other process. Calling a synchronous process to just pass information is probably not the most appropriate approach.

The SQL that is implemented in DB2 for z/OS does not provide an asynchronous option of calling process logic. We need to revert to another solution to pass information to another process in an asynchronous way. Several options are available, such as using WebSphere MQ messaging or using a DB2 table to serve as a buffer between the two processes. Replication solutions can also be used to trigger processing based in an event in DB2.

Using MQ

WebSphere MQ is the *de facto* standard for asynchronous message passing. The performance characteristics of MQ are similar to DB2. The cost of issuing a PUT call is in the same order of magnitude as a DB2 INSERT. The major difference, however, is the behavior of locking, as there is no need to worry about timeouts and concurrency between the PUT and the GET.

If set data needs to be passed between the processes, an additional overhead is incurred, as the MQ message is basically a string. So, if the reply is the result of a cursor, a set, then the rows need to be fetched, concatenated into a long string which is used for the PUT. The receiver then needs to cut the string into pieces and recompose the set. This processing requires additional overhead when compared to the stored procedure alternative since this just needs to add WITH RETURN on the cursor to pass the information to the calling application. The MQ server has the ability to start a user process when a message arrives on the queue. See “MQ interface functions” on page 163 for more details about how to use the DB2 MQ functions.

Using a DB2 table

Use a DB2 table to pass information between processes. The sender performs an insert in the table and the receiver performs a select. Concurrency is an issue if the latency of the information passing is an issue. The insert keeps a lock until commit, which means that the program issuing the select to look for any new rows can suffer timeout when the inserter is not committing frequently. The DSNZPARM SKIPUNC alleviates this problem when the selector is using CS or RS and the table is defined with row locking.

Tip: SKIPUNC is introduced by APARs PQ92833, PQ82390, and PQ79789.

See Chapter 12, “What you need to know about locking” on page 327 for details about locking.

If a low latency between the two processes is required, the DB2 table option is probably not the best approach, since the selecting process needs a very high polling frequency because there is no automatic dispatching upon insert in comparison to the MQ-based solution.

Using replication

If the information that needs to be moved to the other process is already inserted or updated in a DB2 table, replication may be the best way to solve the problem. The cost of capturing the change is totally asynchronous, since it is performed by the capture program. The subsequent invocation of the receiving process can be achieved by opting for the MQ-based replication architecture. Using the MQ-based configuration capture performs the put of the changes upon processing of the commit. Refer to the redbook, *WebSphere Information Integrator Q Replication: Fast Track Implementation Scenarios*, SG24-6487, for information.

6.2.2 What flavors can we use?

Different programming languages give different performance results, but performance is not the only consideration when selecting a programming language for the stored procedure.

There are other considerations such as code reuse. One example is a customer coding all their business logic using Java which was deployed on WebSphere. They also did database side programming by means of stored procedures, but these were written using the stored procedure language. This meant that they had to recode some of the business logic in the stored procedure language. That led to a decrease in productivity, and, more importantly, led to the need for training the programmers on two different languages. In this case, it would have made much more sense to use Java stored procedures, since this allows for code reuse and makes it possible to call the stored procedure logic without passing through the DB2 CALL PROCEDURE statement.

So the choice of programming language strongly depends on your existing development language skills and on the amount of code that you already have that might be reused.

There are some software cost issues involved when selecting the SQL procedure language as your programming language, since a C compiler is required on z/OS. This is not really a problem, since the developer would never be exposed to the C code, but you would need a compiler license, which is not present by default and which adds to the software cost.

If you decide to deploy Java stored procedures, it may be worthwhile to look at zSeries Application Assist Processor (zAAP). You can find more information about zAAP at:

`http://www.ibm.com/servers/eserver/zseries/zaap`

zAAP can help you get a correct price/performance ratio when running significant Java code on z/OS.

6.2.3 Good housekeeping for stored procedures

This section covers best practices for writing stored procedures and explores pitfalls that exist.

Using Development Center

The DB2 Development Center (DC), included in the V8.1 UDB Application Development Client (ADC) component, is the follow-on product to the DB2 Stored Procedure Builder (SPB) in the DB2 V7.2 UDB Application Development Client.

Development Center supports the entire family of DB2 servers using the DRDA architecture. It communicates with the DB2 UDB V8.1 distributed servers (Linux, UNIX®, and Windows®), and with DB2 UDB for OS/390 V6 and Version 7, and DB2 UDB for z/OS Version 8, as well as currently supported DB2 UDB releases on iSeries™.

Development Center supports creating SQL stored procedures on all supported versions of DB2 for OS/390 and z/OS (currently V6, V7, and V8). Java stored procedures can be created with Development Center on DB2 V7 and V8.

You may want to change the environment settings when using Development Center. In order to get more feedback from the precompiler you should change precompiler option NOSOURCE to SOURCE. You do not get any line number in edit mode, and it can be hard to spot the problem without the source being included in the output.

The default option is also not to perform and explain on the BIND. We always recommend you verify the access path after creation of the stored procedure. Changing the bind to incorporate EXPLAIN(YES) is, therefore, a good recommendation. If you are not using a TSO session to create the stored procedure, we recommend you use Visual Explain to verify the access path.

Also, use External name in the CREATE PROCEDURE to make life easier by allowing you to correlate the stored procedure and load module.

Manage the DTT you use correctly

If a declared temporary table is defined in an application process running as a local thread, the application process or local thread that declared the table qualifies for explicit thread reuse if:

- The table was defined with the ON COMMIT DELETE ROWS attribute, which is the default.
- The table was defined with the ON COMMIT PRESERVE ROWS attribute and the table was explicitly dropped with the DROP TABLE statement before the thread's commit operation.
- The table was defined with the ON COMMIT DROP TABLE attribute. When a declared temporary table is defined with the ON COMMIT DROP TABLE and a commit occurs, the table is implicitly dropped, if there are no open cursors defined with the WITH HOLD option.

When the thread is reused, the declared temporary table is dropped and its rows are destroyed. However, if you do not explicitly or implicitly drop all declared temporary tables before or when your thread performs a commit and the thread becomes idle waiting to be reused, as with all thread reuse situations, the idle thread holds resources and locks. This includes some declared temporary table resources and locks on the table spaces and the database descriptor (DBD) for the TEMP database.

Recommendation: Instead of using the implicit drop feature of thread reuse to drop your declared temporary tables, we recommend that you define the declared temporary tables with the ON COMMIT DROP TABLE clause so that the tables are implicitly dropped when a commit occurs.

Remote threads qualify for thread reuse differently from local threads. If a declared temporary table is defined (with or without ON COMMIT DELETE ROWS) in an application process that is running as a remote or DDF thread (also known as Database Access Thread or DBAT), the remote thread qualifies for thread reuse only when the declared temporary table is explicitly dropped, or when using ON COMMIT DROP TABLE on the declare before the thread performs a commit operation. Dropping the declared temporary table enables the remote thread to qualify for the implicit thread reuse that is supported for DDF threads via connection pooling and to become an inactive DBAT. See the redbook *Distributed Functions of DB2 for z/OS and OS/390*, SG24-6952, for more information about distributed threads.

6.2.4 Performance expectations and tips

There is sometimes a cost for providing flexibility, and stored procedures are a good example of this behavior. It is not because you can put code in a stored procedure that you have to do so. Once you decide to use a stored procedure and are convinced of the advantages, you need to be aware of the difference in performance caused by the options on the CREATE PROCEDURE statement. You may wonder why DB2 has a slow and a fast option and why DB2 does not simply provide the fast option. In many cases, that is exactly what DB2 is doing, but some fast options have a manageability impact (tradeoff). Therefore, this section describes the advantages and disadvantages of the options.

Provide dual interface when using local callers

Stored procedures provide several advantages, but you need to be aware of the infrastructure overhead. Make sure to provide a direct path to the stored procedure code without using the DB2 call. A batch program that runs on the same system as the DB2 subsystem should not call the stored procedure, but should call the stored procedure program logic directly. You can save between 250 and 700 µsecs of CPU time on a z900 machine by calling the stored procedure program directly.

Then, distributed clients can call the stored procedure, and a local program running in CICS or batch can just call the application logic used in the stored procedure without going through DB2.

This works fine as long as we do not have to deal with result sets. However, non-Java subprograms cannot pass result sets or cursors from one package to another. So, if you design your stored procedure to return a result set, the call to the stored procedures cannot be replaced by a simple subroutine call to the stored procedure module. This is a known restriction.

To overcome the restriction of “result sets” or “cursors” with subprograms, you use DB2 temporary tables. Table 6-1 shows the difference between using a stored procedure only versus enabling a non-Java subprogram for result set passing. This table shows the extra steps required to enable a non-Java program to handle stored procedure invocation and subprograms.

Table 6-1 Stored procedure only vs. enabling non-Java subprogram for result set passing

Stored procedure	Subprogram and stored procedure
Stored procedure: 1. Declare a cursor WITH RETURN 2. Open the cursor Caller (with SQL CALL): Associate the result set to a cursor and fetch from it.	Subprogram: 1. Declare a DB2 temporary table with structure similar to the result set. 2. Select rows from regular table and insert them into temporary table. 3. Declare cursor on temporary table WITH RETURN. 4. Open the cursor. Caller (with SQL CALL): Associate the result set to a cursor and fetches from it. Caller (with program CALL): Declare a cursor on DTT and fetch from it.

So, the extra steps required for making the result set available after a call as a subroutine involve the definition of the DTT, the fetching of all the rows from the original cursor, and inserting the fetched row in the DTT. There are a few extra considerations when using this

technique since the insert into the DTT causes an increased CPU overhead and our objective is to avoid using an extra 250 to 700 µseconds of CPU time. Another important element is the number of rows that we want to put in the DTT. When opening the original cursor, nothing should happen, assuming we do not have materialization. When moving to the subprogram logic, the cursor is fetched and inserted into the DTT. At what point, do we stop fetching? It may also be beneficial to use a CTT instead of a DTT, since that allows you to avoid the logging part of the insert into the temp table. The CTT has an additional advantage since it supports static SQL, where the DTT will always have to determine the access path at run time. See “Temporary tables” on page 210 for a full comparison between CTT and DTT. This means, however, that we no longer use the TEMP database, but start using whatever database is used to store the work files. This must be considered whenever defining the buffer pools for the work files.

An alternative is the deployment of the stored procedure logic as an included module in the calling application. This means that the caller and the called logic become part of the same precompile unit. This allows the caller to fetch from the subroutine regardless of the programming language. The disadvantage is the rigidity of the source management as any change in the subroutine causes a recompile of the caller. It, however, provides the encapsulation and the performance benefits.

Create options

The create procedure statement provides a lot of options that have an impact on execution performance. Each of the following options are described in more detail in the SQL REFERENCE, we just provide you with the recommendation.

PROGRAM TYPE

We recommend that you specify SUB to avoid the overhead of loading subroutines into memory every time they are called, and deleting subroutines from memory every time they complete execution. Subroutines include not only the programs that are called by your stored procedures, but also any other modules that are included at link-edit time. You should avoid MAIN unless you have applications that do not effectively initialize work areas, and you have no control over your source code. This may be true of vendor packages for which changing the source code is not an option. In all other cases you should use SUB. Beware when using program type sub because the variables may not be initialized when using a reentrable program. It is better to initialize the variables in the program than to rely on the compiler init option.

STAY RESIDENT

We recommend that you specify YES to avoid the overhead of loading the load module for the stored procedure into memory each time it is called, and deleting the load module from memory every time it completes execution. Be aware that even if you specify NO, it is possible that your stored procedure load module remains in memory if there are many tasks calling that procedure.

COMMIT ON RETURN

We recommend that you specify YES for stored procedures that are called from a distributed client application. Specifying YES ensures that locks are released when the stored procedure returns to the calling application. Remember that this option has impact on the caller that goes way beyond simple performance considerations. If the stored procedure performs write operations and uses the commit on return, then the options, that the caller has, become limited since the changes performed by the stored procedure can no longer be rolled back by issuing a ROLLBACK of the caller.

We recommend that you specify NO for stored procedures that are called locally. The COMMIT ON RETURN should be NO for nested stored procedures also because a stored procedure cannot call other stored procedures defined with COMMIT ON RETURN YES.

PARAMETER TYPE

We recommend that you specify either GENERAL WITH NULLS or DB2SQL. Either of these options give you the capability to set IN, OUT, and INOUT parameters to null in your calling programs and stored procedures by setting the associated indicator value to a negative value. Nullifying parameters that are not used during either a call to or a return from a stored procedure reduces the amount of data that is passed. For example, output parameters do not have data in them during the call to the stored procedure, so you can nullify the output parameters in the calling program. For stored procedures that are called by distributed applications, this can result in savings in the amount of data transferred across the network, therefore, a reduction in network transmission time.

Calibrate LE options

The memory that is used by LE is depending on your installation defaults. It may be useful to validate how much memory is used and adapt the LE run time options. The best approach is to activate the LE storage report for a short period of time and then disable the storage report and adapt the run time options based on the report output. The run time keyword to get the storage report is RPTSTG.

> **Important:** Do not activate the storage report all the time, since that will have a negative impact on the CPU consumption of your application.

- If you are unsure about the amount of memory that is assigned to which area, you can use the RPTOPTS=((ON)). By default, the output of both RPTSTG and RPTOPTS are written to SYSOUT, but you can override the DD statement by using the MSGFILE keyword. More detailed information can be found in the *z/OS V1R6 Language Environment® Debugging Guide*, GA22-7560-05, and *z/OS V1R6 Language Environment Customization Guide,* SA22-7564-06.

The statistics for initial and incremental allocations of storage types that have a corresponding run-time option differ from the run-time option settings when their values have been rounded up by the implementation, or when allocations larger than the amounts specified were required during execution. All of the following are rounded up to an integral number of double-words:

- Initial STACK allocations
- Initial allocations of THREADSTACK
- Initial allocations of all types of heap
- Incremental allocations of all types of stack and heap

The report tells you that two areas are too small, the STACK and the ANYHEAP. The HEAP, on the other hand, can be reduced. You can see an extract of the storage report in Example 6-1 and you can find the full report in “Language Environment storage” on page 432.

Example 6-1 Storage report produced by run-time option RPTSTG(ON) (extract)

```
STACK statistics:
   Initial size:                                          4096
   Increment size:                                        4096
   Maximum used by all concurrent threads:                7488
   Largest used by any thread:                            7488
   Number of segments allocated:                             2
   Number of segments freed:                                 0
```

```
....
HEAP statistics:
   Initial size:                                                  49152
   Increment size:                                                16384
   Total heap storage used (sugg. initial size):                  29112
   Successful Get Heap requests:                                    251
   Successful Free Heap requests:                                   218
   Number of segments allocated:                                      1
   Number of segments freed:                                          0
    ....
ANYHEAP statistics:
   Initial size:                                                  32768
   Increment size:                                                16384
   Total heap storage used (sugg. initial size):                 104696
   Successful Get Heap requests:                                     28
   Successful Free Heap requests:                                    15
   Number of segments allocated:                                      6
   Number of segments freed:                                          5
```

Setting up WLM

The application environment option points to a WLM definition which provides a range of performance-related options. A single application environment has a fixed set of options, and you may, therefore, have to set up several application environments to cover your various needs.

Number of TCBs

When the number of TCBs allowed per address space is reached and more requests are coming in, there are scheduling delays due to address space creation overhead. It is important to monitor your accounting report. When accounting class 3 and/or 8 are active, the accounting field QWACCAST provides you with the total elapsed time spent waiting for an available TCB. The field QWACCANM gives you the number of times an SQL call statement had to wait for an available TCB before the stored procedure could be scheduled. Ideally, these values should be zero.

Service class and classification rules

The stored procedure is running in a WLM-established address space, but can be considered as an application object. It should not run at the priority of the DB2 system address spaces. Make sure that you define the proper classification rules to reflect the importance of the work and likely duration of the work.

DB2-provided stored procedures

DB2 comes with a set of procedures that are required for some of the CLI/ODBC metadata calls. DB2 also comes with stored procedures required for distributed administration and for stored procedure management. Some of these procedures have special requirements, such as APF authorization or the need for an WLM application environment, which has NUMTCB set to 1.

Tip one: Make sure to setup separate WLM application environments for the IBM-supplied stored procedures. Add a separate environment with NUMTCB = 1 and isolate the stored procedures that require the load modules to be APF-authorized.

Tip two: Make sure you apply OW55360 (SRM IS TOO QUICK TO UNBIND SERVER ADDRESS SPACES). Without this fix, you may see excessive stop and start of your address space as SRM keeps stopping them when no requests are queued. The fix will force SRM to consider a longer period of time when deciding to stop the address space.

6.3 Using user-defined functions

DB2 is considered an object-relational database. This means that the application has the ability to add new data types and new functions to the database. Adding a new function to DB2 is very similar to adding a stored procedure in DB2. You assign a particular piece of code to a name and DB2 allows you to use the name, and therefore your code, from within your SQL.

6.3.1 Why use user-defined functions?

There are several reasons why you want to deploy user-defined functions (UDFs). The simplest reason is to encapsulate a certain piece of programming logic that you do not want to expose to the outside world. The other reason may be to avoid unnecessary traffic between the application program and the SQL, which may even involve network traffic. All of this is similar to the arguments in favor of using stored procedures. UDFs take this one step further, because they allow you to intergroup non-DB2 information inside of the SQL execution process. Examples are the join between a WebSphere MQ queue and a DB2 table or the exploitation of data that is returned from a sequential file or a CICS transaction.

Another category of problems that we can help solve with a UDF is the expansion of the SQL semantics with functions that are specific to your industry, and either too complex or too sensitive to expect every application programmer to develop the code for independently. The DB2 extenders are a good example of expanding the SQL language with functions and data types that are not common in the typical programming languages, such as special data or images.

6.3.2 What flavors can we use?

UDF comes in variations, so that we can reuse internally defined functions and just rename or combine them, and we can also call external programs. The second variation is the kind of information that is returned by the UDF, which can be in the shape of a column, or manifest as an entire table.

Variations depending on the result that is returned

- Scalar

  The scalar function is really just going to return a single value. This can still involve passing many rows as input to the function when using it in the select list of a query that performs a GROUP BY.

- Table

  The table function is acting as a DB2 table. It can return multiple columns and multiple rows. This gives you the ability to join non-DB2 data with a table without passing through the client application logic.

Variations on the underlying function that is called

- External

  An external function is essentially using the same infrastructure as the stored procedure. It uses WLM to dispatch the load module that is attached to the function in an application environment. It is obvious that you incur the overhead of activating the code in the WLM address space. Beware that the number of invocations can quickly become very high,

since the SQL statement is driving the UDF execution, and not merely a simple CALL, such as with a stored procedure.

- Sourced

  A sourced UDF is reusing an existing function. Whether this involves external code or whether the function is performed internally in DB2 depends entirely on the SQL statement that is used in the RETURN clause. It works either way.

- SQL (needs to be scalar)

  A SQL scalar function returns a single value each time it is invoked. Usually, specifying a function is valid wherever an SQL expression is valid.

MQ interface functions

The Application Messaging Interface (AMI) is a commonly used API for WebSphere MQ that is available in a number of high-level languages. In addition to the AMI, DB2 provides its own application programming interface to the WebSphere MQ message handling system through a set of external user-defined functions. Using these functions in SQL statements allows you to combine DB2 database access with WebSphere MQ message handling.

Conceptually, the WebSphere MQ message handling system takes a piece of information (the message) and sends it to its destination. MQ guarantees delivery despite any network disruptions that can occur. Application programmers use the AMI to send messages and to receive messages. The three components in the AMI are:

- The message, which defines what is sent from one program to another
- The service, which defines where the message is going to or coming from
- The policy, which defines how the message is handled

To send a message that uses the AMI, an application must specify the message data, the service, and the policy. A system administrator defines the WebSphere MQ configuration that is required for a particular installation, including the default service and default policy. DB2 provides the default service and default policy, DB2.DEFAULT.SERVICE and DB2.DEFAULT.POLICY, which application programmers can use to simplify their programs.

A service point represents a logical end-point from which a message is sent or received and includes the name of the MQSeries® queue manager and the name of the queue. A service policy specifies a set of quality-of-service options that are to be applied to this messaging operation. Both the service point and policy are defined in the DSNAMT repository file. See MQSeries Application Messaging Interface for more details.

Commit environment for WebSphere MQ functions

DB2 provides two versions of commit when you use DB2 MQ functions:

- A single-phase commit: The schema name when you use functions for this version is DB2MQ1C.
- A two-phase commit: The schema name when you use functions for this version is DB2MQ2C.

You need to assign these two versions to different WLM environments. This guarantees that the versions are never invoked from the same address space.

You can exploit the single-phase commit, if you want to use the queue for error logging that should not be rolled back when an error occurs in the unit of work. Any DB2 COMMIT or ROLLBACK operations are independent of WebSphere MQ operations. If a transaction is rolled back, the messages that have been sent to a queue within the current unit of work are not discarded.

Scalar MQ functions

Table 6-2 describes some of the DB2 MQ scalar functions. See the *DB2 UDB for z/OS Version 8 Application Programming and SQL Guide,* SC18-7415-02, for a full list of all available functions. Use the scalar functions in any part of the SQL where expressions are allowed.

Table 6-2 A subset of the DB2 MQ scalar functions

MQ scalar function	Description
MQREAD (receive-service, service-policy)	MQREAD returns a message in a VARCHAR variable from the MQ location specified by receive-service, using the policy defined in service-policy. This operation does not remove the message from the head of the queue, but instead returns it. If no messages are available to be returned, a null value is returned.
MQRECEIVE (receive-service, service-policy, correlation-id)	MQRECEIVE returns a message in a VARCHAR variable from the MQ location specified by receive-service, using the policy defined in service-policy. This operation removes the message from the queue. If correlation-id is specified, the first message with a matching correlation identifier is returned; if correlation-id is not specified, the message at the beginning of queue is returned. If no messages are available to be returned, a null value is returned.
MQSEND (send-service, service-policy, msg-data, correlation-id)	MQSEND sends the data in a VARCHAR or CLOB variable msg-data to the MQ location specified by send-service, using the policy defined in service-policy. An optional user-defined message correlation identifier can be specified by correlation-id. The return value is 1 if successful or 0 if not successful.
Note: You can send or receive messages in VARCHAR variables or CLOB variables. The maximum length for a message in a VARCHAR variable is 4,000 bytes. The maximum length for a message in a CLOB variable is 1 MB.	

The MQ scalar functions are working on a message per message basis, and will return a value for every message processed. This also means that we need to process the result of the scalar function in the SQL. What is involved to put all the rows of the employee table in a MQ queue? The MQSEND function is the answer, so let us have a look at the SQL required to do this. See Example 6-2 for the MQSEND function syntax.

You notice that we concatenated all the columns in the employee table, which forced us to cast the non-character data types to character. We also made all the VARCHAR fields fixed length CHAR to make the parsing simpler at the receiving end. Without doing this, it becomes difficult for the receiver to know where FIRSTNME ends and the MIDINIT starts.

Example 6-2 Coding the MQSEND function for the Employee table

```
MQSEND(
EMPNO CONCAT CHAR(FIRSTNME,12) CONCAT MIDINIT CONCAT CHAR(LASTNAME,12) CONCAT WORKDEPT
CONCAT CHAR(HIREDATE) CONCAT JOB CONCAT CHAR(EDLEVEL) CONCAT SEX CONCAT CHAR(BIRTHDATE)
CONCAT CHAR(SALARY) CONCAT CHAR(BONUS) CONCAT CHAR(COMM)
)
```

The result of the MQSEND invocation is the value 1 if the PUT was successful. The next step is to feed the function with the rows of the employee table. See Example 6-3 for a possible SQL that puts all the rows of the employee table in a queue.

The singleton select uses a grouping function to avoid having to fetch the value 1 all the time. Remember that 1 is the result of a successful MQSEND. The CASE expression compares the total number of rows that were processed versus the SUM of the MQSEND output. If all rows are on the queue, then the SUM should be equal to the COUNT of the rows. If the SUM and the COUNT are not equal, we can derive the amount of rows in error from the difference in the two values.

Example 6-3 Singleton SELECT that puts the Employee table in a queue

```
EXEC SQL
SELECT CASE WHEN COUNT(*) = SUM(PUT)
            THEN 'ALL OK'
            ELSE CHAR(COUNT(*)-SUM(PUT)) CONCAT 'ERRORs Occurred'
       END
FROM (SELECT
         MQSEND(EMPNO CONCAT CHAR(FIRSTNME,12) CONCAT MIDINIT CONCAT CHAR(LASTNAME,12)
                CONCAT WORKDEPT CONCAT CHAR(HIREDATE) CONCAT JOB CONCAT CHAR(EDLEVEL)
                CONCAT SEX CONCAT CHAR(BIRTHDATE) CONCAT CHAR(SALARY) CONCAT CHAR(BONUS)
                CONCAT CHAR(COMM)
               ) AS PUT
      FROM DSN8810.EMP) AS B
```

Table MQ functions

DB2 also comes with a set of MQ table functions. Table 6-3 describes a subset of the MQ table functions that DB2 provides, see the *DB2 UDB for z/OS Version 8 Application Programming and SQL Guide,* SC18-7415-02, for a full list of all available functions.

These table functions are useful because they can be used in a FROM clause to use the queue data as input for the select. This way performing a join between an existing DB2 table and information in a queue becomes quite easy.

Table 6-3 DB2 MQ table functions

MQ table functions	Description
MQREADALL (receive-service, service-policy, num-rows)	MQREADALL returns a table that contains the messages and message metadata in VARCHAR variables from the MQ location specified by receive-service, using the policy defined in service-policy. This operation does not remove the messages from the queue. If num-rows is specified, a maximum of num-rows messages is returned; if num-rows is not specified, all available messages are returned.

Archived

MQ table functions	Description
MQRECEIVEALL (receive-service, service-policy, correlation-id, num-rows)	MQRECEIVEALL returns a table that contains the messages and message metadata in VARCHAR variables from the MQ location specified by receive-service, using the policy defined in service-policy. This operation removes the messages from the queue. If correlation-id is specified, only those messages with a matching correlation identifier are returned; if correlation-id is not specified, all available messages are returned. If num-rows is specified, a maximum of num-rows messages is returned; if num-rows is not specified, all available messages are returned.
Notes: 1. You can send or receive messages in VARCHAR variables or CLOB variables. The maximum length for a message in a VARCHAR variable is 4,000 bytes. The maximum length for a message in a CLOB variable is 1 MB. 2. The first column of the result table of a DB2 MQ table function contains the message. For a description of the other columns, see *DB2 UDB for z/OS Version 8 SQL Reference,* SC18-7426-02.	

Performance considerations

The MQ functions are implemented as external functions. This means that invoking the function will require the overhead of the function invocation. This needs to be considered when comparing the advantages of the MQ function to the potential performance overhead caused by the function invocation. In some cases you are forced to use the MQ function, such as the SQL stored procedure language. When using Development Center with the SQL language option, using the MQ function is your only option to interact with MQ. You should consider the performance impact in any other case where coding the native MQ calls is possible.

The following business case is a good example of using the full flexibility of the functions:

- On every change of the employee table, you need to publish the changes in XML format into a queue.

You can code this yourself in your application, but you may also create an after update trigger that does a select from the transition table using the DB2 XML publishing functions and use the MQPUBLISHXML function to send the information into the queue. Example 6-4 has an example of an after trigger that uses the publishing function.

Example 6-4 Create statement for a trigger that invokes a MQ XML publishing function

```
CREATE TRIGGER PUCHA
AFTER UPDATE ON CUSTOMER
REFERENCING NEW_TABLE AS TRANSIENT
FOR EACH STATEMENT MODE DB2SQL
BEGIN ATOMIC
SELECT MQPUBLISHXML('MYPUBLISHER', TRANSIENT.CUSTOMER, '/MIDWEST/CUSTOMERS')
FROM TRANSIENT;
END
```

In other cases, the advantages are less evident. When the application is coded in a programming language that supports MQ calls, and you need to put data on a queue that was fetched into the application for processing, then it is probably best to code the MQ PUT directly into the application program as the alternative. Using an additional SQL to invoke the MQSEND would create a considerable overhead. The MQSEND needs a separate SQL

statement, which needs to activate the external function MQSEND, which will then perform the PUT. So, the difference between the application code and the MQSEND invocation is at least an extra SQL call, plus the invocation overhead of running the external function. Needless to say, it is better to simply code the PUT in the application.

6.3.3 Performance expectations and tips

The implementation of an external user-defined function is the same as a stored procedure. All the considerations that are described in “Performance expectations and tips” on page 158 are therefore applicable to external user-defined functions. There are additional considerations which are specific to the user-defined function and they are covered in this section.

Providing the optimizer with statistics on your procedure improves access path selection. You can add information on the user-defined function in the SYSIBM.SYSROUTINES table. The application program can also override the estimated number of rows that are returned by the UDF through the CARDINALITY keyword at run time.

Deterministic

Deterministic specifies whether the function returns the same results each time that the function is invoked with the same input arguments. In general, it is best to use DETERMINISTIC since it avoids executing the function multiple times if the input values are the same. If you have a function that translates degrees Celsius in degrees Fahrenheit, then there is no point in performing the calculation more than once for the same input value. If -40 Celsius were -40 Fahrenheit, then it continues to be this way, regardless of the number of invocations. The same logic does not apply if you need to build a randomizing function or if your function needs to count the number of occurrences of a value.

NOT DETERMINISTIC

The function may not return the same result each time that the function is invoked with the same input arguments. The function depends on some state values that affect the results. DB2 uses this information to disable the merging of views and table expressions when processing SELECT, UPDATE, DELETE, or INSERT statements that refer to this function. NOT DETERMINISTIC is the default. Some functions that are not deterministic can receive incorrect results if the function is executed by parallel tasks. Specify the DISALLOW PARALLEL clause for these functions.

DETERMINISTIC

The function always returns the same result each time that the function is invoked with the same input arguments. If applicable, specify DETERMINISTIC to prevent non-optimal access paths from being chosen for SQL statements that refer to this function.

DISALLOW PARALLEL

Make sure you only specify the DISALLOW PARALLEL keyword if your code cannot handle the parallelism. Depending on the type of function you are implementing, parallelism can be pretty hard to implement, because you will have multiple instances of the same function active at one point in time that may need the overview of the entire result set and not just the part exposed to that part of the SQL.

FINAL CALL

The final call balances the first call, and like the first call, occurs only if the function was defined with FINAL CALL. The function can set return values for the SQL-state and diagnostic-message arguments. The function should also release any system resources that it acquired. A final call occurs at end of statement (when the cursor is closed for

cursor-oriented statements, or the execution of the statement has completed) or at end of transaction (when normal end of statement processing does not occur).

Only use the FINAL CALL option if your function needs to perform housekeeping functions that need to be performed outside the loop of individual invocations.

6.4 MQListener in DB2 for OS/390 and z/OS

DB2 UDB for OS/390 and z/OS provides an asynchronous listener, MQListener. MQListener is a framework for tasks that read from WebSphere MQ queues and call DB2 stored procedures with messages as those messages arrive.

MQListener combines messaging with database operations. You can configure the MQListener daemon to listen to the WebSphere MQ message queues that you specify in a configuration database. MQListener reads the messages that arrive from the queue and calls DB2 stored procedures using the messages as input parameters. If the message requires a reply, MQListener creates a reply from the output that is generated by the stored procedure. The message retrieval order is fixed at the highest priority first, and then within each priority, the first message received is the first message served.

MQListener runs as a single multi-threaded process on z/OS UNIX System Services. Each thread or task establishes a connection to its configured message queue for input. Each task also connects to a DB2 database on which to run the stored procedure. The information about the queue and the stored procedure is stored in a table in the configuration database. The combination of the queue and the stored procedure is a *task*.

See the section Asynchronous messaging in DB2 UDB for z/OS and OS/390 in the *DB2 UDB for z/OS Version 8 Application Programming Guide and Reference for Java*, SC18-7414-02.

6.5 Using triggers

A *trigger* is a set of SQL statements that is performed on your behalf without the originating SQL asking for the execution, but with the originating SQL being charged for the CPU time and elapsed time. The DB2 for z/OS trigger is always performed synchronously. So all the resources consumed are accumulated in the original thread.

6.5.1 Why use triggers?

A good usage of a trigger is the management of redundancy introduced by denormalizing the model. Adding a summary flag in a parent table and putting a trigger on the dependent table to maintain the flag may be an option. Beware however that it is a synchronous operation, and by definition, all updates become a hot spot if you aggregate at a sufficiently high level. If an asynchronous call is an option, then using WebSphere MQ and a stored procedure may be a better alternative.

The major benefits of triggers are:

- Enforce data integrity rules. No matter which application performs inserts, updates, or deletes on a table, you are sure that the associated business rules are carried out. This is especially important with highly distributed applications.
- Enables migration from other DBMSs which have trigger support.
- Faster application development. Because triggers are stored in the database, the functionality coded in the trigger does not need to be coded in every application.

- Code reusability. A trigger can be defined once on a table, and can be used by every application that accesses the table.
- Easier maintenance. If the business rules change, the applications do not need to be changed.

6.5.2 What flavors can we use?

There are two categories of triggers:

- Statement trigger

  This kind of trigger is invoked once per statement execution.
- Row trigger

  This kind of trigger is being invoked once per effected row.

6.5.3 Good housekeeping for triggers

You may want to be careful with triggers if you have multiple triggers and are worried about the execution sequence of the trigger. DB2 always activates all of the before triggers that are defined on a table before the after triggers that are defined on that table, but within the set of before triggers, the activation order is by timestamp, and within the set of after triggers, the activation order is by timestamp. This means that if you drop and recreate the oldest trigger, it becomes the most recent and goes to the last execution spot. The consequence of this behavior is that you either have to drop and recreate all the triggers in the correct sequence or that you put all the trigger body code in a program which you call by a single trigger. It is also possible to combine all the trigger logic in a single trigger, which solves the sequence issue and reduces the trigger overhead. The stored procedure solution comes at an extra cost as the overhead of calling the stored procedure can be between 250 and 700 µsecs of CPU time on a z900 machine, depending on the programming language of the stored procedure. This number can become an issue if the trigger execution frequency is high.

One may also decide to move the trigger body into a stored procedure to increase the manageability. This way you can use normal source management procedures and do not need to use DDL to manage new versions of the business logic.

Be careful with executing ONLINE LOAD, since that executes the trigger which can cause significant overhead. When large quantities of rows need to be added, we advise you can drop the trigger and execute the trigger code after the data is loaded. We assume that this is possible, and that you write a separate program to apply the trigger logic.

6.5.4 Performance expectations and tips

Triggers add a functionality that is very hard to replace, since the whole purpose of the trigger is to execute a piece of code without the original application being aware of the code. There is a cost for adding a trigger that is executed. So, you should only consider using triggers when there is no application alternative.

Deciding between after row and after statement triggers

The cost of using a trigger is influenced by the different tasks that are involved in processing the trigger. There are three areas that we need to consider:

- The cost of trigger invocation
- For after triggers, the management of the transition tables and transition variables
- The cost of using WHEN clause in a conditional trigger

The cost of trigger invocation

The invocation cost of the trigger is directly related to the type of trigger, row, or statement. A row trigger is called for each qualifying row (in case of a conditional trigger) or for all changed rows. If DB2 modifies no rows, the row trigger is not activated. A statement trigger is invoked for every statement execution. You can determine how many row and statement trigger invocations occurred by looking at the accounting record. If you want more detailed information, use the performance trace using IFCID 325, which records the start and end of the trigger invocation.

Let us look at an example of row versus statement trigger. We used a 2084-318 (z990) machine with two processors active in the LPAR for our testing.

Assume a row trigger on a table into which 165,829 rows are added using a statement that performs an INSERT INTO ... SELECT FROM or that is loaded using UNLOAD LOAD SHRLEVEL(CHANGE). The row trigger would be invoked 165,829 times to perform the insert of the transition variables. See Example 6-5 for the row trigger DDL.

Example 6-5 Row trigger to copy Employee inserts to new table

```
CREATE TRIGGER ROINSEMP
AFTER INSERT ON EMP
REFERENCING NEW AS NU_ROW
FOR EACH ROW MODE DB2SQL
BEGIN ATOMIC
INSERT INTO NEWEMP VALUES
(NU_ROW.EMPNO, NU_ROW.FIRSTNME ,NU_ROW.MIDINIT ,NU_ROW.LASTNAME ,
 NU_ROW.WORKDEPT ,NU_ROW.HIREDATE ,NU_ROW.JOB, NU_ROW.EDLEVEL, NU_ROW.SEX,
 NU_ROW.BIRTHDATE, NU_ROW.SALARY, NU_ROW.BONUS, NU_ROW.COMM
);
END
```

Then SQL in Example 6-6 was used to copy the 165,829 rows from the source table into the EMP table. The row trigger copied the newly inserted rows into the NEWEMP table.

Example 6-6 SQL to insert many rows into the Employee table

```
INSERT INTO EMP
SELECT EMPNO, FIRSTNME ,MIDINIT ,LASTNAME, WORKDEPT, HIREDATE, JOB, EDLEVEL, SEX,
BIRTHDATE, SALARY, BONUS, COMM
FROM SOURCE;
```

You can find an extract of the accounting trace for this insert statement in Example 6-7. The full report is in Example B-2 on page 433. You notice the two CPU components, the nonnested, and the trigger package SQL. In our example, the nonnested activity used 2.810 seconds of CPU time and the trigger package used 3.614 seconds of CPU time. This gives a total CPU cost of 6.423 seconds or 39 µ seconds per inserted row.

Example 6-7 Row trigger accounting extract

```
TIMES/EVENTS  DB2 (CL.2)
------------  ----------
CPU TIME        6.423486
AGENT          6.423486
NONNESTED     2.809844
  TRIGGER       3.613641

SQL DML     TOTAL
-------- --------
```

```
SELECT          0
INSERT     165830
UPDATE          0
DELETE          0

TRIGGERS         TOTAL
------------  --------
STMT TRIGGER         0
ROW TRIGGER     165829
```

We dropped the row trigger and defined a statement trigger that did basically the same thing, which was to copy the transition table into the NEWEMP table. See Example 6-8 for the trigger DDL.

Example 6-8 Statement trigger to copy Employee inserts to new table

```
CREATE TRIGGER STAINSEMP
AFTER INSERT ON EMPLOYEE
REFERENCING NEW_TABLE AS NU_ROW
FOR EACH STATEMENT MODE DB2SQL
BEGIN ATOMIC
INSERT INTO NEWEMP
( SELECT NU_ROW.EMPNO, NU_ROW.FIRSTNME ,NU_ROW.MIDINIT ,NU_ROW.LASTNAME ,
 NU_ROW.WORKDEPT ,NU_ROW.HIREDATE ,NU_ROW.JOB, NU_ROW.EDLEVEL, NU_ROW.SEX,
 NU_ROW.BIRTHDATE, NU_ROW.SALARY, NU_ROW.BONUS, NU_ROW.COMM
FROM NU_ROW
);
END
```

Example 6-9 shows an extract of the accounting trace for this insert statement. The full report is in Example B-3 on page 434. You again notice the two CPU components, the nonnested, and the trigger package SQL. In our example, the nonnested activity used 2.814 seconds of CPU time and the trigger package used 1.872 seconds of CPU time. This gives a total CPU cost of 4.686 seconds or 28 μ seconds per inserted row.

Example 6-9 Statement trigger accounting extract

```
TIMES/EVENTS  DB2 (CL.2)
------------  ----------
CPU TIME        4.686395
AGENT          4.686395
 NONNESTED     2.814296
 TRIGGER       1.872099

SQL DML     TOTAL
-------- --------
SELECT          0
INSERT          2
UPDATE          0
DELETE          0

TRIGGERS         TOTAL
------------  --------
STMT TRIGGER         1
ROW TRIGGER          0
```

There is a significant difference in CPU consumption when we compare the rows and the statement triggers. See Figure 6-3 on page 172 for a chart comparing the two options.

The nonnested cost is very similar since that is mainly the insert on the first table. The trigger CPU consumption doubles when using the rows trigger option. There is no reason to use a row trigger in this example, since we can implement the same behavior with the statement trigger.

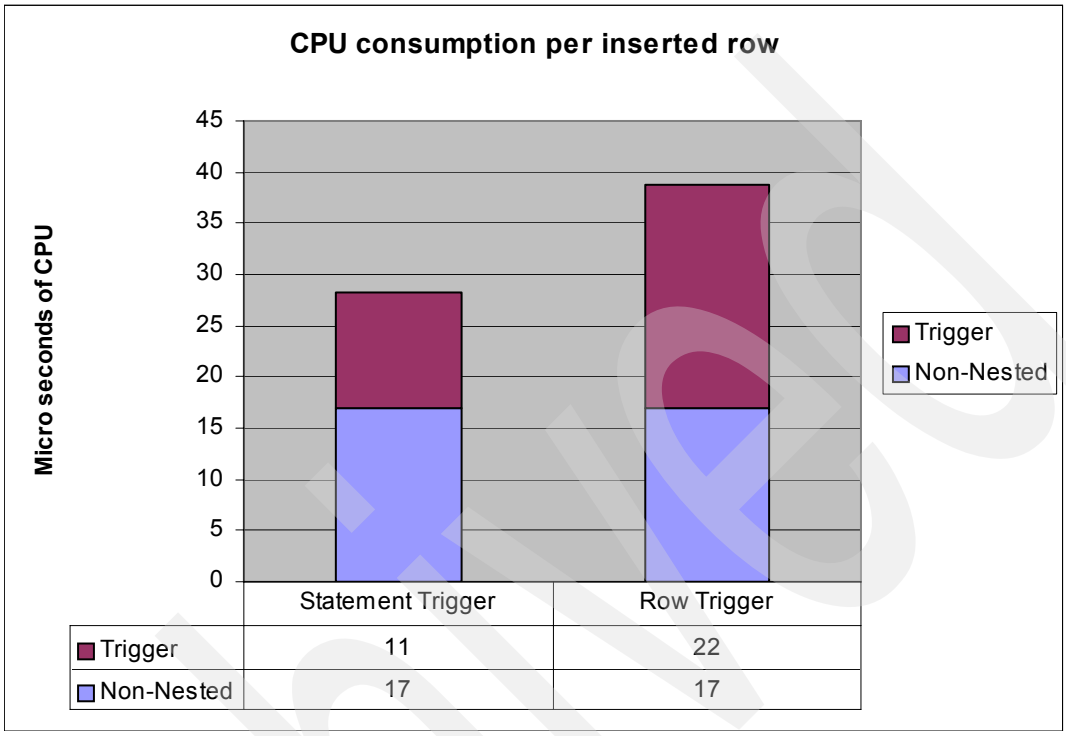


Figure 6-3 Summary of row versus trigger CPU cost per inserted row

The example we use is extreme, because we process 165 829 rows. Is there be a similar difference when fewer rows are involved? We redid the test with fewer rows. This allows us to validate the second element, which is the allocation of the work files.

For after triggers, the management of the transition tables and transition variables

Here we discuss the cost of work file usage on after triggers. The passing of the information to the trigger is also an important cost element. The number of qualifying rows on the row trigger determines if a row trigger is going to use a work file or if a 4 KB memory area is being used to pass the transition variables to the row trigger. The layout of the temporary table is the same as that of the triggered table, the image captured in the temporary table is the entire row, not just the transition variables. This means that the longer the row length, the greater the overhead of a trigger, regardless of the number and size of the transition variable used.

We performed the same insert statement, but used fewer rows in the base insert. We started with 10 rows and kept adding 10 more rows to the insert. The first thing we noticed was that the row trigger avoids the creation of the work file when fewer than 4 KB needs to be passed to the trigger. Our point of reaching 4 KB is around 26 rows. This explains the change in behavior between the measurement point 20 and 30 for the row trigger. The statement trigger used the work file in all cases. See Figure 6-4 on page 173 for an overall picture of the CPU cost per row for both trigger types with a varying number of rows in the insert. We can notice that the cost drops significantly when we move to more rows. Once we reach 70 rows, the cost does not seem to decrease any further.

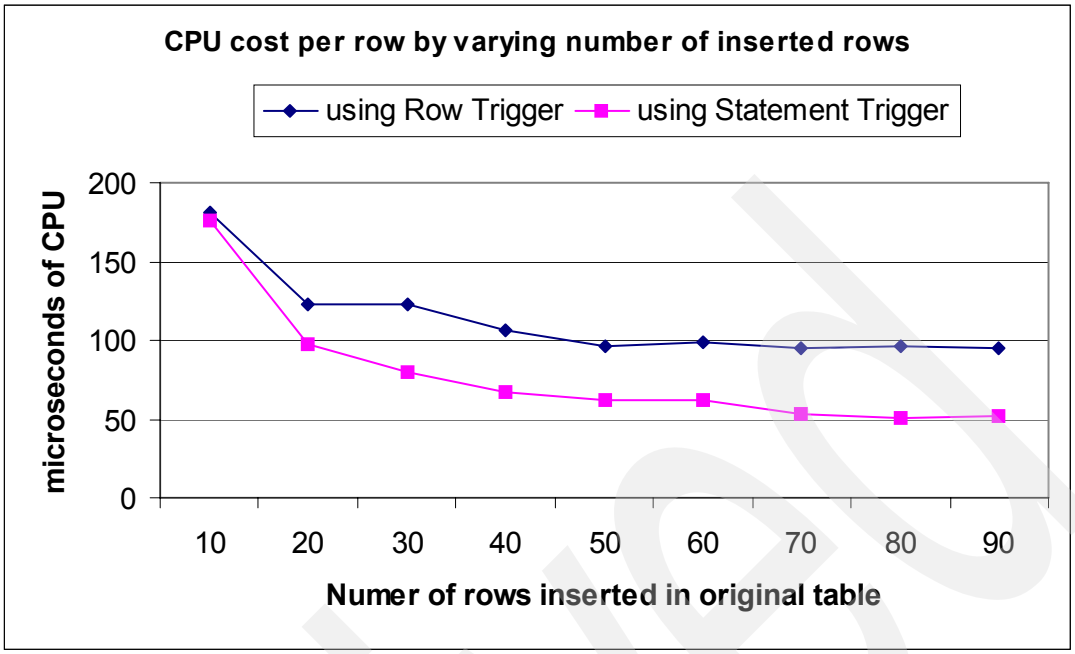


Figure 6-4 Total CPU cost per row insert for row and statement trigger

Note: The statement trigger was always cheaper than the row trigger in our test.

We split out the cost in nonnested and trigger CPU cost. You notice that the cost is different for both elements. See Figure 6-5 for the plot of the trigger cost. This is not significantly impacted by changing the number of rows. The row trigger is again more expensive than the statement trigger.

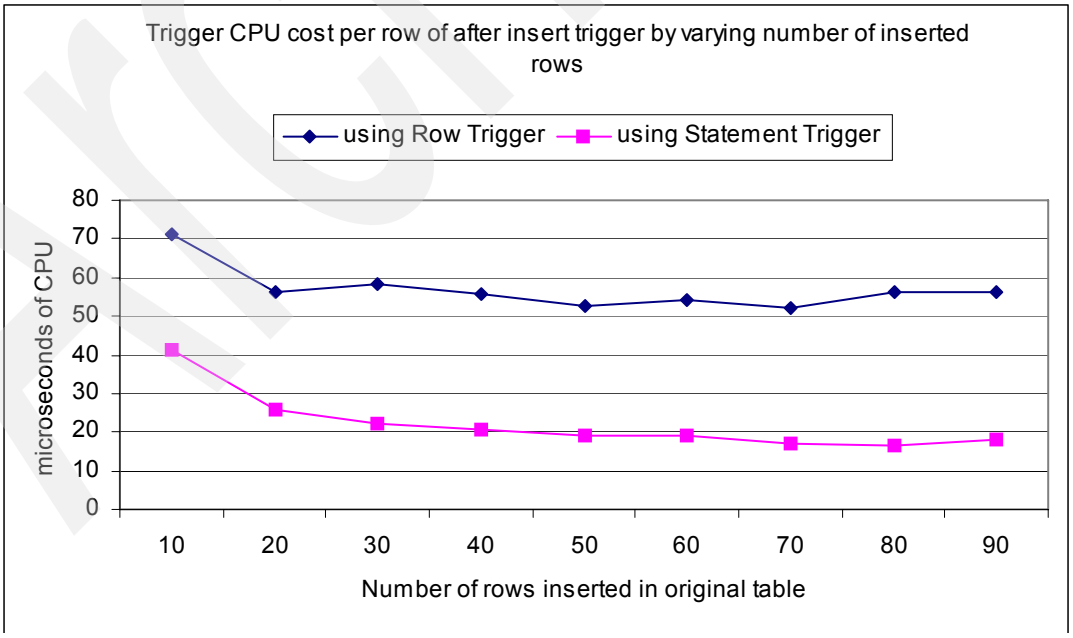


Figure 6-5 CPU of the trigger package for row and statement trigger

Archived

The biggest variation occurs in the nonnested CPU cost. See Figure 6-6 for the details. The cost of the statement trigger on the nonnested side is significantly more expensive than the row trigger, but the difference is not large enough to make an impact on the sum of trigger and nonnested CPU. This cost difference is mainly due to the work file allocation cost which happens in any case on the statement trigger and which can be avoided by the row trigger if fewer than 4 KB need to be passed to the trigger.

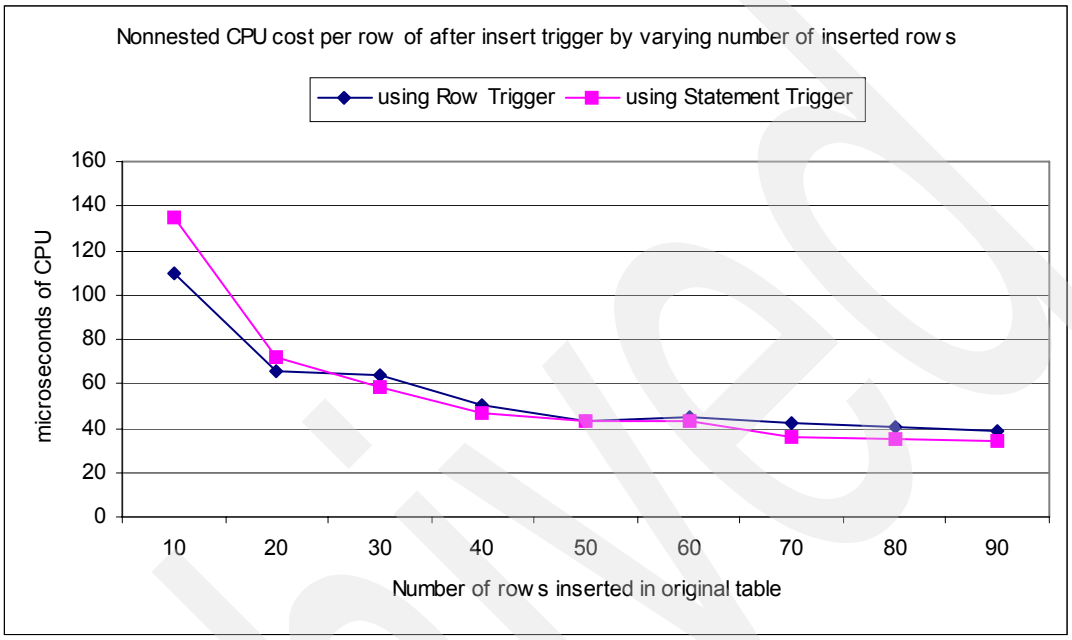


Figure 6-6 Nonnested CPU for row and statement trigger

We notice from the measurements that it is incorrect to assume that the row trigger is only going to deal with a single row. The row trigger logic has the ability to see the entire transition table in the trigger body of every row invocation. It is uncommon to use anything in the body of an after row trigger other than the transition variables, but DB2 is coded to support all cases, therefore, the potential use of a work file. So DB2 will instantiate a work file as large as all of the rows processed in the statement, regardless of whether it is a row or statement trigger. The row trigger has additional logic to avoid the work file if fewer than 4 KB is needed to store all the rows affected by the SQL.

Tip: After statement triggers, always use a work file. After row triggers, use a work file when more than 4 KB is needed to store the rows that were impacted by the SQL.

This means that there is no benefit, from a work file point of view, in using a row trigger when the trigger logic can be implemented as a statement trigger, unless fewer than 4 KB are required. Only use a row trigger when the trigger logic requires the use of row processing. If you know that you are always processing more than 30 rows in the statement and you need row trigger logic, it is cheaper to define a statement trigger and let the statement trigger call a stored procedure using the transition table as a parameter and declare a cursor in the transition table. The stored procedure can now do the row processing work and the stored procedure can use multi-row logic to reduce the overhead.

The cost of using WHEN clause in a conditional trigger

Here we discuss the cost of using the conditional WHEN clause after triggers. Triggers can be made conditional by adding a WHEN clause before the trigger body. The WHEN clause will manifest itself in the accounting report, because every invocation of the condition is counted

as a SELECT. So do not be confused when the numbers in the accounting do not seem to make sense. We added a simple WHEN condition to the trigger that was always true and remeasured the same SQL as before using the 165 829 rows. We saw a significant increase in the CPU consumption in both the nonnested CPU and the trigger body CPU cost. See Figure 6-7 for a graph comparing the results.

Conditional row trigger CPU cost increased more than three times when compared to the basic row trigger and more than six times when compared to the statement trigger.

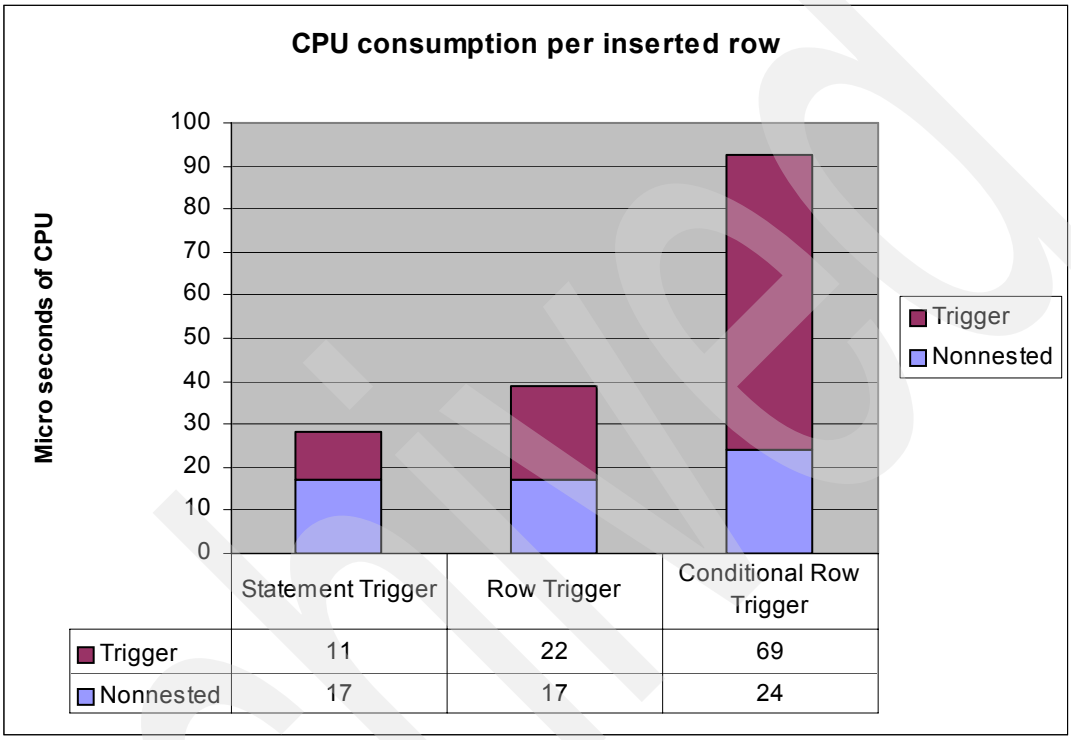


Figure 6-7 Trigger CPU cost when adding a condition

So be careful when adding a condition to the trigger. If the condition is most likely true, it is better not to add the condition and find another way to implement the condition. You may implement the condition in a different way, but you must be sure that you get the same result. One way may be to replace the conditional row trigger by a statement trigger that has a WHERE clause in the SELECT from the transition table.

Tip: Beware that a conditional trigger is much more expensive and implement it using another method when the same result is guaranteed.

Archived

7

XML support

As the use of XML becomes more and more pervasive, sooner or later people in your organization will want to store XML documents or generate XML documents from information stored in DB2 systems.

In this chapter, we briefly describe a number of ways to deal with XML in a DB2 database environment.

We only go into the options that are available today, that is, DB2 for z/OS Version 8, but we can suggest that quite a bit more is coming in the future.

This chapter contains the following topics:

- Introduction
- Storing XML intact
- Decomposing or shredding XML documents
- XML publishing
- The future

© Copyright IBM Corp. 2005. All rights reserved.

7.1 Introduction

In many organizations, relational databases are the backbone for data storage and retrieval. Over the last few years, XML has become the de facto standard for exchanging information between organizations, between businesses and customers, and between departments or applications within the same organization.

Since data tends to live in databases; they are a natural place to store XML data as well. But what is the best way to store XML documents? Do we store the XML document “as is”, or do we need to extract information from it, and store that separately to make it available for other usage.

On the other hand, when we have existing data in a DB2 table, how can we present that information in an XML format to a Web Services application, for example? To answer these questions, as always, you need to ask yourself, which processes are going to use the data? Then determine the best way to store and retrieve the XML data. We look at the problem from two angles. What is the rate of incoming XML documents that need to be stored and what is the retrieval rate? Table 7-1shows this.

Table 7-1 Storing and retrieving XML data

	Low insert rate	**High insert rate**
Low retrieval rate	No problem.	Store intact. Have retrieval do the work.
High retrieval rate	Parse and shred at insert time. This makes retrieval fast.	Potential problem. May need to renegotiate SLA or split processes.

- When the *insert and retrieval rate is low*, you do not need to worry too much about the performance of handling XML data, and it is best to store the XML in a way that is easiest for you to develop. Small homegrown programs that store data in a CLOB, or extract a few fields to store in a separate DB2 table are often good enough to handle this type of situation.
- When the *arrival rate of new documents is low, but you need to be able to retrieve information quickly*, it is best to parse the XML document at insert time (This is a CPU-intensive operation, but it should not be an issue since the insert rate is low.) and store the information in a shredded format (XML Collection). Once the document is shredded into a set of relational tables, it is “business as usual“ to make retrieval as fast as possible, since we assume that retrieval is the critical process here.
- When the *arrival rate of new documents is high, but you just need to be able to store the documents and they are hardly ever retrieved*, it is best to store the XML data as quickly as possible, and “dump” it into an XML Column, or just a regular VARCHAR, CLOB, or DBCLOB. If required, you can extract some data and put it into a side table to make retrieval somewhat faster, or just to be able to extract information that is used as a primary key to be able to retrieve the XML document later, if required.
- When the *arrival (insert) rate and retrieval rate is very high*, that may pose a problem, and you may need to renegotiate your SLAs. If you can tolerate a certain latency between the insert time, and the time the data starts to be interrogated, you can split the process. You can “dump” the XML document in an XML column at insert time, and process it again with a second process to shred it into relational tables to make fast retrieval possible later.

For the remainder of the discussion, we assume that the XML data is stored in DB2. We are not going to explore any options where the XML is stored outside of the database. Databases

provide numerous advantages, such as concurrency, recoverability, and authorization control that you have to build yourself when storing outside of the database.

7.2 Storing XML intact

DB2 for z/OS Version 8 is a so-called XML-enabled database. DB2 Version 8 is a relational database engine that has been enhanced and extended to be able to deal with XML data. However, the underlying data model is a relational model. This means that somehow we have to store XML data in a column of a table using an existing DB2 data type. Since XML documents tend to vary in size and can be very big, VARCHAR, CLOB, or DBCLOB are the most appropriate data types when storing XML documents in DB2 for z/OS Version 8.

When storing intact XML, there are two options:

- Store intact and no extra processing of the XML document is required.
- Store intact, but certain information needs to be extracted from the XML document when it is stored or stored at a later time.

7.2.1 Store intact with no need to process information contained in the XML

This is the easiest problem to solve. You just need to dedicate a column in a table to store the XML data. As mentioned before, VARCHAR, but most likely, CLOB or DBCLOB are the most suitable data types.

The only consideration in this case is that a unique identifier needs to be attached to the document, so it can be identified when it needs to be retrieved later. In this case, the unique identifier is not dependent on the information inside the XML document. This option assumes that there is no need to parse the XML document and use some element or attribute value as an identifier for the document.

Storing XML data this way is just like inserting rows into a regular DB2 table. The only potential problem is if the XML document insert rate is very high, or if the XML documents are very large. Dealing with this type of XML document is no different than any other type of DB2 data dealing with massive inserts. As mentioned before, even in this case, we still want to use a database to store the XML data, because you can take advantage of all the features provided by the database engine, such as scalability, recoverability, and so on.

You can also use XML Extender when dealing this type of XML data, since it provides certain user-defined data types and user-defined functions that allow for easy insert and retrieval, but XML Extender does not provide a lot of added value in this case.

7.2.2 Store intact, but process information contained in the XML documents

This is a much more common case than the previous one, but unfortunately introduces a higher level of complexity as well.

In this case, we store the XML document as such, but we still want to look inside the document, usually to extract some information that is used as an “index” when the XML document needs to be retrieved later. For example, when an XML document containing an order comes into the system, we want to look inside the XML document to find the customer name that sent us the document, so we can store the customer’s name in the same or a separate table, so we can use this information later to retrieve all orders from a certain customer.

To handle this type of XML data, you can either develop a homegrown program that looks inside the XML document to find the customer name, either by doing a simple “search” or by first invoking an XML parser that parses the XML document so you can then retrieve the required customer information from the parsed document. In addition, you can also use DB2 XML Extender’s XML Column feature.

Determining which option to use depends mainly on the complexity of the XML document. If you know that the customer name is always present and that each order can only contain a single customer name, a simple search inside the XML document may do the job. However, if the document is complex, you may want to use a parser, or, even better, XML Extender.

When using the XML Extender XML Column functionality, the XML document is stored as such, but you can define a mapping (a Data Access Definition (DAD) file) indicating which elements and attributes you want to extract for “indexing” purposes. This information is stored in so-called side tables, and can be used when retrieving XML documents later.

Figure 7-1 shows an overview of using XML Column.

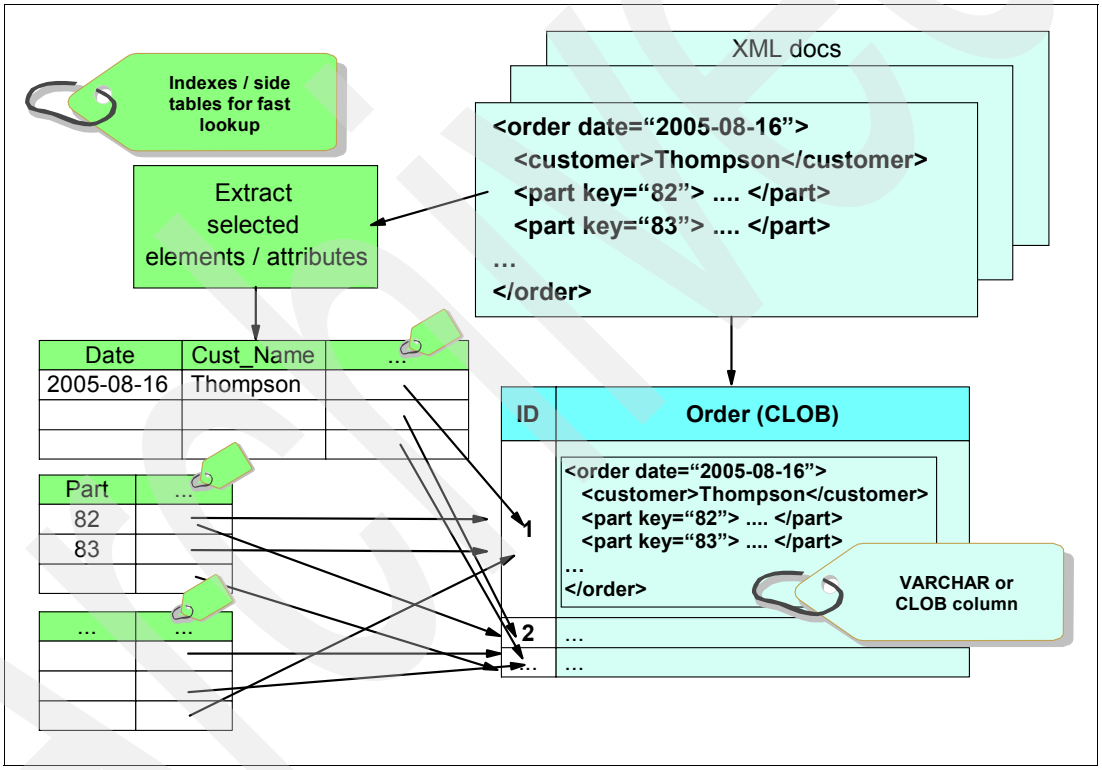


Figure 7-1 XML column

When to use an XML column to store data

You may use XML columns in the following situations:

- The XML documents already exist or come from an external source and you prefer to store the documents in the original XML format. You want to store them in DB2 for integrity, archival, and auditing purposes.
- The XML documents are read frequently and you have a good idea which elements and attributes are used in those searches, but the documents are rarely or not updated.
- The documents have elements with large text blocks, and you want to use DB2 Net Search Extender for structural text searches, while keeping the entire documents intact.

Also keep in mind that white space (blanks and blank sections) may be important. By storing the XML document intact, you can preserve the white space.

Make sure to use side tables to perform searches based on the values of XML elements or attributes. If you know which elements and attributes are frequently used as search arguments, make sure to put them into a side table. In addition, you can build normal DB2 indexes on the side tables to speed up processing even more. Because maintaining the side tables is done at insert, update, and delete time (using triggers), you do not want to create a side table column for every element and attribute. It will increase the insert, update, and delete time considerably.

It is very difficult to add or change side tables after the XML column is populated with data. Therefore, it is imperative to implement a good design of the side tables from day one. We *strongly recommend* that you do not update the side tables directly, because doing so can easily cause data inconsistency problems. You must update the original XML documents stored in the XML columns. Those will trigger automatic updates of the side tables.

When filtering documents at retrieval time, filter as much as possible using side table columns in the WHERE clause. It prevents DB2 from having to parse the XML document, and since you normally have an index on the columns in the side table, DB2 can use an access path that uses the index for fast retrieval of the value and evaluation of the WHERE clause predicate.

Queries using a location path expression (an XPATH-like way to indicate the type of information on which you want to filter) do not scale well because the XML document must be parsed (and potentially parsed multiple times). Therefore, use location path queries only for infrequently occurring ad hoc queries, or ad hoc queries that extract information from a few documents that have been filtered out using predicates on side table columns in the WHERE clause.

Summary

The major advantage of using XML Column is that it allows for fast insert and retrieval of full XML documents. On the other hand, any subdocument level access requires costly XML parsing to evaluate an Xpath expression, retrieve a partial XML document, or perform updates at the subdocument level (for example, replacing an attribute value). The parsing of XML documents is very CPU-intensive, so when dealing with large data volumes, you must consider whether or not parsing is necessary, and if yes, whether it is necessary to do it at insert time, or whether it can be done later (by first “dumping” the information into a “work” table and processing it afterwards), or if the parsing will be done at retrieval time. The latter is an interesting option if the number of retrieval operations that requires parsing is very low, and the requests do not require a subsecond response time.

The fixed mapping rules, between the XML documents and the side tables, make it difficult to make changes to the structure of the XML documents when data already exists in the XML column.

7.3 Decomposing or shredding XML documents

Instead of storing the XML documents intact, for example, in an XML Column as discussed in the previous section, you can also take the incoming XML document and *decompose* or *shred* it, and then store the different “pieces” of the XML document into different columns in one, or most likely, multiple DB2 tables.

To shred an XML document into a set of one or more relational tables, you need to define a mapping between the XML element and attribute structure and the relational tables. This is a fixed mapping, so it requires that the XML documents are well structured to be handled this way. You can use XML Extender to perform shredding of XML documents into DB2 relational tables. This is called *shredding* or *decomposing* into an XML Collection. Again, a DAD file is used to define the mapping (see Figure 7-2).

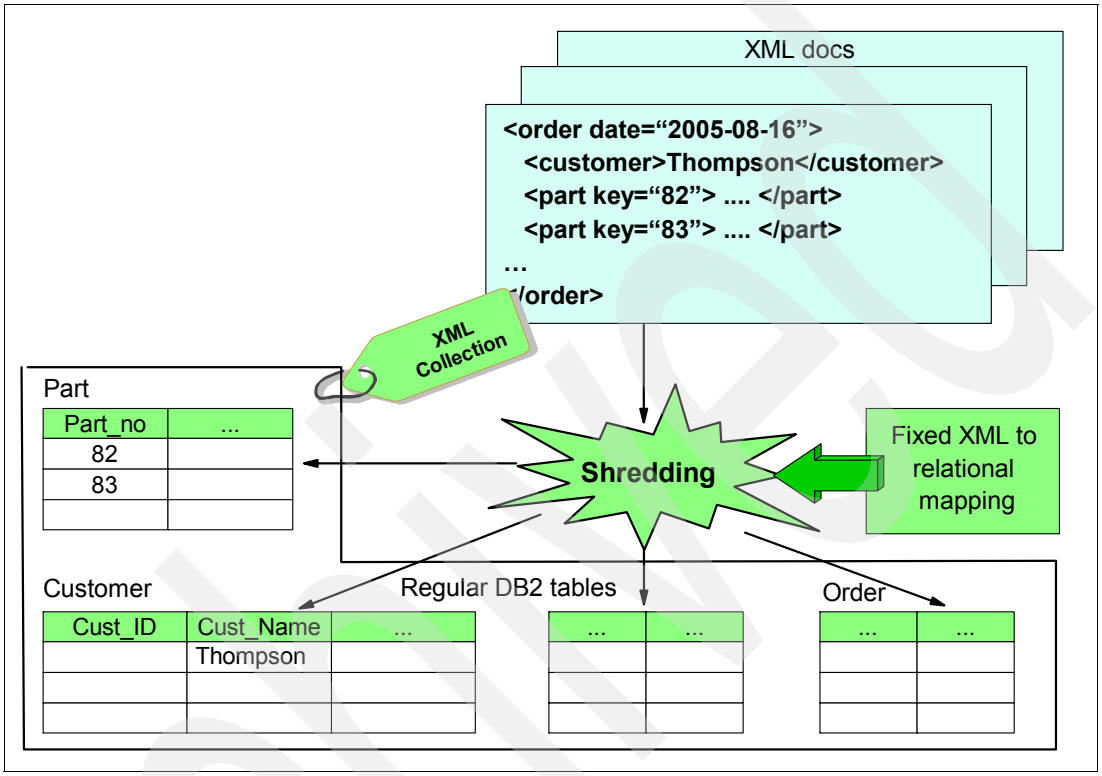


Figure 7-2 XML collection for shredding

In this case, the XML document itself is not stored; only the relevant (shredded) parts are stored in relational tables.

A good usage case for this type of storage is if you have a set of existing applications, but you also want to be able to process data coming into the system as XML documents with those existing applications. To achieve this, you shred the incoming XML documents and put the information into the existing tables used by these applications. The existing applications are not able to tell whether the data is coming from a green screen terminal, or whether the data entered the system via a Web Service that sent in an XML document. So, after shredding, the data resides in "normal" relational tables that can be exploited by any process that knows how how to handle SQL.

We recommend using XML collections in the following situations:

- You have XML documents that map well to an existing relational model, and the XML documents contain information that needs to be stored with existing data in relational tables.
- You want to create different views of your relational data using different mapping schemes.
- You have XML documents that come from other data sources. You are interested in the data but not the tags, and want to store pure data in your database. You want the flexibility to decide whether to store the data in existing tables or in new tables.

- A small subset of your XML documents needs to be updated often, and performance of those updates is critical. Since the data is stored in regular DB2 columns after shredding, you can use normal SQL UPDATE statements (therefore, not using XML Extender functionality, since shredding itself does not support updates) to directly update those columns that require changes.
- If you do not need to store the XML document intact, but still want to store all data contained in the XML document, you can shred the XML document into relational tables, and retrieve only those columns that are currently used by applications using normal SQL SELECT operations.

Note that once the document has been shredded, it is very difficult, if not impossible, to reconstruct the original XML document. After shredding, all white space is removed from the XML document. In addition, in a relational environment the order of rows in a table is not relevant. However, in XML element sequence can be important. Therefore, if you need to be able to reconstruct the original XML document, you may have to add some “artificial” columns to your table to be able to put the elements back in the correct sequence and put back the white space. To avoid the problem, you may want to consider not only shredding the document, but additionally, storing an intact version of it.

The biggest challenge with shredding XML documents using XML Extender is the mapping from the XML schema to the relational schema. With complex documents, the mapping may require dozens of tables to represent the XML schema. This requires large multi-table joins in case we want to reconstruct the XML document (publish it) based on the information inside the XML Collection.

As with XML Columns, the mapping is fixed and does not allow for flexibility in the schema that is to be supported. For example, if the XML schema changes, and needs to allow for multiple occurrences of a certain element or attribute, this may lead to a big change in the relational mapping, as you may need an additional table to handle the change from an 1:1 relationship to a 1:n relationship. And, in addition, you need to extract the information for the documents that are already in that DB2 column and move it to a different (new) table.

XML Collections work best for well structured documents that adhere to a fixed schema that is unlikely to change. If you can control the schema, then XML Collections can work well.

7.4 XML publishing

Relational data is the backbone of many businesses. With XML as a universal data exchange format, the capability of constructing XML data from existing relational data, while preserving the power of SQL, tremendously simplifies business-to-business (B2B) application development.

Therefore, we not only need a way to store XML data in a (relational) database, we also need ways to generate XML based on information that exists in normal relational tables. Generating XML from relational data is also called *XML publishing* or the *composition* of XML documents (Figure 7-3 on page 184).

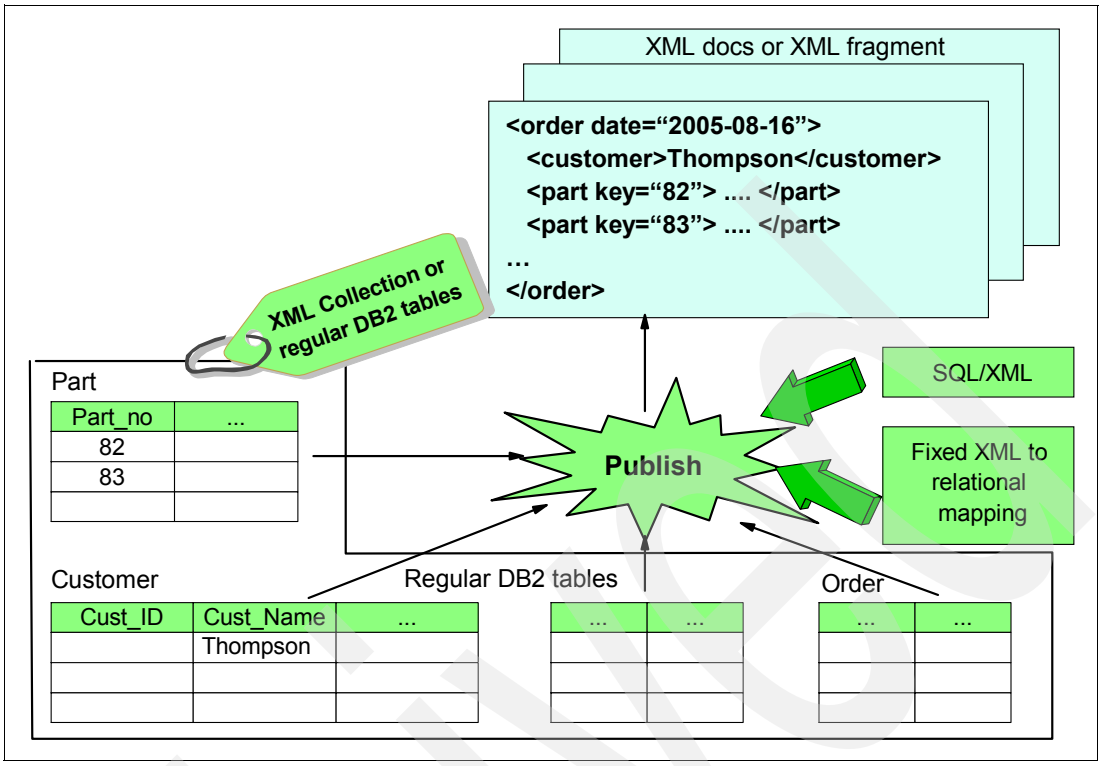


Figure 7-3 XML publishing

There are several ways to publish XML. We list them in the most preferred order:

- Use SQL/XML
- Use DB2 XML Extender
- Write your own program

Write your own program

As with most things, you can always do it yourself, and that may work fine for small scale applications. However, keep in mind that an XML document can have an extremely complex structure, and writing a program to produce correct XML documents is not always that easy. In addition, when documents become larger, or the volumes grow, most homegrown applications do not scale up very well.

Use DB2 XML Extender

You can also use DB2 XML Extender to generate XML documents from data in relational tables, either in an XML Extender XML Collection, or just in a set of regular DB2 tables. During publishing, XML Extender also uses a DAD to map the relational tables to the XML structure.

As is the case for shredding, mapping the existing relational (input) schema to the required XML (output) schema using a DAD can be very complex.

Use SQL/XML

To help integrate the relational and the XML world, the SQLX group at:

http://www.sqlx.org

focuses on extending SQL capabilities and consciously avoids vendor extensions, while encouraging state-of-the-art and projected future developments. SQLX forwards proposals to

the INCITS (International Committee for Information Technology Standards) H2 Database Committee for approval.

SQL/XML is an ANSI and ISO standard that provides support for using XML in the context of a relational database management system. Because SQL is the standard language for accessing and managing data stored in relational databases, it is only natural that enterprises and users want the ability to (access and) generate XML data from their relational data through the use of the SQL interface.

SQL/XML defines a standard that makes it possible to store your XML documents in your SQL database, to query those documents using XPath and XQuery, and to "publish" your existing SQL data in the form of XML documents.

Currently the following SQL/XML publishing functions are available in DB2 for z/OS Version 8:

- XMLELEMENT and XMLATTRIBUTES to construct XML elements with attributes.
- XMLFOREST to construct a sequence of XML elements.
- XMLCONCAT to concatenate XML elements.
- XMLAGG to aggregate XML elements.
- XMLNAMESPACES to declare one or more XML namespaces.
- XML2CLOB is not part of the SQL/XML standard. This function is introduced by DB2, because currently (Version 8), DB2 does not support a native XML external data type. Therefore, the internal XML data type that is used inside the DB2 engine needs to be transformed to a data type (CLOB) that DB2 understands and is able to return to the application. Once the native XML data type becomes available, there will be no need for this function anymore.

Just to give you an idea of what SQL/XML looks like, Example 7-1 shows you how to produce an XML fragment that includes some employee details and their total salary, made up of their normal salary, bonuses, and commission.

Example 7-1 Using XMLELEMENT and XMLATTRIBUTES

```
SELECT
  XML2CLOB(
   XMLELEMENT (NAME "EmployeeSalary",
      XMLATTRIBUTES (E.EMPNO AS "id"),
         XMLELEMENT (NAME "Firstname", E.FIRSTNME),
         XMLELEMENT (NAME "Lastname", E.LASTNAME),
         XMLELEMENT (NAME "TotalSalary",(E.SALARY+E.BONUS+E.COMM))
   )
  )
FROM DSN8810.EMP e WHERE SEX  = 'F'
```

The query returns a result set, where each row contains an XML fragment such as Example 7-2.

Example 7-2 Result of XMLELEMENT and XMLATTRIBUTES

```
<EmployeeSalary id="000010">
   <Firstname>CHRISTINE</Firstname>
   <Lastname>HAAS</Lastname>
   <TotalSalary>000057970.00</TotalSalary>
</EmployeeSalary>XMLFOREST
```

The SQL/XML functions are integrated into the DB2 engine, in contrast to the XML Extender publishing functions that are user-defined functions. In general, SQL/XML functions outperform XML Extender composition by an order of magnitude, especially when generating large XML documents. For example, during a lab measurement with a 5 MB document, SQL/XML was 20 times faster. Also, note that XML Extender cannot insert more than 10,240 rows into any table when composing an XML document.

Summary

The preferred tool for XML publishing is SQL/XML. Generally speaking, it is more flexible and versatile than the DB2 XML Extender publishing functions, and provides better performance.

However, if for some reason you cannot use SQL/XML to get the job done, you can try to accomplish your goal using XML Extender, and as a last resort, if XML Extender is also not up to the job, write your own application to publish relational data as XML.

For more information about the use of DB2 and XML, see *XML for DB2 Information Integration*, SG24-6994.

7.5 The future

In the previous sections, we described the functionality that is available with DB2 for z/OS Version 8 and DB2 UDB for z/OS Version 8 XML Extender. We also described some of the restrictions in the current architecture, mainly related to storing XML data into a relational database, such as:

- The mapping between the XML and the relational schema can be very complex, and may require dozens of relational tables to adequately shred an XML document. Later on, during publishing all these tables have to be joined together again to produce the XML document.
- The mapping also has to be fixed. If the XML schema changes, one of the major advantages of XML is its flexibility to handle documents with different schemas, the mapping has to change, and the definition of the underlying DB2 tables also has to change, and most likely some data has to be moved from one table to another, or two tables or columns have to be merged.
- When accessing documents that are stored intact, any subdocument level access requires XML parsing, which is an expensive operation.

For this reason, IBM has been working on allowing XML documents to be stored in a native XML format and data type (not stored as a CLOB or VARCHAR, or decomposed). DB2 will be enhanced to be able to store XML documents using a native (hierarchical) storage model that is fully integrated with DB2 with sophisticated XML indexing mechanisms.

In order to prepare for this new way to store and deal with XML inside the DB2 engine, it is probably a good idea to also store the XML document somewhere in an intact format. This way, whenever the native XML store arrives, you will always be able to take those “old” documents and store them in the native XML store.

For an indication about the future directions of DB2 with XML, see:

http://www.ibm.com/software/data/db2/udb/viper/

Even though these articles are geared towards DB2 on Linux, UNIX, and Windows, they are also indicative of the XML functions that DB2 for z/OS is going to support, although not all of these functions may be available at the same time, and the implementation will certainly depend on platform characteristics.

Part 3

Embedding SQL in the application

In this part, we introduce the SQL language of DB2. We describe the basic rules on SQL programming and the programming templates, then we proceed with more advanced techniques and relevant DB2 environment definitions.

This part contains the following chapters:

- Chapter 8, “SQL fundamentals” on page 189
- Chapter 9, “Programming templates” on page 215
- Chapter 10, “Advanced programming techniques” on page 263
- Chapter 11, “Infrastructure topics” on page 289

© Copyright IBM Corp. 2005. All rights reserved.

Archived

8

SQL fundamentals

This chapter reviews general good practices for coding SQL statements. We do not discuss special manners for implementing SQL, but basic concepts and rules you should take into consideration while developing new applications using either static or dynamically executed SQL statements.

It is important to have an idea about the performance of the application processes in the early stages of developing an application. Nobody likes fixing poorly performing applications after they have started living in a production system. It is very important to think in terms of coding SQL statements in a “the result is correct and the performance fits within our SLAs” way rather than coding the SQL statement and only checking that DB2 returns the correct rows after an unspecified amount of time.

Caring about performance and availability in the early stages of development processes can make your future life easier, since you will not get complaints on poorly performing applications from your customers or systems people. The difficult part is to set aside some time in your project to squeeze the most out of the opportunities SQL provides and to take some performance measurements in your development system rather than starting these activities later in the production environment.

You should check to see what each new release of DB2 for z/OS is bringing to make your life easier. There could be new functions implemented which can easily replace solutions you previously implemented in your application program. If you just look at the additional list of indexable predicates provided by DB2 for z/OS Version 8, how many of your applications may now use less complicated techniques to ensure index access?

This chapter helps you to take basic SQL issues into account and to avoid very common mistakes. It contains the following sections:

- ”Number of SQL calls”
- ”Number of rows searched”
- ”SELECT column list”
- ”ORDER BY clause”
- ”Subquery to join transformation”
- ”Indexable, stage 1, and stage 2 predicates”
- ”Considerations on the CAST function”
- ”Selective predicates”

© Copyright IBM Corp. 2005. All rights reserved.

- ”Subqueries”
- ”QUERYNO in SQL statements”
- ”Reduce trips to DB2”
- ”Cursor versus searched update and delete”
- ”COMMIT frequency”
- ”Temporary tables”
- ”Retrieving values generated at insert”
- ”Current statistics”

Archived

8.1 Number of SQL calls

Since every execution of a DB2 call, either using a call level interface (CLI), SQLJ or JDBC, or operations that you can embed using your application programming language, requires the usage of system resources, you need to keep the number of SQL calls as low as possible. This is true for statements you use to access your data as well as those that do not require access from disk storage.

It is easy to read the same data again instead of keeping the previously read values in memory, and it is easier to code SET :HOST-VARIABLE = SPECIAL REGISTER than getting the same result by using functions provided by your application programming language. However, it is more costly. Producing some overhead once might not be critical, but have you ever thought of how often your application is called and the overhead instructions that can be executed every day? Probably thousands of times, or even millions of times a day? When you think of service modules invoked by many other application modules, lots of CPU cycles can be saved by simply replacing those statements once!

Simple code sequences, such as string operations or building certain values such as dates or timestamps in your application programming language instead of invoking SQL and letting DB2 do the work, saves about 3,000 instructions per call. This is the amount of machine instructions only used for the round-trip for calling DB2 via EXEC SQL and returning from DB2 to your application, assuming that basic functions use nearly the same code sequences.

Check to see if there may be a better performing solution from an application point of view before invoking SQL to just solve programming challenges.

On the other hand, whenever you access your DB2 data and have to apply different functions on the retrieved columns, do not do it in the application. Things such as multiply new column value, join data, evaluate predicates, or apply filtering predicates, should be done using SQL functionality.

Example 8-1 shows a simple instance of unnecessarily applying functions within the application programming language.

Example 8-1 Applying functions using application programming language

```
SELECT COLa FROM T1
 WHERE COLb = :HV-COLb
  INTO :HV-COLa

HV-COLa := HV-COLa + 1

UPDATE T1
   SET COLa = :HV-COLa
 WHERE COLb = :HV-COLb
```

Example 8-2 shows the simple SQL coding technique which you can use to reduce the additional overhead of an extra roundtrip to DB2.

Example 8-2 Applying functions using SQL

```
UPDATE T1
   SET COLa = COLa + 1
 WHERE COLb = :HV-COLb
```

Make sure your application does not execute single update statements for simplifying application coding by rows where no column attribute has changed. This causes DB2 to look

for the row by performing I/Os and wasting CPU time. If you update multiple columns of a certain row, combine the changes and issue the update statement once, not for each affected column.

Rollback instead of application logic?

In general, an application program should check certain conditions before applying changes to a database. Do not apply your changes first and then later check for data consistency to roll back in case of errors.

The additional overhead caused by locking and undoing all the changes you have made after your last COMMIT results in unnecessary consumption of CPU and I/Os, probably slowing down concurrently running processes by stealing resources. In a transaction, try to put all applied changes to the database right before COMMIT to keep the amount of time while a lock is held (and your application owns a resource exclusively) as short as possible. For more information about DB2 locking, see “Serialization mechanisms used by DB2” on page 329.

Think of a ticket sales system where each person is only allowed to buy six tickets and three have already been bought by a certain person. If the same person asks for another five tickets, do you want to mark five more tickets as “sold” before figuring out that this is more than the allowed amount and perform a ROLLBACK? Or do you want the application to check if the person is allowed to buy another five tickets before applying any updates?

Depending on chances of failure for your transactions, you may say that 99% of your transactions complete successfully. In this case, follow the optimistic approach, it is cheaper to roll back for 1% of all transactions than perform data consistency checks in 99% of all transactions which are OK.

The same rule applies when reading the same data more than once, instead of keeping a copy in the working storage of your application program. Whenever you access DB2 data, make sure that this data is not already available to your application. When you deal with business cases requiring the same data access twice, you might consider using ROWID for getting a more efficient access path. For more detailed information about ROWID access, see “Data access using ROWID” on page 276.

8.2 Number of rows searched

Include as many predicates as possible to limit the number of rows that need to be scanned by DB2 to return the correct result set. Do not perform filtering in the application since it consumes more resources to pass a row to the application than letting DB2 verify that the row does not qualify.

For instance, if you need to calculate the number of employees of a department, there is no point in retrieving all of the employee rows to count them in your application. You can ask DB2 to perform this calculation on your behalf. This is a trivial example, but the logic applies in general.

Providing as many predicates as possible is even more important when you access a partitioned table. By specifying the partitioning key, you allow DB2 to look for qualifying rows only in the specified partition; otherwise, all partitions have to be scanned producing unnecessary I/Os. The access path can even get worse if your DB2 subsystem does not exploit I/O parallelism. See also “Page range screening” on page 101.

If you are sure about a certain value for a specific column in your WHERE clause while developing the application, code it directly, do not use host variables. Do not move your constant values to host variables. See Example 8-3 on page 193.

Example 8-3 Constant values in host variables

```
MOVE 1        TO HV-COLa

SELECT COLc FROM T1
 WHERE COLa = :HV-COLa              -- :HV-COLa always equals 1
   AND COLb = :HV-COLb              -- :HV-COLb depends on business case
```

Rather, code your SQL statement. See Example 8-4.

Example 8-4 Constant values in SQL statement

```
SELECT COLc FROM T1
 WHERE COLa = 1
   AND COLb = :HV-COLb              -- :HV-COLb depends on business case
```

Assuming you do not use the REOPT(VARS) bind option, the use of host variables can make the optimizer choose an access path at bind time which is less efficient than the optimizer would have chosen if you had provided the literal value. Generally, the optimizer's decisions are based on available indexes, the way the SQL statement is coded, and the information you pass to DB2. If you have that information, help DB2 to make the best choice.

8.3 SELECT column list

Limit the columns in your SQL select statement to those your application really needs. You incur extra CPU cost for passing the columns from DB2 to your application. But this is not the only consideration, retrieving additional columns can cause an additional access to the data pages instead of an index-only access, resulting in additional I/Os, accumulating higher CPU, and elapsed time.

Only select the columns you really need in your application and do not select the columns that you only use in your WHERE clause with equal predicates. You can avoid DB2 pushing unnecessary values to your application, reducing CPU consumption.

Using GROUP BY and ORDER BY when selecting all columns from a table requires work files used by DB2 to perform sorts larger than the data. Consider that VARCHAR columns involved in a SORT are padded to full length to allow sorting them.

When using an isolation level of UR, the possibility exists that the value of a column in the index used to qualify a row is different than the value on the data page. This scenario is shown in "UR: Uncommitted read" on page 337. For this reason you may consider including columns in your select list that are also specified in the WHERE clause with equal predicates.

You can specify columns in the ORDER BY clause that are not present in the SELECT list.

> **Important:** If you use SELECT * on VIEW and ALTER ADD columns on the underlying table, you must drop and recreate the view for those new columns in order to become part of the view. It does not happen automatically.

8.4 ORDER BY clause

Whenever you need data back in sequence, specify an ORDER BY clause. Even if DB2 decides to use an index to access data today and therefore qualifying rows are returned in index sequence, it does not mean that it accesses the data in the same way tomorrow. If you

specify ORDER BY, and DB2 can use index access, it avoids the additional sort. If for whatever reason, the access path changes and the index is no longer used, DB2 performs the additional sort guaranteeing the data sequence you requested.

8.4.1 ORDER BY sort avoidance

Since DB2 for z/OS Version 7, columns can be *logically removed,* during a forward index scan, from an ORDER BY clause and an index key, if their values are constant. This enables DB2 to increase the instances of sort avoidance. With DB2 V8, this is also applicable to backward index scan (using dynamic prefetch).

Assume an index on COLb, COLa, COLe, COLd, and COLc.

In Example 8-5, the predicates for columns COLb, COLd, and COLe specify a constant value. Therefore, DB2 can logically remove these columns from the ORDER BY clause without changing the order requested by the user. DB2 can also logically remove these columns from the index key without changing the ordering capability of the index.

Note that logically removing columns from an index key has no impact on the number of I/Os performed for the index compared to not logically removing the columns.

Example 8-5 Logically removed ORDER BY columns

```
SELECT COLa, COLb, COLc, COLd FROM T1
 WHERE COLb = 5
   AND COLd = 7
   AND COLe = 2
ORDER BY COLa, COLb, COLc, COLd
```

Introduced in DB2 for z/OS Version 8, backward index scan can also be used to avoid an additional SORT for a specified ORDER BY clause:

- Use an ascending index to provide a descending sort order.
- Use a descending index to provide an ascending order.

To benefit from index backward scan, the following conditions must be met:

- The index must be defined on the same columns as ORDER BY.
- The ordering of the index must be exactly opposite of what is requested by ORDER BY.

See Example 8-6 for a sample of backward index scan.

Example 8-6 Using backward index scan

```
If index defined as COLa DESC, COLb ASC, DB2 can

 - Forward scan for ORDER BY COLa DESC, COLb ASC
 - Backward scan for ORDER BY COLa ASC, COLb DESC

A SORT is required for

 - ORDER BY COLa ASC, COLb ASC
 - ORDER BY COLa DESC, COLb DESC
```

Consider rebinding static plans and packages when:

- An ascending index exists for SELECT ... ORDER BY ... DESC
- A descending index exists for SELECT ... ORDER BY ... ASC

Query performance may improve significantly after rebinding your plans and packages.

Note that the ORDER BY columns do not have to be in the list of selected columns.

8.5 Subquery to join transformation

Depending on several conditions, DB2 can perform a subquery to join transformation for SELECT, UPDATE, and DELETE statements between the result table of a subquery and the result table of an outer query.

A complete list of conditions for a subquery to join transformation can be found in *DB2 for z/OS Application Programming and SQL Guide,* SC18-7415-02.

A major advantage of subquery to join transformation is the flexibility for DB2 to decide the most efficient order to process the tables.

It is possible that, in some cases, subquery to join transformation will degrade performance. This is most likely to happen when the outer query returns a small number of rows and the subquery returns a large number of rows. But you can also find the same effect the other way around. In such a case, transformation provides no performance benefit, but there is additional overhead to prepare the statement with transformation activated.

Tip: PTF UQ50928 for APAR PQ45052 provides you with the ability of disabling correlated subquery to join transformation in DB2 Version 7 at the system-wide level.

You can find a candidate for a noncorrelated subquery to join transformation in Example 8-7.

Example 8-7 Candidate query for subquery to join transformation

```
SELECT * FROM T1
 WHERE COLa IN
      (SELECT COLa FROM T2
        WHERE COLb = '10');
```

Since you do not tell DB2 that the select statement is read-only, DB2 is looking for an access path where it can maintain the current cursor position to allow for updates on the result set.

If you specify FOR FETCH ONLY, then DB2 does not need to know the current position of your cursor on the base table, because you are not going to update qualifying rows. Therefore, a more efficient access path for read-only cursors may be available to the optimizer. See Example 8-8 to allow the optimizer for another join sequence.

Example 8-8 Specifying FOR FETCH ONLY to improve optimizer's choices

```
SELECT * FROM T1
 WHERE COLa IN
      (SELECT COLa FROM T2
        WHERE COLb = '10')
FOR FETCH ONLY;
```

Tip: While FOR UPDATE OF forces a certain access path, FOR FETCH ONLY provides more opportunities to the optimizer to choose the most efficient access path.

To find out if DB2 transforms a subquery into a join, look at the column QBLOCKNO in your plan table:

- DB2 does not transform a subquery into a join if the value of QBLOCKNO is greater than the value of the outer query.
- A subquery is transformed into a join if both inner and outer query have the same value of QBLOCKNO. The join method is indicated by a value of 1, 2, or 4 in column METHOD.

8.6 Indexable, stage 1, and stage 2 predicates

To benefit most from your physical database design while developing your application processes, you need to take advantage of the existing indexes. Try to find the access path which satisfies your needs in CPU time and elapsed time.

If the predicate is applied in the earlier stage, the performance is improved. If the predicate is used as an argument for a matching index, DB2 does not need to read data pages not satisfying the predicate. On the other hand, if the predicate is stage 2, DB2 has to read rows that do not satisfy the predicate and send them to further analysis by DB2.

If an index is used, index screening filters the rows that qualify. Once the rows are retrieved from a query, they are passed through two stages of processing:

- Stage 1 predicates can be applied at the first stage.
- Stage 2 predicates cannot be applied until the more complex analysis in the second stage. They are also known as *residual predicates*.

At run time, during the first stage of processing, DB2 will apply the simple stage 1 predicates and pre-qualify a row. They are evaluated first to reduce processing cost and eliminate the number of rows to be evaluated by the complex predicates. Only those rows that pre-qualify proceed to the next stage, which requires additional processing time, where DB2 finally qualifies a row or rejects it. Internally, predicates are rearranged to separate stage 1 from stage 2 predicates.

The effort is to try to code only for stage 1 predicates, since they are better than stage 2 predicates, and, only if not achievable, stage 2 predicates. Whenever possible, avoid rejecting a row by your application after retrieving it, you can refer to this as stage 3 processing.

Figure 8-1 on page 197 shows one example of the execution of one SQL statement using an index and shows the difference between stage 1 and stage 2 predicates.

The SELECT statement reads all orders from the table TB1 that has order number between 15000 and 17000. DB2 specifies the use of the index on column COLa starting at key value 15000. DB2 uses the index tree to find the leaf page containing the value 15000 and scans forward until it accesses the leaf page containing the value 17000. Then it stops the scan. GETPAGEs are issued for each index page referenced.

In complex SQL statements with multiple predicates, they are applied in this sequence:

1. Matching index predicate
2. Other index key predicates applied (index screening)
3. All other stage 1 predicates
4. All stage 2 predicates

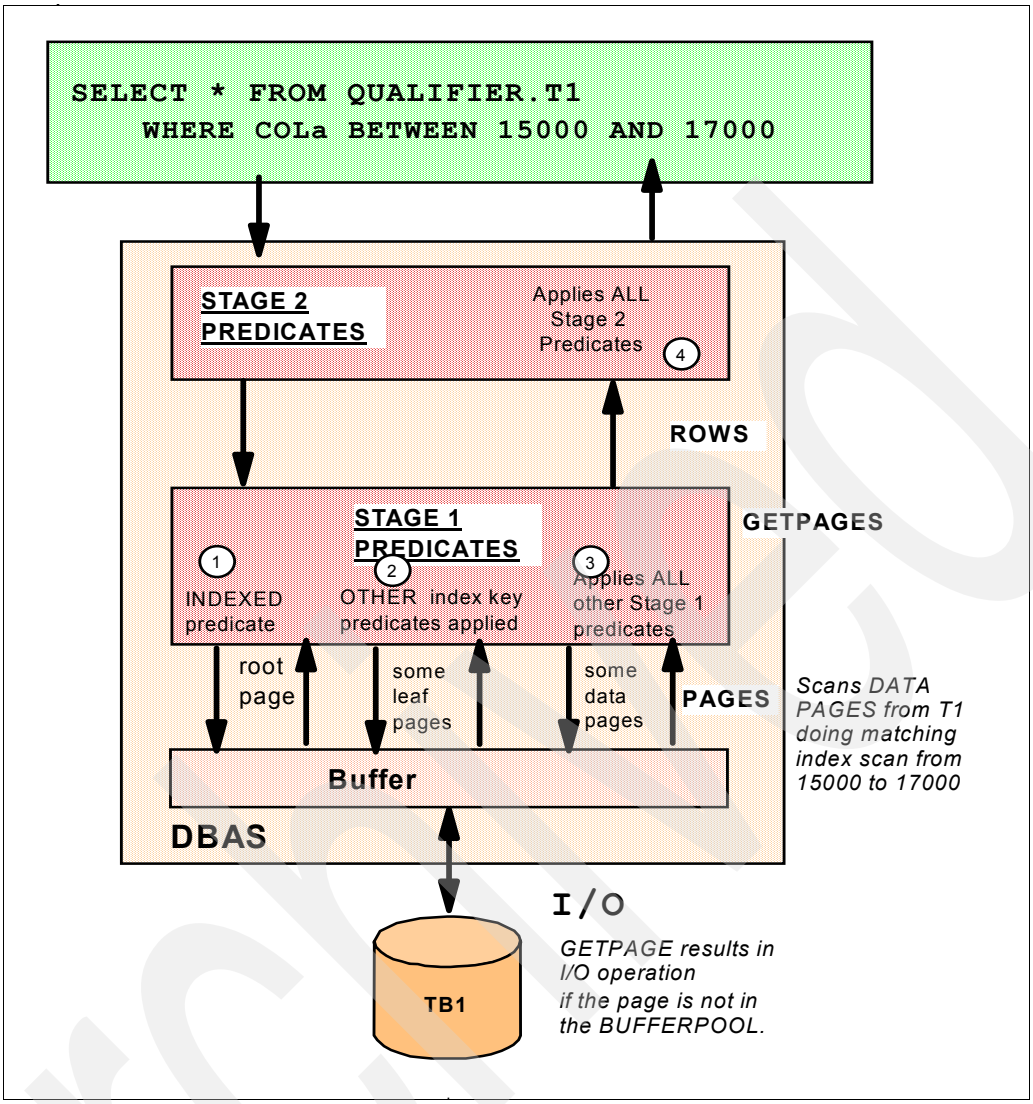


Figure 8-1 Analysis of stage 1 and stage 2 predicates

Make sure that left hand side and right hand side predicates' data type, length, and nullability attributes match each other when coding your SQL statement, even if DB2 is able to use index access on nonmatching predicates. Use noncolumn expressions if possible. Avoid CASTing between data types, if this is not absolutely necessary to get index access. You should check for each new release of DB2 to see if some of the formerly CASTed statements can be eliminated to save CPU costs.

With DB2 for z/OS Version 8, more predicates have become indexable, even if data type, length and nullability attributes do not match. This allows some stage 2 predicates to become stage 1 predicates, and stage 1 predicates allow more index usage, and it applies to both join and local predicates. DB2 allows index access when host variable and target column are not the same data type and length, for example:

- Column is DECIMAL, host variable is INTEGER
- Column is CHAR (3), literal or host variable is CHAR (4)

The potential for performance improvement with no application change is very significant.

Tip: This improvement is being made available also in DB2 V8 compatibility mode, not just new-function mode, via APAR PK12389.

Basically, besides significantly improving performance for many applications, the enhancement looks at programming languages that do not support the full range of SQL data types and lengths, such as:

- C/C++ has no decimal data type on workstations
- Java has no fixed length CHAR data type

See Table 8-1 for a list of predicates you still need to rearrange in DB2 for z/OS Version 8, which are not rearranged by the optimizer.

An indexable predicate is a predicate that can use a (matching) index access as an access path. Indexable predicates are always stage 1, but not all stage 1 predicates are indexable.

Table 8-1 Moving predicates to a better performing stage

Stage 2 nonindexable	Stage 1 nonindexable	Stage 1 Indexable
QTY * 2 = HV-dec		QTY = :HV-dec / 2
YEAR(DCOL) = 2003		DCOL BETWEEN ‘2003-01-01’ AND ‘2003-12-31’
:HV BETWEEN COLa AND COLb		:HV >= COLa AND :HV <= COLb
DCOL + 10 YEARS < CURRENT DATE		DCOL < CURRENT DATE - 10 YEARS
LOCATE (‘P’, LASTNAME) > 0	LASTNAME LIKE ‘%P%’	
	DCOL <> ‘9999-12-31’	DCOL < ‘9999-12-31’
	GENDER <> ‘F’	GENDER = ‘M’

You can find a complete list of indexable predicates in *DB2 UDB for z/OS Version 8 Application Programming and SQL Guide,* SC18-7415-02.

8.6.1 Mismatched numeric types

Figure 8-2 on page 199 shows an example of a predicate using mismatched numeric types (that are stage 2 in Version 7 and become stage 1 and indexable in Version 8 new-function mode).

In this example, the selection of rows from the employee table has to scan the table space in Version 7 because the predicate COLb, defined as decimal (12,2), is compared to a host variable, defined as floating point that is stage 2.

The same predicate is stage 1 in Version 8 and it allows matching index scan to find the rows for the execution of the SQL.

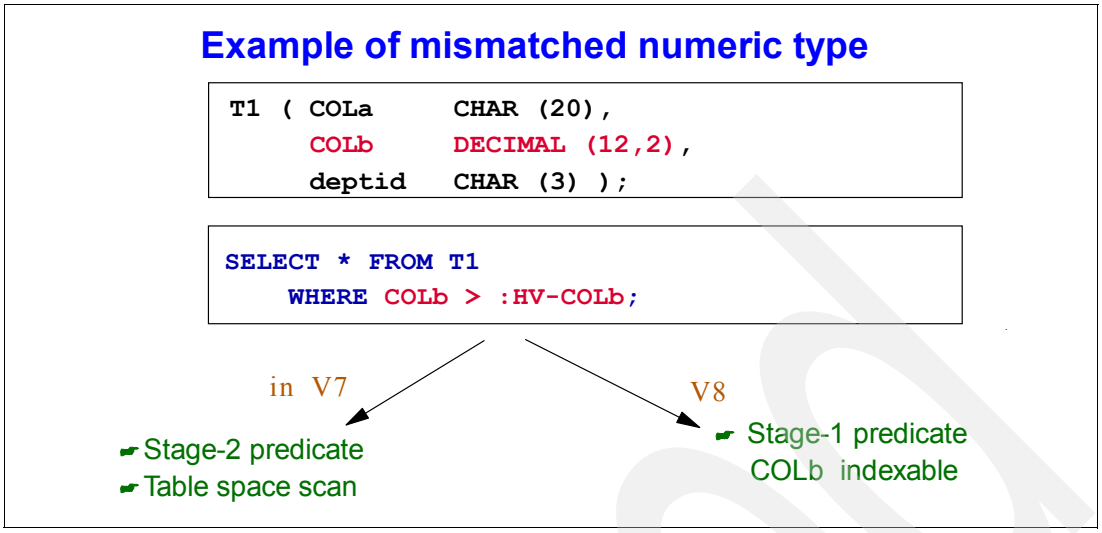


Figure 8-2 Example of mismatched numeric types

But not all mismatched numeric types become stage 1. The exceptions that are still stage 2 predicates are:

- REAL → DEC(p,s) where p >15

  where REAL is the right hand side predicate → DEC(p,s) is the left hand side predicate
- FLOAT → DEC(p,s) where p >15
  - REAL or FLOAT column in the outer table of a join
  - REAL or FLOAT literal or host variable
  - DEC_column > REAL_hostvar

REAL and FLOAT values cannot be converted to decimal (the index column) with precision > 15 without possibly changing the collating sequence.

8.6.2 Mismatched string types

Figure 8-3 on page 200 shows examples of mismatched string types that were stage 2 in Version 7 and are stage 1 and indexable in DB2 for z/OS Version 8.

Archived

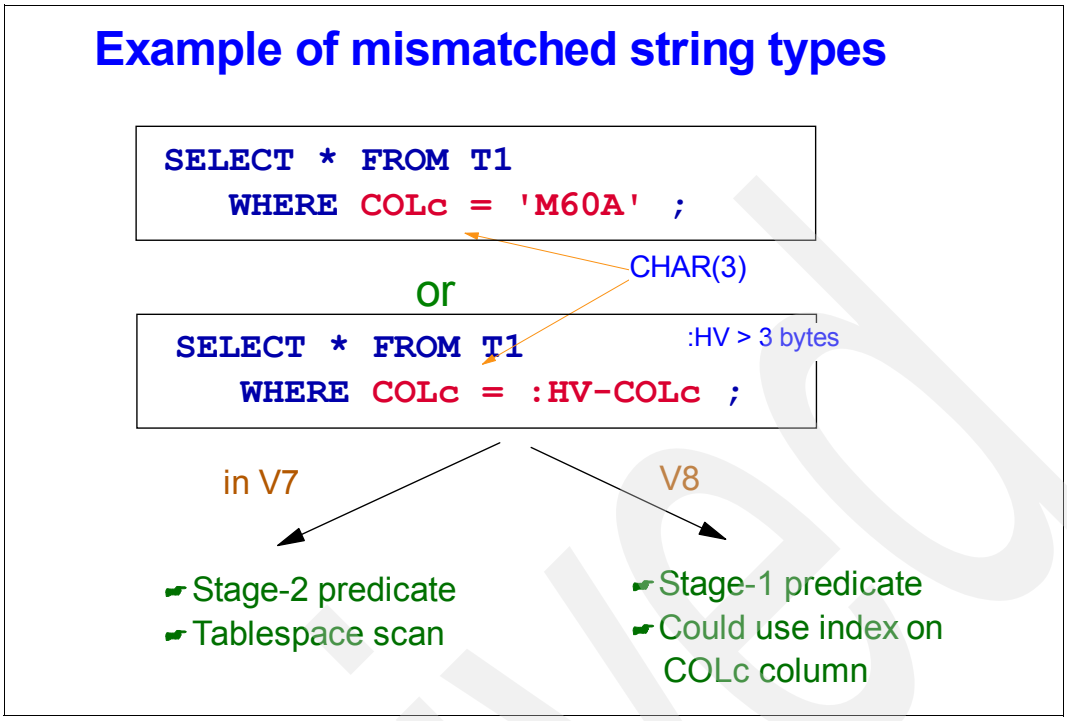


Figure 8-3 Example of mismatched string types

In the example, the selection of rows from table T1 has to scan the table space in DB2 for z/OS Version 7 because the column COLc is defined as CHAR(3) and is compared to a string of four characters. This is a stage 2 predicate in DB2 for z/OS Version 7.

The same predicate is stage 1 in DB2 for z/OS Version 8 and it allows a matching index scan.

Not all mismatched string types become stage 1. The exceptions that are still stage 2 predicates are:

- graphic/vargraphic → char/varchar

  where, for example, graphic is the right hand side predicate → char is the left hand side predicate
- char/varchar(n1) → char/varchar(n2)

  where n1>n2 and not equal predicate
- graphic/vargraphic(n1) → graphic/vargraphic(n2)

  where n1>n2 and not equal predicate
- char/varchar(n1) → graphic/vargraphic(n2)

  where n1>n2 and not equal predicate

8.6.3 Performance

Performance measurements show the type of benefits that can be achieved using mismatched numeric data types and mismatched string types in predicates that were stage 2 in DB2 for z/OS Version 7 and are stage 1 in DB2 for z/OS Version 8. For performance measurements on mismatched data types, see *DB2 UDB for z/OS Version 8 Performance Topics,* SG24-6465.

8.6.4 Recommendations

Consider rebinding plans or packages to get better access paths after migrating to DB2 for z/OS Version 8. Also, consider creating an index for the mismatched columns to enable index access.

8.7 Considerations on the CAST function

With versions prior to DB2 Version 8, you could use the CAST function to make both data types the same. However, if you do this, you generally dictate the join sequence, because DB2 favors the index and picks the simple column side as inner table. Let us look at the SELECT of Example 8-9, where the CUST_NO is defined as DEC(15.0) in ORD_DEC and INTEGER in CUST.

Example 8-9 Dictating join sequence

```
SELECT *
FROM CUST AS C, ORD_DEC AS O
WHERE C.CUST_NO = O.CUST_NO;
```

If you rewrite the predicate as “CUST_NO = CAST(O.CUST_NO AS INTEGER)”, the major advantage is that an available index on C.CUSTO_NO will now be used. Furthermore, since ORD_DEC.CUST_NO is DEC(15,0), this has a bigger range than the INTEGER type. Unless you are really sure that all the values in the ORD_DEC.CUST_NO are within the range of INTEGER, the CAST may fail and DB2 may return an error. In DB2 for z/OS Version 8, unless you really want to dictate the join sequence, it is better not to add any CAST and let DB2 choose the join sequence based on costing. Generally this provides better performance.

8.8 Selective predicates

When you use SQL as a 4th generation programming language, you describe the data you want and not where the data can be found. This does not mean you do not have any influence in how your DBMS is looking for qualifying rows. Even if the optimizer in most cases decides to use reasonable access paths you may have chosen by yourself, there are cases where you can guide the optimizer in the right direction. You should follow the rules shown below when coding your SQL statements:

- When predicates are ANDed together, code the most selective predicate first.

  By coding the most selective predicates first, the number of rows qualifying for applying the next predicate can be reduced dramatically, resulting in a reduced number of I/Os and elapsed time.

- When predicates are ORed or INed together, code the most selective predicate last.

  An IN or OR statement is true as soon as the first expression is true. As soon as a row qualifies, DB2 can stop looking at remaining predicates.

8.9 Subqueries

The same rules mentioned in ”Selective predicates” apply to usage of subqueries and are listed below.

- Code the most selective, first.

  If fewer rows qualify on a first subquery, only a smaller number of rows qualifies for predicate evaluation on other predicates, avoiding unnecessary I/Os.

DB2 allows you to take control of the execution of your subquery statement, providing you the ability to switch subqueries on and off, depending on your current needs. This technique can assist you if no dynamic SQL is implemented, but still different conditions exist where you may want to execute a subquery or not, depending on the input data entered by users.

See Example 8-10 for an example of controlling execution of a subquery.

Example 8-10 Control execution of subqueries

```
SELECT  COLa, COLb FROM T1
 WHERE  COLa = :HV-COLa
   AND EXISTS
            (SELECT 1 FROM T2
              WHERE 0       = :HV-SUBQUERY-1
                AND COLc    = :HV-COLc
                AND T1.COLb = T2.COLb)
    OR  EXISTS
            (SELECT 1 FROM T3
              WHERE 0       = :HV-SUBQUERY-2
                AND COLd    = :HV-COLd
                AND T1.COLb = T3.COLb));
```

By providing variable HV-SUBQUERY-1 with value 0, DB2 executes the first subquery. The same applies to subquery two and variable HV-SUBQUERY-2. Any value other than 0 stops DB2 from executing the subquery.

Tip: For DB2 V7 and DB2 V8, APAR PQ92434 (respectively with PTF UQ94790 and UQ94791) enhances the performance of SQL statements with predicates involving host variables.

8.10 QUERYNO in SQL statements

When coding your application programs, make sure you provide a unique QUERYNO for each SQL statement you code. The QUERYNO clause specifies the number to be used for this SQL statement in EXPLAIN output and trace records (therefore, also available in monitoring tools you might use).

The number is used for the QUERYNO columns of the plan tables for the rows that contain information about this SQL statement. This number is also used in the QUERYNO column of SYSIBM.SYSSTMT and SYSIBM.SYSPACKSTMT. This allows faster tracking back to a problem causing query.

If the clause is omitted, the number associated with the SQL statement is the statement number assigned during precompilation. The number may change if the application program changes as well. See Example 8-11 on page 203 for assigning QUERYNO 1 on an SQL statement.

Example 8-11 Including QUERYNO in a SQL statement

```
SELECT COLa FROM T1
 WHERE COLb = 1
QUERYNO 1
```

8.11 Reduce trips to DB2

In many cases, you can perform the same action in different ways using the power of SQL. Whenever possible, code your SQL in such a way to reduce the number of times that your application must call DB2 since each roundtrip to DB2 requires CPU time.

The most obvious benefit occurs when OPEN/FETCH/CLOSE or OPEN/FETCH/UPDATE CURSOR/CLOSE is changed to singleton SELECT or UPDATE.

However, in general, this does not necessarily mean that combining many “small” SQL statements into one very large SQL statement is always better performing. The access path is still the critical element, check the new access path carefully, because it may result in more elapsed and CPU time than all the single statements.

We now examine three instances of singleton SELECTs.

8.11.1 OPTIMIZE FOR *n* ROWS

In general, DB2 assumes that the application will retrieve all qualifying rows for a declared cursor statement. This assumption is most appropriate for batch environments. In the case of interactive applications, you may not want to retrieve the whole result set, but just the first ***n*** rows. The access path to retrieve only *n* rows instead of all qualifying rows can be completely different. See Example 8-12 for how to code the optimization clause.

Example 8-12 Specifying OPTIMIZE FOR n ROWS

```
DECLARE CURSOR C1 FOR
  SELECT COLa FROM T1
   WHERE COLb = 1
  OPTIMIZE FOR 10 ROWS;
```

DB2 uses the OPTIMIZE for *n* ROWS clause to choose appropriate access paths that minimize response time. Possible effects of using OPTIMIZE FOR *n* ROWS are:

- The join method can change. Nested loop join is the most likely choice, because it has low overhead cost and appears to be more efficient if you want to retrieve only a few rows.
- An index that matches the ORDER BY clause is more likely to be picked. This is because no sort is needed for ORDER BY.
- List prefetch is less likely to be picked.
- Sequential prefetch is less likely to be requested by DB2 because it infers that you only want to see a small number of rows.
- In a join query, the table with the columns in the ORDER BY clause is likely to be picked as the outer table if there is an index on that outer table that gives the ordering necessary for the ORDER BY clause.

Use OPTIMIZE FOR *n* ROWS to avoid sorts because it tells DB2 to select an access path that returns the first qualifying rows quickly. This means, that whenever possible, DB2 avoids

any access path involving a sort. For values > 1 you may not necessarily avoid a sort. Fetching the entire result set is possible even if a smaller number of *n* is provided.

In the special case of using OPTIMIZE FOR 1 ROW, DB2 disables list prefetch. If you continue fetching more than one row, list prefetch is not turned on dynamically.

Important: Specifying a small number on OPTIMIZE FOR *n* ROWS clause, but fetching the entire result set can significantly increase elapsed time. There will be no SQLCODE 100 issued after reading *n* rows if there are still more qualifying rows left.

8.11.2 FETCH FIRST *n* ROWS ONLY

In some applications, you execute queries that can return a large number of rows, but you need only a small subset of those rows. Retrieving the entire result table can be inefficient. You can specify the FETCH FIRST *n* ROWS ONLY clause in a SELECT statement to limit the number of rows in the result table to *n* rows. After fetching *n* rows, SQLCODE 100 is issued. If fewer rows qualify, SQLCODE 100 is issued after the entire result set has been fetched which can be fewer than the value of n.

Using FETCH FIRST *n* ROWS allows you to specify ORDER BY to retrieve qualifying rows in a desired order. Note that this can cause an internal sort if the ORDER BY clause is not supported by an index, resulting in additional I/Os for reading all qualifying rows. All qualifying rows are sorted, even if only FETCH FIRST 1 ROW is specified.

Tip one: You can also use ORDER BY in combination with FETCH FIRST 1 ROW ONLY. If you do not access data using an unique index, additional getpages occur.

Tip two: If you specify FETCH FIRST *n* ROWS ONLY but not OPTIMIZE FOR *n* ROWS in a SELECT statement, DB2 optimizes the query as if you had specified OPTIMIZE FOR *n* ROWS. If both clauses are used, the value for OPTIMIZE FOR *n* ROWS clause is used for access path selection.

8.11.3 Existence checking

You can use a singleton SELECT statement for existence checking (to see if at least one row qualifies) if you do not actually need the data itself. Instead of selecting column values, simply select a constant value to avoid unnecessary columns passed to your application.

Often a singleton SELECT is used to perform existence checking and your application does not need to know the content of the row. For application logic SQLCODEs 0 and -811 are treated in the same way.

With the addition of the FETCH FIRST *n* ROWS ONLY clause in V7, you can code the FETCH FIRST 1 ROW ONLY clause on a SELECT INTO statement to indicate that only one row is to be returned to the program, even if multiple rows match the WHERE criteria. In this case application performance can be improved by adding FETCH FIRST 1 ROW ONLY because it prevents DB2 from looking for more qualifying rows, saving I/Os and reducing elapsed time for your query. For a simple example, see Example 8-13.

Example 8-13 Existence checking with singleton SELECT

```
SELECT 1 FROM T1
 WHERE COLa = 10
FETCH FIRST 1 ROW ONLY;
```

In Example 8-14, you do not need the ORDER BY since you do not retrieve a column. However, sometimes you want to check if one row exists and retrieve the most recent one (if more than one exist), so an:

```
ORDER BY TIMESTAMP_COL DESC
```

can help, since the alternative is to use a cursor.

A DB2 V8 new-function mode enhancement allows you to specify an ORDER BY clause to affect which row should be returned on a singleton select. When you use the FETCH FIRST 1 ROW ONLY with the ORDER BY clause, the result set is retrieved and ordered first, and then the first row is returned. For example, using the sample employee table, for all employees with a salary of more than $40,000, move the salary of the employee who has been employed the longest in host variable :HV1. You can code this query as you see in Example 8-14.

Example 8-14 SELECT INTO using ORDER BY

```
SELECT SALARY
FROM DSN8810.EMP
INTO :HV1
WHERE SALARY > 40000
ORDER BY HIREDATE FETCH FIRST ROW ONLY;
```

8.12 Cursor versus searched update and delete

DB2 provides two methods to manipulate rows from within an application program:

- Searched update and delete statements
- Positioned update (cursors using FOR UPDATE OF)

You may think of using searched update and delete statements to simplify application coding, but have you ever thought of the impact on other user's transactions or batch jobs?

Using searched UPDATE or DELETE statements means manipulating an unknown number of rows if the WHERE clause does not complement available unique indexes. An unknown number of rows affected by DML can lock many resources, preventing others from accessing the data. Depending on your lock escalation system parameter NUMLKTS, and the LOCKMAX parameter specified at table space level, lock escalation for each affected table can occur, preventing all concurrent access to the table not using uncommitted read. So, depending on the number of affected rows, a searched update or delete statement might not be a good solution.

Do not use a singleton select or a cursor declared FOR FETCH ONLY to read rows you are going to manipulate afterwards. If you use ISOLATION (CS) or ISOLATION (UR) for your package, DB2 cannot guarantee that your data stays the same until you update or delete the row you have read before.

Depending on the number of rows qualifying your WHERE clause, you should consider using a cursor as you see in Example 8-15 for deleting more than one row. Using a cursor allows you to take intermediate commit points.

The second way to manipulate data, using a FOR UPDATE OF cursor, can help with concurrency.

Example 8-15 Declaration of FOR UPDATE OF cursors

```
DECLARE C1 CURSOR WITH HOLD FOR
  SELECT COLb, COLc FROM T1
```

```
   WHERE COLa = :HV-COLa
FOR UPDATE OF COLb, COLc;
```

You can delete a previously fetched row by issuing the embedded statement you see in Example 8-16.

Example 8-16 DELETE using FOR UPDATE OF cursors

```
DELETE FROM T1
 WHERE CURRENT OF C1
```

For updating a fetched row, see Example 8-17.

Example 8-17 UPDATE using FOR UPDATE OF cursors

```
UPDATE T1
   SET COLb = :HV-COLb
      ,COLc = :HV-COLc
 WHERE CURRENT OF C1
```

If there is no need to retrieve old column values, you can save the additional overhead for passing column values to your application program by simply selecting a constant value instead of different columns.

Avoid updating primary keys if you use RI to prevent your applications from encountering referential integrity error scenarios.

Since your application program gets back control after each execution of an embedded SQL statement, using a cursor gives you the ability to take control of COMMIT frequency instead of performing a searched UPDATE or DELETE.

Additionally, you can minimize concurrency problems by specifying the FOR UPDATE OF clause, because on each fetch of a FOR UPDATE OF cursor, DB2 acquires an update lock (U lock) on a page where a fetched row resides (a row lock is taken if the associated table space was defined using LOCKSIZE (ROW)). This prevents concurrently running processes from manipulating the row your application has fetched, but not from reading the row. In case you update or delete the row, the U lock is propagated to an exclusive lock (X lock), which prevents even readers from accessing the row (assuming no UR is used on concurrently accessing applications). As soon as your application issues a COMMIT, all table, page, and row locks are released and your changes are visible for other applications.

If you cannot use the FOR UPDATE OF clause on a cursor, which is the case if you have to access data using an ORDER BY clause, you can specify USE AND KEEP UPDATE LOCKS on your select statement.

Using the statement shown in Example 8-21, DB2 takes an update lock at the time data is accessed. If materialization occurs, DB2 takes the update locks at OPEN CURSOR time and releases them at COMMIT.

Example 8-18 Use of USE AND KEEP UPDATE LOCKS

```
DECLARE C2 CURSOR WITH HOLD FOR
  SELECT COLb, COLc FROM T1
   WHERE COLa = :HV-COLa
  ORDER BY COLb
  WITH RS
  USE AND KEEP UPDATE LOCKS;
```

With isolation level CS, the behavior is different: DB2 releases the locks as it moves to the next row or page depending on your locksize.

A basic requirement for using USE AND KEEP UPDATE LOCKS is specifying isolation level RR or RS on the statement. Additionally, a searched update can be issued, specifying the same qualifying predicates in the WHERE clause.

Without using FOR UPDATE OF cursors and KEEP UPDATE LOCKS, you would access your qualifying data using a singleton select or a FETCH ONLY cursor and issue an UPDATE or DELETE statement afterwards.

To make sure data has not changed if you issue separate UPDATE or DELETE statements, you can code a retry logic to check if a certain row has been modified in the meantime by another process. If the only intention is to delete using FOR UPDATE OF clause, you do not need to retrieve column values, just select a constant value so no data has to be passed to your application program when rows are fetched. The only reason to select column values is if you have to log the data you are going to delete.

But be aware that a single SQL statement can manipulate a large number of rows. Even if you obviously only delete 10 rows from a table, DB2 can delete 15 million rows because of referential integrity cascading deletes. For more suggestions about deleting a large amount of data, see “Massive insert, update, and delete” on page 264.

The main concerns of those locks caused by deleting a huge amount of rows are:

- A COMMIT cannot be issued during the SQL statement execution.
- The locks acquired for the updated rows are held until a COMMIT after completion of the statement.
- The long duration of the locks may cause lock contention and eliminate online concurrency.
- The unit of work is long, so rollback times in case of abend are long, too. At restart, the delete process starts over from the beginning by deleting all rows again.

We recommend you use a cursor instead of the searched update or delete statement which allows you to commit your changes frequently and thereby releasing resources. For cascading deletes even the cursor technique does not provide you the ability to issue COMMITs during cascading deletes.

SQLERRD(3) in SQLCA contains the number of rows affected after each INSERT, UPDATE, and DELETE statement (but not rows deleted as a result of CASCADE delete). For the OPEN of a cursor for a SELECT with INSERT or for a SELECT INTO, SQLERRD(3) contains the number of rows inserted by the embedded INSERT. The value is 0 if the statement fails and -1 for a mass delete from a table in a segmented table space.

8.12.1 SQL precompiler option NOFOR

To control the DB2 precompiler or an SQL statement coprocessor, you specify options when you use it. There are various options available you can specify. The options allow you to control how the SQL statement processor interprets or processes its input, and how it presents its output. You can find more information about SQL preprocessor options in *DB2 UDB for z/OS Version 8 Application Programming and SQL Guide,* SC18-7415-02.

If you are using the DB2 precompiler, you can specify SQL processing options in one of the following ways:

- With DSNH operands
- With the PARM.PC option of the EXEC JCL statement
- In DB2I panels

If you are using the SQL statement coprocessor, you specify the SQL processing options in the following way:

- For C or C++, specify the options as the argument of the SQL compiler option.
- For COBOL, specify the options as the argument of the SQL compiler option.
- For PL/I, specify the options as the argument of the PP(SQL('option,...')) compiler option.

DB2 assigns default values for any SQL processing options for which you do not explicitly specify a value. Those defaults are the values that are specified in the APPLICATION PROGRAMMING DEFAULTS installation panels.

In static SQL, NOFOR eliminates the need for the FOR UPDATE OF clause in DECLARE CURSOR statements. When you use NOFOR, your program can make positioned updates to any columns that the program has DB2 authority to update. When you do not use NOFOR, if you want to make positioned updates to any columns that the program has DB2 authority to update, you need to specify FOR UPDATE with no column list in your DECLARE CURSOR statements. The FOR UPDATE clause with no column list applies to static or dynamic SQL statements.

Whether you use or do not use NOFOR, you can specify FOR UPDATE OF with a column list to restrict updates to only the columns named in the clause and specify the acquisition of update locks.

You imply NOFOR when you use the option STDSQL(YES).

If the resulting DBRM is very large, you may need extra storage when you specify NOFOR or use the FOR UPDATE clause with no column list.

8.13 COMMIT frequency

Have you ever heard about the *fear of committing*? You can have a list of reasons why a single application does not commit, but not committing is even worse if you are not alone in the system (which is usually the case). If you do not commit, you:

- Do not free locked resources.
- Prevent others from accessing your manipulated data.
- Can cause timeouts for concurrently running applications.
- Can prevent utilities from completing.
- May run into painful rollback scenarios.

Because of the reasons shown above, issuing COMMITs frequently is a basic rule for allowing availability and concurrency in high availability environments. If you have no idea about commit frequency, you can start using the following rule-of-thumb:

- Make sure no lock is held for more than five seconds.
- Make sure no single object is locked for more than 10% of elapsed time in U- or X-mode (avoid hot spots).
- Always UPDATE/DELETE using WHERE CURRENT OF CURSOR.

- COMMIT takes 1 to 4 ms to complete, so do not overcommit.

  Deleting 10,000 rows including 10,000 commit points adds 10,000 x 4 ms = 40 sec to elapsed time.

If you follow these recommendations, you will not have any locking nightmares.

You can find more information on COMMITs in “Commit” on page 348.

For non-select SQL operations, you can calculate the time of objects being locked using the following formula:

Time of object being locked = (A x B x C) / D

where:

- *A* is the number of pages/rows that have changed during your commit interval.

  The value of A can be reduced as follows:

  – Use shorter commit intervals.
  – If you use page locking, consider changing the clustering sequence.

- *B* is the transaction rate; respectively, the number of commit intervals per minute for batch jobs.

  The value of *B* cannot be reduced in online environments (amount of transactions). Decreasing the value of *B* batch environments is useless, since the value for *C* increases.

- *C* is the lock duration.

  The value of C can be decreased as follows:

  – Reduce lock duration by doing all non-critical work before updating a hot spot.

- *D* is the number of pages/rows that are candidates for applying data manipulations.

  To enlarge the number of D, consider the following:

  – Row level locking.
  – If you use page level locking, consider changing the clustering sequence.
  – If you use page level locking, consider using MAXROWS.

The goal is to minimize values for A, B, and C, and it is to maximize the value for D.

Depending on your isolation level and usage of CURSOR WITH HOLD, issuing COMMITs is also important for read-only jobs.

You can find details about CURSOR WITH HOLD in “Cursors WITH HOLD” on page 350. For more information about COMMIT, see “Commit” on page 348.

Be aware that different attachments require different statements to commit or roll back the changes your application has made. See Table 12-7 on page 350 for a list of commands you have to issue to terminate a unit of work in several environments.

For IMS, CICS, or RRSAF environments, where DB2 is not the only resource manager where data has to be committed or rolled back, you must use the IMS calls, the CICS function, and the RRS commit rather than trying to use SQL. Talk to the COMMIT coordinator in charge.

Otherwise DB2 can only issue:

- SQLCODE -925 if you execute DB2 COMMIT
- SQLCODE -926 if you execute DB2 ROLLBACK

and in both cases, neither COMMIT nor ROLLBACK is performed.

8.13.1 SAVEPOINTs

From a DB2 point of view, ROLLBACK means to back out changes up to the last commit point. If you are need to roll back to different points in time inside your unit of work, consider using SAVEPOINTs. SAVEPOINTs allow you to set a savepoint within a unit of recovery to identify a point in time within the unit of recovery to which relational database changes can be rolled back. Be aware that only changes inside of DB2 are backed out, any other resources like IMS, CICS, or sequential files are not affected. A SAVEPOINT statement can be embedded in your application program or issued interactively. It is an executable statement that can be dynamically prepared. For an example of how to use savepoints, see Example 8-19.

Example 8-19 Using SAVEPOINTs

```
SAVEPOINT A UNIQUE ON ROLLBACK RETAIN CURSORS;
...
SAVEPOINT B UNIQUE ON ROLLBACK RETAIN CURSORS;
...
SAVEPOINT A UNIQUE ON ROLLBACK RETAIN CURSORS;
```

While savepoint A is not defined as unique, it can be reused inside the same application program. Three options are available for savepoints:

- UNIQUE

  Specifies that the application program cannot reuse the savepoint name within the unit of recovery.

  Omitting UNIQUE indicates that the application can reuse the savepoint name within the unit of recovery. If a savepoint was not created with the UNIQUE option, the existing savepoint is destroyed and a new savepoint is created.

- ON ROLLBACK RETAIN CURSORS

  Specifies that any cursors that are opened after the savepoint is set are not tracked, and, therefore, are not closed upon rollback to the savepoint. Although these cursors remain open after rollback to the savepoint, they may be unusable. For example, if rolling back to the savepoint causes the insertion of a row on which the cursor is positioned to be rolled back, using the cursor to update or delete the row results in an error.

- ON ROLLBACK RETAIN LOCKS

  Specifies that any locks that are acquired after the savepoint is set are not tracked, and therefore, are not released on rollback to the savepoint.

8.14 Temporary tables

In this section, we list the differences looking at created global temporary tables (CTTs) and declared global temporary tables (DTTs).

For example, you can use temporary tables for:

- Passing data between different programs inside an application process
- Storing non-DB2 data to join with data residing in DB2
- Storing temporary result sets

See Table 8-2 on page 211 for a comparison between CTTs and DTTs.

Table 8-2 Comparing CTTs and DTTs

Created global temporary tables (CTTs)	Declared global temporary tables (DTTs)
An empty instance is created at OPEN, SELECT INTO, INSERT, or DELETE operation on CTT.	An empty instance is created when a DECLARE GLOBAL TEMPORARY TABLE statement is executed.
Instance of CTT is destroyed when application process ends.	DTT is dropped when application process ends (see note below).
Any program in the application process can reference the same instance of CTT.	Any program in the application process can reference the same instance of DTT.
Instance of CTT is emptied if application process issues a COMMIT and no CURSOR WITH HOLD is opened on CTT.	Instance of DTT is only emptied if DTT is defined with ON COMMIT DELETE ROWS, application process issues a COMMIT and no CURSOR WITH HOLD is OPENED on DTT.
ROLLBACK includes CTT.	ROLLBACK includes DTT.
CTT is not allowed to be referenced in CREATE INDEX.	DTT is allowed for CREATE INDEX (schema name of the index has to be 'SESSION'). However first INSERT into DTT with index takes a long time.
LOCK TABLE is not allowed, but since the CTT is dedicated to the application process, LOCK TABLE does not make sense.	LOCK TABLE is not allowed, but since the DTT is dedicated to the application process, LOCK TABLE does not make sense.
No support for query parallelism.	I/O and CPU parallelism are supported.
DELETE from CTT only without WHERE clause.	DELETE FROM DTT with WHERE clause.
UPDATE of rows in CTT not allowed.	UPDATE of rows in DTT is supported.
CTT is visible in system catalog.	DTT is not visible in system catalog.
CTT is visible in SYSPACKDEP.	DTT is not visible in SYSPACKDEP.
CTT references are visible in SYSPACKSTMT.	DTT references are visible in SYSPACKSTMT.
Data access can be dynamic or static SQL.	Data access by implicitly dynamic SQL since the instance of the DTT is created at run time.
CTT is stored in work file database, and can span multiple files.	DTT is stored in TEMP database, and cannot span multiple files.
Qualifier for CTT chosen at CREATE TABLE.	Qualifier for DTT is 'SESSION'.
CONSTRAINTS cannot be defined on columns of a CTT.	CONSTRAINTS can be defined on columns of a DTT.
IDENTITY columns cannot be defined on columns of a CTT.	IDENTITY columns can be defined on columns of a DTT.
A column cannot be defined WITH DEFAULT on a CTT.	A column can be defined WITH DEFAULT on a DTT.
A VIEW can be created on a CTT.	A VIEW cannot be created on a DTT.
No logging available for CTTs.	Undo logging is available for DTTs.
SAVEPOINTs are not supported.	SAVEPOINTs are supported.

Note that several options on a DECLARE GLOBAL TEMPORARY TABLE statement can be specified to empty a DTT:

- ON COMMIT DELETE ROWS

  If no open cursor WITH HOLD accesses the DTT, all rows will be deleted.
- ON COMMIT PRESERVE ROWS

  All rows are preserved beyond COMMIT (thread reuse is not allowed).
- ON COMMIT DROP TABLE

  If no open cursor WITH HOLD accesses the DTT, the DTT is dropped.

Note also that a DTT is always used “under the covers” when using static scrollable cursor to store the result set of the query.

On a final note, make sure to specify the ON COMMIT DROP TABLE option, or explicitly DROP your DTT before committing in a distributed environment. Not doing so will prevent the connection from becoming inactive, and the DBAT being pooled at commit time.

Tip: When using a DTT, the optimizer has only very limited statistical information available about the data in the declared temporary table. The DTT is only declared at execution time. Therefore, the optimizer may not be able to come up with the best possible access path in all circumstances.

A CTT, on the other hand, is created beforehand and has a definition in the catalog. You can insert statistics in the catalog that can help the optimizer during access path selection. But be careful, because the table definition and statistics only exist once, but every instance of the table may have very different characteristics in terms of data.

8.15 Retrieving values generated at insert

New in DB2 Version 8, you can retrieve DB2 generated values at insert time. By providing this solution, you do not have to perform another roundtrip to DB2 just to receive a value assigned by DB2. This also saves additional CPU time as well as elapsed time.

In Example 8-20, we provide the DDL for a table where COLa is an identity column.

Example 8-20 Table including generated values by DB2

```
CREATE TABLE T2
      (COLa     INTEGER GENERATED ALWAYS AS IDENTITY
      ,COLb     CHAR(30) NOT NULL
      ,COLc     CHAR(30) NOT NULL WITH DEFAULT 'Text'
      ,COLd     DATE NOT NULL WITH DEFAULT);
```

Using the syntax provided in Example 8-21 on page 213, you can retrieve:

- Automatically generated values as ROWID or identity columns
- Default values for columns
- All values for an inserted row, without specifying individual column names
- All values that are inserted by a multi-row INSERT
- Values that are changed by a BEFORE trigger

Example 8-21 Retrieving values during INSERT

```
SELECT COLa, COLb, COLc, COLd
  FROM FINAL TABLE
      (INSERT INTO T2 (COLb, COLd)
               VALUES ('Name', '01.06.2004'));
```

You can retrieve by DB2 generated values using a singleton SELECT statement or a cursor. If you use a cursor for retrieving the rows that are going to be inserted, rows are always inserted at OPEN CURSOR. If an error occurs during SELECT from INSERT, it depends where the error has occurred whether the statement was successful or not.

- If SELECT INTO returns an error SQLCODE, no rows are inserted into the target table and no rows are returned (in effect, providing column values which are not allowed).
- If OPEN CURSOR returns an error SQLCODE, no inserts are done.
- If FETCH CURSOR returns an error SQLCODE, the result table contains the original number of rows that was determined during OPEN CURSOR processing. The application can decide whether to COMMIT or ROLLBACK.

The ability to retrieve values generated at update, such as

```
SELECT COLa from T1 (UPDATE T1 SET COLa = COLa + 1)
```

is not yet available.

8.16 Current statistics

If SQL statements do not perform very well, make sure that DB2 internal statistics collected by the RUNSTATS utility were up to date when your package was bound before manipulating statistics and using optimizer hints to push access paths back to the “good” way. In case your data has changed significantly, consider running RUNSTATS and REBIND to ensure that the optimizer can choose the most efficient access path. If you use Visual Explain, use the built-in statistics advisor to get information about your statistics. In general, consider REBINDing your package if an SQL statement does not perform as expected before rewriting the statement. Also, keep in mind that empty tables may be a common starting point for many new applications; therefore, no appropriate statistics exist. Also consider REBINDing your package if a former empty table is populated over time and statistics may be helpful in supporting the optimizer’s decisions.

Archived

Archived

9

Programming templates

In this chapter, we discuss basic application design techniques that you can use to help you achieve optimum performance and availability.

Generally, in order to achieve our goals, we look at two aspects:

- The first one is related to performance. We look at the database, from the logical design model to its physical design implementation and we gather information on processes: online transactions and batch processing.
- The second one is related to availability. We evaluate online and batch applications executing DML against the database, keeping in mind concurrency and integrity.

Today's requirements dictate that online transactions are able to run even during heavy workloads. We discuss the issues you should be aware of and alternative solutions for integrity, availability, and performance.

This chapter contains the following topics:

- Cursor versus singleton select
- UPDATE and DELETE using cursors
- Browsing techniques
- Advanced inquiry techniques
- Scrollable cursors
- Using multi-row FETCH
- Data integrity
- Intelligent exception handling
- Data access topics

© Copyright IBM Corp. 2005. All rights reserved.

9.1 Cursor versus singleton select

A common scenario encountered in application design is one in which a selection predicate may qualify multiple rows, but in most cases will only qualify one. The simplest way to design this access is to use a cursor using OPEN/FETCH/CLOSE processing, but this is also the most expensive. For best performance, consider the following technique when designing your application.

Use singleton SELECT first, then a cursor if the SQLCODE is -811

Implementing this mechanism depends a lot on the probability of multiple rows meeting the selection criteria. Consider using this technique, in Figure 9-1, if only one row qualifies more than 80% of the time.

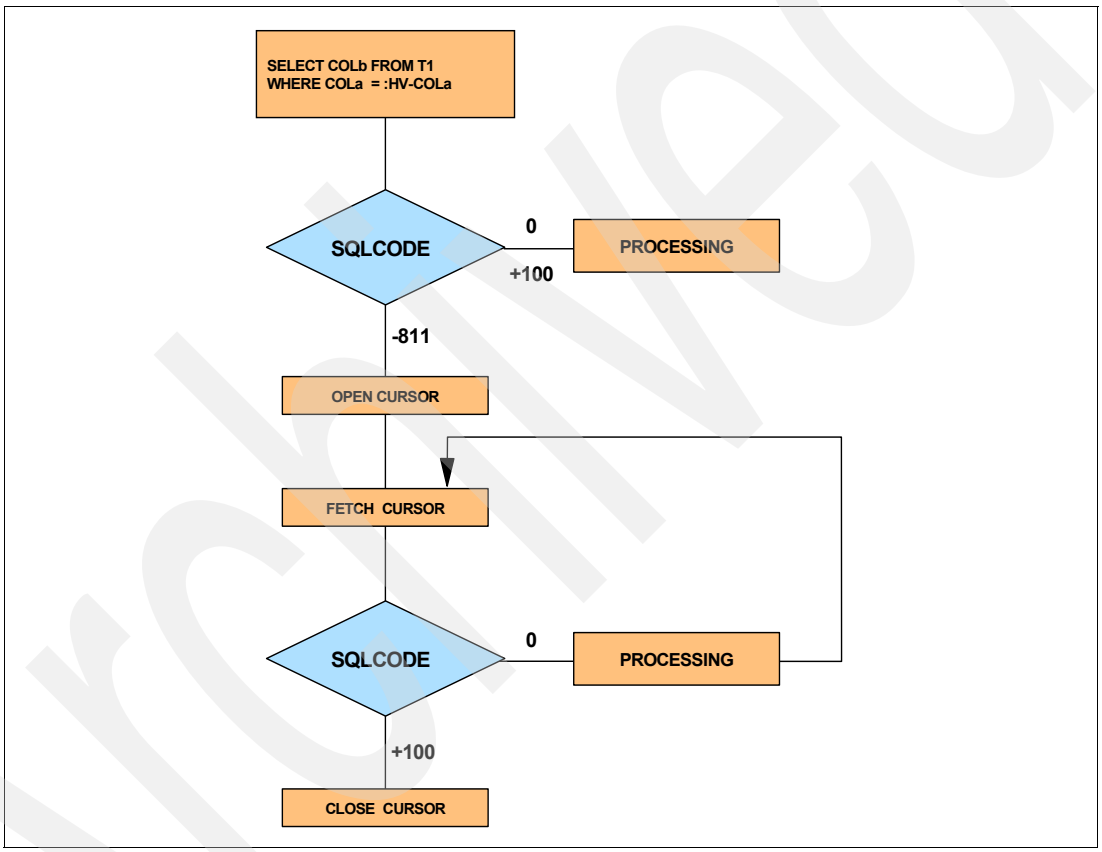


Figure 9-1 Program structure for most likely retrieving one row

Perform the singleton SELECT statement:

- If SQLCODE = 0 or +100, you have the result.
- If SQLCODE = -811, perform cursor processing using OPEN/FETCH/CLOSE.

Also, consider using this only if it is acceptable to not get any rows back. The architected behavior for negative SQLCODE is no data returned.

The benefit of this technique is the reduction of the CPU required to perform the OPEN and CLOSE of the cursor when only a single row exists. Obviously, if multiple rows are expected most of the time and you must return the rows, you should define a cursor and you should use OPEN/FETCH/CLOSE processing.

Archived

9.2 UPDATE and DELETE using cursors

One of the most powerful features of SQL is the ability to perform update and delete operations on an entire set of data with the execution of a single statement. However, the application has no control over the size of the unit of recovery when the set of qualifying data is large. In order to take control of your commit frequency during delete and update operations, you should use a cursor declared with FOR UPDATE OF and then perform UPDATE and DELETE operations using the WHERE CURRENT OF clause. This allows your application to manage the size of the unit of recovery by issuing COMMITs periodically.

When moving to a page where a row resides that is going to be updated or deleted, DB2 acquires an update lock (U lock) on the page. As soon as a row on that page is updated or deleted, the U lock is promoted to an exclusive lock (X lock). Besides allowing your application to control the commit scope, this technique also reduces the opportunity for deadlock, as discussed in “Code FOR UPDATE OF for updatable cursors” on page 342.

Depending on the design of the application, restart coding may not be necessary for applications that are only deleting rows from a table. This is the case when all rows that qualify for the cursor are deleted, since the rows already deleted will not be returned to the cursor on restart.

When deleting, be aware of any cascading deletes that may occur because of referential integrity. If a large number of dependent rows may be affected, consider deleting rows from the dependent tables first. This technique avoids lock escalations on tables not directly accessed. In cases where only a small number of rows are affected in dependent tables, adjust your commit frequency to account for the number of affected rows in dependent tables.

Important: Your unit of recovery can contain more rows than are deleted by the cursor because of RI cascading deletes.

9.3 Browsing techniques

A common online application requirement is to browse a large number of rows and to present them to the user at a terminal. The front end may not be able to hold the entire result set at once, so the application must be able to handle this mismatch in size.

We describe a straightforward browsing technique that applies predicates to the columns that are used in the repositioning process. The technique uses a limited or well-defined set of predicates to set the boundaries of the result set. More challenging inquiry problems are addressed in “Advanced inquiry techniques” on page 227.

There are two design options to solve this basic browsing problem:

- The browsing application retrieves the entire result set and stores the rows for subsequent piecemeal retrieval by the application.
- The browsing application retrieves the result set in smaller sets, typically only the data that fits on one screen.

The retrieval of all qualifying rows is typically considered on fat clients, which can use scrolling techniques on the client to move through the result set. Retrieval of all rows in a single request can reduce network traffic and CPU consumption for data transfer.

Retrieving parts of a result set is typically the only option when working with thin clients or dumb terminals that can only present the data. Network traffic overhead and CPU consumption increase as more trips to the data server are required.

Both options have their advantages and disadvantages. You can use different techniques to implement these design options. There are two key considerations when comparing the options: the integrity of the displayed data and performance.

Regardless of which option you choose, DB2 for z/OS Version 8 provides you the ability to reduce the number of fetches by using multi-row FETCH, which performs better than repeated single row fetches. For more information about multi-row fetch, see “Using multi-row FETCH” on page 239.

9.3.1 Integrity of displayed data

The first consideration when choosing a browsing technique is the data integrity requirement of the application. Unless you use ISOLATION(RR) in a conversational mode, the result set you fetched will be exposed to other applications and can be updated.

For the design option where all data is retrieved in the first invocation of the transaction and subsequently stored in temporary storage, all of the rows displayed reflect approximately the same point in time in the database. However, the result set can become very stale, meaning that rows in the result set may have been updated or deleted, and new qualifying rows might have been inserted.

For the design option where only the data that fits on one screen is retrieved and not the entire result set, DB2 repositioning covers multiple points in time of the database. If the transaction disconnects at each screen scroll, each scroll operation reflects its own point in time and will return data that is current at that point. Between scrolling operations, rows may have been added to or removed from the result set and rows displayed on the screen might be updated or deleted, but a smaller set of rows are exposed for a shorter time.

The option you use will depend on your requirements. If the data is going to be used to perform subsequent updates, you might choose to browse using a smaller result set to minimize the exposure of your displayed rows to updates by other application processes.

9.3.2 Performance

The performance of the two browsing techniques depends greatly on the size of the result set and the efficiency of the access path used to retrieve the result set. First, we examine the performance of the technique that retrieves and stores the entire result set on the first invocation of the search, and then we will present two possible cases of repositioning in DB2 to read only the next portion of the result set.

Retrieve and store the result set in temporary storage

Figure 9-2 on page 219 shows the processing used to display multiple screens of data when the result set has been retrieved at the first invocation of the search and then stored in temporary storage for use in subsequent browsing. The following steps are performed:

1. The first set of qualifying rows are moved to the screen for display.
2. All qualifying rows are read from the DB2 table and written to temporary storage.
3. A scroll forward action is requested.
4. The next set of qualifying rows is read from temporary storage and moved to the screen for display.
5. A scroll forward action is requested.
6. The next set of qualifying rows is read from temporary storage and moved to the screen for display.

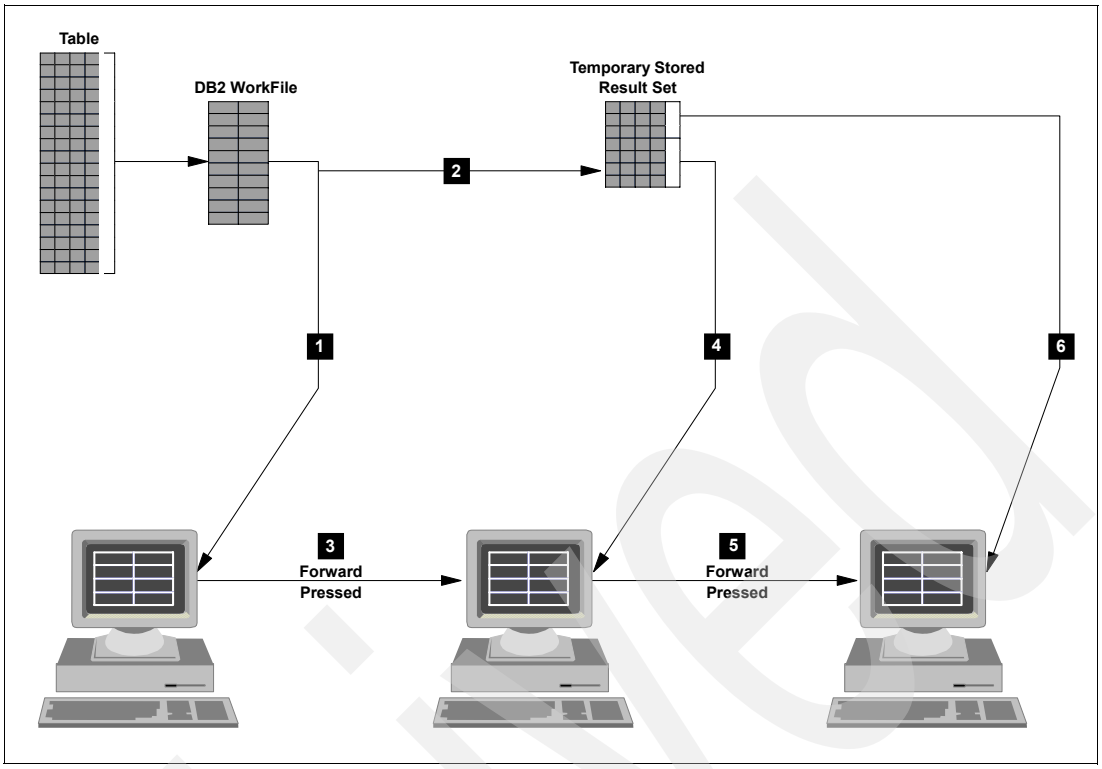


Figure 9-2 Retrieve and store the result set in temporary storage

Figure 9-2 also shows the use of a DB2 work file. Whether or not a work file is required will depend upon whether the cursor materializes. However, the advantage of this browsing technique is that cursor materialization will only be performed once at the first invocation of the search. Whether the cursor materializes or not, the entire result set must be retrieved before the first screen is displayed, so the performance difference should not be significant.

The performance of this technique will depend upon the size of the result set.

DB2 repositioning with efficient access path

An efficient access path is one in which the cursor does not have to materialize, and the cursor is able to position on the first matching row with a single probe. When the cursor does not have to materialize the SQL statement can retrieve just the rows needed very efficiently. Figure 9-3 on page 220 shows the processing used to display multiple screens of data using this approach. The following steps are performed:

1. The first set of qualifying rows are read from the table and moved to the screen for display.
2. A scroll forward action is requested.
3. The next set of qualifying rows is read from the table and moved to the screen for display.
4. A scroll forward action is requested.
5. The next set of qualifying rows is read from the table and moved to the screen for display.

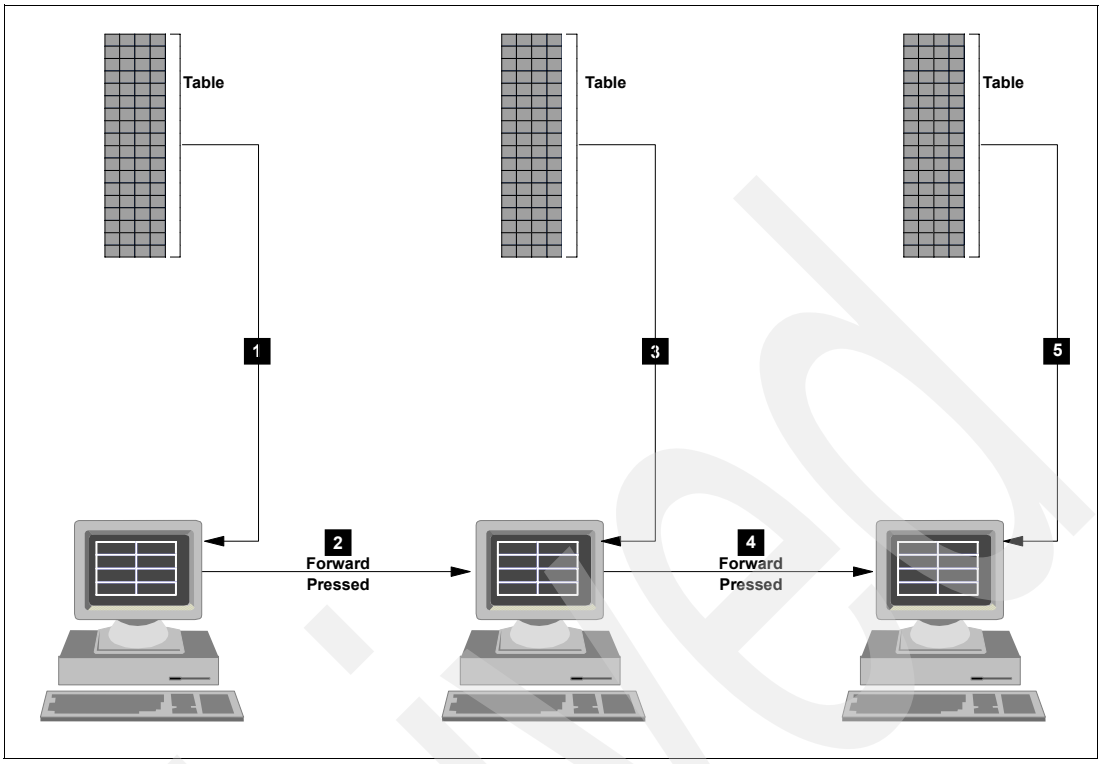


Figure 9-3 DB2 repositioning with efficient access path

In the initial invocation, the transaction retrieves the exact number of rows that fit on one screen. The key values of the last row displayed on the screen are stored in persistent temporary storage for use in the forward process. Once the user presses the Forward key, the information of the last row displayed is used to open the cursor resulting in the subsequent rows. Again the exact number of rows contained on the screen is retrieved. The process is repeated for all subsequent screens.

With this method, the performance does not depend upon the size of the total result set, only on the number of rows that are retrieved to fill the screen.

DB2 repositioning without efficient access path

For the purposes of our discussion, an inefficient access path is one in which the cursor must materialize, or one in which the matching columns of the access path are fewer than the number of columns being used for repositioning. Figure 9-4 on page 221 shows the processing used to display multiple screens of data using DB2 repositioning and a materialized cursor. The following steps are performed:

1. The entire result set is read into a DB2 work file and sorted, and then the first set of qualifying rows are read from the DB2 work file and moved to the screen for display.
2. A scroll forward action is requested.
3. The remainder of the result set is read into a DB2 work file and sorted, and then the next set of qualifying rows are read from the DB2 work file and moved to the screen for display.
4. A scroll forward action is requested.
5. The remainder of the result set is read into a DB2 work file and sorted, and then the next set of qualifying rows are read from the DB2 work file and moved to the screen for display.

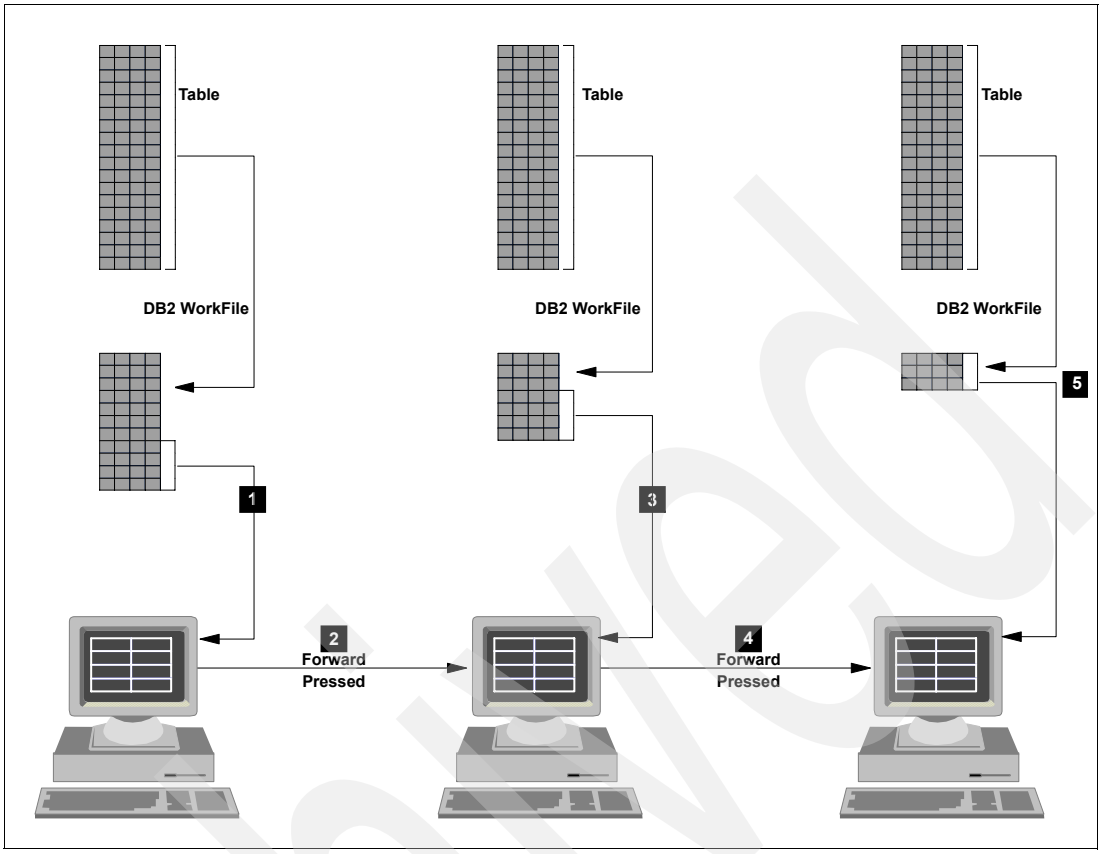


Figure 9-4 DB2 repositioning without efficient access path

In the case of cursor materialization, each OPEN CURSOR will read and sort all remaining rows in the result set before the first one is displayed. In theory, each subsequent screen should require fewer resources as the result set becomes smaller.

When the number of matching columns is fewer than the number of columns used for cursor positioning, the opposite is typically true, each subsequent screen may require more resources as more rows that have already been processed are matched in the index and then subsequently filtered out. The performance should never get as bad as the materialized cursor, however, since no sort is performed.

With this method, the performance depends on the size of the result set, and the amount of rows from the result set that has already been displayed.

Summary

It is obvious that the design option using DB2 repositioning with an efficient access path is by far the most efficient solution. By only retrieving the rows as they are needed on the screen, the costs are incremental. A user who finds what they want on the first screen does not incur the cost of reading all of the rows. This option requires the existence of an index that supports both the search predicates on the repositioning columns and the ORDER BY clause of the cursor.

The design option using temporary storage has the highest cost on the first invocation, since it must retrieve and store all of the rows, but subsequent scrolling will not incur this cost since the rows are retrieved from temporary storage and not from DB2. If you anticipate that the entire result set will eventually be displayed in most cases, this may be an appropriate design.

Archived

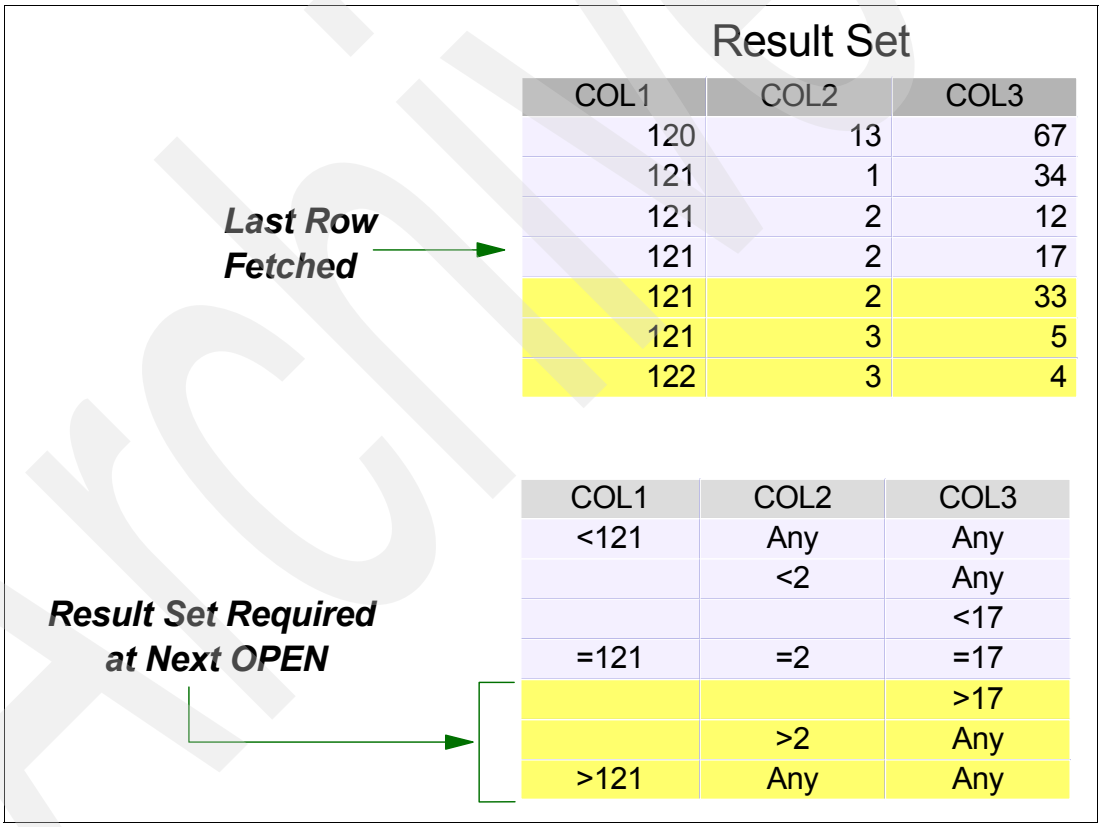
However, if the size of the result set exceeds the capacity of your temporary storage, you cannot consider this design.

DB2 repositioning with an inefficient access path is the least desirable design alternative, especially if it involves cursor materialization. The magnitude of the problem is due to the repetitive high cost of the initial screen display and the forward browsing actions.

9.3.3 A browsing technique solution using DB2 repositioning

In this section, we consider only the cases where the predicates and repositioning columns provide the best filtering match. See “Predicates and repositioning columns do not match” on page 230 for a discussion of the cases where the two do not match.

We assume that the program is coded pseudo-conversational, so the cursor is closed the second time you go into the transaction. Therefore, you have to reopen the cursor in such a way that the rows that have been previously displayed are no longer included in the result set. You have to change the values of the host variables in the cursor to retrieve the appropriate rows of the result set. Figure 9-5 shows the result set and the logical conditions that have to be added to the original predicates.



COL1	COL2	COL3
120	13	67
121	1	34
121	2	12
121	2	17
121	2	33
121	3	5
122	3	4

COL1	COL2	COL3
<121	Any	Any
	<2	Any
		<17
=121	=2	=17
		>17
	>2	Any
>121	Any	Any

Figure 9-5 Cursor repositioning

You should perform the sequence of these four actions to allow for repositioning:

1. Determine on which unique columns you reposition.

   The columns on which you reposition must be unique. You can also reposition on nonunique columns, provided you stop filling a screen only if you reach a break point in a value of the column. This can be quite difficult to achieve in a browsing application, because you usually do not control the size of the screen.

2. Add an ORDER BY on those columns.

   You must add an ORDER BY to the CURSOR because you need to process the rows in the proper sequence. This does not necessarily imply a sort, if you have the proper index defined. Failing to add the ORDER BY may result in unpredictable output since the last row of a screen can contain any value.

3. Store the repositioning key value of the last screen in persistent storage.

   Since you are coding the application pseudo-conversationally, you need to store the repositioning values in a persistent way to be able to use them when the user requests a forward browse.

4. Add predicates to skip the part that is already processed.

   You must add additional predicates in order to retrieve the result set. However, the result set should exclude the rows that have already been displayed, and we discuss several alternative techniques for you to use below.

Cursor repositioning using OR

The first option, which maps most closely to the logic layers of Figure 9-5 on page 222, consists of stacking these layers combined with OR. In the cursor repositioning examples, we use a table, TREPOS, that has a unique clustering index on COL1, COL2, and COL3. Example 9-1 shows the cursor C1, which we code using OR predicates.

Example 9-1 Cursor declaration for restart: The common way

```
DECLARE C1 CURSOR FOR
SELECT COLa, COLb, COLc, COLd, COLe
FROM T1
WHERE COLa > :HV-COLa
 OR  (COLa = :HV-COLa
  AND COLb > :HV-COLb)
 OR  (COLa = :HV-COLa
  AND COLb = :HV-COLb
  AND COLc > :HV-COLc)
ORDER BY COLa, COLb, COLc

                                        SRTN SRTC
MTHD TNAME  ACTYP MCOLS ACNAM  IXONLY  UJOG UJOG
---- ------ ----- ----- ------ ------- ---- ----
0    T1     I     0     I1     N       NNNN NNNN
```

Using the cursor declaration you see in Example 9-1, DB2 is using a nonmatching index scan, which cannot be considered an efficient access path since the CPU consumption can be quite high if you retrieve rows that are located at the end of the index.

We will not implement this solution because it does not provide us with the most efficient access path.

The reason the access path is a nonmatching index scan is the use of the OR to combine the predicates. It is considered a non-Boolean predicate, which makes it unusable for a matching index scan in a single index processing. DB2 may still use the three clauses as Boolean predicates if a multiple index access path is selected. This may even result in a far worse situation as DB2 needs to process the RIDs of the entire result set while processing the indexes, after which a sort needs to be performed to get the resulting rows in the sequence of the ORDER BY. Depending on the size of the result set, DB2 may even switch to a table space scan.

Cursor repositioning using AND NOT

The AND NOT solution partially solves the problem of the non-Boolean predicates used in the OR solution. Example 9-2 shows the C2 cursor, which is coded using AND NOT.

Example 9-2 Cursor declaration for restart: Using AND NOT predicates

```
DECLARE C2 CURSOR FOR
SELECT COLa, COLb, COLc, COLd, COLe
FROM T1
WHERE     COLa >= :HV-COLa
 AND NOT (COLa  = :HV-COLa
     AND  COLb  < :HV-COLb)
 AND NOT (COLa  = :HV-COLa
     AND  COLb  = :HV-COLb
     AND  COLc <= :HV-COLc)
ORDER BY COLa, COLb, COLc

                                        SRTN SRTC
MTHD TNAME  ACTYP MCOLS ACNAM  IXONLY  UJOG UJOG
---- ------ ----- ----- ------ ------- ---- ----
0    T1     I     1     I1     N       NNNN NNNN
```

In this case, DB2 uses a matching index scan on one column. DB2 cannot use more than one matching column because NOT is used on the other predicates.

The efficiency and resource consumption of this solution depend on the cardinality of the first column. The overhead is proportional to the cost of scanning the index entries of the rows that were already retrieved by the application. Any row with a column value of COL1 equal to the repositioning value and a value of COL2 that is smaller than the repositioning value has to be reprocessed. The same is true for equality of COL2 and a value smaller or equal to the repositioning value of COL3.

Assume a repositioning scenario where the last row of the previous screen has the three column values of 121, 1, and 2,817.

Since the access path has only MATCHCOLS 1, DB2 has to scan all index entries with a value of COL1 equal to 121 up to the value 1 for COL2 and 2,817 for COL3. All these index entries have to be processed, but do not qualify the predicates, and are, therefore, not included in the result set.

This access path may be acceptable if the number of key values DB2 has to process before it reaches the first qualifying row is small. As the number of rows that match on COL1 becomes larger, this number will grow. The performance will also vary depending upon where you are positioned in the result set.

Cursor repositioning using OR and additional predicates

Although the AND NOT solution partially solves the problem, you may decide not to use it since the use of the negative condition is counterintuitive.

A more intuitive approach, resulting in the same access path and performance, is based on the OR solution. You simply add an additional predicate combined with an AND that is matching on the first column. Example 9-3 shows the C3 cursor, which is coded using OR and an additional matching predicate.

Example 9-3 Cursor declaration for restart: Using OR and additional predicates

```
DECLARE C3 CURSOR FOR
SELECT COLa, COLb, COLc, COLd, COLe
```

```
FROM T1
WHERE COLa >= :HV-COLa
 AND (COLa  > :HV-COLa
  OR (COLa  = :HV-COLa
 AND  COLb  > :HV-COLb)
  OR (COLa  = :HV-COLa
 AND  COLb  = :HV-COLb
 AND  COLc  > :HV-COLc))
ORDER BY COLa, COLb, COLc

                                        SRTN SRTC
MTHD TNAME  ACTYP MCOLS ACNAM  IXONLY  UJOG UJOG
---- ------ ----- ----- ------ ------- ---- ----
0    T1     I     1     I1     N       NNNN NNNN
```

In this case, DB2 uses a matching index scan on one column. DB2 cannot use more than one matching column, because of the OR used to combine the other predicates.

As in the previous example, the efficiency and resource consumption of this solution depend on the cardinality of the first column, and may vary depending upon where you are in the result set.

Cursor repositioning using multiple cursors

One method to improve the performance of cursor repositioning is to use multiple cursors, one for each column that is used for positioning. Table 9-1 shows the relevant values for a cursor to reposition when using three columns.

Table 9-1 Relevant values for multiple cursor repositioning

COLa	COLb	COLc	MATCHCOLS
= :HV-COLa	= :HV-COLb	> :HV-COLc	3
= :HV-COLa	> :HV-COLb	Any	2
> :HV-COLa	Any	Any	1

Consider using one cursor for each column of the unique index used for repositioning. See Example 9-4 on page 221.

Example 9-4 Program logic for browsing using multiple cursors

```
DECLARE C1 CURSOR FOR
  SELECT COLc, COLd, COLe
    FROM T1
   WHERE COLa = :HV-COLa
     AND COLb = :HV-COLb
     AND COLc > :HV-COLc
   ORDER BY COLa, COLb, COLc
   QUERYNO 1

DECLARE C2 CURSOR FOR
  SELECT COLb, COLc, COLd, COLe
    FROM T1
   WHERE COLa = :HV-COLa
     AND COLb > :HV-COLb
   ORDER BY COLa, COLb, COLc
   QUERYNO 2
```

```
DECLARE C3 CURSOR FOR
  SELECT COLa, COLb, COLc, COLd, COLe
    FROM T1
   WHERE COLa >:HV-COLa
   ORDER BY COLa, COLb, COLc
   QUERYNO 3

                                                SRTN SRTC
STMTNO MTHD TNAME  ACTYP MCOLS ACNAM  IXONLY  UJOG UJOG
------ ---- ------ ----- ----- ------ ------- ---- ----
3      0    T1     I     1     I1     N       NNNN NNNN
2      0    T1     I     2     I1     N       NNNN NNNN
1      0    T1     I     3     I1     N       NNNN NNNN
```

EXPLAIN indicates a matching index scan for all three cursors with, respectively, a value for MATCHCOLS of 3, 2, and 1. This may seem to be a problem because you have accesses with fewer MATCHCOLS than columns. However, cursors C2 and C3 are only opened when the first qualifying row is the first row encountered in the index. Therefore, there will be no rows qualified on the first one or two columns that do not also match on the second and third.

9.3.4 Backward browsing

The solutions we have presented so far cover the problem of forward browsing. We have not considered the case where the user wants to move backward in the result set. Backward browsing presents a different problem, because you want to retrieve a part of the result set that has already been displayed.

A possible solution is the use of a separate cursor with inverted predicates and an ORDER BY DESC on the repositioning columns. The necessity of defining an additional index for backward scrolling was eliminated in DB2 for z/OS Version 8 when DB2 was enabled to read index structures backwards.

For backward browsing, you need to store the key columns of the first row and the last row displayed on the screen to ensure forward and backward browsing.

In case the thread stays alive while users can scroll forward and backward, you can consider using scrollable cursors as described in “Scrollable cursors” on page 232.

9.3.5 Considerations for browsing on clients

If you use a workstation, including a client application other than a terminal emulation for maintaining your data, you can also consider clients handling browsing issues.

For example, if your front end interface is a fat client, consider passing all qualifying rows in unsorted sequence to the client to provide user-defined sort sequences, handled by a client application.

In case your front end interface is a thin client, consider sorting data on the mainframe, passing only the first qualifying rows in required sequence.

9.4 Advanced inquiry techniques

The solutions covered in “Browsing techniques” on page 217 solve many of the browsing problems, however, there are additional challenging cases that you may need to address:

- What if the user can choose from as many as 20 different criteria that can be chosen in any sequence or combination?
- How do you handle the case where the filtering predicate is not related to the columns used for repositioning?
- How do you handle the requirement for five different indexes to deliver the inquiry performance without affecting the response times of your inserting transactions?

9.4.1 Many criteria available, but only a few selected

We assume that you need to code a transaction where the user is allowed to provide values for a specific number of predicates out of a list of potential selection criteria. Figure 9-6 shows the data entry screen that allows the user to fill in up to nine criteria.

Figure 9-6 Screen with many selection criteria

How do you efficiently handle this large number of possible predicate combinations?

You have three different ways of solving this problem. You can use:

- Static SQL
- Static SQL with REOPT(VARS)
- Dynamic SQL

We examine the three possibilities.

Static SQL

The problem with the static SQL option is the inability to dynamically adapt to the run-time host variable values. At coding time, you must incorporate all possible combinations or provide a mechanism to support those combinations.

You have three options:

- Code one cursor
- Code as many cursors as there are predicate combinations
- Use a solution that finds the balance between the two extremes

Again, check the three solutions.

Code one cursor

In this case, you code one SQL statement that is used for all possible predicate combinations. You need to provide a mechanism to disable predicates in the cursor, since there is no relationship with the supplied values and the predicates present in the SQL statement. A number of techniques enable you to do that:

- Use BETWEEN

  Provide with high values and low values in the host variables if the predicate is not filled in.

- Add OR 0 = :HV

  If the predicate is not filled in by the user, the host variable HV gets a value of 0. This will effectively disable the predicate, since 0 is equal to 0. Given the OR, there is no need to verify the predicate.

- Enabling subqueries with AND 0 = :HV

  If AND 0 = :HV is coded inside a subquery, providing a value of 0 for host variable HV lets DB2 execute the subquery.

- Use LIKE

  This works with alphanumeric data. If the predicate is not supplied by the user, the host variable gets a value of '%', which includes all potential values, and, therefore, disables the predicate.

There are a number of considerations when using this approach. Access path selection becomes an issue as a single cursor can only have one access path. Furthermore, the disabling technique may have an impact on the access path selection. If you use the OR solution, the predicate is applied at stage 2, even if it could have been used as an index matching stage 1 predicate. The LIKE technique also causes problems. Assume that DB2 selects an access path that uses a matching index scan on the column on which you specify LIKE. Everything works fine if the user provides a value for that column. However, what happens if the user does not supply a value for that column? You end up with an extremely inefficient access path.

An additional consideration is the cost of applying redundant predicates. If the user supplies only three out of nine criteria, your WHERE clause contains six redundant predicates. These cause CPU consumption because they are combined through AND, and therefore have to be evaluated under all circumstances.

The conclusion is that this is not a good solution, unless there is an indexable predicate that can always be applied to provide an acceptable static access path.

Code as many cursors as there are predicate combinations

Coding as may cursors as there are predicate combinations is the opposite of using just one cursor. The problem you face is the usability of this solution, since you have to code for all possible combinations. If a new column is added to your front end, the number of possible combinations goes up by the power of two. For only four columns, there are 15 combinations (n^2 combinations - 1, when no data is filled in). The number of possible combinations quickly becomes unmanageable. There are also good DB2 reasons not to use this solution, because

the size of the DBRM can have a tremendous impact on the EDM pool and allocation and deallocation cost.

This solution is not really practical, unless the number of possible combinations can be reduced to a manageable number.

Use a balance between the two previous extremes

Assuming you code all the different cursors of the previous solution, what would be their access path? If we take a closer look at the Explain output for these statements, we notice that a lot of these statements use the same access path. This makes sense because the number of indexes that are defined on the table is also limited. Defining more cursors than access paths is a waste of resources.

When you analyze the total number of predicate combinations, you notice that not all combinations make sense. There is no point in coding a cursor using predicates that have a 50% filter factor. In general, it does not make sense to define a cursor if the number of entries that qualify the combination of predicates exceeds 200. Applying this rule greatly reduces the number of combinations. Therefore, do not allow all possible combinations as search criteria, but provide those making sense to the user (for instance, a search on *Sm* on the NAME column (over 1,000,000 customers) does not make sense since nobody scrolls down more than 200 hits).

A second element in the analysis is the need for an efficient access path. Since this is a browsing problem, you need an index to avoid having to sort the result set for repositioning. The number of indexes that you define on a table is limited by the boundaries set by your user requirements and updating transactions. The number of cursors that you define in the program should match the indexes defined on the table.

Static SQL with REOPT(VARS)

The benefit of REOPT(VARS) is limited when you want to efficiently handle a large number of possible predicate combinations. There is, of course, the possibility that reoptimization gets you a better access path, but you need to include all the predicates even though they may not be provided by the user.

The cost difference between static SQL with REOPT(VARS) and dynamic SQL is only the authorization checking. It is almost certain that the cost of the redundant predicates using static SQL with REOPT(VARS) is higher than the authorization checking. If you still want to use static SQL with REOPT(VARS), although there is no real benefit, you should never use the OR solution to disable a predicate, because DB2 cannot use this as a matching index predicate even if the value for the host variable is available. The preferred solution is the LIKE option because it provides full flexibility. Numeric fields can use the BETWEEN option. You still have the overhead of the predicate evaluation for predicates that are not specified by the user.

REOPT(VARS) is definitely good practice for long-running SQL statements.

Dynamic SQL

The advantage of dynamic SQL is the ability to optimize the WHERE clause, which should not contain predicates other than those supplied by the user. This is a performance benefit because there are no redundant predicates. However, compared to the static SQL option, you have a cost increase due to PREPARE. The cost of the PREPARE can vary widely depending on SQL and database complexity. The fastest PREPARE is estimated to break even with the elimination of one redundant predicate applied to 10,000 rows (or 1,000 rows using 10 predicates). The breakeven point really depends on the number of rows on which the predicates are applied, and, of course, the cost of PREPARE and the number and type of

predicates. This assumes that both statements use the same access path. If the access path of the dynamic SQL is more efficient, the difference is more important.

It is not always true that dynamic SQL consumes more CPU than static SQL. If the PREPARE overhead of dynamic SQL is spread out over many rows that are fetched, the CPU time difference between static and dynamic SQL becomes smaller.

The problem you face with this coding is repositioning. You need to add an ORDER BY to the CURSOR to allow you to reposition. What happens if you code an ORDER BY on columns on which the user did not specify any condition or that have a very inefficient filtering? If you know which predicates the user is providing, you should be able to ORDER BY those columns when you dynamically define the cursor. You have to add additional logic in your application to consider these issues.

For more information about dynamic SQL, see *DB2 for z/OS and OS/390 Squeezing the Most Out of Dynamic SQL*, SG24-6418.

9.4.2 Predicates and repositioning columns do not match

It is possible that the predicates with the highest filtering are not the same as the columns you want to use to reposition the cursor. If this is the case, you may want to reconsider the solution you selected to implement your browsing application. The filter factor of the columns used for repositioning logic has a strong impact on the overall efficiency of your solution.

The complexity of the problem depends on the type of predicates with a high filter factor. If that predicate is an = predicate, the problem is similar to the standard browsing problem in “Browsing techniques” on page 217. You just need to define an index on the columns that have the = predicate, followed by the columns used in repositioning. This way the single index can address both problems. The predicates with the high filter factor will be applied to the index, and DB2 does not have to perform a sort because the same index can provide the required sequence. Be aware that there are cases where DB2 may still have to perform a sort, although the repositioning columns are in the index. If DB2 decides to use list prefetch or multiple-index access, it has to perform an additional sort, therefore losing the advantage of adding the columns to the index.

Another solution is to adapt the sort sequence so that it corresponds to the index that provides the best filtering. Changing the sort sequence so that it maps to the best filtering has an impact on the functional behavior of the application and may not be acceptable to your user. The problem that arises, if you decide to change the sort sequence, is the synchronization between the SQL coding and the selected access path. This synchronization can become quite complex if you use dynamic SQL or static SQL with REOPT(VARS). You should always verify the access path and make sure that DB2 chooses the index that matches your ORDER BY. If DB2 decides to use an index that does not match your ORDER BY, changing the sort sequence of your statement does not work, because you end up with qualifying predicates that do not match the ORDER BY sequence. If you adjust the ORDER BY sequence inside your application, you have to adjust the WHERE clause as well to allow for repositioning.

The filter factor of the repositioning predicates depends on the position within the result set. Figure 9-7 on page 231 illustrates the change in filter factor of the two sets of predicates:

- Actual predicates

  The actual predicates reflect the conditions entered by the user. The user requests to obtain all rows with a COL1 > value and COL2 > value. These predicates remain the same until the user decides to issue another inquiry.

- Repositioning predicates

  Repositioning predicates are used to eliminate the part of the result set that has already been displayed. The values of the host variables vary from low values on the first screen to high values when the end of the result set is reached.

Upon initial invocation of the transaction, the combined filter factors of the repositioning predicates is 1, since all rows are still included in the result set.

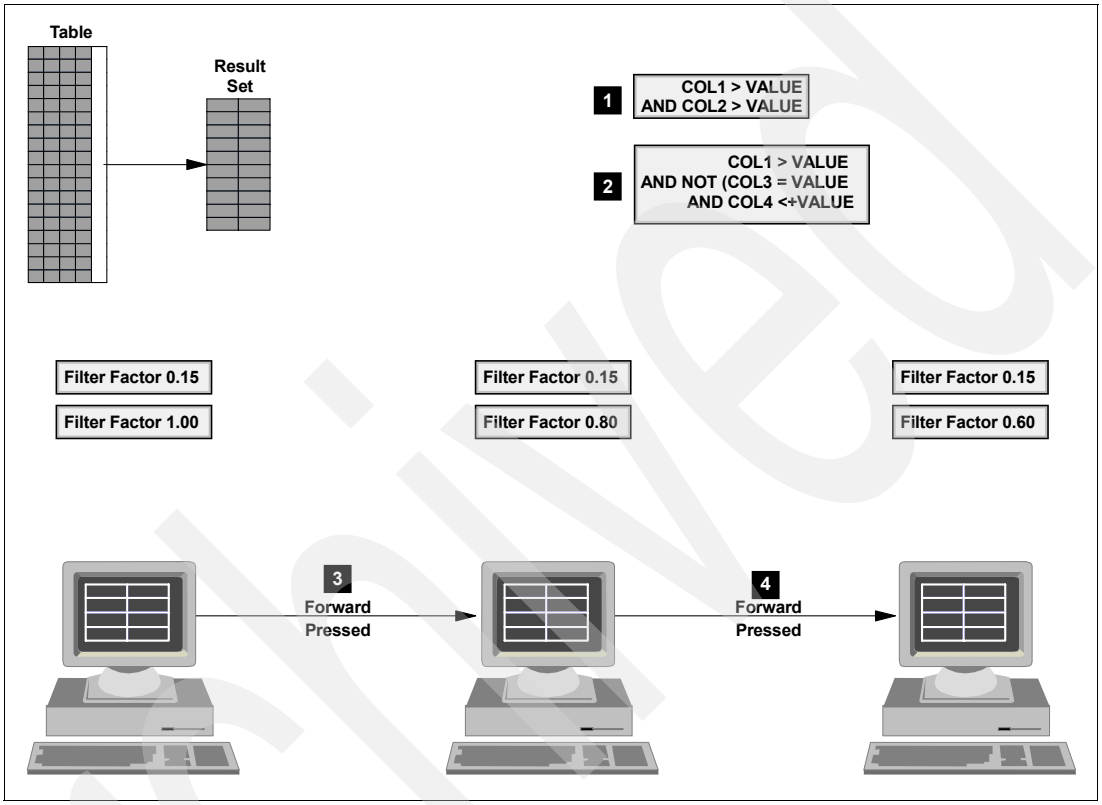


Figure 9-7 Filter factor of the repositioning predicates

If you code your application using dynamic SQL, the first time the statement is executed, you should not use repositioning predicates. There is no point in adding predicates that specify that the column value must be larger than the low values. The filter factor of the actual predicates is, in this case, 15%. If the user subsequently hits the Forward key, the filter factor of the repositioning predicates changes. The size of the result set of the predicate set roughly corresponds to five screens. The filter factor of the repositioning predicates when displaying the second screen is therefore 0.80. The filter factor of the repositioning predicates again improves when the user moves on to the third screen, to become 0.00 when the end of the result set is reached.

These filter factors seem to imply that the solution varies according to the position within the result set. The optimal solution does not really depend on the position within the result set because the different cases are not independent. You cannot get to the third screen unless you have processed the first two screens. You should therefore tackle the problem of the first screen, before considering the subsequent screens.

Since the best option is really determined by the individual filter factors, we do not advise you to use the OPTIMIZE FOR 1 ROW clause. Using this option destroys DB2's ability to assess the filter factors, and DB2 may choose an inefficient access path. The best option is probably the use of dynamic SQL or static SQL with REOPT(VARS) to make sure that the optimizer

has the most efficient information. Retrieve all rows from the cursor and store them in persistent application storage for subsequent retrieval. This gives you an uneven distribution of response time, but the overall efficiency of your application is maximized.

The only problem you are faced with in this solution is the potential for an excessive elapsed time of the OPEN CURSOR. The elapsed time of the OPEN of the cursor is related to the efficiency of the access path and the size of the result set. If the user provided you with predicate values that return a high number of qualifying rows, the process might take a long time. Consider activating the Resource Limit Facility (RLF) to provide an upper boundary for the resource consumption, but this approach is applicable only to dynamic SQL.

You can also add the repositioning predicates to the cursor. Although the predicates would not have a real value in terms of filtering rows, DB2 may decide to use the index on them to eliminate a sort. If the sort is eliminated, the OPEN is very fast, and it does not take a lot of resources to find the first set of rows given the limited filtering of the additional predicates. Your application can display an error warning once it has reached the FETCH limit you imposed in your application. Another option is to return only a maximum number of rows using FETCH FIRST *n* ROWS ONLY, since people tend not to scroll on result sets containing more than 200 hits. Therefore, your resource consumption can be minimized as well for large result sets, provided that the cursor does not have to materialize.

Another option is the use of a COUNT(*) before opening the cursor that retrieves all columns. The COUNT would read the index entries into the buffer pool if not already present and give you the exact number of qualifying rows. Use this solution only if you have a high probability of unacceptable result sets, because the general use of the COUNT leads to excessive resource consumption.

Increasing the number of indexes

In general, inquiry applications are concurrent with the applications that change the DB2 tables. The inquiry applications want to increase the number of indexes to boost the performance of the queries. The increase will cause a deterioration in the elapsed time of the applications that change the data because the cost of index maintenance goes up. An additional consideration is the locking impact of the inquiry applications.

You can solve this problem by creating a second set of tables that are used by the inquiry application. See “What if your insert rate is not compatible with your select requirements?” on page 373 for more information about techniques to maintain the second copy.

9.5 Scrollable cursors

Scrollable cursors were initially introduced in DB2 for z/OS Version 7. They provide support for:

- Fetching a cursor result set in backward and forward directions
- Using a relative number to fetch forward or backward
- Using an absolute number to fetch forward or backward

The technique used in DB2 Version 7 was based on declared temporary tables automatically created by DB2 at run time. A main concern of copying rows to declared temporary tables was that those rows were not sensitive to subsequent inserts which qualified for your result set. This changed with DB2 Version V8.

9.5.1 Static scrollable cursors

Prior to DB2 Version 8, you could declare a scrollable cursor using the following keywords:

- INSENSITIVE, causing the result set to be static. The cursor is not sensitive to updates, deletes, and inserts to the base table.
- SENSITIVE STATIC, causing the result set not to be static during scrolling of the result set. The behavior of the cursor depends on the FETCH statement. You can specify on the FETCH statement either:
  - INSENSITIVE, meaning that changes are visible to the cursor made by itself. They are visible to the application because DB2 updates both the base table and the result table when a positioned update or delete is issued by the application.
  - SENSITIVE, which is the default for cursors declared as SENSITIVE STATIC, changes are visible to the cursor made by itself. Furthermore, committed deletes and updates from other applications are also visible. However, as part of a SENSITIVE FETCH, the row is verified against the underlying table to make sure it still exists and qualifies. Inserts are not visible for the cursor.

All scrollable cursors mentioned above use a declared temporary table which is populated at OPEN CURSOR time. You can refer to those cursors as static, scrollable cursors. See Figure 9-8 for a visual explanation of result sets for static scrollable cursors.

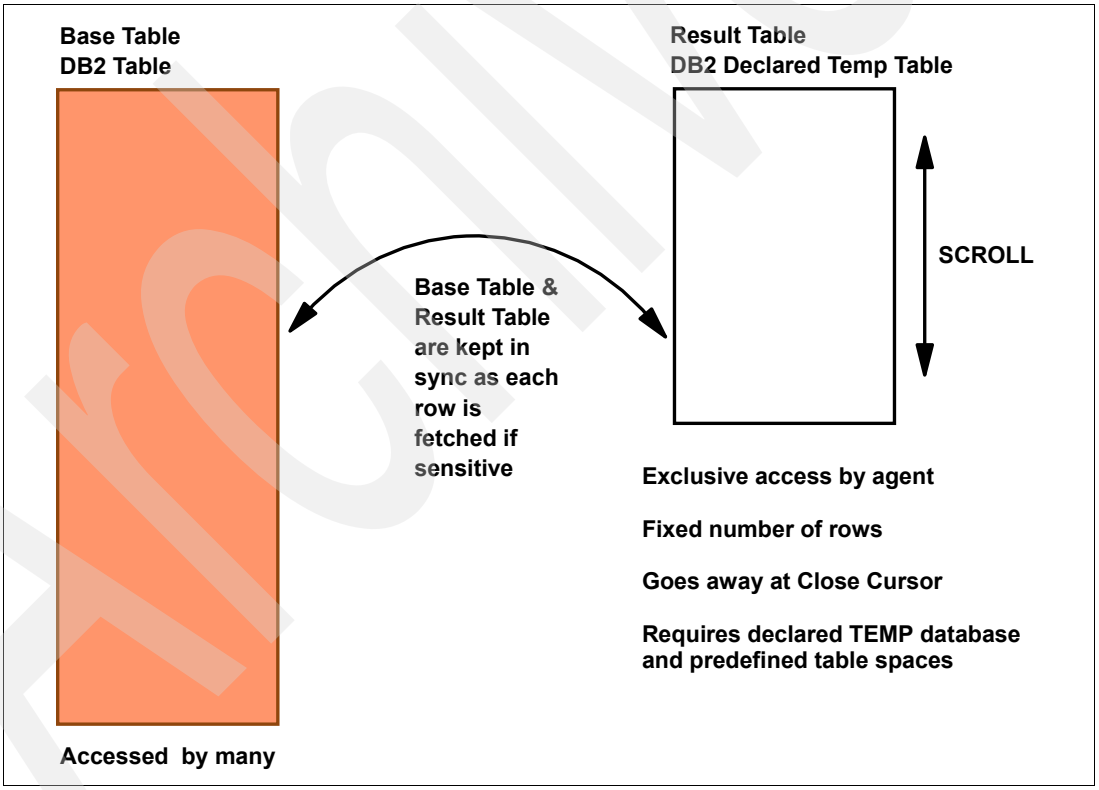


Figure 9-8 Temporary result set for static scrollable cursors

Note: Scrollable cursors support large objects (LOBs), but no temporary LOB table space is created. Data consistency is ensured by storing the LOB descriptor in the result table, therefore, the associated row in the LOB base table cannot be deleted.

9.5.2 Dynamic scrollable cursors

This functionality was enhanced in DB2 Version 8, allowing scrollable cursors to view inserts of concurrently running processes, using SENSITIVE DYNAMIC SCROLL keywords.

A dynamic scrollable cursor does not materialize the result table at any time. Instead, it scrolls directly on the base table, and is therefore sensitive to all committed INSERTs, UPDATEs, and DELETEs. Dynamic scrollable cursors can use index scan and table space scan access paths. Since DB2 supports backwards index scan, there is no need for creating indexes in an inverted order to support backwards scrolling.

All types of cursors support multi-row fetch: scrollable, non-scrollable, static, and dynamic. For more information about multi-row fetch, see “Using multi-row FETCH” on page 239.

For dynamic scrollable cursors, two new keywords were introduced in DB2 for z/OS Version 8:

- ASENSITIVE, which is the default

  DB2 determines the sensitivity of the cursor. If the cursor is not read-only, SENSITIVE DYNAMIC is used for maximum sensitivity.

  The cursor is implicitly going to be INSENSITIVE if the SELECT does not allow it to be sensitive (for example, if UNION or UNION ALL is used).

- SENSITIVE DYNAMIC

  Specifies that the size of the result table is not fixed at OPEN cursor time. A FETCH statement is always executed against the base table, since no temporary result set is created at OPEN time. Therefore, the cursor has complete visibility of changes:

  - All committed inserts, updates, and deletes by other application processes or cursors outside the scrollable cursor
  - All positioned updates and deletes within the cursor
  - All inserts, updates, and deletes by the same application process, but outside the cursor

Using the keyword ASENSITIVE allows DB2 to decide whether a cursor is either INSENSITIVE or SENSITIVE DYNAMIC, bearing in mind that the cursor should always be as sensitive as possible. A cursor meeting the following criteria is considered to be a read-only cursor having an effective sensitivity of INSENSITIVE, otherwise, the effective sensitivity is SENSITIVE DYNAMIC:

- The first FROM clause identifies or contains:
  - More than one table or view
  - A catalog table with no updatable columns
  - A read-only view
  - A nested table expression
  - A table function
  - A system-maintained materialized query table (MQT)
- The first SELECT clause specifies the keyword DISTINCT, contains an aggregate function, or uses both.
- The SELECT statement of the cursor contains an INSERT statement.
- The outer subselect contains a GROUP BY clause, a HAVING clause, or both clauses.
- It contains a subquery, such that the base object of the outer subselect, and of the outer subquery, is the same table.

- Any of the following operator or clauses are specified:
  - A UNION or UNION ALL operator
  - An ORDER BY clause (except when the cursor is declared as SENSITIVE STATIC scrollable)
  - A FOR READ ONLY clause
- It is executed with isolation level UR and a FOR UPDATE is not specified.

Example 9-5 shows the declaration of a SENSITIVE DYNAMIC scrollable cursor.

Example 9-5 Declaring a SENSITIVE DYNAMIC scrollable cursor

```
DECLARE C1 SENSITIVE DYNAMIC SCROLL CURSOR FOR
  SELECT COLa, COLb, COLc
    FROM T1
   WHERE COLa > 10;
```

For maximum concurrency, we recommend using ISOLATION(CS). Note that the isolation level is promoted to CS even if you use BIND option ISOLATION(UR) and the SELECT statement contains FOR UPDATE OF clause.

Benefits of dynamic scrollable cursors are:

- Performance improved by sort avoidance for ORDER BY

  (SENSITIVE DYNAMIC requires an ORDER BY supported by an index)
- No temp table and work file
- Enhances usability of SQL
- Enhances portability
- Conforms to SQL standards

9.5.3 FETCHing options for scrollable cursors

In general, the sensitivity to changes can be specified in two ways: on DECLARE CURSOR statements and on FETCH statements.

Depending on your cursor definition, there can be several implications for your FETCH syntax on SENSITIVE DYNAMIC scrollable cursor. Table 9-2 lists the dependencies on DECLARE and FETCH statements for scrollable cursors.

Table 9-2 Dependencies on DECLARE and FETCH statements for scrollable cursors

Specification on DECLARE	Specification on FETCH	Comment	Visibility of changes
INSENSITIVE	INSENSITIVE	Default INSENSITIVE	None
INSENSITIVE	SENSITIVE	Not allowed	
SENSITIVE	INSENSITIVE	Allowed	See own (same cursor) updates and deletes
SENSITIVE	SENSITIVE	Default SENSITIVE	See own changes and others' committed updates and deletes

Specification on DECLARE	Specification on FETCH	Comment	Visibility of changes
SENSITIVE DYNAMIC	INSENSITIVE	Not allowed, even if ASENSITIVE and DB2 decides for SENSITIVE DYNAMIC	
SENSITIVE DYNAMIC	SENSITIVE		See own and others' committed changes

You can fetch data from your current positioning using the FETCH orientation keywords you find in Table 9-3:

Table 9-3 FETCH orientation keywords for scrollable cursors

Keyword in FETCH statement	Cursor position when FETCH is executed
BEFORE	Before the first row
FIRST or ABSOLUTE +1	On the first row
LAST or ABSOLUTE -1	On the last row
AFTER	After the last row
ABSOLUTE	On an absolute row number, from before the first row forward or from after the last row backward
RELATIVE	On the row that is forward or backward a relative number of rows from the current row
CURRENT	On the current row
PRIOR or RELATIVE -1	On the previous row
NEXT	On the next row (default)

For further details about fetch orientation keywords, refer to *DB2 for z/OS Version 8 SQL Reference,* SC18-7426-02.

Table 9-4 compares different kinds of scrollable cursors looking at change-visibility and materialization of the result table.

Table 9-4 Comparing scrollable cursors

Cursor type	Result table	Visibility of own changes	Visibility of others' changes	Updatability
Non-scrollable (SQL contains a join or sort)	Fixed, work file	No	No	No
Non-scrollable	No work file, base table access	Yes	Yes	Yes
INSENSITIVE SCROLL	Fixed, declared temp table	No	No	No
SENSITIVE STATIC SCROLL	Fixed, declared temp table	Yes (INSERTs not allowed)	Yes (no INSERTs)	Yes

Cursor type	Result table	Visibility of own changes	Visibility of others' changes	Updatability
SENSITIVE DYNAMIC SCROLL	No declared temp table, base table access	Yes	Yes	Yes

9.5.4 Updating using static scrollable cursors

In general, positioned updates and deletes using scrollable cursors are possible and are based on the temporary result table for static scrollable cursors and on the underlying base table for dynamic scrollable cursor. A positioned UPDATE or DELETE is always allowed if the cursor is not read-only and the page or row lock was acquired successfully.

But how does DB2 perform updates and deletes? Let us look at DB2's processing.

For packages and plans containing updatable static scrollable cursors, ISOLATION (CS) lets DB2 use optimistic locking. DB2 can use optimistic concurrency control to shorten the amount of time that locks are held in the following situations:

- Between consecutive FETCH operations
- Between FETCH operations and subsequent positioned UPDATE or DELETE statements

Optimistic locking consists of the following steps:

- When the application opens the static scrollable cursor, DB2 fetches the qualifying rows into the DTT. When doing so, DB2 will try to use lock avoidance to minimize the amount of locking required.
- When the application requests a positioned UPDATE or DELETE operation on a row, DB2 finds and locks the corresponding base table row and reevaluates the predicate to verify that the row still satisfies the search condition.
- For columns that are in the result table, compares current values in the row to the values of the row when the result table was built. DB2 performs the positioned update or delete operation only if the values match.
- An UPDATE or DELETE is disallowed if either the row fails to qualify the WHERE clause or the values do not match.

If the row passes the conditions above, the following actions are performed for UPDATE statements:

- Update the base table
- Re-FETCH from the base table to reevaluate the predicate
- Update the result table
  - If the search condition fails, mark row as an update hole
  - If the search condition satisfies, update row with latest values

For DELETE statements, DB2 performs the following actions:

- Delete the base table row
- Update the result table
  - Mark row as a delete hole

However, optimistic locking cannot be used for dynamic scrollable cursors. For dynamic scrollable cursors, each lock is acquired on the underlying base table while fetching, similar to non-scrollable cursors.

9.5.5 Change of underlying data for scrollable cursors

In general, when your application scrolls on a result set and not on the underlying table, data in the base table can be changed by concurrent application processes. You can refer to these data changes as update or delete holes in the context of scrollable cursors.

Static scrollable cursors declared as SENSITIVE using a temporary result set have created the necessity for detecting update and delete holes, because you may want to know if somebody is burning down the building while you are in it. Changes performed by others are only visible if your FETCH statement uses the SENSITIVE keyword:

- DELETE HOLE

  A delete hole is created when the underlying base table row has been deleted. Delete holes are not prefetched. If the evaluation fails, SQLCODE +222 is returned.

- UPDATE HOLE

  An update hole is created when the corresponding base table row has been modified such that the values of the rows do not qualify the row for the query any longer. Every SENSITIVE FETCH reevaluates the row against the predicate. If the evaluation fails, the row is marked as an update hole and a SQLCODE +222 is returned.

  An update hole can turn into a row again on a subsequent FETCH SENSITIVE of an update hole.

Since you should be aware of changes applied by your application process, they are visible to your cursor even if they are not committed (as it is with any other insert, update, or delete activity you perform). The reason is DB2 updates your result set immediately after the base table is updated.

Since cursors declared with SENSITIVE DYNAMIC scroll directly on the base table, in general no update or delete holes might occur. The only special case is if an application issues a FETCH CURRENT or FETCH RELATIVE +0 statement to fetch the previously fetched row again, but the row was deleted or updated so that it no longer satisfies your WHERE clause, then DB2 returns SQLCODE +231. For example, this can occur if you use lock avoidance or uncommitted read and so no lock is taken for a row your application retrieves.

Note that the order of your result set is always maintained. If a column for an ORDER BY clause is updated, then the next FETCH statement behaves as if the updated row was deleted and reinserted into the result table at its correct location. At the time of a positioned update, the cursor is positioned before the next row of the original location and there is no current row, making the row appear to have moved.

9.5.6 SQLCODEs for scrollable cursors

DB2 issues the SQLCODEs listed in Table 9-5 when your application deals with scrollable cursors:

Table 9-5 SQLCODEs for scrollable cursors

SQLCODE	Description
+222	Update or delete hole detected.
+231	Cursor position invalid.
-222	Update or delete attempted against an update or delete hole.
-224	Result table does not agree with base table.

SQLCODE	Description
-225	FETCH statement incompatible with non-scrollable cursor.
-228	FOR UPDATE OF clause specified for read-only cursor.
-243	Sensitive Cursor cannot be defined for specific SELECT statements.
-244	Sensitivity option specified on FETCH conflicts with the sensitivity option in effect for your cursor.

9.5.7 Summary of scrollable cursors

Scrollable cursors are not going to work in IMS environments, because IMS programs generally end at the end of the transaction and close all cursors. In CICS environments, you may keep affected threads alive using conversational transactions over several scroll up and down operations, which is usually not the case. The reason is that if the thread ends, DB2 cleans up the temporary tables being used to hold the entire result set and loses cursor positions as well for dynamic scrollable cursors.

Note: Talk to your application people and DB2 systems programmer before implementing solutions based on scrollable cursors. You need to verify the requirements and possibly get a TEMP DB set up.

If you have no ability to scroll backwards, choose forward only cursors. If you have to maintain your cursor position to go back and forth, choose scrollable cursors in non-CICS and non-IMS environments. If you only need a snapshot of data in your tables, use INSENSITIVE cursors. In case you need actual data, choose the SENSITIVE DYNAMIC option for your scrollable cursor to receive the most actual data.

Whenever you need a scrollable cursor to scroll backwards, use backward index scan to avoid sort for ORDER BY when possible.

Important: The prerequisite of scrollable cursors is keeping a cursor open for maintaining the current cursor position. As soon as a thread terminates (which is the case for most IMS and CICS transactions), cursor position is lost and the application has to reposition. You can have a maximum of 2,000 local active threads and DBATs at the same time on each DB2 subsystem.

9.6 Using multi-row FETCH

DB2 Version 8 introduces multi-row fetch, allowing you to FETCH zero or more rows of the result table. By fetching more than one row, your applications reduce the number of SQL calls by fetching a rowset using one single FETCH statement as you see in Figure 9-9 on page 240. This means less crossing of the SQL Application Programming Interface (API) and lower or reduced CPU usage for this function.

Instead of fetching a single row, you can fetch a set of rows referred to as a *rowset*.

Multi-row fetch supports dynamic and static SQL and also scrollable and non-scrollable cursors. Therefore, the same keywords are available for multi-row fetch statements that you can find in “FETCHing options for scrollable cursors” on page 235. As with scrollable cursors, a multi-row FETCH can only be used inside an application program.

To benefit from performance enhancements, you have to be aware that affected applications require:

- Different handling techniques
- More application storage, depending on the rowset size

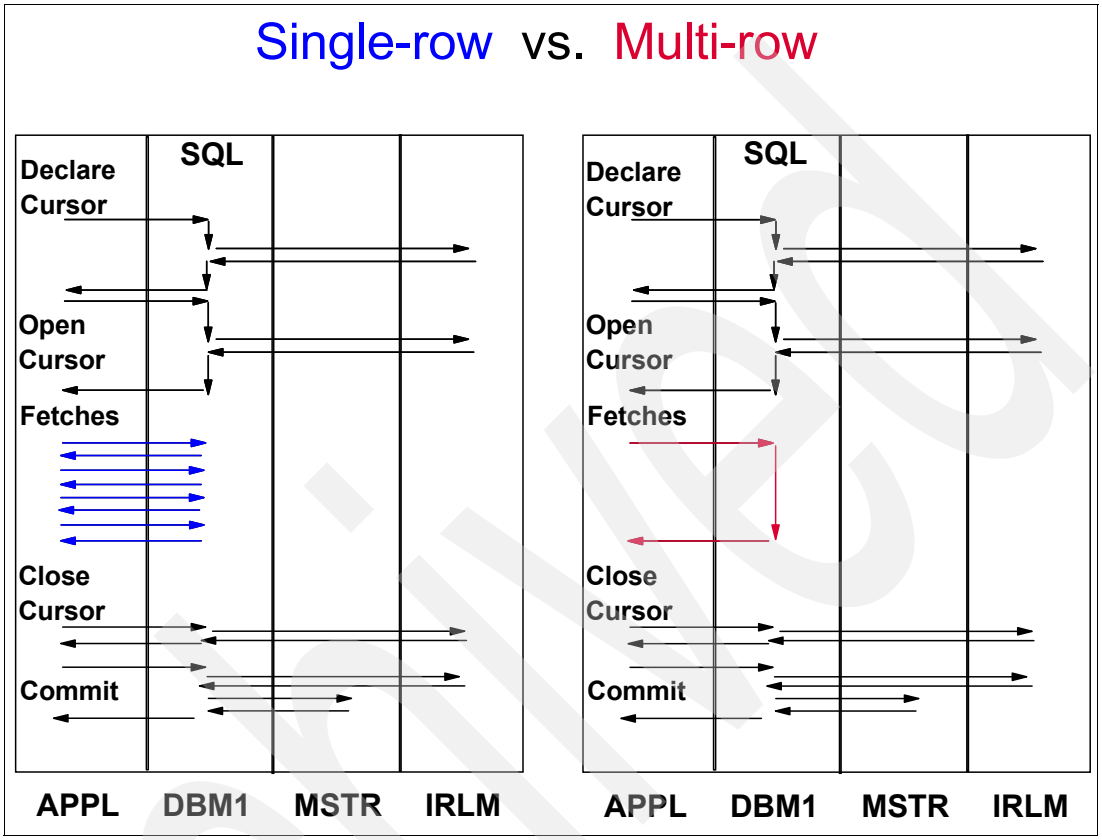


Figure 9-9 Comparing single-row fetch and multi-row fetch

9.6.1 Application preparation

The values derived from DB2 are stored in an array of attributes describing the table structure in which each element of the array corresponds to a value for a column. The array has to be defined using the application programming language. See Example 9-6 for COBOL.

Example 9-6 Host variable definitions for fetching 10 rows at a time

```
01 T1-VARS.
  05 COLa        PIC S9(9) COMP-4 OCCURS 10 TIMES.
  05 COLb        PIC X(32)        OCCURS 10 TIMES.
  05 COLc                         OCCURS 10 TIMES.
    49 COLc-LEN  PIC S9(4) COMP-4.
    49 COLc-DATA PIC X(60).
```

See Example 9-7 for a cursor declaration for usage regarding multi-row fetch.

Example 9-7 Cursor declaration for multi-row fetch

```
DECLARE C1 CURSOR WITH ROWSET POSITIONING FOR
  SELECT COLa, COLb, COLc FROM T1
```

Note that you do not specify the size of the rowset on the DECLARE CURSOR statement. That is done at FETCH time. For a DECLARE CURSOR statement, the default is WITHOUT ROWSET POSITIONING. You must specify WITH ROWSET POSITIONING for cursors fetching multiple rows, otherwise DB2 issues SQLCODE -249 at the time your application tries to fetch the result set. Using the WITH ROWSET POSITIONING clause specifies that this cursor can be used to return either a single row or multiple rows, as a rowset, with a single FETCH statement. The FOR *n* ROWS clause of the FETCH statement controls how many rows are returned on each FETCH statement. The maximum number of rows for a rowset is 32,767. Cursors declared WITH ROWSET POSITIONING may also be used with row positioned FETCH statements.

Important: Using WITHOUT ROWSET POSITIONING does not mean that there is no DRDA blocking when using distributed access. It only indicates that one row at a time is returned to the application host variables.

You can use GET DIAGNOSTICS attribute DB2_SQL_ATTR_CURSOR_ROWSET to determine after OPEN CURSOR if a cursor is enabled for rowset positioning:

- The value **N** indicates that this cursor only supports row-positioned operations.
- The value **Y** indicates that this cursor supports rowset-positioned operations.

To create the required array for multi-row fetch, you can manipulate the output of DCLGEN. The output only contains one level of the array, which has to be multiplied a certain number of times, matching the maximum number of rows you plan to fetch simultaneously. See Example 9-8 for an example of the multi-row FETCH statement.

Example 9-8 New FETCH syntax

```
FETCH NEXT ROWSET
 FROM C1
  FOR n ROWS
 INTO :COLa, :COLb, :COLc
```

The number of *n* can be either a host variable or implemented in the statement itself. After executing a multi-row FETCH, the number of rows retrieved is stored in SQLERRD(3).

If FOR *n* ROWS is not specified on a FETCH statement and a cursor is declared with FOR ROWSET POSITIONING, the size of the rowset will be the same as the previous FETCH statement for this cursor.

If a FETCH statement specifies the ROWSET keyword, and not the FOR *n* ROWS clause, the size of the rowset is implicitly set to the size of the rowset that was most recently specified in a prior FETCH statement. If a prior FETCH statement did not specify the FOR *n* ROWS clause or the ROWSET keyword, the size of the current rowset is implicitly set to 1.

The following rowset-positioned clauses are available in DB2 Version 8 for multi-row FETCH:

- NEXT ROWSET
- PRIOR ROWSET
- FIRST ROWSET
- LAST ROWSET
- CURRENT ROWSET
- ROWSET STARTING AT ABSOLUTE
- ROWSET STARTING AT RELATIVE

You can find a detailed description of rowset-positioned clauses in *DB2 for z/OS Version 8 SQL Reference,* SC18-7426-02.

9.6.2 Determining the result set

The multiple-row-fetch syntax block is similar to the existing single-row-fetch block in DB2 Version 7, except that there is an additional clause FOR *n* ROWS, where *n* can either be an integer constant or a host variable. When using a host variable, you can vary the number of rows fetched in a rowset for each FETCH, if necessary.

Let us assume that you have coded FOR 5 ROWS, but DB2 only has returned three rows. You can figure out how many rows DB2 has returned in the following ways:

- SQLERRD(3) in SQLCA

  When DB2 processes a FETCH statement, and the FETCH is successful, the contents of SQLERRD(3) in the SQLCA is set to the number of returned rows.

- ROW_COUNT in GET DIAGNOSTICS

  If the previous SQL statement was an OPEN or a FETCH that caused the size of the result table to be unknown, ROW_COUNT in GET DIAGNOSTICS returns the number of rows in the result table. If the previous SQL statement was a PREPARE statement, ROW_COUNT in GET DIAGNOSTICS returns the estimated number of rows in the result table for the prepared statement. Otherwise, if the server only returns an SQLCA, the value zero is returned.

We recommend you use fields in SQLCA to save invoking GET DIAGNOSTICS.

Using multi-row FETCH, you have to check for three conditions instead of two conditions (SQLCODE is 0 or 100) for single-row FETCH statements:

- SQLCODE 0

  If SQLCODE is set to 0, each row in the array containing the rowset is occupied. Checking SQLERRD(3) is optional, since it contains the maximum number of rows of the rowset.

- SQLCODE 100 and SQLERRD(3) > 0

  When DB2 processes a multiple-row FETCH statement, the contents of SQLCODE is set to 100 if the last row in the table has been returned with the set of rows. The number of rows returned is stored in SQLERRD(3).

- SQLCODE 100 and SQLERRD(3) = 0

  No rows are retrieved.

If you fetch beyond the end of the result set, you will receive an end of data condition:

- When there are only 5 rows left in the result table and you request FETCH NEXT ROWSET FOR 10 ROWS, 5 rows and SQLCODE 100 with SQLERRD(3) with a value of 5 will be returned.

If you fetch beyond the beginning of the result set, you will receive an end of data condition:

- When you are positioned on rows 2, 3, 4, 5, 6, and 7, and you request FETCH PRIOR ROWSET FOR 10 ROWS, 2 rows and SQLCODE 20237 with SQLERRD(3) with a value of 2 will be returned.

9.6.3 Application code changes

Since multi-row FETCH allows you to fetch more than one row, your application code has to change slightly if you use multi-row cursors:

- Using single-row FETCH, application code consists of a loop to fetch data until SQLCODE 100 or until all rows on your scrolling application are occupied.

- Using multi-row FETCH, application code
  - Can process the entire rowset if SQLCODE is 0
  - Can process the number of entries in the rowset as stored in SQLERRD(3) if SQLCODE is 100 and no additional FETCH is required
  - Does not have to process any rows if SQLCODE is 100 and SQLERRD(3) has a value of 0
- Online scrolling applications can issue one multi-row FETCH and continue processing all retrieved values until the number of rows stored in SQLERRD(3) is reached or all rows on your scrolling application are occupied.
  - Requires one FETCH statement and one loop to process the retrieved rows
- Non-online scrolling applications can process the rows after the conditions mentioned above are checked.
- Depending on your result-set you need one loop for FETCHing data and one loop for processing the retrieved rows.

In Figure 9-10 on page 244, we provide a possible structure for using multi-row FETCH statements inside of your application program.

Benefits using multi-row fetch:

- Improved performance by eliminating multiple trips between application and database.
- Possible reduced network traffic in distributed access depending on the block fetch options already in place (RQRIOBLK parameter in DB2 Connect™). Both ambiguous and updatable cursors will benefit.
- DB2 will always ship 1 ROWSET per network message (no longer 32 KB blocking).

Archived

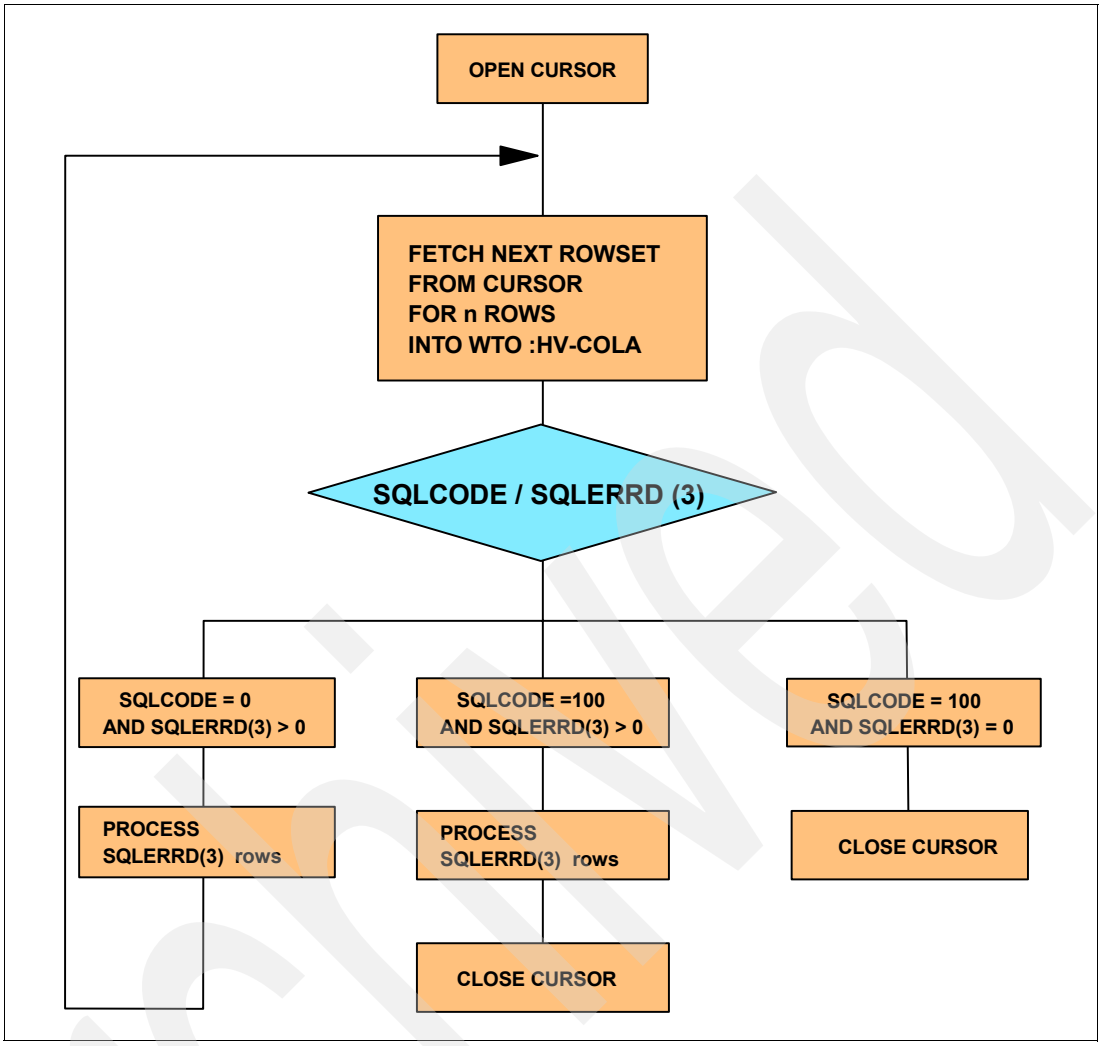


Figure 9-10 Using multi-row FETCH inside your application

9.6.4 Multi-row update and multi-row delete

Cursors defined for multi-row fetches also allow you to update or delete the result set you have previously retrieved or even parts of the result set. The cursor definition is nearly the same for a single-row cursor. See the definition of an updatable multi-row cursor in Example 9-9.

Example 9-9 Multi-row cursor definition with FOR UPDATE OF

```
EXEC SQL
  DECLARE C1 CURSOR WITH ROWSET POSITIONING FOR
    SELECT COLa, COLb, COLc, COLd FROM T1
       FOR UPDATE OF COLa
END-EXEC
```

Example 9-10 shows the syntax for updating the entire result set.

Example 9-10 Updating a result set of a multi-row cursor

```
EXEC SQL
  UPDATE T1
```

```
     SET COLa = COLa + 1
   WHERE CURRENT OF C1
END-EXEC
```

A multi-row cursor also allows you to update only a single row of the result set. The syntax is different, as you can see in Example 9-11 where only the second row is updated.

Example 9-11 Updating a single row of a result set of a multi-row cursor

```
EXEC SQL
  UPDATE T1
     SET COLa = COLa + 1
   FOR CURSOR C1 FOR ROW 2 OF ROWSET
END-EXEC
```

The syntax we show for updating a rowset also applies to DELETE statements.

9.7 Data integrity

As numbers of concurrent users and parallel transactions increase over time, it is extremely important to ensure data integrity when different applications and transactions manipulate the same data. In the following sections, we describe how to make sure your data does not enter an inconsistent state caused by concurrently running online and batch applications.

To guarantee data consistency, you should consider the following objectives in your application design:

- Protect the same data being updated by concurrently running processes
- Roll back the data to a point where the data is consistent when an abnormal termination occurs

9.7.1 Concurrent updates

When running applications concurrently, it is even more important to make sure those applications do not interfere with each other when accessing the same data. In worse cases, updates can get lost or applications can even destroy data integrity by violating integrity rules across multiple columns or rows.

In Figure 9-11 on page 246, we describe a scenario of two online update processes running concurrently. Both applications have read CUSTNO 18 at nearly the same time, so the represented data is consistent and still the same.

Assume that the UPDATE statements in the applications are coded as shown, the data in DB2 depends on the user who first saves the changes, to be more specific on the elapsed time between both SELECT and UPDATE statements. The second update will be the update you can find in the database, since the first update is only visible between both updates.

In the following sections, we describe how you can prepare your applications for concurrency, and prevent these problems.

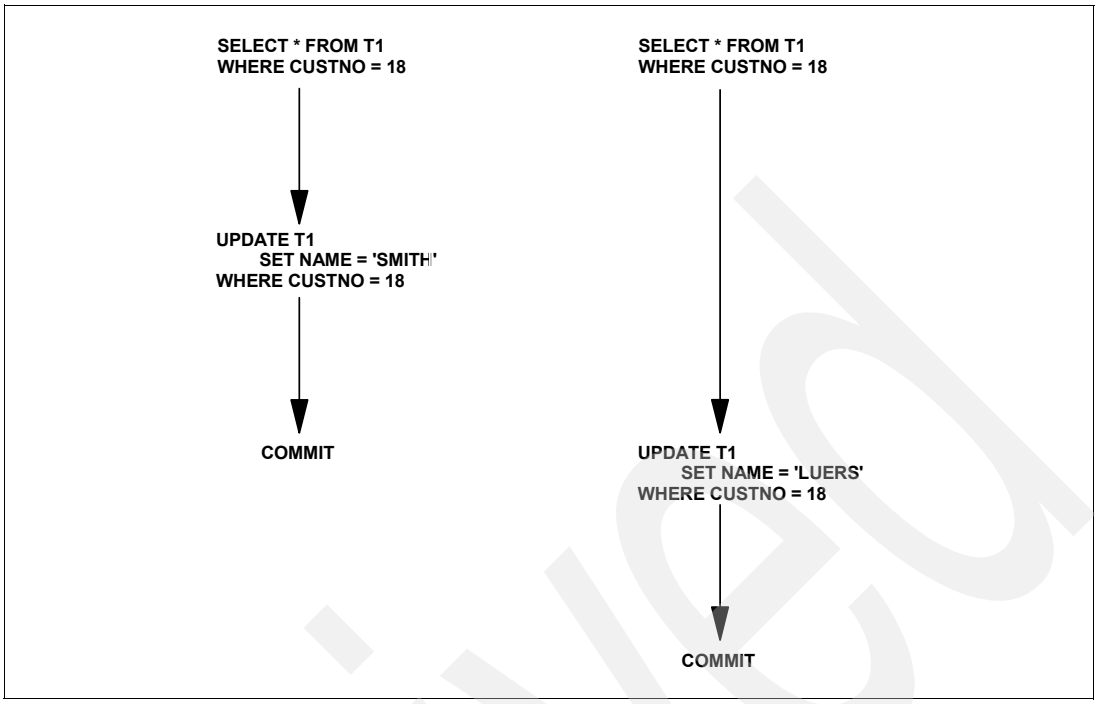


Figure 9-11 Concurrently updating processes

9.7.2 Last update column

A common approach to deal with concurrent updates for certain rows is to add a timestamp column to a table which contains the timestamp when a certain row has been last updated. The idea is to reference the previously read value in your UPDATE statement.

See Example 9-12 for the DDL we assume in Figure 9-12 on page 247, containing a unique index on CUSTNO.

Example 9-12 DDL for table T1

```
CREATE TABLE QUALIFIER.T1
( CUSTNO     DECIMAL(10,0) NOT NULL
, NAME       CHAR(32)      NOT NULL WITH DEFAULT
, COLa       CHAR(10)      NOT NULL WITH DEFAULT
, ...    ...           ...
, TS_LAST_CH TIMESTAMP NOT NULL )                  -- Timestamp of last update
 IN DBNAME.TSNAME;
```

For a visual explanation of application programming techniques for ensuring concurrency, see Figure 9-12 on page 247.

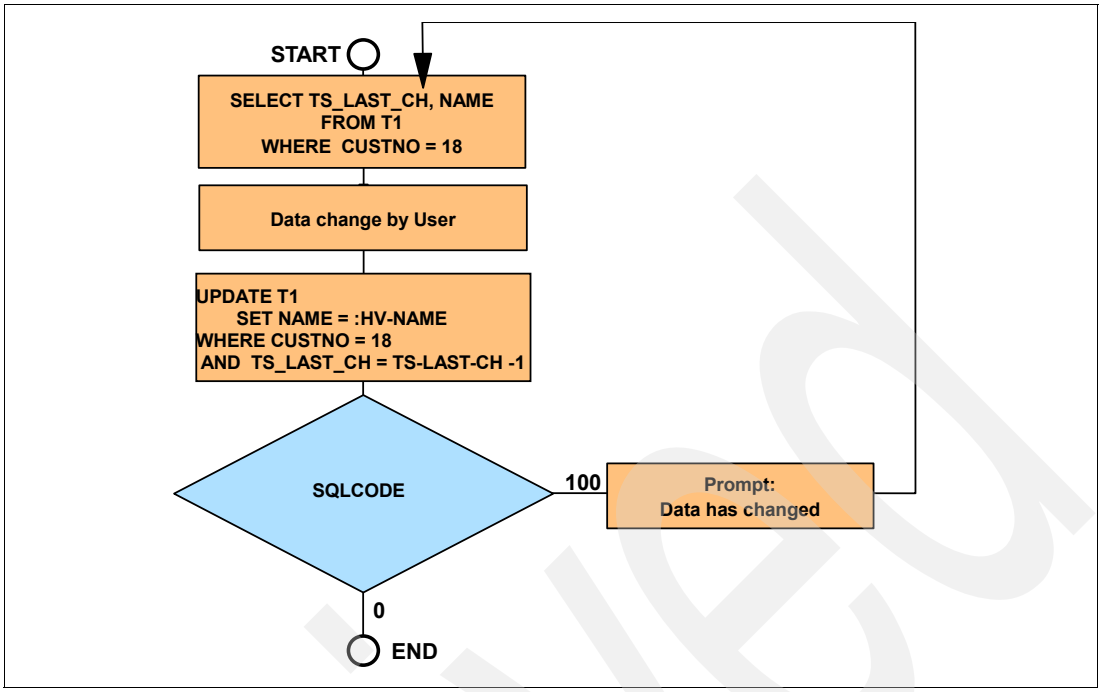


Figure 9-12 Coding for online concurrency

Right at the beginning of the business case, the application reads data for CUSTNO 18 and displays it on the screen while the timestamp when the data was read is stored in host variableTS-LAST-CH-1. After an unspecified amount of time, the user changes the data and sends it back to the application. To ensure no concurrent process has updated the data, include the previously retrieved value for TS-LAST-CH-1 in the WHERE clause of your UPDATE statement.

To provide the previously read timestamp to the updating transaction, you can consider two techniques:

- Timestamp becomes an additional output parameter for read-transactions and an input-parameter for updating transactions.
- Timestamp is kept in temporary storage among transactions.
  - Scratch pad area for IMS transactions
  - Transient data queue for CICS transactions

In case you use a change logging table and have no plans to keep the timestamp of most recent data changes in the same table as your operational data, you can consider using change logging tables and perform the check if a change has occurred on the change logging table.

If available, storing values as user-identification, terminal-number, table name, key values of affected rows, and type of DML (insert, update, or delete) can help you in error situations as well as provide information regarding who has changed a certain value in the database.

However, if you implement change logging tables, keep in mind that additional tables in your transaction increase elapsed time, the amount of I/Os, and CPU consumption for each executed transaction.

Archived

Pessimistic locking

In the case of a high probability of previously read rows being updated by concurrently running processes, you can flag the previously read rows as *update pending* using application logic. Concurrent processes are not allowed to update these rows. Be aware that using this technique enormously decreases the availability of your data.

Normally, a record lock is taken on a read and this lock is only released when the application code has finished with the record and releases the lock. This form of locking is known as *pessimistic locking* because your application assumes that previously accessed data will change during processing time.

Optimistic locking

Applications using optimistic locking expect successful updates and no concurrent data updates during processing time. The approach of optimistic locking compares previously read column values to actual column values in the WHERE clause of your UPDATE statement. Assume you are going to update COLb of table T1 WHERE COLa = 10 and the old value of COLb is stored in host variable HV-COLb-OLD. To ensure that no concurrent application has already updated the value of COLb, you can code the update as shown in Example 9-13.

Example 9-13 Update using optimistic locking

```
UPDATE T1
   SET COLb = :HV-COLb-NEW
      ,COLc = :HV-COLc-NEW
      ,COLd = :HV-COLd-NEW
 WHERE COLa = 10
   AND COLb = :HV-COLb-OLD
   AND COLc = :HV-COLc-OLD
   AND COLd = :HV-COLd-OLD;
```

Unless you do not list all old column values in your WHERE clause, the context in which an update is applied may have been changed. It depends on your business process if concurrent updates on different columns of one single row are allowed or not. Assuming COLa is unique, Example 9-14 shows two update statements, both of them affecting the same row, manipulating different columns. If values for COLb and COLc depend on each other, the old values for those columns have to be included in the WHERE clause, too.

Example 9-14 Concurrent update on independent columns

```
Transaction 1:

UPDATE T1
   SET COLb = 20
 WHERE COLa = 10
   AND COLb = 15;

Transaction 2:

UPDATE T1
   SET COLc = 30
 WHERE COLa = 10
   AND COLc = 25;
```

If all previous column values are checked in a WHERE clause as shown in Example 9-15 on page 249, the first update statement will succeed, while the second statement will receive SQLCODE 100 since COLb or COLc has changed.

Example 9-15 Concurrent update on dependent columns

```
Transaction 1:

UPDATE T1
   SET COLb = 20
 WHERE COLa = 10
   AND COLb = 15
   AND COLc = 25;

Transaction 2:

UPDATE T1
   SET COLc = 30
 WHERE COLa = 10
   AND COLb = 15
   AND COLc = 25;
```

You can consider the technique mentioned above using a TS-LAST-CH instead of providing old column values a special case of *optimistic locking*.

9.7.3 Work in progress tables

Work in progress tables are considered if you have to package more than one unit of work from a DB2 point of view together in one business unit of work by “tying” different DB2 units of work together. If you apply data changes after every single unit of work to your operational data, other applications can read data that may be inconsistent. An additional issue is the case of an abend. How do you undo the changes made by the first two transactions if the third fails? See Figure 9-13 on page 250 for a visual explanation of work in progress tables.

Archived

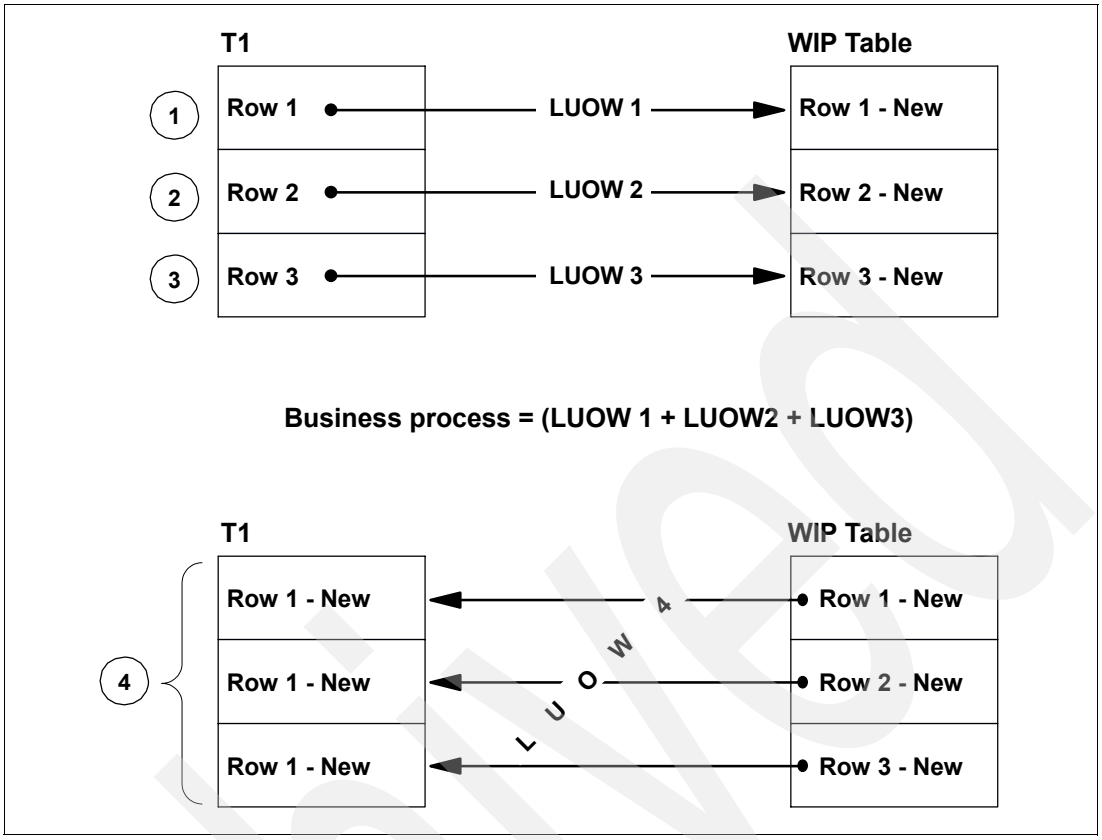


Figure 9-13 Work in progress table

Upon invocation of the first unit of work, the information that is subject to change is copied from the actual table into a work table, and a logical application lock is taken in the actual table to signal this event to other transactions that also want to update the same information. The other transactions that are part of the same application logical unit of work interact with the information in the work in progress table. Once the application logical unit of work completes, the updated information from the work in progress table can be stored in the actual table. This process is within the scope of a DB2 logical unit of work, so DB locking mechanisms can be used to guarantee consistency.

The work in progress table solves the consistency issues of the application, but might create another challenge if multiple transactions running in parallel use the same table. The work in progress table becomes a hot spot.

To tie certain transactions together, consider using artificial keys. Therefore, a logical transaction identifier can be passed from the front end (which is aware of a logical unit of work containing *n* physical units of work) to application processes in the back end.

There are several ways we can deal with the hot spot. Using isolation UR can be a solution for those statements that retrieve data. UR isolation does not, however, provide a solution for UPDATE and DELETE. The use of row locking is appropriate, unless this table is used in a data sharing environment. In a data sharing environment, consider using MAXROWS 1.

To remove any contention problem, it is best to add a column that uniquely identifies your application logical unit of work. Adding this column to all WHERE clauses guarantees that you do not really need UR, although there might be a performance benefit, provided that the additional column is a matching predicate in the index and that the optimizer uses a matching index access path.

A work in progress table can vary in size dramatically, starting with zero rows, growing during the day and containing again zero rows at the end of the day (assuming your business cases are finished during online hours). This makes it even harder to collect proper statistics to ensure appropriate access paths. Consider using the VOLATILE attribute for a work in progress table when you want to privilege index access even if no statistics are available. You can find more information about volatile tables in “Volatile tables” on page 23.

9.8 Intelligent exception handling

Intelligent exception handling is a general requirement. In the following sections, we describe possible design approaches to equip your application processes with a kind of logic to simplify your daily business.

9.8.1 Using DSNTIAR

You can use DSNTIAR to convert a SQLCODE into a text message. The module DSNTIAR is provided in each installation of DB2 for z/OS and is located in SDSNLOAD library. To execute DSNTIAR, no active thread is needed. The input used by DSNTIAR is SQLCA structure. You may want to run GET DIAGNOSTICS right after an SQL statement affecting multiple rows returns a non-zero SQLCODE.

Programs that require long token message support should code the GET DIAGNOSTICS statement instead of DSNTIAR. Keep in mind that to use GET DIAGNOSTICS, you invoke SQL again, and you need an active thread and a package. If you only want to translate SQLCODE into an error message, DSNTIAR consumes less CPU than GET DIAGNOSTICS.

> **Tip:** If your CICS application requires CICS storage handling, you must use the subroutine DSNTIAC instead of DSNTIAR.

Furthermore, DSNTIAR needs an output area and the length of each line that is moved into the output area. See Example 9-16 for how to implement DSNTIAR in your application program to retrieve output lines at 79 bytes per line.

Example 9-16 Implementation of DSNTIAR

```
Variable definitions:

05 LRECL                                PIC S9(9) COMP value +1027.

05 ERROR-MESSAGE.
  10 ERROR-LEN                          PIC S9(4) COMP VALUE +1027.
  10 ERROR-TEXT                         OCCURS 13 PIC X(79).

Invocation:

CALL ‘DSNTIAR’ USING SQLCA
                     ERROR-MESSAGE
                     LRECL
```

In general, not every module should deal with DSNTIAR. Instead, consider implementation of a central error handling module. Using central modules, the output of all SQL error messages appears in the same structure, simplifying the tasks of application development and error

investigation. You can consider the following parameters as input to a central error handling routine, so that certain applications do not have to deal with DSNTIAR:

- Perform abend yes/no
- Perform no abend, but rollback
- Perform no abend, but dump storage

Additionally, it makes sense to pass GET DIAGNOSTICS ALL output for parsing to a central error handling routine and returning the results in a common structure. Therefore, an additional parameter can be:

- Analyze GET DIAGNOSTICS ALL output

In case a central error handling routine accesses DB2 tables (for example, SYSIBM.SYSPACKAGES if SQLCODE -805 is passed from the calling module), ISOLATION (UR) should be used to minimize contention problems at all levels.

If a central error handling module is in charge of performing an abend, it is important to pass information about the causing module and the affected SQL statement for output.

In case you decide to dump storage without taking an abend, you should ensure that sufficient time has passed between taking two separate dumps.

Since CICS does not allow you to issue displays, CICS requires special handling routines.

9.8.2 GET DIAGNOSTICS

DB2 for z/OS Version 8 provides a new function complementing SQL statements affecting multiple rows and extending the information available in SQLCA: the GET DIAGNOSTICS statement. You can look at this statement as a facility to obtain much more detailed information about what has happened during the execution of a SQL statement.

The statement enables applications to retrieve diagnostic information about the last SQL statement that has been executed. GET DIAGNOSTICS returns SQL error information for an overall SQL statement and for each condition if multiple errors have occurred. It supports SQL error message tokens greater than 70 bytes as it is provided by SQLCA. This statement can only be embedded in application programs and cannot be dynamically prepared.

The GET DIAGNOSTICS statement is important to provide the information from all of the extended names and multi-row operations. You will need to switch from using SQLCA to GET DIAGNOSTICS for diagnostic information when your application includes long names or multi-row operations.

The GET DIAGNOSTICS statement can return a lot more information about the statement, about conditions, and connections. Additionally, it can return multiple conditions for statements affecting multiple rows since it can return the error message associated with an error, similar to DSNTIAR.

Do not issue GET DIAGNOSTICS after each invocation of an SQL statement, instead of checking SQLCODE. Use GET DIAGNOSTICS only if you have received an error SQLCODE to gather further information not provided by SQLCA. Invoking GET DIAGNOSTICS adds the additional overhead of one more roundtrip to DB2 to your application process. If you use GET DIAGNOSTICS, make sure that your application program does not issue another SQL statement before gathering information, since only the last SQL statement executed from within your program can be examined, which is the same rule for looking at SQLCA.

See Figure 9-14 on page 253 for an example of when to use GET DIAGNOSTICS.

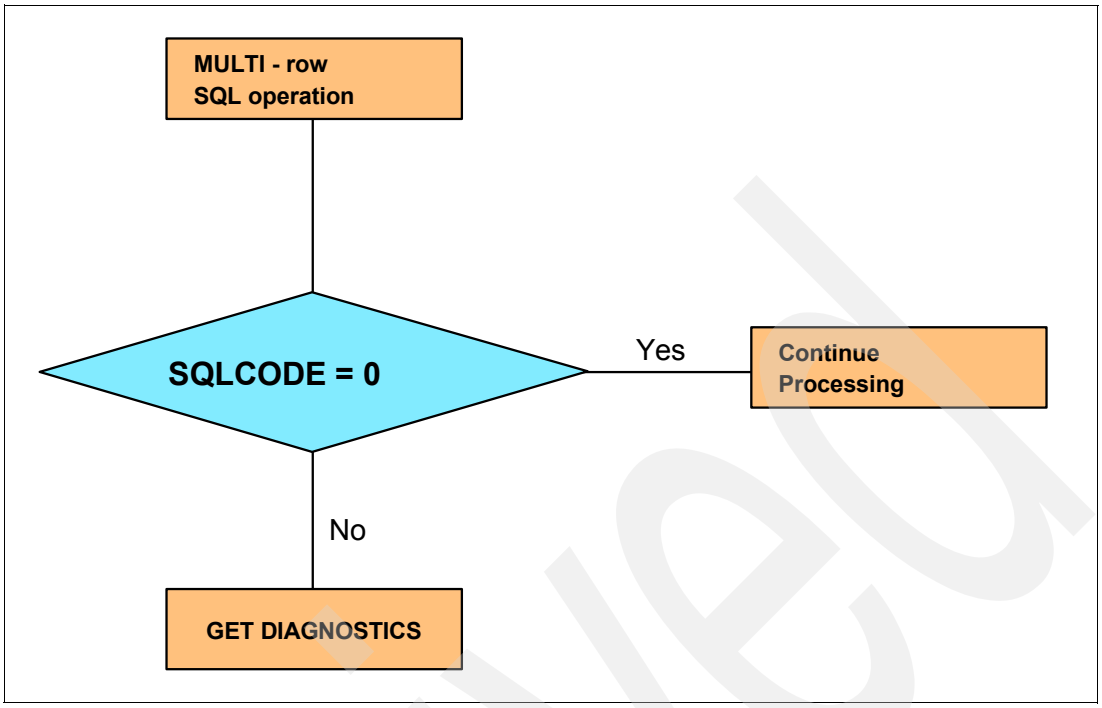


Figure 9-14 When to use GET DIAGNOSTICS

Note that the statement can be any SQL statement affecting multiple rows or a SQL statement affecting objects using long names provided by DB2 for z/OS Version 8. For multi-row FETCH, see “Determining the result set” on page 242 regarding how to determine the number of returned rows.

Additionally, any SQLCODE that is reasonable for your application process can allow you to continue without executing GET DIAGNOSTICS.

The need for GET DIAGNOSTICS especially appears when you deal with objects where SQLCA cannot provide the information you need, when looking at multi-row operations. Use GET DIAGNOSTICS to handle multiple SQL errors that may result from one SQL statement, for example, for multi-row INSERTs using NOT ATOMIC CONTINUE ON SQLEXCEPTION. Some of the information available through GET DIAGNOSTICS is also available in SQLCA.

> **Important:** Do not use GET DIAGNOSTICS instead of checking SQLCODE. Use GET DIAGNOSTICS only if you have received an error SQLCODE to gather further information not provided by SQLCA.

GET DIAGNOSTICS consists of the following groups of items:

- Statement items, which contain information about the SQL statement as a whole
- Condition items, which contain information about each error or warning that occurred during the execution of the SQL statement
- Connection items, which contain information about the SQL statement if it was a CONNECT statement

Since DB2 for z/OS Version 8 allows you to use longer names for your DB2 objects and columns, some of them will potentially no longer fit into the SQLCA. The only way to retrieve

them is by using GET DIAGNOSTICS. Possible candidates for long names in output of GET DIAGNOSTICS are:

- Table names
- Column names
- CATALOG_NAME
- CURSOR_NAME
- DB2_ORDINAL_TOKEN_N
- SERVER_NAME
- DB2_AUTHORIZATION_ID
- DB2_SERVER_CLASS_NAME

You can find a complete list of parameters and data types for GET DIAGNOSTICS in *DB2 for z/OS Version 8 SQL Reference,* SC18-7426-02.

Using GET DIAGNOSTICS

To retrieve information about multiple errors that occurred in a multi-row operation, you have the following choices:

- Execute GET DIAGNOSTICS in a loop until you have retrieved the information for all errors.
- Use GET DIAGNOSTICS ALL statement.

GET DIAGNOSTIC - A loop

See Example 9-17 for a short description about how to use GET DIAGNOSTICS after a multi-row INSERT statement was executed.

Example 9-17 GET DIAGNOSTICS on multi-row INSERT

```
INSERT INTO T1 FOR 5 ROWS
VALUES (:HV-COLa-ARRAY)

test for SQLCODE 0

NOT ATOMIC CONTINUE ON SQLEXCEPTION

GET DIAGNOSTICS :NUMERRORS = NUMBER

I:=0

Loop until I = NUMERRORS
I:=I + 1

GET DIAGNOSTICS CONDITION :I :HV-SQLCODE = DB2_RETURNED_SQLCODE
```

First, the number of errors and warnings detected by the execution of the previous SQL statement, other than a GET DIAGNOSTICS statement, is retrieved into host variable NUMERRORS. If the previous SQL statement returned an SQLSTATE of 00000 or no previous SQL statement has been executed, the number returned is one. For each error occurred, GET DIAGNOSTICS is invoked again to retrieve more detailed information.

GET DIAGNOSTICS ALL

Using GET DIAGNOSTICS ALL returns all diagnostic information for the previously executed statement in one string. The format of the string is a semicolon separated list of all of the available diagnostic information in the form:

```
<item-name>[(<condition-number>)]=<value-converted-to-character>;...
```


Archived

as shown in Example 9-18.

Example 9-18 Example of GET DIAGNOSTICS ALL output

```
GET DIAGNOSTICS :HV-STRING ALL STATEMENT

NUMBER=1;RETURNED_SQLSTATE=02000;DB2_RETURNED_SQLCODE=+100;
```

Host variable HV-STRING identifies a variable described in the program in accordance with the rules for declaring host variables. The definition of the host variable equals the definition of a host variable for a VARCHAR data type. If the length of the host variable is insufficient to hold the returned diagnostic string, the string is truncated, a warning is returned, and the GET_DIAGNOSTICS_DIAGNOSTICS item of the diagnostics area is updated with the details of this condition.

The information returned can be parsed by your application program. Not every application program should parse the structure by itself, but a central error handling routine should be implemented to process the long string.

By using GET DIAGNOSTICS ALL, you can avoid unnecessary calls of GET DIAGNOSTICS function for each affected row. We recommend using GET DIAGNOSTICS ALL statement.

GET DIAGNOSTICS provides a text representation of all the information gathered about the execution of an SQL statement. The combined information parameters are:

- ALL

  All diagnostic items available set for the last executed SQL statement are combined within one string.
- STATEMENT

  Indicates that all statement-information-item-name diagnostic items that are set for the last SQL statement executed should be combined into one string.
- CONDITION

  Indicates that all condition-information-item-name diagnostic items that are set for the last SQL statement executed should be combined into one string.
- CONNECTION

  Indicates that all connection-information-item-name diagnostic items that are set for the last SQL statement executed should be combined into one string.

The maximum size for a host variable containing GET DIAGNOSTICS ALL output is 32,672 bytes. Depending on the number of rows in error, information is likely to be truncated.

You can find a complete list of host variable definitions in *DB2 for z/OS Version 8 SQL Reference,* SC18-7426-02.

Return code for GET DIAGNOSTICS

DB2_GET_DIAGNOSTICS_DIAGNOSTICS contains textual information about errors or warnings that may have occurred in the execution of the GET DIAGNOSTICS statement. The format of the information is similar to what would be returned by a GET DIAGNOSTICS :HV = ALL statement.

Important: GET DIAGNOSTICS does not change the content of SQLCA.

Refer to *DB2 for z/OS Version 8 SQL Reference*, SC18-7426-02, for more detailed information about GET DIAGNOSTICS.

> **Important:** If you use GET DIAGNOSTICS immediately following an SQL statement using private protocol access, DB2 returns an error.

9.8.3 Retry logic

You may encounter situations where it can be a good idea to let applications handle unexpected SQLCODEs to get the correct result, instead of prompting error messages to users. The handling of the SQLCODEs affects the probability of a retry being successful. While the probability of encountering a deadlock a second time afterwards may be low, it is more unlikely that an unavailable resource is available immediately after the first SQL call has failed. Table 9-6 lists error SQLCODEs where application-controlled retries may be useful.

Keep in mind that SQLCODE -911 (victim of a deadlock) rolls back the current unit of work to the previous COMMIT and closes all open cursors. Issuing the same statement again without repeatedly processing the previously executed application logic and maintaining counters also as work-dependent variables can result in the loss of data integrity. See Table 9-6 for ideas for possible parameters to consider when coding for automatic retries.

Table 9-6 Candidate SQLCODEs for application retry logic

Error SQLCODE	Description
-904	Unsuccessful execution caused by an unavailable resource.
-911	The current unit of work has been rolled back due to deadlock or timeout, a ROLLBACK statement is issued on behalf of the application process. All updates during the current unit of work have been undone.
-913	Unsuccessful execution caused by deadlock or timeout, a ROLLBACK statement is not issued, but, in case of deadlock, the application process itself is requested either to issue a ROLLBACK statement or to terminate.

When a deadlock or timeout occurs in TSO, RRS, batch, and CAF environments, DB2 attempts to roll back the SQL for one of the application processes. As a result, the application process receives one of two possible SQL error codes: -911 or -913.

If you get a -913, and the situation is a deadlock, retry work can just make the situation worse, you need to roll back. To find out, check the reason code or SQLERRD(3) in the -913 message text listed in Example 9-19.

Example 9-19 Message text for -913

```
-913 UNSUCCESSFUL EXECUTION CAUSED BY DEADLOCK OR TIMEOUT. REASON CODE reason-code, TYPE OF
RESOURCE resource-type, AND RESOURCE NAME resource-name
Explanation: The application was the victim in a deadlock or experienced a timeout. The
reason code indicates whether a deadlock or timeout occurred. Refer to message DSNT500I
under DB2 Messages for an explanation of 'resource-type' and 'resource-name'. Refer to
Table 3 in Appendix C, “Problem determination,” on page 731 for an explanation of resource
type codes.
```

```
System Action: The SQL statement cannot be executed. If the SQL statement being executed
was a cursor FETCH, DB2 closes the cursor.
SQLERRD(3) also contains the reason-code which indicates whether a deadlock or timeout
occurred. The most common reason codes are:
        00C90088 - deadlock
        00C9008E - timeout
```

Depending on your transaction monitor (IMS or CICS), different actions can take place. See *Application Programming and SQL Guide for DB2 for z/OS Version 8,* SC18-7415-01, for further details.

The number of automatic retries you may want to implement can differ depending on more than one parameter. Consider the current workload on your LPAR and the currently specified DSNZPARMs. You may think twice before implementing five retries on SQLCODE -913 when timeout limits are set to 60 seconds in your current installation of DB2. Implementing certain numbers for automatic retries in application logic must not be static. See Example 9-20 on page 258 for dynamic program controls on error handling.

According to different reasons for receiving timeouts or deadlocks, we provide you a list containing the probability of completing a unit of work after deadlock or timeout conditions have occurred.

You should retry issuing the same statement if you receive one of the following conditions:

- SQLCODE -911 with reason code 00C90088 after repositioning all cursors

Passing the same statement to DB2 does not make sense for any of the following conditions:

- SQLCODE -904 with reason code 00C9008E

The reason code is placed in SQLERRD(3) in SQLCA and contains the reason code which indicates whether a deadlock or a timeout or the resource unavailable condition has occurred.

The information passed to the application in deadlocks and timeouts is:

- IMS
  - In deadlocks, IMS reprocesses the transaction (internal abend U777 occurs); the transaction is not aware of being rescheduled.
  - In timeouts, the application receives SQLCODE -911.
- CICS
  - If ROLBE=YES is specified for the transaction, after a deadlock or timeout SQL code -911 is passed to the transaction and back out is performed.
  - If ROLBE=NO is specified for the transaction, the transaction receives SQLCODE -913, and the rollback decision is passed to the application.

The recommendations for high transaction rates are:

- Deadlocks
  - IMS enqueues the transaction, and no action is required.
  - In CICS, specify ROLBE=YES, and let the user retry the operation.
- Timeouts
  - Do not execute retry logic in IMS, because region occupancy may be severely affected.
  - In CICS, specify ROLBE=YES, and let the user retry the operation.

9.8.4 Error handling control table

An error handling control table (EHCT) can be used for applications determining their own correct commit frequencies and behavior on exception situations depending on various parameters. In Table 9-7, we provide you with an idea of possible attributes of an EHCT.

Table 9-7 Error handling parameters

Attribute	Content
APPLICATION_NAME	Error handling instructions for named application
START_TIME	Starting time for described behavior
END_TIME	Ending time for described behavior
COMMIT#ROWS	Commit every *n* rows
COMMIT#SECS	Commit every *n* seconds
SQLCODE	SQLCODE requiring special handling
NUMBER_RETRIES	Number of retries for SQLCODE

The table you see above gives you ideas of parameters you can decide to use to force the correct behavior for your applications.

Example 9-20 gives you an idea of possible descriptions for an application.

Example 9-20 Error handling table

```
APPLICATION_NAME  START_TIME  END_TIME  COMMIT#ROWS  COMMIT#SECS  SQLCODE  NUMBER_RETRIES
----------------  ----------  --------  -----------  -----------  -------  --------------
APP1              05:00:00    20:29:59  50           3            NULL     NULL
APP1              05:00:00    20:29:59  NULL         NULL         -904     0
APP1              05:00:00    20:29:59  NULL         NULL         -911     1
APP1              20:30:00    04:59:59  1000         8            NULL     NULL
APP1              20:30:00    04:59:59  NULL         NULL         -904     5
APP1              20:30:00    04:59:59  NULL                      -911     5
```

As you see in the example, application APP1 is forced to behave differently during online and batch times regarding commit frequency and the number of possible retries in certain exception situations caused by unexpected SQLCODEs.

Application APP1 has to COMMIT after manipulating 50 rows or at least every three seconds during online hours, whatever comes first. The number of retries for SQLCODE -904 is zero, while it is 1 for SQLCODE -911. During batch hours, APP1 commits every 1,000 rows or 8 seconds, the number of retries for each SQLCODE -904 and -911 is five.

The table should just give you an idea of how to set up an error handling table. Of course, table design can be different, for example, you can also consider using a column for 904#RETRIES instead of using different rows.

Keep in mind that SQLCODE -911 includes a rollback to solve the deadlock situation and issuing the failing statement again is probably not the solution you may want. In this case, the rolled back unit of work from a DB2 point of view has to be completely repeated, usually requiring more application logic to handle this situation automatically using an error handling table.

Independent from using an error handling control table, make sure you provide all necessary information you need to solve the problem in SYSOUT. Consider using DSNTIAR to get the full explanation for the received SQLCODE. For more information about DSNTIAR, see “Using DSNTIAR” on page 251.

For online environments, avoid abends in online transactions after receiving exceptional SQLCODEs whenever possible.

9.8.5 Determining the correct load module

SQLCODE -805 might be a common error code if you deal with multiple load libraries and with package versioning at the same time. Indicating reason code 3 for SQLCODE -805, the most common reason is that the executed load module and entries in SYSIBM.SYSPACKAGE do not match. But how does DB2 find out if the SQL has not even changed?

The solution is that a consistency token is stored in the load module. The same token is stored in SYSIBM.SYSPACKAGE. Both tokens have to be the same so that DB2 can be sure that the correct package is used for a specific version of the application program during run time. While executing a load module, DB2 checks SYSIBM.SYSPACKAGE for a package associated to the load module containing the same consistency token. If no matching package can be found, SQLCODE -805 with reason code 3 is issued.

If more than one version of the load module is available in your concatenated job step libraries, you can locate the correct load module as shown below. First, see Example 9-21 for querying the catalog to retrieve the possible consistency tokens from the catalog:

Example 9-21 Consistency token from SYSIBM.SYSPACKAGE

```
SELECT TIMESTAMP, HEX(CONTOKEN) FROM SYSIBM.SYSPACKAGE
 WHERE COLLID = 'Your Collection'
   AND NAME   = 'PGM1'
ORDER BY TIMESTAMP DESC
```

DB2 only returns one row if you have not enabled package versioning. You can find a possible result set of the query in Example 9-22.

Example 9-22 Result set for package associated to module PGM1

```
                            CONTOKEN
TIMESTAMP                     HEX
--------------------------  ----------------
2005-08-10-15.59.07.213882  17AE17EE0850B336
```

The consistency token is stored in the load module using hex values. A simple FIND command on the retrieved value will not succeed. The first four bytes of the string you are looking for are the last four bytes of the hexadecimal token as shown in SYSIBM.SYSPACKAGE. The last four bytes in your FIND command equal the first four bytes in the CONTOKEN_HEX column. See Example 9-23 on page 260 for the FIND command for the hexadecimal consistency token as shown above.

Example 9-23 Checking the load module

```
Command ===> F X'0850B33617AE17EE'

--------------------------------------------------------------------------------
...Ç.. ..........&·..Þ.Ô..à................Y.Ô.....Y.»...Ø...E...X...¢.ø.Z.Ð....
00060240000100030**5B31A1E**024000000202030001 1E1E00000E180018010C000E121B170E1A0001
000808000011000C**80367E7E**084000002E2C0205EE987B090E087B03700A150A07784030291C0802
--------------------------------------------------------------------------------
```

If you can locate the value in your load module, it is likely that you found the right module. However, this is not a documented, robust technique.

9.9 Data access topics

In the following sections, we discuss techniques which can be used inside the application to control data access.

9.9.1 Restricting an application's data access

If your application needs to protect data from being accessed by other applications, consider locking tables or sets of tables using logical locking. In this context, when we talk about logical locking, we assume a control table containing different rows for critical tables as mentioned above. In the case where a table is not allowed to be updated by other processes, the control table contains at least one entry per affected table (or set of tables). Example 9-24 shows table T1 not allowed to be updated by other applications and table T2 not allowed to be read by other applications.

Example 9-24 Logical locking table

TABLE_NAME	JOB_NAME	TIMESTAMP	LOCK_TYPE
T1	JOB1	2005-07-12-22.29.41.362282	UPDATE
T2	JOB2	2005-07-13-08.14.20.825462	READ

Using this method implicitly forces applications accessing tables contained as rows in a control table to read this table first to access the operational data. If no row can be found, all access is granted. Depending on the lock type, only reads or updates are currently allowed on the specified table.

This technique may only be used for critical tables where retrieval of inconsistent data cannot be tolerated (such as a result of application problems) or exclusive access is needed to ensure data consistency during data updates (such as a job step scope unit of work). You can also use logical locking in case you encounter application problems corrupting your data and you decide not to allow users to update, but only to view data until the problem is fixed.

Note that all locks taken by DB2 do not survive the end of a thread.

9.9.2 Applications to switch between isolation levels

In the following paragraphs, we describe a technique that an application can use to switch between more than one isolation level without affecting the program logic. A common use may be an application dynamically switching between using isolation levels CS and UR. See Example 9-25 on page 261 for the required BIND PACKAGE statements for the application.

Example 9-25 BIND PACKAGE statements

```
BIND PACKAGE (COLLI001) -
  MEM (PGM1)            -
   UALIFIER (QUALI001) -
  ACT (REP)             -
  CURRENTDATA (NO)      -
  ISOLATION (CS)        -
  RELEASE (COMMIT)      -
  VALIDATE (BIND)       -
  EXPLAIN (YES)         -

BIND PACKAGE (COLLU001) -
  MEM (PGM1)            -
  QUALIFIER (QUALI001) -
  ACT (REP)             -
  CURRENTDATA (NO)      -
  ISOLATION (UR)        -
  RELEASE (COMMIT)      -
  VALIDATE (BIND)       -
  EXPLAIN (YES)         -
END
```

The only difference between both BIND PACKAGE statements shown above is the name of the collection and the isolation level. You can use the special register CURRENT PACKAGESET to switch between both packages, both accessing the same tables. The necessary variable declarations needed for package switching are shown in Example 9-26.

Example 9-26 Host variables for SET CURRENT PACKAGESET

```
01 PARM-PACKAGESET              PIC X(18) VALUE 'COLLx001'.
01 FILLER                       REDEFINES PARM-PACKAGESET.
  05 FILLER                     PIC X(4).
  05 PARM-PACK-ISL               PIC X(1).
  05 FILLER                     PIC X(3).

01 PACKAGE-OF-CALLER             PIC X(18).
```

A change of special register CURRENT PACKAGESET is required to get advantage of switching isolation levels from within the application, it has to be set before your first data access using SQL inside your application. You only need to change the special register if it differs from the CURRENT PACKAGESET which is already used. Only in this case you have to switch it back at the end of your program. See Example 9-27 for a short pseudo-code description of when to set CURRENT PACKAGESET.

Example 9-27 Setting CURRENT PACKAGESET

```
Program entry:

EXEC SQL
  SET :PACKAGE-OF-CALLER = CURRENT PACKAGESET
END-EXEC

For ISOLATION (CS):    MOVE ‘I’ TO PARM-PACK-ISL

For ISOLATION (UR):    MOVE ‘U’ TO PARM-PACK-ISL

IF PARM-PACKAGESET NOT = PACKAGE-OF-CALLER THEN
  EXEC SQL
    SET CURRENT PACKAGESET = :PARM-PACKAGESET
```

```
  END-EXEC
END-IF

Main program

Program exit:

IF PARM-PACKAGESET NOT = PACKAGE-OF-CALLER THEN
  EXEC SQL
    SET CURRENT PACKAGESET = :PACKAGE-OF-CALLER
  END-EXEC
END-IF
```

The host variable for assigning the new value for CURRENT PACKAGESET must use the same format as the register itself, which is CHAR(18).

Important: If you change special register CURRENT PACKAGESET, either make sure all other programs involved in a certain thread support the same collection or reset CURRENT PACKAGESET before exiting your program. Otherwise, DB2 issues SQLCODE -805 at the time when the next SQL is invoked inside your thread.

If you change CURRENT PACKAGESET special register, ensure that PKLIST entries in your BIND PLAN statement support the used packages.

You can retrieve the isolation level to use from the application which is calling your module.

Archived

10

Advanced programming techniques

Today's systems and environments need to be available 24 hours a day, 7 days a week, and 52 weeks per year. Besides minimizing outages as far as possible, they also have to perform better than ever before in order to meet the variable workloads and the good response times needed by e-business requirements.

In this chapter, we discuss several more advanced programming methods, the type that TV shows call "do not try them at home". Before you start implementing any of the techniques we mention in this chapter, talk to your system programmer or DBA and evaluate the impact of these programming methods on your whole environment.

As an application programmer, you have indirect influence on the amount of data logged by DB2, so you may want to warn your systems DBA to pay attention to log data volumes and discuss solutions together when these volumes exceed 10 MB/second.

This chapter contains the following topics:

- Massive insert, update, and delete
- Delete considerations
- Special data accesses
- Optimizing repeat processes
- Batch process design

© Copyright IBM Corp. 2005. All rights reserved.

10.1 Massive insert, update, and delete

During everyday business operations, sooner or later you encounter a bottleneck in your system where your current setup does not satisfy the requirements. A typical example is the shrinking batch window impacting heavy batch workloads where millions of rows have to be processed.

In this section, we discuss several techniques to increase the throughput of your system regarding updates of large amounts of data.

There is a major difference between changing only one row and many thousands, if not millions, of rows. When changing a single row, you normally use index access to locate the row, ending up in a synchronous I/O to read the page and apply your changes. For online applications, this is not critical so far.

But as soon as you start applying changes to many rows, you must be concerned about the way the changes are done since ending up in millions of synchronous I/Os is definitely going to be too costly. The way you apply huge amounts of data changes to your populated tables depends very much on what you care about most. There is again a tradeoff between availability and performance, the choice depending on your priorities and business requirements.

As you push the envelope of performance, solving one performance bottleneck often highlights another potential bottleneck elsewhere in the system. Keep this in mind particularly with massive insert, update, and delete activity.

For instance, while our typical recommendation is to perform all update activity at the end of the unit of recovery to reduce lock duration, performing a mass insert, update, or delete that logs a huge amount of data right before committing can result in a very long commit process because all of that data must be externalized. Spreading massive insert, delete, and update activity across the entire unit of recovery increases the chances that the log data has already been written by the deferred write of a previous process before your final commit is issued.

10.1.1 Insert considerations

We start by giving you general information about how DB2 inserts data. We then describe two approaches of inserting large amounts of data into a table. The first one is about multi-row insert, the second one is about the LOAD utility.

DB2 inserts

When you insert large amounts of data into a DB2 table, there are four typical potential bottlenecks that you need to resolve with assistance from your database administrator:

- Random and sequential I/O to index and data
- Active log write I/O
- Lock and latch contention
- CPU time

When you insert data, you have to distinguish between random inserts and sequential inserts, both in the data and in the index. By definition, inserts that are in the sequence of the clustering index will be sequential in the clustering index and the table. This same insert, however, will be random relative to the order of existing nonclustered indexes. A random insert into the table will also be random with respect to the clustering index, but may match the sequence of some nonclustered index.

For each random insert into an index, you have the potential of *n* synchronous I/Os where *n* depends on the number of index levels. Maintaining indexes during inserts can become the primary bottleneck if you have multiple indexes on large tables. If possible, size your buffer pool large enough to contain all nonleaf pages to reduce I/O to the top levels of the index. A random insert in the table usually has one synchronous I/O for reading the data page where the row is going to be inserted. Both the index and the table space will encounter asynchronous deferred write I/O after the row is inserted.

For random inserts into an index, specify PCTFREE and FREEPAGE that are greater than zero to reduce index splits. We recommend you start with PCTFREE 10 FREEPAGE 31 and then adjust it accordingly depending upon your index split activity. Specifying free space on the table space will allow DB2 to insert rows in clustering sequence and reduce the need to REORG. However, by specifying PCTFREE 0 FREEPAGE 0 on the table space, you can force insert activity to the end of the data set, and, thereby, reduce the amount of random I/O when inserting in the table space. This technique will require a REORG of the table space to return the rows to clustering order.

Sequential inserts according to an ever ascending or descending clustering index go to the end of the data set, so DB2 is able to significantly reduce both read and write I/Os. By inserting sequentially, only one page is written for multiple rows inserted, assuming more than one row fits in a data page. When insert activity is in accordance with the clustering index, specifying PCTFREE 0 FREEPAGE 0 on the clustering index and data pages reduces the size of the table and index and improves page density. Keep in mind the order of the nonclustering indexes, and specify free space for them if inserts are not in the same sequence as the index key.

Consider running ONLINE REORG after you have inserted large amounts of data randomly.

The amount of data being logged is nothing you can influence from an application's point of view if you just perform necessary updates. You can find information about how to minimize the generated log data in "The effect of column placement on performance" on page 18. Of course, if you insert a row, all columns are logged.

You can also reduce the volume of logged data by using DB2's data compression.

You may see lock and latch contention during high volume insert processing, this is caused by the following typical lock waits:

- Lock waits by other concurrently running threads (see "Using a DB2 table" on page 155).
- Asynchronous preformat wait.
- When you insert or load data into a table containing an identity column or a sequence (see "Sequences and identity columns" on page 414).
- Inserts in nondata-sharing generally do not wait for locks, because conditional locks are used.

If it becomes necessary to avoid contention on preformatting pages, you can LOAD or REORG your data using the PREFORMAT option. Keep in mind that there is no page preformatting for declared temporary tables and work files. Generally DB2 preformats in tracks or in cylinders depending on your unit of allocation, with a maximum of two cylinders of data pages, depending on the SECQTY you have specified. With the sliding secondary quantity allocation, the preformatted size tends to be 2 cylinders.

Note: Make sure you have applied PK05644 for preformatting the correct amount of space.

CPU usage is mostly affected by the number of indexes that DB2 has to maintain during data insertion. We recommend you use QUBE for an upper bound estimate.

To give you an idea of how to perform a upper bound estimate of the amount of time a certain query may take to execute, see “Determining the duration of your process” on page 113.

In the following paragraphs, we describe the approaches of inserting large amounts of data into a table.

10.1.2 Multi-row insert

With versions prior to DB2 for z/OS Version 8, you do inserts into a table in one of the following ways:

- INSERT with VALUES

  Use it to insert a single row into the table using values provided or referenced.

- INSERT with SELECT

  Use it to insert one or more rows into the table using values from other tables or views.

DB2 for z/OS Version 8 has introduced a third way of inserting data using SQL: Multi-row insert.

You code INSERT with FOR *n* ROWS to insert multiple rows into a table or view using a single SQL call. With multi-row insert, DB2 uses an array of host variables to insert *n* rows where *n* can be a maximum of 32,767 for a single INSERT statement. From the application point of view, there is only one SQL invocation. Therefore, performance can improve by eliminating multiple trips between your application and the database engine. See Figure 10-1 on page 267 for a possible structure of application logic using multi-row insert inside your application.

Archived

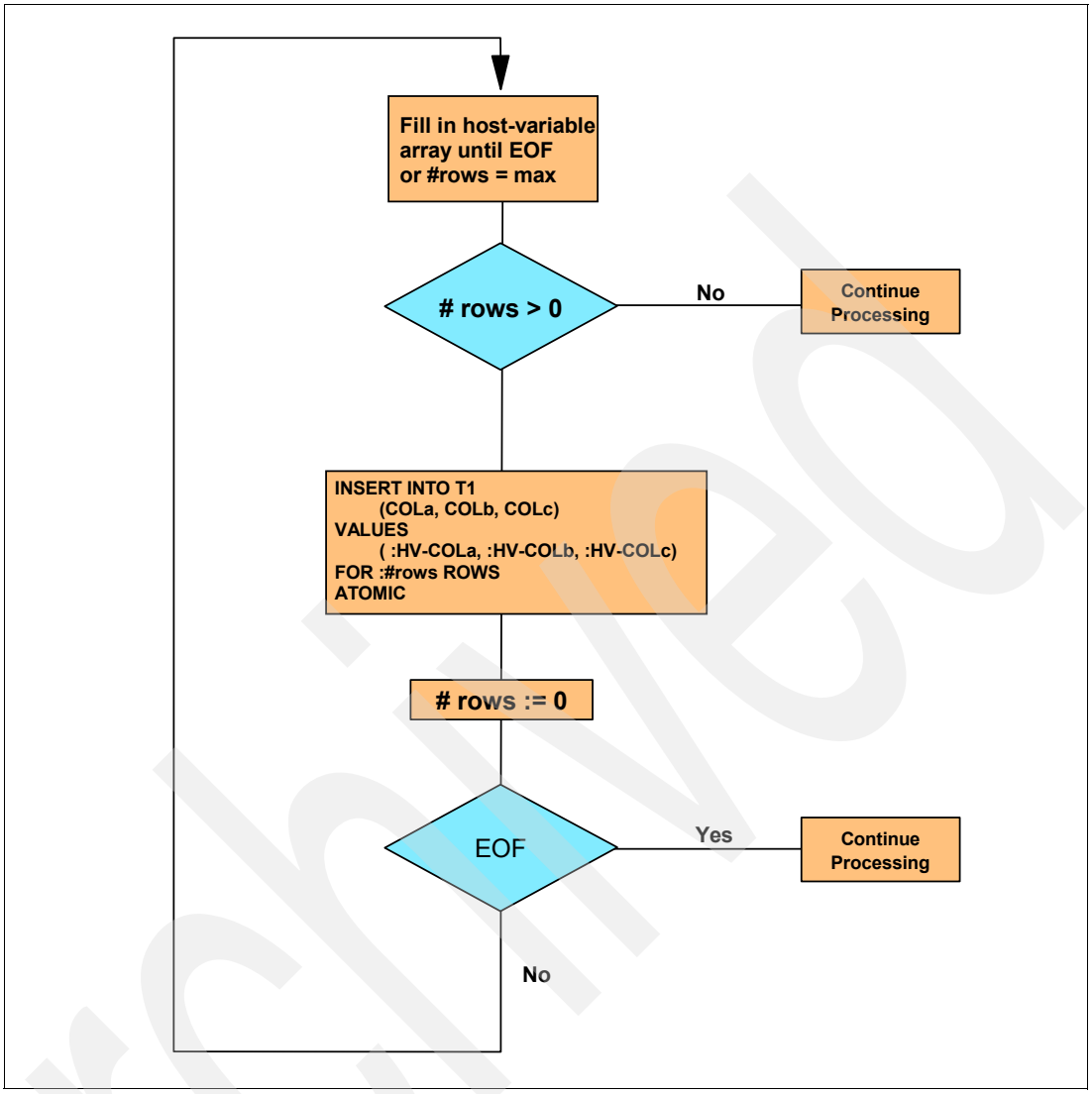


Figure 10-1 Application logic for multi-row INSERT

Note that *max* specifies the maximum number of rows you are going to use on the multi-row insert statement. The variable *#rows* specifies the current amount of used host variable arrays.

ATOMIC or NOT ATOMIC

Additionally, DB2 provides the options ATOMIC and NOT ATOMIC CONTINUE ON SQLEXCEPTION to specify if a multi-row INSERT has to succeed or fail as a unit, or if you want DB2 to proceed despite a partial failure affecting one or more rows.

- ATOMIC is the default and specifies that all of the rows should be treated as a single unit. If any row of the SQL statement fails, then all changes made to the database by any of the individual row inserts are undone.
- NOT ATOMIC CONTINUE ON SQLEXCEPTION specifies that all of the rows should not be treated as a single unit. If any row of the SQL statement fails, no attempt will be made

Archived

to undo other successful inserts. SQLCODE will provide indication that all rows have failed, or all rows have been inserted successfully, or at least one row has failed.

The relevant SQLCODEs issued during multi-row processing are:

- +252: All were successful, but warnings have occurred
- -253: At least one failed
- -254: All inserts failed

You can use the GET DIAGNOSTICS statement to keep track of this. For more information, see “GET DIAGNOSTICS” on page 252.

A consideration regarding ATOMIC or NOT ATOMIC CONTINUE ON SQLEXCEPTION is the amount of data you are inserting. Inserting 32,767 rows into a table with rows 32 KB long (the row is in a 32 KB page and there is only one row per page) consumes 1 GB of space in the application and would log more than 1 GB of data. A rollback would definitely be very painful.

From a restart or repositioning point of view, ATOMIC is easier to handle, since it is an *ALL or NOTHING* type of process. From a performance point of view, NOT ATOMIC CONTINUE ON SQLEXCEPTION, combined with efficient application logic, is the preferred solution.

Choosing a reasonable number of rows

A multi-row insert is part of a unit of work just as any other DML, so make sure your COMMIT frequency is adjusted properly, referring to the number of rows you are going to insert. Committing the changes can consume a large amount of time if a large number of rows are inserted. See “Commit” on page 348 for a detailed description for the actions performed by DB2 at the time a COMMIT is issued.

When starting on multi-row inserts, do not use a high number of rows to insert at once. Inserting the maximum number of possible rows using one invocation of SQL does not necessarily improve performance, because it is possible that no intermediate log writes are performed by concurrently running applications, resulting in a huge amount of log writes at COMMIT time. The reason is that control interval (CI) log writes to DB2’s log buffer can be triggered by another committing application.

Let us assume you use a high number of rows and perform 32,000 inserts at the end of your thread. It is very likely that no other application inside the same subsystem is committing in between, so you end up causing the wait time for your commit point by yourself. Additionally, the more rows you insert, the more logging has to be done. Instead, inserting approximately 100 rows at a time when you have collected the values in your host variable array and continue processing, gives other applications the chance to take commit points, and, therefore, write to the log a CI also containing your inserts, therefore reducing your commit wait time.

Generally, a number of 100 rows per SQL may be the maximum value starting point for multiple-row operations, and provides a good investment for performance.

CPU time may be reduced by up to 40% by avoiding application programming interface (API) overhead for each insert call, using a rowset of 100 inserts per call in a local environment (TSO). The improvement of CPU usage depends on:

- Network performance (if running in a distributed environment) by reducing network traffic by avoiding message send and receive for each row you have inserted
- Number of indexes
- Number of columns
- Row size
- Number of rows inserted per call

Multi-row insert and triggers

Another consideration is the effect on triggers. With a multi-row INSERT statement, the time at which triggers are processed depends on the atomicity option in effect for the statement:

- ATOMIC: The inserts are processed as a single statement, any statement level triggers are invoked once for the statement, and transition tables include all of the rows inserted.
- NOT ATOMIC CONTINUE ON SQLEXCEPTION: The inserts are processed separately, any statement level triggers are processed for each inserted row, and transition tables include the individual row inserted. With this option in effect when errors are encountered, processing continues, and some of the specified rows do not end up being inserted. In this case, if an insert trigger is defined on the underlying base table, the trigger transition table includes only rows that are successfully inserted.

ATOMIC is easier to restart and reposition from since it is an ALL or NONE type of process. This does not mean that it is the recommended approach. You need to choose between the two options looking at data integrity and availability of updated data.

Suggestions on multi-row insert

Being able to insert multiple rows can improve performance, particularly across the network.

Multi-row insert can affect the concurrency of other users accessing data in the same table. Minimize contention by adjusting the size of the host variable array, committing between inserts and preventing lock escalation.

Do not do heavy multi-row inserts immediately before committing your changes, because no intermediate log writes may have occurred and your COMMIT takes longer.

Multi-row insert can be implemented as either static or dynamic SQL.

Watch out for implementing multi-row inserts in case you use referential integrity or your application persists on row insertion at a specific point in time of your unit of work. This may be the case if subsequent parts of the application process refer to that particular row. Otherwise, your application logic can always be three blocks down the road while the insert is still pending because your host variable array is not filled up at that time.

Example of multi-row insert

The input variables for multi-row insert are provided as an array. The definition is the same as for multi-row FETCH statements as seen in Example 9-6 on page 240. Each language has its own conventions and rules for defining a host variable array. Each array represents cells for multiple rows of a single column.

In Example 10-1, you can find a statement for inserting more than one row at once.

Example 10-1 Inserting n rows at once

```
INSERT INTO T1
         (COLa, COLb, COLc)
  VALUES (:HV-COLa-ARRAY, :HV-COLb-ARRAY, :HV-COLc-ARRAY :HV-IND-ARRAY)
     FOR n ROWS
     ATOMIC
```

In Example 10-2, :HV-COLa and ‘ABCD’ are used for each row of data, while all values for COLb come from the array.

Example 10-2 Inserting n rows at a time combined with scalar host variables

```
INSERT INTO T1
         (COLa, COLb, COLc)
  VALUES (:HV-COLa, :HV-COLb-ARRAY, ‘ABCD’)
     FOR n ROWS
```

For further information about the performance aspects of multi-row insert, see *DB2 UDB for z/OS Version 8 Performance Topics,* SG24-6465.

10.1.3 LOAD SHRLEVEL CHANGE

The classic LOAD drains all access to a table space, so you had to write application programs for inserting data to avoid drains and allow availability to data. But by adding SHRLEVEL CHANGE RESUME YES on your LOAD utility statement, you can operate the LOAD utility like an SQL insert program, maintained by DB2, with the following characteristics:

- Claims instead of drains
- Inserts trying to maintain clustering order
- Fires triggers
- Operates with LOG YES only
- Uses existing free space

Inside ONLINE LOAD, there is an intelligent scope of COMMIT. Because you have no influence on the commit frequency chosen by DB2, data may partially be visible before the utility has completed. In case of restarting the utility, make sure nobody changed the input data set, since DB2 keeps track of the data already loaded in SYSUTIL. You can also use multiple data sets for loading different partitions to parallelize your workload. See Example 10-3 for a sample ONLINE LOAD statement using the data set associated to DD SYSREC00 for loading rows into table TB1.

Example 10-3 Sample statement for using ONLINE LOAD

```
LOAD DATA INDDN (SYSREC00)
     SHRLEVEL CHANGE
     RESUME YES
     LOG YES
     INTO TABLE QUALIFIER.TB1
```

10.1.4 LOAD

When doing a large number of inserts, deletes, and updates (massive processing, that is, a large number of rows in the table are affected by the process), usually the following technique provides for better performance, but decreases availability:

- Instead of issuing UPDATE, DELETE, or INSERT statements as you scan the entire table, write a record to a sequential file using these rules:
  - Write an updated record to reflect the new image of the row instead of UPDATE
  - Omit records to be deleted instead of DELETE
  - Write a new record instead of INSERT
  - Write a record for each existing unchanged row
- Possibly sort the output file in clustering sequence

- Reload the new table from this sequential file, using the LOAD utility with REPLACE LOG NO options.

When scanning the table for writing output records, make sure that the table is accessed using the clustering order. The records you are going to load should be loaded in clustering order, too.

Keep in mind that LOAD REPLACE empties all tables in the affected table space.

See Figure 10-2 for a visual explanation of exploiting parallelism for LOAD REPLACE on partitioned tables.

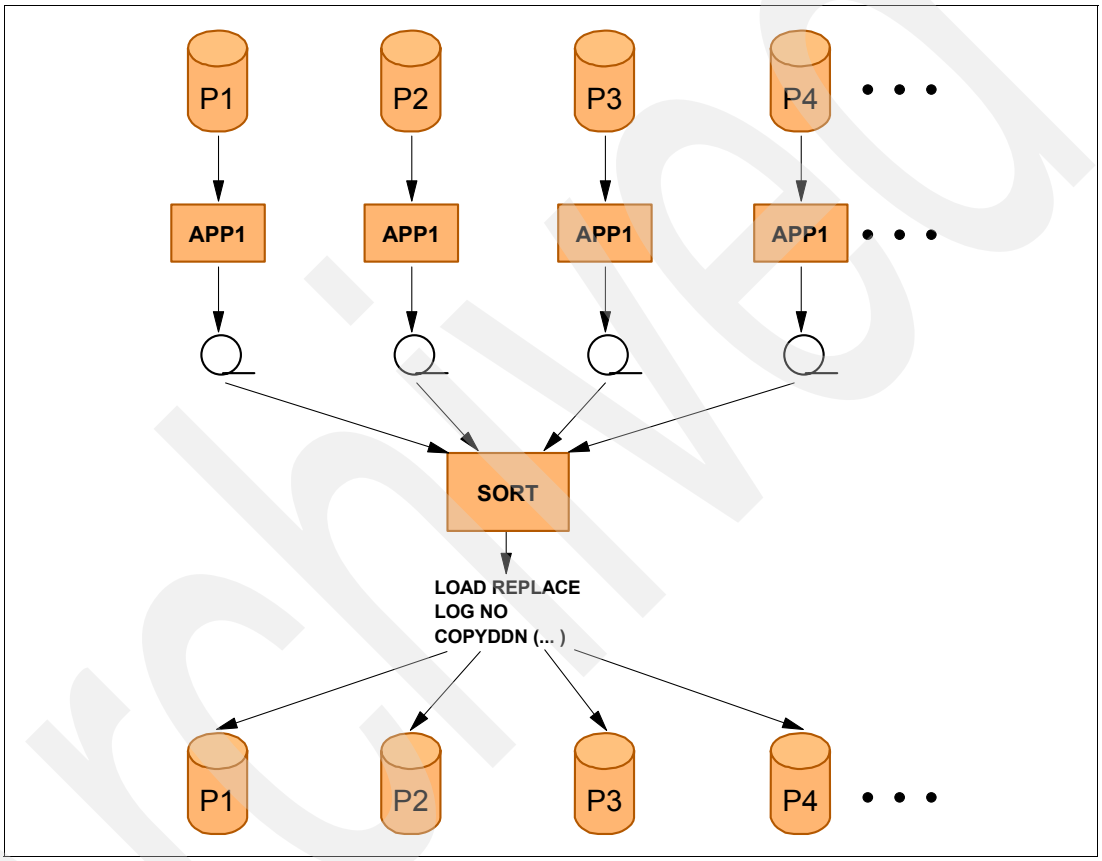


Figure 10-2 Mass change activity using LOAD REPLACE

The scan of the table should use a clustering index if the rows must be retrieved in a specific order for matching a sequential input file. Many parallel jobs can read the input table and write the changed data to output sequential files. Deleted rows can be written to a separate file for backup. The output of the parallel jobs can be merged and the LOAD with REPLACE option can be used to get the data into the table. An external sort may be more efficient rather than sorting the rows while retrieving them.

The technique mentioned above is efficient, because it can utilize such performance-oriented functions as:

- Sequential prefetch when scanning the existing table
- Load interface instead of insert interface (SQL INSERT statements) to insert rows into the table
- LOG NO option, eliminating logging of data and related overhead

In addition, there is no need to reorganize the table space after the process, because the LOAD utility creates organized tables and indexes.

The technique we have described can pay off if more than 10% of the rows in a table are manipulated.

You can also use a version of this technique when you need to insert a massive number of rows at the end of a table or partition. In this case, data in the table or partition is still valid, but you add new data at the end. You can use the LOAD utility with RESUME YES option to achieve this. To keep the table or partition organized, make sure that the new keys are higher than the highest existing key in the table or partition. Rows inserted by the LOAD utility are always stored at the end of the table or partition. The clustering sequence is not maintained for data, and the LOAD utility cannot use available distributed free space in the table space.

When you use LOAD REPLACE LOG NO, the associated table space changes to COPYPEND status and will not allow any update activity. You can avoid COPYPEND by specifying NOCOPYPEND in your LOAD statement. Using NOCOPYPEND prohibits recovery using the log as long as no full image copy is taken. We recommend using INLINE COPY during LOAD processing.

Recommendation

We recommend you use multiple-row insert if you have to do your inserts from within an application program. Use ONLINE LOAD if you have to push a sequential file to your table instead of writing a standalone application. ONLINE LOAD, introduced in DB2 for z/OS Version 7, can reduce CPU consumption by 40% with the same data availability as insert, which is similar to the multi-row insert. Since you are not able to control commit frequency during ONLINE LOAD, some data may be visible to other application processes before the utility has finished.

More information about insert versus LOAD measurement comparison is available at:

http://www.ibm.com/developerworks/db2/library/techarticle/0203shibamiya/0203shibamiya2.html

10.2 Delete considerations

If a large number of rows are affected by a singleton DELETE statement, the primary concerns are:

- An intermediate commit cannot be issued during the SQL statement execution.
- The locks acquired for the updated rows are held until a commit point after completion of the statement.
- The long duration of the locks may cause lock contention and reduce concurrency.
- The unit of work is long, so rollback times in case of abend are long. At restart, the delete process starts over from the beginning by deleting all rows again.

Avoid singleton DELETE statements for manipulating a large number of rows in your table. In the following paragraphs, we describe possible ways around the primary concerns.

10.2.1 Mass delete

A delete is considered a mass delete if all rows in a table are deleted by simply flagging the space map pages of the affected table space pages and the associated index pages as empty. Therefore, a mass delete can be done in nearly no time, regardless of the number of

affected rows. However, almost no rows are logged during this kind of processing, since ROLLBACK consists of undoing changes to the space map pages.

There is no real mass delete if any of the following is true for the affected table (this means the deletion of each row is processed independently):

- The table is the parent table of a referential constraint.
- The table is defined as DATA CAPTURE (CHANGES).
- A delete trigger is defined on the table.
- The table resides in an unsegmented table space.

If DB2 cannot use this mass delete technique, the DELETE statement results in the logging of all database data changes, requiring CPU time and I/Os.

If all rows in a table are deleted, all segments become immediately available for reuse at the COMMIT point, without requiring the REORG utility to execute. But since mass deletes only empty space map pages, consider running REORG TABLESPACE after you have performed a mass delete to reclaim the logically empty pages which are still stored in your VSAM data sets for data and indexes.

10.2.2 Deleting only qualifying rows

One way to reduce long units of work is to split the SQL statement and divide it into a certain number of parts. Using the clustering index is a good idea to limit the key range affected by the DML.

Assume a clustering index is built on COLa in table T1, the DELETE statement can be coded as you see in Example 10-4 and will be one OUW.

Example 10-4 Delete affecting many rows: The ugly way

```
DELETE FROM T1
 WHERE COLb = 1;
```

In Example 10-5, we split the long unit of work into smaller units of work, reducing locking contention and providing you with the ability to issue intermediate COMMIT points.

Example 10-5 Delete affecting many rows: The good way

```
DELETE FROM T1
 WHERE COLa BETWEEN :HV-COLa-1 and :HV-COLa-2
   AND COLb = 1;
```

If you are aware of the column distribution statistics and know how to set appropriate values of HV-COLa-1 and HV-COLa-2, you can use this technique to keep your data available and avoid concurrency problems. This solution should perform better than using a cursor to delete the data, however a cursor driven solution gives you easier control over commit-frequency.

Use a cursor driven update if you have no idea about the column distribution statistics. In this case, make sure that you declare the cursor with FOR UPDATE OF column-name and do not select a value in your SELECT statement. If you have to keep a protocol of the data you delete, you have to list all relevant columns in the SELECT column list. See Example 10-6 on page 274 for a simple example of how to declare a FOR UPDATE OF cursor.

Example 10-6 Cursor declaration for update

```
DECLARE CURSOR C1 WITH HOLD FOR
  SELECT 1 FROM T1
   WHERE COLb = :HV-COLb
     FOR UPDATE OF COLa;
```

After fetching a row, you can issue DELETE FROM T1 WHERE CURRENT OF C1 to delete the previously fetched row. Choose a reasonable commit frequency. See “Error handling control table” on page 258 for a dynamic control on commit frequency.

REORG SHRLEVEL CHANGE DISCARD

Another way of deleting a large number of rows and minimizing the impact on other concurrent processes is ONLINE REORG using DISCARD. See Example 10-7 for a sample statement of ONLINE REORG discarding all rows where COLa = 10.

Example 10-7 Online REORG using DISCARD option

```
REORG    TABLESPACE DBNAME.TSNAME
         DISCARD FROM TABLE T1
         WHEN (COLa = 10)
         SHRLEVEL CHANGE
         COPYDDN SYSCOPY
         DISCARDDN SYSDISC
         FASTSWITCH YES
         DEADLINE NONE
         MAPPINGTABLE MAP_TBL
         TIMEOUT TERM
         LONGLOG TERM
```

All deleted rows are stored in the data set assigned to DD statement SYSDISC. When using REORG SHRLEVEL CHANGE, you ensure concurrency while deleting the rows qualified by the WHEN clause in the REORG statement.

LOAD REPLACE

Use LOAD REPLACE if you have to empty all tables in a multi-table table space or the table in a single-table table space. This is the fastest way since DB2 just deletes the underlying VSAM data set and defines a new one, unless you specify REUSE keyword or a user-defined table space. REUSE specifies that DB2 resets the underlying VSAM files logically without deleting and redefining them. You only specify one table in your LOAD statement when using REPLACE. See Example 10-8 for a sample statement of LOAD REPLACE.

Example 10-8 Using LOAD REPLACE

```
LOAD DATA INDDN (SYSREC00)
     REPLACE
     LOG NO
     INTO TABLE QUALIFIER.TB1
```

DD statement SYSREC00 can be assigned as DUMMY to force loading no data. Note that all tables in the simple or segmented table space where the loaded table resides are emptied as well.

If you have multiple tables in a multi-table table space and only want to delete all the rows from one, the best option is running a mass delete statement or REORG SHRLEVEL CHANGE DISCARD.

If you use LOAD REPLACE with a DUMMY data set in input to empty a partition, this is the best performing method, but its advantages decrease with the number of nonpartitioned indexes.

Recommendation

We do not have a general recommendation about how you go about deleting large amounts of data. The solution depends on several factors, such as affected table spaces, number of affected rows, and service level agreements. What do you rely on most, performance or availability? Check the options we have mentioned and decide which solution is best for your environment.

10.2.3 Update considerations

When rows are updated, all database changes result in logging. You have no influence on the amount of data logged during update processing since this depends on the location of the column being updated as chosen at table design time, and possibly the changed data capture.

When updating large amounts of data, most of “Delete considerations” on page 272 apply.

Assume you need to update all columns COLc in table T1 and the new content of the attribute depends on the current values of COLa, COLb (in a WHERE clause), and COLd (as an input to calculating the new value). See Example 10-9 for a simple example.

Example 10-9 Updates with different WHERE clauses

```
UPDATE T1
   SET COLc = COLd * :HV-xxx
 WHERE COLb = 10
   AND COLa = :HV-COLa;

...

UPDATE T1
   SET COLc = COLd * :HV-yyy
 WHERE COLb = 15
   AND COLa = :HV-COLa;
```

When we look at a mass update affecting most rows in a table, most likely not all rows will be updated using the same value. CASE expressions provide you with excellent scalability, and at the same time greatly reduce the number of SQL statements. See Example 10-10 for an update statement using a CASE expression instead of multiple update statements.

Example 10-10 Update using CASE expression

```
UPDATE T1
   SET COLc =
  CASE COLb WHEN 10 THEN COLd * :HV-xxx
            ...
            WHEN 15 THEN COLd * :HV-yyy
  END
 WHERE COLa = :HV-COLa
```

A combination of range predicates and CASE expressions to express your update statement is a good way to split this large unit of work into multiple, more manageable units of work.

See Example 10-5 on page 273 for splitting a long unit of work into smaller units of work, reducing locking contention, and providing you the ability to issue intermediate COMMIT points.

You can avoid concurrency problems by using possible range predicates, the same way we mentioned in “Deleting only qualifying rows” on page 273. Use this technique if you are aware of the column distribution statistics and know how to set appropriate values of HV-COLa-1 and HV-COLa-2 to keep your data available and avoid concurrency problems. This solution should perform better than using a cursor to update the data; however, the cursor driven solution gives you control over commit frequency.

If your updates come along randomly, updating rows in no specific order, consider sorting the input values first, before passing them to DB2 to keep the sequence of updated rows the same as the order of the clustering index to benefit most from DB2’s sequential mechanisms. Sorting input data might not be possible in all cases, for instance, when you process the rows while inserting them, and changing their order may destroy your business process.

10.3 Special data accesses

In this section, we describe two unusual situations of data access. The first one is direct row access by means of the ROWID, and the second one is the case of a program looping on updates.

10.3.1 Data access using ROWID

If an application selects a row from a table that contains a ROWID column, the ROWID value implicitly contains the location of the row. If you use that ROWID value in the search condition of subsequent SELECTs, UPDATEs, or DELETEs, DB2 can navigate directly to that row without using an index access or a table space scan to find the row. This access method is called *direct row access*, and is a very efficient access path, because DB2 does not have to use index access or table space scan to locate the row.

You can only use direct row access on a table that contains the ROWID data type. The host variable containing your ROWID has to be the ROWID data type. Before you can use a ROWID in your WHERE clause, you have to retrieve it from DB2 at least once. See Example 10-11 for an example of how to select a ROWID, including the required definitions.

Example 10-11 Selecting a ROWID column

```
Variable definitions:

01 HV-ROWID USAGE IS SQL TYPE IS ROWID.

First select:

SELECT COLb, COLc, ROWID
  INTO :HV-COLb, :HV-COLc, :HV-ROWID
  FROM T1
 WHERE COLa = :HV-COLa;
```

However, if you expect to use direct row access, but for some reason the row has moved (this is usually the case if a REORG has been executed on the particular table space in the

meantime), DB2 may decide not to use direct row access at execution time. This may lead to serious performance problems because an alternate access path needs to be used. See Example 10-12 for a query using direct row access.

Example 10-12 Query using direct row access

```
UPDATE T1
   SET COLd  = :HV-COLd
 WHERE ROWID = :HV-ROWID
```

Note that an IN list can also be used for multiple ROWIDs to allow your query to qualify for direct row access. However, just because a query qualifies for direct row access does not mean that this access path is always chosen. If DB2 determines that another access path is better, direct row access is not chosen.

If parallelism, RID list processing, or direct row access are options, DB2 uses direct row access. If direct row access fails, DB2 does not revert to parallelism or RID list processing, but to the backup access type. You can find the backup access type in your PLAN_TABLE. In the worst case, a backup access path will result in a table space scan.

Do not write your applications so that they rely on direct row access, since it might not work after the row has moved. Furthermore, consider the following backup scenarios:

- Do not let your application remember ROWID columns across a unit of work since ONLINE REORG can move them in between because the claim is released at COMMIT.
- An index on ROWID can be an efficient backup access path.
- Add at least one indexable column to your WHERE clause to avoid table space scan as a backup access path.

Use direct row access if you are unable to keep the previously read values in application storage and need to read a certain row more than once.

10.3.2 Never-ending update

Have you ever seen your application updating certain rows using a cursor and getting into a loop? Do you think this is impossible? Figure 10-3 on page 278 shows a chart containing columns COLa and COLb where COLa is contained in index IX1. The cursor only uses WHERE COLa > 10 and updates COLa by adding a value of 10 using a separated UPDATE statement.

Since DB2 chooses matching index scan on IX1 as a valid access path for your cursor, updating a row containing COLa = 15 to a value of 25 also updates index IX1, therefore, moving the already processed value of 15 to the end of the index tree after value 20. Therefore, the "same" row qualifies again on a subsequent fetch for the same cursor.

Archived

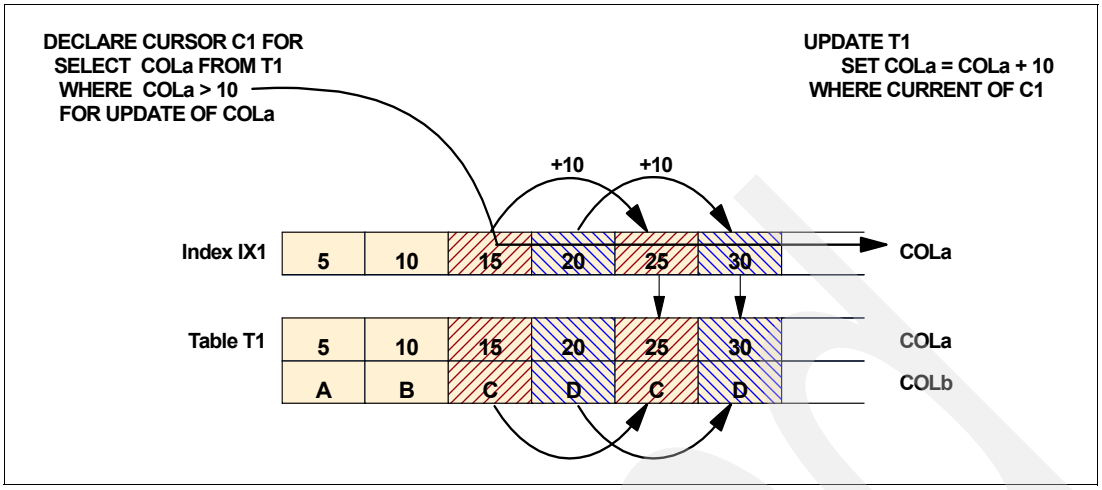


Figure 10-3 Never-ending update using matching index scan

To avoid this problem, the most effective solution is using FOR UPDATE OF, forcing DB2 to change the access path as soon as a column it uses for data access is specified in the FOR UPDATE OF clause. Therefore, coding FOR UPDATE OF COLb can still use the same access path, while using FOR UPDATE OF COLa, in our example, results in forcing DB2 to use list prefetch. With list prefetch, DB2 retrieves the RIDs of the rows you are going to access with the "same" access path without using FOR UPDATE OF, stores them in a RID list, and accesses data according to this list. Updates to the end of the index by the same application are not visible any more to the application.

Usage of list prefetch is indicated in your plan table with a value of 'L' in the PREFETCH column.

Note: If DB2 materializes the result set or a different access path is used, you do not encounter a never-ending update.

10.4 Optimizing repeat processes

Most of today's batch workloads contain repeatedly executed processes. In this section, we provide useful information regarding optimizing repeat processes. DB2 internally optimizes such processes. See "Repetitive SQL procedures" on page 429.

10.4.1 Prefetch mechanisms

Prefetch is a mechanism for reading a set of pages (usually 32) from disk into the buffer pool with one asynchronous I/O operation. DB2 uses the following prefetch mechanisms:

- Sequential prefetch
- Sequential detection (sometimes also called dynamic prefetch)
- List prefetch

See *DB2 for z/OS Version 8 Administration Guide*, SC18 -7413-02, for a complete description of the prefetch mechanisms and when you use prefetch.

Even if all mechanisms provide for asynchronous I/O, there are differences in the number of pages read and the cost of I/O per page:

- Sequential prefetch is decided by the optimizer at bind time, and each page read costs in a range of 0.03 to 0.3 msec. per 4 KB page.
- Sequential detection is triggered by DB2 at statement execution time if sequential access is detected. Each I/O operation is performed for a group of pages (usually 32). Pages, which are not within the prefetch range, can still be read using synchronous I/O by the application.
- List prefetch reads only those pages that are actually needed, but it requires a sort operation on the RIDs.

Batch programs issuing many accesses to large tables will benefit from the largely reduced synchronous I/O times provided by sequential prefetch and sequential detection mechanisms.

Because list prefetch costs are higher than sequential prefetch or sequential detection costs, you should design your processes to avoid massive list prefetch accesses and favor the use of the clustering sequence. Use list prefetch to retrieve a small number of rows. If the statement is included in a program loop, avoid a large number of executions of that loop. For more details about list prefetch, see “List prefetch” on page 67.

Sequential prefetch and sequential detection should be the preferred I/O operations for large batches. To achieve this, batch processes must be designed so that accesses to data are done on the clustering index sequence. Use list prefetch only in specific and limited situations.

10.4.2 Split programs into two parts

If you take a look at your online and batch workloads and how they are designed, you are likely to see a difference in the way these workloads access and manipulate data. Therefore, you should consider different techniques to access your data between online and batch environments.

We show the most likely differences for online and batch processing and the recommended coding technique in Table 10-1.

Table 10-1 Differences between online and batch applications

Requirement	Online	Batch
Data access	Random I/O by singleton SELECT	Most likely sequential access using a CURSOR, scanning for specific data ranges, access in clustering order
Insert activity	Random inserts	Heavy insert activity, using multi-row insert or LOAD SHRLEVEL CHANGE
Update activity	Random updates	Heavy update activity using cursor or reload data, access in clustering order
Delete activity	Random deletes	Mass deletes or LOAD REPLACE
Unit of work	Transaction scope	COMMIT interval contains *n* logical units of work

Requirement	Online	Batch
Concurrency	Check for concurrent transactions and batch jobs	Commit frequency
Restart	Issue new transaction	Maintain restart information, Implement restart logic (reposition cursors, maintain variable values)
Sequenced numbers	Generated by using DB2 sequences or identity columns	Generate using application logic (only if independent from DB2-generated numbers)
Error handling	Prompt message to user or intelligent exception handling routine	Abend or intelligent exception handling routine

There exists no general code for online and batch applications. In most cases, you need different coding to fit the requirements of the two environments.

10.4.3 Sorted input

Sort your external input data for processing DB2 tables in clustering order of the affected tables to minimize synchronous I/Os and to allow DB2 to take advantage of prefetch mechanisms. Sorting data might not be possible in all cases, for instance, if rows have to be processed in a different order.

10.4.4 Parallelism exploitation

When DB2 accesses data from a table or index in a partitioned table space, it can initiate multiple parallel operations. The response time for data or processor-intensive queries can be significantly reduced. Queries can only take advantage of parallelism if you enable parallel processing. Before designing applications based on parallelism, talk to your system DBA.

DB2 checks for the possibility of using parallelism in the following situations:

- Sysplex parallelism

  Splits a large query across different DB2 members in a data sharing group.

- CPU parallelism

  Enables true multi-tasking within a query. A large query can be broken into multiple smaller queries. These smaller queries run simultaneously on multiple processors accessing data in parallel. This reduces the elapsed time for a query.

- I/O parallelism

  Manages concurrent I/O requests for a single query, fetching pages into the buffer pool in parallel. This processing can significantly improve the performance of I/O-bound queries.

If any one of the above is true, the chosen form of parallelism is used. If none of the above is true, the SQL statement is processed sequentially.

To enable parallel processing, you need to:

- Specify DEGREE (ANY) at BIND or REBIND for static SQL
- Set CURRENT DEGREE special register to ‘ANY’ for dynamic SQL

Before using these options, talk to your systems DBA. RLF can disable parallelism, and DSNZPARMs can enable/disable sysplex query parallelism

See *DB2 for z/OS Version 8 Administration Guide,* SC18-7413-03, for more information about enabling parallelism.

See *DB2 Data Sharing Planning and Administration,* SC18-7417-02, for more information about sysplex query parallelism.

10.4.5 Hot spots

Heavily updated portions of a table are called *hot spots*. The hot spots typically become a bottleneck for data throughput because DB2 has to serialize updates to those pages to ensure data correctness and consistency. From an application point of view, whenever you update a certain hot spot, make sure your application holds the requested resource for the shortest possible amount of time, issuing a COMMIT immediately after a hot spot is updated and your logical unit of work completes. You can find more information about hot spots in "Hot spots" on page 353.

10.4.6 Influencing the access path

If you cannot tune your queries by the methods we already have mentioned in this book, you can still influence the access path by using two further methods:

- Optimizer hints

  By using optimizer hints, you simply tell the optimizer which access path DB2 has to use for a certain query. We strongly recommend you talk to your system DBA before using optimizer hints.

- Manipulating statistics

  You can look at manipulating statistics as lying to DB2 and fooling the optimizer. Since optimizer hints have been introduced in DB2 for OS/390 Version 6, manipulating statistics is strongly discouraged.

- VOLATILE tables

  It favors the use of index for accessing a specific table.

10.4.7 Caching SQL results

As we have already mentioned in "Number of SQL calls" on page 191, instead of repeatedly accessing data, cache the results of your SQL statements if you will need them again. Even if you might think of asking DB2 more than once for the same data, and it might be most likely that the same data is still in the buffer pool, accessing main storage is faster than asking DB2 for qualifying rows since no physical I/O is required (if z/OS paging is not happening).

Considerations for caching SQL results are:

- If application cache is used, no roundtrip to DB2 is required.
- If data does not reside in the buffer pool any more, at least one synchronous I/O is required to access data again using SQL.
- In case of bad access-paths, the same overhead occurs twice for selecting data.
- Be extremely careful with caching of non-read-only data in a data sharing environment.

Acquiring main storage for caching SQL results is a smart solution, if:

- A central service module handles main storage access (reads, writes, and deletes).

- Different logical results can be cached using different names provided during run time.
- Main storage can be acquired dynamically.
- Data length and logical primary keys can be defined dynamically to access data in main storage.
- An internal index structure can be provided by the central service module.

Developing efficient central storage handling routines can be challenging, but it can also save you a lot of I/Os and elapsed time for your batch workloads.

For example, long-running batch jobs reading huge amounts of data can benefit from this solution if lookup-tables are not accessed using SELECT any longer, but they are stored and accessed in main storage after being retrieved once at batch jobs' start.

10.5 Batch process design

Typically, a batch cycle consists of a series of programs. In general, the programs must run at a particular time (daily, weekly, or monthly), in a certain sequence, and they must complete in a given time frame. This time frame is commonly called the *batch window*.

Several performance issues are associated with batch applications, and their importance and effect may vary in different environments. The most important performance factors from an operational point of view are:

- Elapsed time

  Many factors may influence the elapsed time of a batch application, starting from the I/O configuration and MVS system parameters and ending with DB2 parameters and application logic. It is mostly influenced by random I/Os on DB2 objects, such as tables and indexes. Usually, elapsed time is the main concern and the target is to achieve a total turnaround time for the batch cycle that may fit into the batch window. Use of parallel processes can reduce total elapsed time.

- Restart

  Long-running batch programs should contain checkpoint- and restart-routines.

- Concurrency

  If the batch window becomes too small, all batch programs should be designed to run both in batch or online environments.

- Commit frequency

  Choose the frequency of commits depending on service level agreements and type of accesses. Read-only programs need to commit to release the claims that prevent utilities from draining. Update processes need to allow concurrent read activity. Keep in mind that today a long-running batch job will grow in volumes and complexity. See "Error handling control table" on page 258 for a dynamic control on commit frequency.

A batch program can execute SQL statements under several attachments. The most commonly used attachments are:

- TSO
- IMS batch message processing (BMP)
- DB2 DL/I batch support
- Call Attach Facility (CAF)
- Recoverable Resource Services Attachment Facility (RRSAF)
- Mass processing by data propagation

Performance is pretty similar using each of the attachment facilities, so we do not make a distinction from a performance point of view among these attachments.

10.5.1 Maximizing throughput

You can influence both CPU and I/O performance when designing the application. Considering that the CPU is the most expensive resource in your system, its utilization must be carefully controlled. Having a bottleneck on the I/O devices and leaving the CPU idle is a misuse of resources. Therefore, batch applications should be made CPU-bound, as opposed to I/O-bound, to enable high utilization of all available resources. To reduce DB2 CPU consumption, batch applications should be designed in a way that:

- Benefits from index look-aside (see “Index look-aside” on page 426.)
- Minimizes number of SQL calls (see “Number of SQL calls” on page 191.)
- Minimizes the amount of data accessed (see “Number of rows searched” on page 192.)
- Minimizes the number of I/Os

A large number of data accesses usually require a large number of I/O operations. The cost of I/O operations directly affects elapsed time. DB2 includes mechanisms that help to reduce the time for I/O operations basically by means of prefetch. By operating asynchronously to the originating application program and reading several pages in a single I/O operation, sequential prefetch is one of the major DB2 functions that can be used in batch executions. To reduce the time spent for I/O operations, application programs should be designed so that:

- Most I/O operations are asynchronous
- More useful data is retrieved per I/O (group data accessed by the process)
- The number of I/O operations is reduced to avoid revisiting the same data

When dealing with large batch processes, other considerations that can influence the design are:

Lookup tables

If you refer to lookup-tables in your batch processes, reading the look-up table into main storage can dramatically reduce the number of I/Os in accessing the look-up table every time you need it. Access main storage for retrieving the values instead.

Restart using sequential input and output files

An important factor to consider when designing a time-consuming batch application is the time required for the backout and restart operation in case of abnormal termination. The application should be composed of many units of work. COMMIT should be performed frequently with respect to both application logic and performance. Checkpoint interval is important in determining the time for restart as well.

Using commits, however, requires special application logic to handle cursor repositioning and restart situations including file repositioning to the last commit point. When designing your batch application, make sure you try to access data in clustering order, so you can be sure to benefit from sequential prefetch algorithms from proper indexing. An example is repositioning your cursor during restart. For repositioning, save the last key before committing your changes just like you do on all needed variables. You can externalize those values in a DB2 table or using external files. See Figure 10-4 on page 284 for a brief overview of a restart implementation.

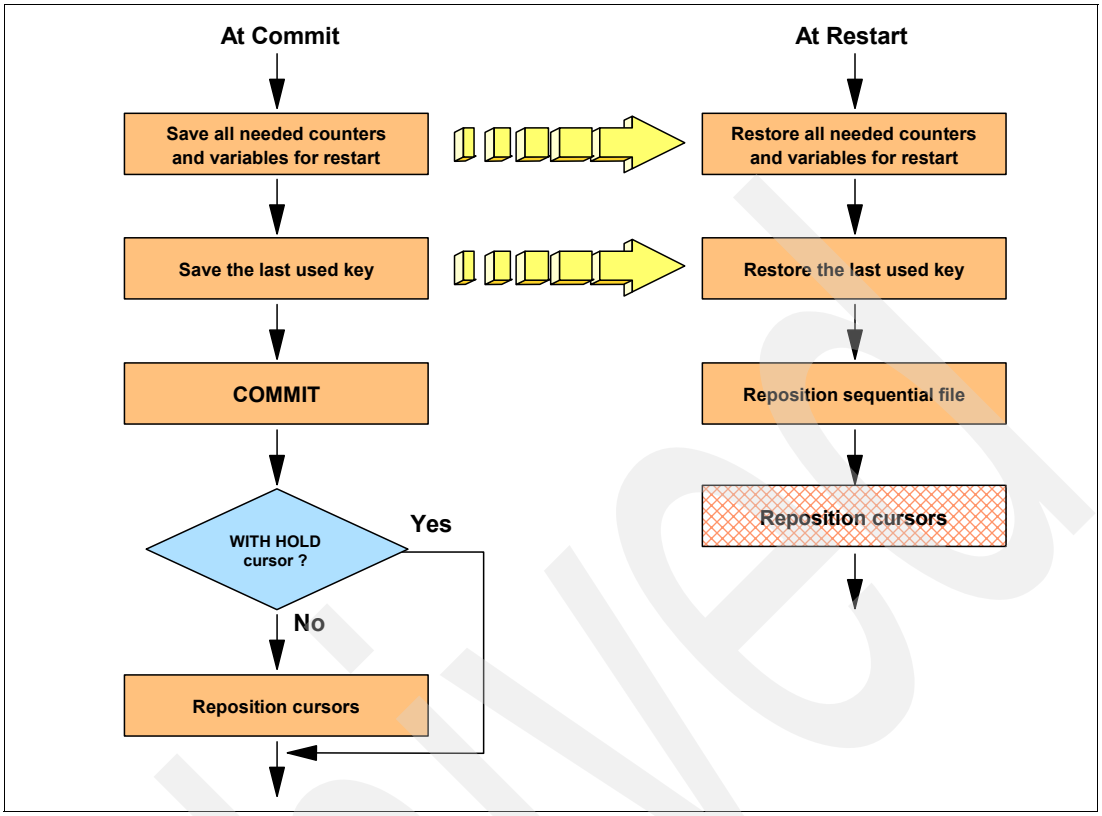


Figure 10-4 Program logic for commit and restart

We provide checkpoint and restart considerations concerning sequential input/output files. Different types of sequential files are commonly used as input/output files in DB2 batch programs. The common file types we use are:

- QSAM files are the most commonly used sequential files. QSAM manages all aspects of I/O buffering automatically. This is convenient for application development, but implies more logic in case of restarts.
- BSAM files require the program to manage its own input and output buffers. It must pass a buffer address to the READ macro and fill its own output buffer before issuing a WRITE macro. BSAM is usually used for specific requirements, and may take advantage of the NOTE and POINT macro statements for repositioning. BSAM is more complex to handle than QSAM.
- GSAM files are part of IMS and therefore require the existence of IMS/DB. They may be used in IMS batch and BMP jobs. Because of the two-phase commit protocol implemented between DB2 and IMS, the commitment of updates is synchronized across the two subsystems. Also, running the job under BMP batch allows the use of the IMS checkpoint and restart facility, in which case GSAM files can be automatically repositioned at restart time. There are special considerations for repositioning GSAM files.
- Recoverable VSAM using RRS (recoverable resource services) manages backouts on behalf of DB2 and IMS. RRS uses a two-phase commit mechanism to ensure that all updates to all resources are committed.

When dealing with sequential files, techniques must be implemented to:

- Control the buffer management of output files and synchronize it with DB2 commit phases
- Reposition the input/output files when restarting
- Handle potential duplicate output records as a result of restart

Figure 10-5 illustrates the need of repositioning sequential files unaffected by COMMIT and ROLLBACK.

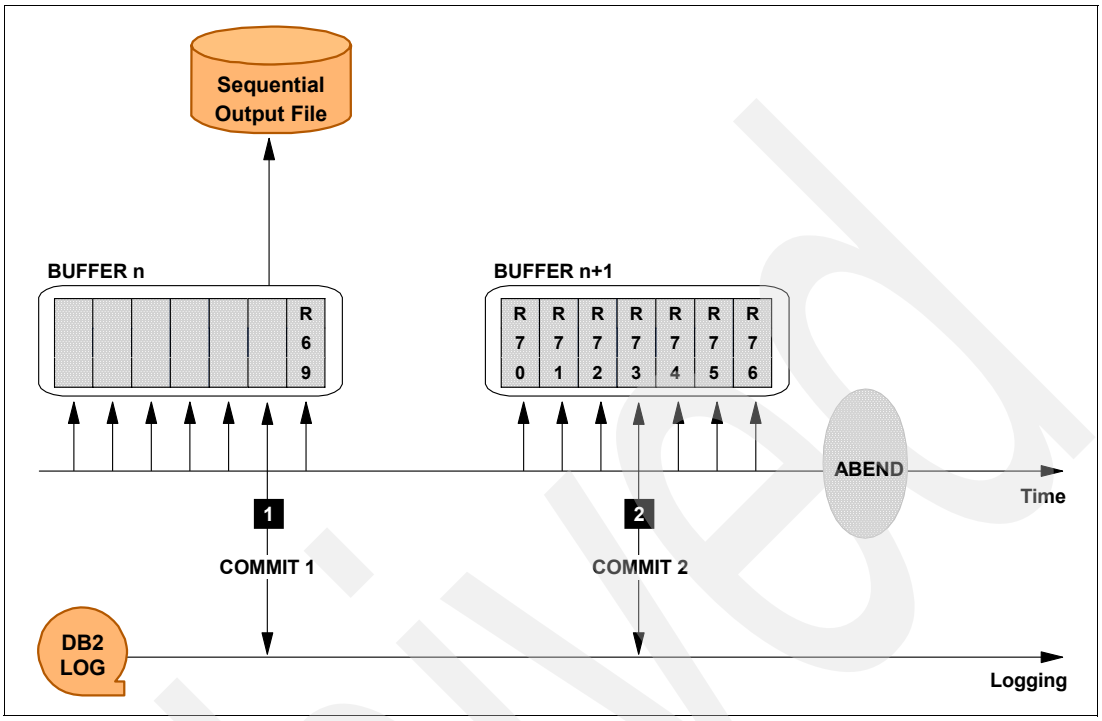


Figure 10-5 Sequential output files

The application program issues two COMMIT statements and writes sequential data to QSAM files. All DB2 changes up to these points in time are externalized to the DB2 log. The QSAM buffers are not externalized at the same point in time, depending on your buffer management. For this instance, it is most likely in an abend scenario that your QSAM data does not correspond with your DB2 data. Therefore, QSAM files can contain data which is associated to DB2 rows affected by ROLLBACK. Or the QSAM file might not contain all data up to the last commit point if an abend has occurred directly after a COMMIT.

Besides closing and reopening the data set (which consumes a lot of elapsed time for the unlikely event of an abend, decreasing your performance dramatically) to ensure externalization of your allocated QSAM buffers, you can think of ESTAE (Extended Specify Task Abnormal Exit) routines to clean up your environment (that is, closing files to force buffer externalization) in case of an abnormal termination.

If your application abends with abend code B37 (not enough disk space available), you can lose your output buffers, regardless of an established ESTAE routine.

You will need to talk to your z/OS systems programmer to implement ESTAE routines.

Instead of repositioning on your output data sets, you can consider using GDG (generation data group) data sets. At each restart, you create a new generation of the GDG. When you are concerned about eliminating duplicate records in sequential data sets as a result of a restart, the elimination of those records can be postponed until the end of all application phases by running an additional DFSORT for suppressing records with duplicate control fields, therefore, involving all created generations. Example 10-13 on page 286 shows the required DFSORT statement.

Example 10-13 Eliminating duplicate records using DFSORT

```
SORT FIELDS=(1,3,CH,A)
SUM FIELDS=NONE
```

Restart using DB2 tables for input and output files

Since sequential file handling during restart operations can be challenging, consider using the following technique to help guarantee data consistency:

- LOAD your input files into DB2 table A without indexes
- Read your input data from DB2 table A using a FOR UPDATE OF cursor
- Write your output records to DB2 table B
- Delete the rows you have already processed from table A using WHERE CURRENT OF CURSOR

In case of a restart, your cursor can simply reposition on table A since only nonprocessed rows are in the table.

There will be an additional overhead of doing INSERTS to a DB2 table, instead of writing to a sequential file, but the special checkpoint and restart considerations for sequential files will be eliminated. To reduce overhead for roundtrips to DB2 for each insert statement, consider using multi-row insert if your commit frequency contains a reasonable number of output records. Often an application does one write to a sequential file after having executed several SQL statements. The overhead of the INSERTs may then be reduced.

Cursor repositioning

Assume you have a table containing columns COLa to COLf and a unique index on columns COLa, COLb, and COLc. Furthermore, assume your application has to perform a restart on saved values for indexed columns COLa, COLb, and COLc. The most common approach to code a restartable cursor is the one shown in Example 10-14.

Example 10-14 Cursor repositioning: A common approach

```
DECLARE C1 CURSOR WITH HOLD FOR
  SELECT   COLd, COLe, COLf FROM T1
   WHERE   COLa >= :HV-COLa
     AND ((COLa  = :HV-COLa
     AND   COLb  = :HV-COLb
     AND   COLc  > :HV-COLc)
      OR  (COLa  = :HV-COLa
     AND   COLb  > :HV-COLb)
      OR  (COLa  > :HV-COLa))
```

The parentheses are only used to visualize how the predicates are grouped. The recommendation for cursor repositioning in batch is similar to repositioning in the browsing section, see Table 9-1 on page 225 for the relevant values for a cursor to reposition. Since each row in the table is part of the WHERE clause, as you can see in Example 10-14, the number of MATCHCOLS is different for each OR predicate.

For each OR predicate, DB2 has to perform index-screening, resulting in additional getpages you may want to avoid. To avoid these additional getpages and the amount of time used for repositioning the cursor, consider using one cursor for each column of the unique index used for repositioning. See Example 10-15 on page 287.

Example 10-15 Cursor repositioning: Minimizing getpages

```
DECLARE C1 CURSOR FOR
  SELECT   COLc, COLd, COLe
    FROM T1
   WHERE COLa = :HV-COLa
     AND COLb = :HV-COLb
     AND COLc > :HV-COLc

DECLARE C2 CURSOR FOR
  SELECT   COLb, COLc, COLd, COLe
    FROM T1
   WHERE COLa = :HV-COLa
     AND COLb >:HV-COLb

DECLARE C3 CURSOR FOR
  SELECT   COLa, COLb, COLc, COLd, COLe
    FROM T1
   WHERE COLa >:HV-COLa
```

EXPLAIN indicates a matching index scan for all three cursors with, respectively, a value for MATCHCOLS of 3, 2, and 1. This seems to be a problem, but it is not, because cursors C2 and C3 are opened when the first qualifying row is the first row encountered in the index.

Simultaneous batch and online processing

Batch windows are defined as the time when batch jobs are scheduled and no online transactions are executed. But how do you explain your Web shop's downtime to your customers for four hours during peak times at night because of a "batch window"? Looking at continuous operations and high availability batch windows are disappearing because of business needs.

Therefore, batch processes have to be designed in a way to allow them to run concurrently with online transactions. This is a summary of factors to consider:

- Reduce the number of locks on data pages.
  - Use page or even row level locking for all affected table spaces.
  - Commit frequently in your batch applications.
- Batch processes should support checkpoint and restart functionality.
- You should be able to adjust commit intervals easily and dynamically.
- Prefer specifying commit intervals in units of time, not as a number of records processed.
- Each COMMIT statement consumes resources.
- Checkpoint interval becomes an important design factor for long-running batch programs, that is, any program that runs longer than the expected response time for a transaction.
- Logical backout using point-in-time recovery is not possible, because you can lose changes performed by online transactions.

Archived

Archived

11

Infrastructure topics

In this chapter, we describe functions that are independent of the application code, but are part of the infrastructure or environment where the application resides. The settings of these functions can impact the performance and concurrency characteristics of the application.

This chapter contains the following topics:

- Database BIND options
- Thread reuse
- Blocking in a distributed environment
- Plan and package authorization
- Static SQL, dynamic SQL, and reoptimization
- CICS Open Transaction Environment
- Rebinding plans and packages
- The role of catalog statistics
- System parameters
- Considerations when using Java to access DB2

© Copyright IBM Corp. 2005. All rights reserved.

11.1 Database BIND options

In this section, we look at the BIND options that have an impact on performance. Most of them change the behavior of the application's performance or concurrency characteristics, but are transparent to the application code itself. Others require that you also code your applications properly to get the most benefit out of using the option.

11.1.1 CURRENTDATA

CURRENTDATA is an important BIND option. Unfortunately, it serves a dual purpose. It is related to lock avoidance (in a local and remote environment), and it allows for blocking of ambiguous cursors in a distributed environment.

Tip: CURRENTDATA(NO) serves two purposes:

- It promotes lock avoidance, but only for applications using isolation level cursor stability, ISOLATION(CS).
- It also promotes blocking, irrespective of the isolation level.

For a detailed description about the impact of CURRENTDATA on lock avoidance, see "ISOLATION levels" on page 336.

An ambiguous cursor is a cursor that is in a plan or package that contains either PREPARE or EXECUTE IMMEDIATE SQL statements, and for which the following statements are true:

- The cursor is not defined with the FOR READ ONLY clause or the FOR UPDATE OF clause.
- The cursor is not defined on a read-only result table.
- The cursor is not the target of a WHERE CURRENT clause on an SQL UPDATE or DELETE statement.

When your application does not require data currency, using CURRENTDATA(NO) enables the use of block fetch for ambiguous cursors. Before using this option, think about the business implication, but most applications should be able to cope with CURRENTDATA(NO) data. The default value for CURRENTDATA is YES. Block fetching can provide a significant performance improvement; instead of sending a single row per network message, DB2 will send as many rows as will fit in a query block. The default query block size is 32 KB, but the size can go up to a maximum of 2 MB.

Recommendation: Use DRDA blocking as much as possible, either by using a read-only cursor, or by specifying CURRENTDATA(NO) when using an ambiguous cursor.

In case your application cannot take advantage of DRDA blocking, you can try using multi-row FETCH to achieve the same reduction in network messages that DRDA blocking provides. See "Multi-row operations" on page 301 for more details.

11.1.2 ACQUIRE

The acquire parameter indicates when resources for DBRMs specified in the MEMBER list during the BIND PLAN command will be acquired. The two options that you can use are:

USE Acquires table space locks only when the application program first uses them.

ALLOCATE Acquires all table space locks and loads all DBDs referenced in the plan, when the plan is allocated. The value has no effect on dynamic SQL statements, which always use ACQUIRE(USE). If you use ACQUIRE(ALLOCATE), you must also use RELEASE(DEALLOCATE).

Packages, local or remote, associated with the plan always acquire their resources when the application first accesses them; packages always use ACQUIRE(USE).

ACQUIRE(ALLOCATE) can increase the plan size, because additional items become resident in the plan.

Because packages always use ACQUIRE(USE), and most applications use packages, and ACQUIRE(USE) is usually the best choice for concurrency, the use of ACQUIRE(ALLOCATE) is extremely limited.

11.1.3 RELEASE

The RELEASE parameter determines when “certain” DB2 resources are released or reset.

The things impacted by the RELEASE parameter are:

- When table space level locks are released
- When plans and packages are released from the EDM pool
- When storage acquired by the thread is released
- When sequential detection information is reset
- When index look-aside information is reset
- When xPROCs are destroyed

There are two values that you can specify for the RELEASE parameter:

COMMIT Release resources when the program commits

DEALLOCATE Release resources when the program ends

By looking at the list of resources that are affected, it should be obvious that the use of RELEASE(DEALLOCATE) is certainly not a free lunch.

Using RELEASE(DEALLOCATE) tells DB2 that you want to keep the resources around so they do not need to be reacquired after a commit. This is good if you have a long-running batch job, or want to get the best performance out of thread reuse in an online environment.

The advantages of the use of RELEASE(DEALLOCATE) are:

- It reduces CPU impact for simple transactions. Acquiring the locks and PTs may represent a large portion of the CPU cost of a simple transaction. Saving this may mean a significant performance boost for simple, short running transactions.
- It reduces table space (TS) lock activity.
- It reduces the number of TS locks propagated to the CF in a data sharing environment.
- It reduces XES and false global lock contention (IS, IX-locks) when not using the locking protocol 2 that was introduced in Version 8.
- For batch with many commits, RELEASE(DEALLOCATE) avoids resetting sequential detection, index look-aside, and xPROCs at commit.

Archived

However, the downsides of using RELEASE(DEALLOCATE) should be looked at as well, and in many big installations, they often outweigh the benefits. The disadvantages of using RELEASE(DEALLOCATE) are:

- Virtual storage capacity
  These threads accumulate ever more storage for statements that are not being used. The storage for unused statements can be left around until deallocation time. For these threads, storage contraction is not effective (nor will this be the case when full system storage contraction needs to be performed). The storage impact of RELEASE(DEALLOCATE) is often an underestimated problem that can have severe consequences.
- The plans and packages that are bound with RELEASE(DEALLOCATE) stay around much longer and will cause growth in the EDM Pool's storage consumption. This is especially problematic in a CICS environment with heavy thread reuse when using a single plan.
- The chances of getting into resource contention increase. Because RELEASE(DEALLOCATE) only releases table space locks at deallocation time for static SQL, and dynamic SQL when full caching is active, you may receive timeouts when:
  - (Re)binding plans and packages
  - Executing DDL against any of the objects referenced by any of those plans and packages
  - Trying to perform a mass delete on segmented table space
  - Lock escalation has occurred
  - Executing an SQL LOCK TABLE statement

Recommendation: Use RELEASE(COMMIT), unless you really need the (little bit of) extra performance RELEASE(DEALLOCATE) can provide, and your system can tolerate the side effects of the use of RELEASE(DEALLOCATE) discussed above.

See also "Thread reuse" on page 296 for recommendations for using RELEASE(DEALLOCATE) and thread reuse together.

A measurement done by the DB2 performance department in the IBM Silicon Valley Laboratory comparing workload using RELEASE(DEALLOCATE) versus RELEASE(COMMIT) using the DB2 Version 8 data sharing environment showed only a 5% Internal Throughput Rate (ITR) and 7% CPU time difference. The gap used to be much wider in Version 7, 17% for both ITR and CPU time, mainly due to the extra XES contention in the old locking protocol (protocol level 1). This shows that you must be very close to some capacity limit that is difficult to remove to be able to justify the "inconveniences" caused by the use of RELEASE(DEALLOCATE).

It is worth noting that the RELEASE attribute is ignored in a distributed environment where DB2 for z/OS work is done on behalf of a DRDA client. In a distributed environment, RELEASE(COMMIT) is always used. Also note that the RELEASE parameter does not apply to page and row locks. In almost all circumstances, they are released at commit time.

11.1.4 ISOLATION

The parameter is only mentioned here in an attempt to list all performance-related bind options. For a full description of the ISOLATION bind parameter, see "ISOLATION levels" on page 336.

Note that you can override the plan's or package's ISOLATION level by specifying the isolation level at the individual statement level using the WITH clause.

Also note that if the package specifies ISO(UR) and you do an INSERT, that DB2 will "promote" the isolation level so it is appropriate to process the statement according to its semantics.

11.1.5 VALIDATE

We recommend the use of the VALIDATE(BIND) because it saves the authorization check at execution time. This is especially important if the plan or package is executed very often, because each time the plan or package is executed, DB2 will verify whether the object and/or authorization is now in place.

The only "valid" reason to use VALIDATE(RUN) is when a certain object does not yet exist, but you already have to create the plan that contains DBRMs. BIND PLAN using VALIDATE(BIND) results in a BIND error if the object does not exist. When using BIND PACKAGE, you can use VALIDATE(BIND) SQLERROR(CONTINUE) to get around the problem. The SQLERROR option only exists on the BIND PACKAGE command.

Note that the default value for the BIND PLAN and BIND PACKAGE command is VALIDATE(RUN). This reduces the chances of BIND errors, but may have a performance impact if DB2 needs to do checking at run time, because the necessary objects or authorizations were not in place at BIND time.

We highly recommend that you rebind the plan or package as soon as the objects and authorizations are in place.

11.1.6 DEFER(PREPARE)

In a DRDA environment, when DEFER(PREPARE) is used to bind the plan or package, it reduces network traffic when applications perform PREPARE statement. When DEFER(PREPARE) is used, the PREPARE statement does not trigger network traffic to the database server. DB2 will hold the PREPARE and wait for the corresponding EXECUTE, DESCRIBE, or OPEN call. PREPARE and EXECUTE/DESCRIBE/OPEN are sent on one network message, and the messages are passed back in one message, instead of two messages.

All messages in relation to the PREPARE statement for dynamic SQL statements that refer to a remote object will be deferred until either:

- The statement executes; that is, the program issues an EXECUTE statement.
- The program asks for a metadata description of the result of the statement, for example, using DESCRIBE. The metadata description is the column attributes, such as data type and length. Because this information only exists on the server, a message had to be sent to the server to get this information.
- Note that the PREPAREd statement text with an INTO clause is not deferred.

As a consequence, when using DEFER(PREPARE), DB2 does not immediately flow the PREPARE when it is encountered in the program, and the application will receive an SQLCODE 0, even though the statement may contain invalid text or objects. It is not validated until the EXECUTE or DESCRIBE statement is executed by the application. This means that the application program has to handle SQL codes that are normally returned at PREPARE time, but can now occur at EXECUTE/DESCRIBE/OPEN time.

When using REOPT(VARS), DB2 automatically enables the DEFER(PREPARE).

In DB2 ODBC, the configuration parameter DEFERREDPREPARE defaults to 1 to enable the defer option.

11.1.7 ENCODING

The ENCODING bind option specifies the application encoding scheme (EBCDIC, ASCII, or UNICODE) or the CCSID for all host variable static statements in the plan or package. The CURRENT APPLICATION ENCODING SCHEME special register does the same for dynamic SQL statements.

When the application uses a different encoding scheme than the data, you can use this option to convert the data that is retrieved from the DB2 tables' encoding scheme into the correct encoding scheme that is expected by the application. Although the conversion is done very efficiently, it can still have a performance impact if a large number of rows need to be returned to the application.

From a performance perspective, it is best to store the data in the same encoding scheme as the applications that use the data. However, in many situations that is impossible, especially in today's global economy where data may be stored in Unicode, but is manipulated by applications that are running in Unicode, ASCII, and EBCDIC. See "Choosing an encoding scheme" on page 23.

11.1.8 DEGREE

The DEGREE keyword on the BIND plan or package indicates whether or not you want the optimizer to consider running the static SQL in this plan or package for parallel execution.

1 No parallelism will be attempted.

ANY DB2 will allow for parallel processing for static SQL.

The DEGREE BIND parameter only affects static SQL. Whether or not dynamic SQL statements are eligible for parallel processing is determined by the CURRENT DEGREE special register. Applications can change the CURRENT DEGREE special register by using the SET CURRENT DEGREE statement. Initially, the CURRENT DEGREE special register is set to the CDSSRDEF DSNZPARM

We recommend you set CDSSRDEF to 1. You should use parallelism selectively where it provides value, rather than globally.

An additional DSNZPARM PARAMDEG can be used to limit the maximum degree of parallelism for a parallel group in the system. By specifying a value for this DSNZPARM, you limit the degree of parallelism, so that DB2 cannot create too many parallel tasks that use virtual storage. The default value of 0 means that DB2 will choose a maximum degree of parallelism that is based on the system configuration, which may be higher than the system can afford.

Another option that has an impact on query parallelism is the Resource Limit Facility (RLF). RLF can be used to disable certain types of parallelism for dynamic SQL. The following RLFFUNC values in the RLST are related to query parallelism:

'**3**' The row disables query I/O parallelism.
'**4**' The row disables query CP parallelism.
'**5**' The row disables Sysplex query parallelism.

Recommendation: Do not bind all plans and packages with DEGREE(ANY) or set CDSSRDEF DSNZPARM to ANY, but rather do it at the individual plan, package, or statement level. In addition, use the PARAMDEG DSNZPARM to put a cap to the maximum parallel degree, so the system does not get into trouble.

11.1.9 Handling collections

DB2 packages are grouped in collections. Collections have no physical counterpart; they are just a way to logically group packages together. The handling of collections is different for local or remote applications. When running locally, each application has an associated plan. In the BIND PLAN command, you can specify the PKLIST keyword to indicate the list of packages and collections you want DB2 to search. When running remotely, all applications use the DISTSERV plan. DISTSERV is an internal plan that is used to execute remote packages. You cannot specify a PKLIST for the DISTSERV plan.

PKLIST

For a local application (and for applications where the DRDA application requester is an DB2 for z/OS system), the PKLIST determines the collections that DB2 will look into for a matching package and consistency token. DB2 goes through the entries in the PKLIST sequentially. Therefore, the more collections are present in the PKLIST, the more times DB2 may need to find the correct package. When the PKLIST entry contains a " * ", like " MYCOLL.* ", DB2 will do a lookup into SPT01 to see if a matching package exists. Therefore, from a performance standpoint:

- Reduce the number of coll-id* collections in the PKLIST. It will reduce the number of probes that DB2 has to perform to find a package with a matching server-collection-package-consistency_token.
- Put the collections that are most likely to contain the package first in the PKLIST.

Whether or not this is feasible will be to a great extent determined by the way your application development/production environment is structured.

We recommend that first of all, your collection setup matches your application/production environment. Their setup tends to dictate the collection setup more than performance considerations. However, do not get carried away by having too many collections using an " * " in the PKLIST.

SET CURRENT PACKAGE SET

When the CURRENT PACKAGE SET special register is set, it will instruct DB2 to search into that collection indicated by the value of the special register. For example, if the CURRENT PACKAGE SET special register is set to "COLL1", DB2 will only look into the COLL1 collection for a matching package, even if the PKLIST specifies PKLIST(*.*) — look in all collections.

If, in a distributed environment, an application needs to be able to search multiple collections, for example, UNITTEST, SYSTEST, and PROD, you can use SET CURRENT PACKAGE SET = 'UNITTEST' and execute the SQL statement. If an SQLCODE -805 (package not found) is returned, you can issue another SET CURRENT PACKAGE SET = 'SYSTEST' statement and try to execute the SQL statement. If you receive another SQLCODE -805, you can try with the last collection by issuing SET CURRENT PACKAGE SET = 'PROD' and execute the SQL statement a last time. If it still fails, something is wrong, and you can terminate the application.

SET CURRENT PACKAGE PATH

Note that in the scenario above, we issued the SET CURRENT PACKAGE SET statement multiple times, and each time, since this is a distributed application, we make a trip across the network to the DB2 server to find a matching package. If you need to search many collections, this can be an expensive operation. In addition, you need to have extra coding in the application to handle the SQLCODE -805 and set the special register again and retry.

You can also use the SET CURRENT PACKAGE PATH special register to pass a list of collections in which the server can search for a matching package. So, instead of potentially issuing three SET CURRENT PACKAGE SET statements, you can also issue:

```
SET CURRENT PACKAGE PATH = 'UNITTEST,SYSTEST,PROD';
```

and invoke the SQL statement. This will potentially save two trips over the network.

We recommend the use of SET CURRENT PACKAGE PATH over the use of SET CURRENT PACKAGE SET, because it can save network overhead and allows for more flexibility.

11.2 Thread reuse

For simple transactions, the CPU cost of creating a thread is an important part of the total cost of the transaction. When these transactions are used frequently, an effort can be made to minimize the thread creation cost. DB2 can reuse an existing thread, by performing just the sign-on, and skipping the create thread.

Each time a transaction reuses a thread, the cost of creating a thread is avoided. So the design objective should be to achieve as much thread reuse as possible without distorting the application structure and the operational procedures for the BIND process.

Thread reuse is often used together with the RELEASE(DEALLOCATE) BIND option. See "RELEASE" on page 291 for details. However, you do not have to always use both.

Archived

Recommendations: When considering thread reuse and RELEASE(DEALLOCATE), use the following guidelines:

1. First ask yourself, do we really need this extra performance boost?
   It is best to reserve thread reuse and the use of RELEASE(DEALLOCATE) for
   - High volume OLTP programs. A good rule of thumb is to only consider thread reuse and RELEASE(DEALLOCATE) for transactions that are executed more than once per second (on an average hour).
   - Batch programs that issue many commits
2. If so, consider implementing just thread reuse without using RELEASE(DEALLOCATE). This allows you to avoid thread creation and termination for each transaction, and spares you from experiencing some of the side effects from using RELEASE(DEALLOCATE), like the reduced availability, and increased EDM pool size that is required.
3. If you really need RELEASE(DEALLOCATE), as well as thread reuse, make sure that you:
   - Use packages, so you can specify the RELEASE parameter at the package level. If a lot of DBRMs are bound into a plan that is bound with RELEASE(DEALLOCATE), all objects in there will keep their locks and the big plan will stay allocated in the EDM pool until deallocation of the thread.
   - Use RELEASE(COMMIT) at the plan level. Do not use RELEASE(DEALLOCATE) for your plans when using packages, because it is used as a default value for the allocated packages if the RELEASE parameter was not explicitly specified at BIND PACKAGE time.
   - Use RELEASE(DEALLOCATE) only on those packages that are executed frequently and are performance sensitive. If a package that was bound with RELEASE(DEALLOCATE) is only executed once, it will still stay in the EDM pool and keep its table space locks until the thread deallocates. In this case, you only have the disadvantages of RELEASE(DEALLOCATE) and none of the advantages (since the package is only used once).

The conditions to reuse a thread are different for each transaction manager.

11.2.1 IMS message processing regions

Normally, in IMS, every time a new transaction is scheduled and an SQL call is executed, a thread is created. If in one IMS schedule, five transactions are executed, the DB2 thread is only created once (at first DB2 call), and reused four times.

Besides the multiple executions of a transaction per schedule, wait for input (WFI) transactions always reuse the thread, because there is only one schedule, since it is always the same program that is executing. However, each WFI transaction requires a dedicated processing region.

If you enable a message processing program (MPP) region to allow for so-called pseudo-WFI, IMS tries to reuse the thread by leaving the thread and existing program loaded until the next transaction arrives. If the transaction is the same as the one currently executing, IMS will reuse the thread automatically.

If you use pseudo-WFI, your accounting figures can look somewhat strange since a DB2 accounting record is written when a new DB2 transaction comes in (and does its DB2 sign-on). So the elapsed time when the region is idle or the region is used by a different

transaction not accessing DB2 (and the CPU time in the latter case) is accounted to the previous transaction.

Your application design should be a compromise between a higher thread reuse ratio and a modular design.

11.2.2 IMS/ESA fast path regions

All transactions processed by an IMS Fast Path (IFP) region are connected to the same DB2 plan. An IFP always reuses the thread. An IFP is equivalent to a WFI region when the thread is reused. However, note the differences between an IFP and a WFI:

- There are limitations for the input and output messages in IFPs.
- IFPs may process several transaction or routing codes.

IFPs are the right choice for processing simple and high-volume DB2 transactions. This approach has efficient message processing by the IFP Expedited Message Handler and a 100% thread reuse ratio in DB2, giving the lowest possible transaction CPU cost for a given DB2 access pattern.

11.2.3 Thread reuse in CICS

For a CICS-DB2 thread, there are several factors to consider to achieve adequate CPU consumption and therefore better performance.

A task with SQL requests can obtain a thread to connect to DB2 by two ways:

- Reuse an existing thread that another task used previously, or
- Allocate a new thread

The most efficient way for a transaction to execute an SQL statement is to reuse a thread rather than allocate a new one. A thread can be reused *only* if the DB2ENTRY associated with the transaction is the same as the previous transaction *and* the new transaction uses the same plan.

Also, to reuse a thread, the DB2 thread definition (DB2ENTRY or DB2CONN for pool entries) must have either:

- THREADWAIT=YES to allow the transaction to wait for a thread already in use, or
- PROTECTNUM > 0 (protected thread) to allow unused threads to wait for a transaction that might otherwise run immediately if allowed to create a new thread.

A thread becomes available for reuse after SYNCPOINTs for *terminal driven transactions* if the thread is in the initial state. Initial state means that all modifiable special registers are at their initial value and there are no held cursors.
For *non-terminal driven transactions* that specify NONTERMREL(NO), threads are only released at end of task (EOT).
When the option NONTERMREL(YES) is used on the DB2CONN definition, it causes non-terminal driven transactions to release their threads at SYNCPOINT, and they become subject to the same rules as terminal driven transactions.

The modifiable special registers include:

- CURRENT APPLICATION ENCODING SCHEME
- CURRENT DEGREE
- CURRENT LOCALE LC_CTYPE
- CURRENT OPTIMIZATION HINT
- CURRENT PACKAGESET

- CURRENT PATH
- CURRENT PRECISION
- CURRENT RULES
- CURRENT SERVER
- CURRENT SQLID

For a complete list and description, see Chapter 2. Language elements, section Special registers of the *DB2 UDB Version 8 SQL Reference*, SC18-7426-02.

Unprotected threads are terminated immediately upon thread release if no other task is waiting. This means that a task that releases its thread at an intermediate SYNCPOINT may have to go through thread creation multiple times within one transaction. Note that unprotected threads can be reused. It is a common fallacy that you must use a protected thread to get thread reuse. Therefore, pool threads can be reused, though less likely if many different plans are executing in the pool.

The advantages of reusing threads is more significant in short and high volume transactions. In a short transaction, the CPU savings of avoiding the creation of a new thread is significant, while they are less significant for long transactions. Thread deallocation is asynchronous and does not take as much CPU, so it is less of a consideration when tuning.

CICS sign-on

Though less significant than thread reuse, sign-on reuse may provide valuable performance gains for very short transactions (for instance, a single SQL statement executed in the task).

Sign-on always occurs when a thread is used for the first time. A thread is signed-on again when the thread is reused and any of the following occur:

- The primary authorization ID changes: The AUTHID or AUTHTYPE parameter value will greatly affect the frequency that the primary authorization changes:
  - Best choice is a value that is constant such as a AUTHID(string) or AUTHTYPE(SIGN). The worst choice is a value that may change frequently, such as AUTHTYPE(TERM) or AUTHTYPE(USERID). Note a thread can be reused but a resign-on will occur if the authorization values of the thread have to be changed.
  - Prior to CICS TS V1.2, note that three values on AUTH=(x1,x2,x3) are tried in order until a value is available for use as primary authorization id. The attachment does not use all three values as a collective authorization list.
- The first SQL statement after a SYNCPOINT is ACCOUNTREC(UOW). This provides the ability to correlate DB2 accounting with CICS accounting on a per UOW transaction basis. For transactions with multiple UOWs per transaction, multiple DB2 accounting records have to be correlated. This can be overcome by specifying ACCOUNTREC(TASK).
- The TXID changes on a DB2ENTRY used by multiple transactions when ACCOUNTREC(TXID) is specified. It can be avoided by using ACCOUNTREC(TASK) or ACCOUNTREC(NONE).
- The last transaction left an open held cursor or left a special register in a modified state.

However, keep in mind that if no accounting records are produced, it may complicate performance analysis, because multiple executions of transactions are lumped together in the same DB2 accounting record.

If you do not care about DB2 security and set grant access to PUBLIC, you should also specify CACHESIZE(0) for the BIND PLAN.

Avoiding a sign-on wherever possible will improve the performance of transactions within a DB2ENTRY.

When a thread has open held cursors or modified special registers, the thread cannot be reused at intermediate SYNCPOINTS (before end of task (EOT)). A partial sign-on will occur at EOT to restore the thread to the initial state. That is why it is a good practice to close all cursors, especially the ones with the WITH HOLD option, when they are no longer needed, to release locks and make the thread reusable. The extra effort it takes to restore the special register to the initial value may well be worth the performance increase achieved by making the thread reusable.

Also, there are certain considerations regarding the use of plan and packages:

- Use a single large plan with many packages rather than using Dynamic Plan Switching (DSP), PLANEXITNAME(*nnn*) in DB2 ENTRY to switch between many smaller plans. Using DSP reduces the possibility of reusing an existing thread. Also, a transaction can only switch plans when the thread is reusable.
- Package definitions should also be tuned. List the most frequently used packages first and use wildcarding where possible. A large collection does not present a significant performance impact.

11.2.4 Thread reuse in a distributed environment

Thread reuse in a distributed environment is related to the setting of the specification of the CMTSTAT DSNZPARM. We recommend you set its value to INACTIVE. This allows the connection to become inactive (it stays around in the DDF address space, waiting for a new request to come in over this connection) and the DBAT (that is doing the actual work in DB2 on behalf of the remote application in the DBM1 address space) is put back into a pool, so it can be reused by another connection. Therefore, technically speaking, the correct term in a DDF environment is *DB2 thread pooling*.

Each time a DDF thread commits, with CMTSTAT=INACTIVE, DB2 will try to separate the connection from the DBAT, and put the DBAT back into the pool so it can be reused by any other incoming connection request. DBATs can be reused by any incoming connection, regardless of things, such as inbound location and user ID. The only factors that prevent a thread being returned to the pool at commit time for reuse are programming constructs that maintain “context” after a commit statement. There are four situations to be aware of:

- If the application has any open WITH HOLD cursors, the thread has to hold the cursor open.
- If the application has any DECLAREd global temporary tables that have not been explicitly dropped at commit time, regardless of the ON COMMIT PRESERVE ROWS or ON COMMIT DELETE ROWS options, the thread has to keep the table in memory for further use.
- If the application has touched any package that has been bound with KEEPDYNAMIC YES, the thread has to keep the prepared SQL statement in memory for reuse.
- If the application uses a held LOB locator.

With DRDA connections, if any of these situations exists, the active thread cannot be pooled (cannot become an inactive thread). It is cleansed of storage after commit, but it is held exclusively for the use of the application connection that has used it.

Therefore, to reduce system resources and reuse DBATs as much as possible, make sure you use the features described above judiciously. Remember that the maximum number of DBATs (MAXDBAT) in a subsystem is 1,999.

The most likely reason that can prevent successful DB2 thread pooling is the use of WITH HOLD cursors, whether intended or not. In client/server, network computing environments,

applications, based on the ODBC API (ODBC, CLI, ADO, Visual Basic®, OLE DB, and .NET) and the CLI-based JDBC driver (not the new Universal Driver), pick up the CLI configuration settings from the DB2CLI.INI file. One of the CLI configuration settings is a parameter called CURSORHOLD. The default value of CURSORHOLD is 1, which causes cursors to be held over commit points, and prevents the DB2 threads from being reused until the application disconnects from the database.

Recommendation: In most cases, you should change DB2CLI.INI to explicitly specify CURSORHOLD=0. If you want to use a WITH HOLD cursor, it would be better to specify it explicitly on the DECLARE CURSOR statement within the application program (overriding the DB2CLI.INI default).

A parameter to consider in the DB2CLI.INI file is AUTOCOMMIT, where the default value is 1. While this is good in the context of DB2 thread pooling, it is not the right choice for transactional systems, where the application wants to control the unit of work boundaries, and you want to save the extra network messages caused by flowing a commit after every SQL statement. On the other hand, having AUTOCOMMIT=1 is good for environments that use query tools. Autocommit forces them to release claims on the DB2 for z/OS system, as well as allowing the connection to become inactive.

Recommendation: In most cases, the safest approach is that transactional systems explicitly change AUTOCOMMIT to 0 in the application. However, make sure the application commits regularly, otherwise thread pooling will not be used.

Another solution is to have a database entry in the CLI.INI file that query tools use that has AUTOCOMMIT=1, and another entry that transactional systems use that specifies AUTOCOMMIT=0.

11.3 Blocking in a distributed environment

In a distributed environment, it is crucial to reduce the number of trips over the network. Network speed has increased dramatically over time, and it is an extremely variable component when compared to CPU speed. There are several ways to optimize network traffic in a distributed environment.

DRDA blocking

DRDA Block fetching can provide a significant performance improvement; instead of sending a single row per network message, DB2 will send as many rows as can fit in a query block. The default query block size is 32 KB, although it can be larger, up to a maximum of 2 MB. Although the maximum is now 2 MB, DB2 for z/OS as well as DB2 Connect only use 32 KB query blocks. DB2 Connect supports query blocks up to 64 KB. You specify the DRDA query block size in DB2 Connect using the RQRIOBLK parameter.

Recommendation: Use DRDA blocking as much as possible, either by using a read-only cursor, or by specifying CURRENTDATA(NO) when using an ambiguous cursor.

Blocking can mean a dramatic performance boost.

Multi-row operations

In cases where DRDA blocking is not used, for example, when doing an INSERT into a remote table, when using an updatable cursor, or an ambiguous cursor together with CURRENTDATA(YES), a single row will flow across the network instead of a block of rows.

To get around this restriction, you can use multi-row operations.

When using a multi-row operation, a multi-row fetch, or a multi-row insert, DB2 always retrieves a rowset in a single trip across the API, or across the network in a distributed environment. The maximum rowset size is 32 KB, up to a maximum of 10 MB when the rowset has to flow over a network. Therefore, by using inserting a rowset instead of a single row, you can simulate DRDA blocking during insert. All rows in the rowset will be sent across the network in a single network message. The same applies for multi-row fetch operations. Even for an updatable multi-row cursor, DB2 will fetch the entire rowset across the network in a single message.

Using multi-row insert may open a number of interesting options, for example, when doing replication. Most of today's replication implementations tend to use a pull scenario. Assume that you need to propagate to multiple DB2 for z/OS servers for Business Intelligence (BI) purposes using Data Propagator. You get good performance if you use a pull scenario today. In this scenario, the target servers pull the data from the server to replicate it. The main reason for using a pull scenario is that it allows Data Propagator to use DRDA block fetch. With DB2 V8, we can use multi-row operations in DB2, so you can also consider using a push scenario. Data Propagator can push out the data to the other remote servers, using a remote multi-row insert. This way the DB2 source server is in control.

DB2 always sends a rowset over the network for a multi-row operation (up to a maximum of 32 KB rows or 10 MB). The number of rows indicated by the rowset size that needs to be inserted will be sent over the network as a single message.

The same applies to ambiguous cursor in plans and packages bound with CURRENTDATA(YES) or updatable cursors. They can also "simulate" DRDA blocking by using a rowset cursor. As with multi-row insert, a single rowset is sent in a single network message.

OPTIMIZE FOR *n* ROWS

DB2 for z/OS uses the OPTIMIZE FOR *n* ROWS clause for two purposes:

- The first purpose is to influence the DB2 optimizer in the selection of its access path. When OPTIMIZE FOR *n* ROWS is specified, this tells DB2 optimizer that the result set for the query will be *n* rows. By giving this value, the optimizer can choose the best access path to retrieve *n* rows.
- The second purpose is to control the way *DRDA* query blocks are sent on the network. The default query block size is 32 KB.

OPTIMIZE FOR many ROWS

When you specify a large value for *n* on the OPTIMZE FOR *n* ROWS clause in a DRDA environment, this triggers DRDA continuous block fetch, a sort of prefetch in a distributed environment.

OPTIMIZE FOR few ROWS

When you only want a few rows given back from the answer set, you can specify a small value for *n*. When blocking can be done in a DRDA environment, a small value limits the number of rows in a query block being transmitted. For example, if a query block can hold 20 rows, but you specify OTPIMIZE FOR 10 ROWS, then 10 rows are put in a query block and sent across the network. You might want to do that, for instance, if only 10 rows will fit on a screen. Note that OPTIMIZE for 1, 2, or 3 ROWS always sends 16 rows.

11.4 Plan and package authorization

In a performance-oriented design, you should minimize the cost of authorization checking. We describe the functions which can reduce this cost.

11.4.1 DB2 caching authorization IDs

You can tell DB2 to cache authorization IDs for plans and packages. Caching authorization IDs can greatly improve performance, especially when a limited number of IDs are reused frequently. There is an individual cache for each plan, and one global cache for all packages.

The authorization ID is cached in storage each time the execute plan authority check or the execute package check succeeds. When a plan is to be run or when a package execute authority needs to be checked, if the executor authorization ID is already cached for this plan or package, the authorization check is skipped. This eliminates authorization checking cost for the most frequent authorization IDs that execute a plan or authorization IDs that execute packages. There is no user action required to take advantage of or benefit from this performance improvement.

11.4.2 Minimize the cost of plan authorization checking

When checking the authorization to execute a plan, DB2 uses a cache to improve performance. Checking is fastest when the EXECUTE privilege is granted to PUBLIC. PUBLIC does not use cache, so it is faster than getting a hit in authorization cache. After that, the second best way, in regards to performance, to check authorization, is when the plan is reused by any ID that already appears in the cache. The size of the cache is set in the BIND PLAN subcommand via the CACHESIZE parameter. If the parameter is not specified, the plan cache size defaults to the AUTHCACH DSNZPARM. However, note that anyone can execute a plan that has been granted to PUBLIC, and this causes a serious security exposure.

Following are our recommendations for frequently executed plans:

If you need to get greater transaction cost savings in authorization checking than those provided with DB2 plan authorization cache, consider:

- Granting the EXECUTE privilege of the plan to PUBLIC, and
- Setting the size of the plan authorization cache to 0 for the plan

When granting to PUBLIC, you should at least consider some partial ways to do authorization ID checking. These techniques, however, do not provide for auditing. For example:

- Include the ENABLE/DISABLE options when you bind the plan.

  You can restrict system connection types that can use a plan. System connection types are TSO, CICS, CAF, RRS, IMS, IMS MPP, IMS BMP, DL/I batch, and remote connections. Always just enable the plan for those connection types through which the plan should be invoked.

- Use IMS, CICS, and WebSphere authorization checking.

  IMS and CICS have transaction to program control. If a user comes in via a transaction monitor, the transaction monitor can verify that the user is authorized to execute the transaction.

  WebSphere can provide robust security management including authentication, authorization, and auditing infrastructure for WebSphere-based applications. These facilities interact with the zSeries Security Server (or equivalent) to augment the security infrastructure provided directly within WebSphere.

Configurations within the Enterprise JavaBean, with the global security settings, and within the datasource definition all affect how the identity is determined for accessing DB2.

If you check DB2 authorization with the plan authorization cache, consider:

- Granting the EXECUTE privilege of the plan to a limited number of authorization-IDs
- Setting the size of the plan authorization cache adequately for the plan

  The CACHESIZE option allows you to specify the size of the cache to acquire for the plan. DB2 acquires storage from the EDM storage pool. The default CACHESIZE value is 1,024 or the size set at installation time. The size of the cache that you specify depends on the number of individual authorization IDs actively using the plan. The overhead takes 32 bytes, and each authorization ID takes 8 bytes of storage. The minimum cache size is 256 bytes (enough for 28 entries and overhead information) and the maximum is 4,096 bytes (enough for 508 entries and overhead information). The specified value rounds up to the next highest value that is a multiple of 256.

11.4.3 Minimize the cost of authorization checking for packages

You can also consider using the package authorization cache. As mentioned before, the package authorization cache is not for an individual package, but it is a global pool used to store all cached package authorizations. This performance enhancement provides a run-time benefit for:

- Stored procedures
- Local packages that are in a package list and for which the plan owner does not have execute authority at bind time but does at run time.
  Remember that in a local environment the plan owner needs to have the privilege to execute the package. The authorization ID associated with the user that is running the local plan does not need any privilege on the invoked packages.
- Local packages that are not explicitly listed in a package list but are implicitly listed by collection-ID.*. The benefit is also provided for use of *.* or *.package-ID.
- Packages that are invoked from a remote location.

Our recommendations for frequently executed packages are:

- Grant the EXECUTE privilege of all packages in a collection (collection-ID.*) to some authorization-IDs or grant the EXECUTE privilege of the package to PUBLIC, so long as security is not required.
- Set the size of the package authorization cache adequately.

  You can set the size of the package cache by using CACHEPAC DSNZPARM. The default value of 32 KB is enough storage to support about 400 collection-ID.package-id entries or collection-ID.* entries. For each increase of 100 packages, add 8 KB of storage to the cache.

11.4.4 Routine authorization cache

When invoking a stored procedure, you not only need to be able to execute the package associated with the stored procedure, but you also need the necessary authorization to execute the stored procedure. In other words, you need EXECUTE privilege on the stored procedure. A certain authorization ID that is allowed to invoke a stored procedure can also be cached, the so-called routine cache, that is shared by all stored procedures. You define the size via the CACHERAC DSNZPARM. (The routine cache is also used to cache information about who is allowed to invoke user-defined functions, but this is beyond the

scope of this discussion.) A 32 KB routine authorization cache is large enough to hold authorization information for about 380 stored procedures or user-defined functions.

11.5 Static SQL, dynamic SQL, and reoptimization

You can code your SQL as either static or dynamic SQL.

For this discussion, we classify something as *static SQL*, if the access path is determined at bind time, and stored in the plan or package.

We use the term *dynamic SQL* when the statement is dynamically prepared (bound) (at least the first time) when it is executed. When the dynamic statement cache (DSC) is active, it will be stored in there, so it can be reused the next time the exact same statement is executed.

In this section, we do not really address whether or not a program should use static or dynamic SQL from a functional point of view, or from a manageability point of view (better control over when the access path changes), our focus is on the performance impact of using one versus the other, and the use of reoptimization.

11.5.1 Dynamic versus static SQL

Which coding technique should you use? When making that decision, consider at least these three issues:

- Functionality
- Performance
- The level of control that you have over your application

We look at these issues.

Functionality

The major functional advantage of dynamic SQL is flexibility. If you do not know the predicates until you actually PREPARE the statement at execution time, dynamic SQL may be a good solution. This feature can be useful for query environments as described in “Many criteria available, but only a few selected” on page 227. With some types of application coding, you do not have all of the choices, for example, ODBC and JDBC. See the redbook, *DB2 for z/OS and OS/390: Squeezing the Most Out of Dynamic SQL*, SG24-6418.

Performance

Assuming an equivalent access path, static SQL with REOPT(NONE), since NOREOPT(VARS) should no longer be used, will provide the best performance, since DB2 does not have to determine the access path at execution time.

If you expect that DB2 can find a better access path when it knows the actual values of the host variables, that is the first reason to use dynamic SQL or reoptimization. But if DB2 does not find a better access path, you waste a number of CPU cycles.

When your SQL uses a host variable instead of a literal, the optimizer cannot exploit nonuniform distribution statistics. This may not be a problem if your data has a uniform distribution. However, if your data is skewed, a single access path cannot provide optimal efficiency for all values applicable to your host variable. Therefore, the solution is to have multiple access paths. We discuss the different options in the sections that follow.

The level of control you have over the application

It the application is a homegrown application, you can recode the application to handle different options with a different SQL statement.

A common way to handle this problem is the use of multiple statements instead of just one, as coded in Example 11-1. Multiple statements enable you to have multiple access paths.

Example 11-1 Use different statements for different host variable values

```
SELECT (HV1);
    WHEN ('Value 1')
      DO;
         EXEC SQL SELECT COL1, COL2, COL3
                    FROM TABLE WHERE COL4 = 'Value 1';
      END;
    WHEN ('Value 2')
      DO;
         EXEC SQL SELECT COL1, COL2, COL3
                    FROM TABLE WHERE COL4 = 'Value 2';
      END;
    OTHERWISE
      DO;
         EXEC SQL SELECT COL1, COL2, COL3
                    FROM TABLE WHERE COL4 = :HV1;
      END;
```

However, if it is a packaged or vendor application, your only option is probably reoptimization.

11.5.2 Why do we need reoptimization?

When you want to “reuse” an access path, you either use static SQL, or you use dynamic SQL with parameter markers in combination with the dynamic statement cache (DSC).

When you use static SQL, the plan or package is bound once at bind time, and then the same access path is reused every time that statement is executed.

When you use dynamic SQL and you want to avoid determining the access path each time the statement is executed, you can use the DSC. The first time the statement is prepared, DB2 determines the access path and stores all the information in the DSC. Anyone else who wants to execute the exact same statement (including blanks) can then reuse the statement in the DSC and save the overhead of binding the statement at execution time.

However, as always, there is a price to pay. In order to have a “flexible” static SQL statement, you use host variables and not fixed values or fixed literals. This way, you can vary the value of the host variable at execution time, but still execute the same static SQL statement.

The same concept applies to dynamic SQL. In order to improve the chances of finding a matching statement in the DSC, you should use parameter markers (“?”) in your dynamic SQL statements when you prepare them, and provide the values for the parameter markers at execution time. Remember that the complete statement string has to match before the statement in the cache can be reused. Therefore, if you specify WHERE COLA=’ABC’, what are the chances that another program will also need the values ‘ABC’ from COLA. But if you use WHERE COLA=? at PREPARE time, it is more likely that the statement will be reused from the DSC.

The “disadvantage” of using host variables and parameter markers from an optimization point of view is that their value is not known at BIND or PREPARE time. As a consequence, the

optimizer has to make certain assumptions, and may not be able to come up with the best possible access path for every possible value of the host variable or parameter marker that will be specified at execution time.

11.5.3 Reoptimization as a solution

REOPT is a bind option that controls when DB2 builds the access path information for dynamic SQL applications. By default, the access path is built at PREPARE.

Three new options, REOPT(ONCE), REOPT(NONE), and REOPT(ALWAYS) can be specified in the BIND and REBIND commands for plans and packages.

REOPT(ONCE) defers access path selection until OPEN. The values of host variables at OPEN time are used to build the access path. The resulting access path is cached in the global dynamic statement (prepared) cache. REOPT(ONCE) is valid only in new-function mode. REOPT(ONCE) only applies to dynamic SQL statements, and is ignored if you use it with static SQL statements. DB2 for z/OS caches only dynamic statements. If a dynamic statement in a plan or package that is bound with REOPT(ONCE) runs when dynamic statement caching is turned off (DSNZPARM CACHEDYN=NO), the statement runs as if REOPT(ONCE) is not specified.

REOPT(NONE) is the default option. NOREOPT(VARS) is a synonym for REOPT(NONE). REOPT(VARS) can be specified as a synonym for *REOPT(ALWAYS)*. By using REOPT(ALWAYS), you tell DB2 that you want the access paths to be reevaluated at execution time, using the values of the host variables and parameter markers that are in effect when the statement is executed. The reevaluation applies to any statement that uses host variables, parameter markers, or special registers.

REOPT(ONCE) is the choice for DSC. REOPT(ALWAYS) is the choice when the SQL is embedded, but you want dynamic SQL optimization and no DSC.

Note: You can use REOPT(ONCE) and KEEPDYNAMIC(YES) on the same plan or package, but REOPT(ALWAYS) and KEEPDYNAMIC(YES) are mutually exclusive.

11.5.4 When to use reoptimization

Do not change your existing applications if they work efficiently. Using REOPT(ALWAYS) would only increase the CPU consumption and not provide any enhancement.

Reoptimization can be used to solve the problem explained in Example 11-1 on page 306, for example, when you have no control over the application code. If you rebind the plan or package with REOPT(ALWAYS), you no longer need to test the different cases and invoke the appropriate statement. All the values you provide at run time behave as though they are literals without the need to code them as literals. You should, however, only make the change, if you have a large number of specific values in the program.

As mentioned before, a good reason to choose dynamic SQL over static SQL is the cost of applying dummy predicates. When you do not know which predicates are going to be used, you are forced to provide all of them, equipped with a technique to disable each of the predicates. A common technique is the use of the “(*predicate* OR 0=*:hv*)” construct. If the predicate is not present, you supply a value of 0 for the :hv. This disables the predicate as 0=0. There is no need for DB2 to verify the predicate since this check is irrelevant. However, this check is applied at stage 2, because the 0=:hv predicate is a stage 2 predicate.

The performance question we have to answer is, “What is more expensive, the PREPARE cost when using dynamic SQL, or the redundant checks included in the static SQL statement?”

The cost depends on the stage at which the predicate is applied, which depends on the coding technique used. The real question is the number of rows to which the predicate is applied. If you code the redundant predicate as a stage 1 predicate and there are other stage 2 predicates that provide considerable filtering, you may want to change your stage 1 predicate to a stage 2 predicate and move it to the end of the query. Doing so reduces the number of times the predicate is applied and therefore the CPU overhead.

Applying a typical search argument to one row costs approximately 0.2 to 0.3 microseconds of CPU time on a z900 processor. You can apply one redundant predicate to 10,000 rows, assuming a PREPARE costs 30 msec, and still achieve the same cost as the dynamic SQL option.

If you exceed this number, code your SQL as dynamic SQL. Make sure to code the predicate as a stage 2 predicate to reduce the number of rows that need to be scanned. If your WHERE clause does not contain any stage 2 predicates, try to use stage 1 predicates for your redundant predicates. An example of a stage 1 predicate is the use of LIKE. You can disable the predicate by specifying only a '%' in the host variable.

11.5.5 Summary

If you have a good access path, use static SQL. Neither dynamic SQL nor REOPT provides you any benefit. These options only increase your CPU consumption without any added value.

If your static SQL suffers from occasional performance problems as a result of skewed data distribution, using REOPT(ALWAYS) can be a solution. You should isolate the SQL that causes the problem in a separate package to minimize the overhead, as REOPT can only be specified for a plan or package, not at the statement level.

If your selection criteria vary considerably and the number of rows to be processed is important, consider dynamic SQL. If the dynamic SQL issues one PREPARE and subsequent OPEN, FETCH, and CLOSE with different values that have a skewed distribution, consider using REOPT(ALWAYS). If you do so, make sure that any DESCRIBE is coded after you issue an OPEN or EXECUTE. If you fail to do so, DB2 will perform the PREPARE twice.

11.5.6 Performance coding with dynamic SQL

Without going into the detail of application coding for dynamic SQL, we still want to provide some coding tips when using dynamic SQL.

Dynamic statement cache

To reduce the CPU overhead of using dynamic SQL, you should enable the (global) dynamic statement cache via the CACHEDYN DSNZPARM. DB2 will start caching dynamically prepared statements in the dynamic statement cache (DSC), and when a PREPARE request is received, will first check the DSC for a matching statement. If a match is found, the prepared copy from the DSC is handed to the application, and a normal prepare is avoided. This is also called a *short prepare*.

Another advantage of using the DSC is that dynamic SQL statements that exist in the global cache do not acquire an S lock on the DBD like “plain” dynamic SQL does, therefore, providing better concurrency.

Using parameter markers

To give DB2 a better chance to find a matching statement in the DSC, you should use parameter markers “?” in your dynamic SQL statements.

No prepare after commit

If you want to optimize dynamic SQL performance even more, you can use the KEEPDYNAMIC(YES) BIND option in combination with the use of the DSC. In this case, the optimized statement is kept in local thread storage (local statement cache) across a commit. Normally prepared statements are destroyed at commit. But to be able to take advantage of this feature, the application must avoid to issue the PREPARE after the commit. The coding should look like:

```
PREPARE
OPEN, FETCH, ... CLOSE
COMMIT
OPEN, FETCH, ... CLOSE
COMMIT
OPEN, FETCH, ... CLOSE
COMMIT
```

Note that there is only one PREPARE. The other times, the prepare is avoided. If you re-prepare after the commit, the local statement cache is not used, and the statement is either copied from the global DSC, or re-prepared from scratch.

To control the amount of storage occupied by the local statement cache, use the MAXKEEPD DSNZPARM. Using KEEPDYNAMIC(YES) can have a serious impact on your system’s virtual storage usage in the DB2 DBM1 address space, and should only be used after consulting with your DB2 and MVS system programmers.

11.6 CICS Open Transaction Environment

Prior to CICS TS 2.2, an application runs on the quasi-reentrant task control block (QR TCB) inside the CICS application only region. All SQL statements run under separate TCBs that are only used for processing DB2 requests. Every time the application issues an SQL call, the application switches from the QR TCB to one of the private TCBs that are used for SQL calls, and switches back to the QR TCB once the SQL call has completed. This method uses extra CPU costs per call, because the task is switching back and forth.

CICS TS 2.2 improves this method by exploiting the Open Transaction Environment (OTE), that was first introduced in CICS TS 1.3. In order to exploit the OTE support, your transactions have to be threadsafe. You have to mark these transactions as threadsafe in the program property tables in CICS. If a service is not threadsafe, CICS switches back to the QR TCB as it does for non-OTE environment. You indicate that a transaction is threadsafe by means of the CONCURRENCY(THREADSAFE) option.

If your CICS DB2 application is not defined as threadsafe, or if it uses EXEC CICS commands or exits that are not threadsafe, TCB switching takes place and some or all of the performance benefits of OTE exploitation are lost. Note in particular that the use of a non-threadsafe exit program on the CICS-DB2 mainline path (for example, a program enabled at XRMIIN or XRMIOUT) may causes more TCB switching than that experienced than before.

Recommendation: From a performance point of view, we highly recommend you use threadsafe programs. They will reduce the number of TCB switches dramatically.

The best case scenario is that a program only does two TCB switches. The first one, when the first SQL call is executed, switching from the QR TCB to the so-called open TCB. The second switch is when the thread ends, and we switch back to the QR TCB. With OTE, the same TCB can be used by the CICS DB2 task-related user exit, the SQL requests that CICS makes to DB2, and any subsequent application code.

For more information, see Technote # 1160142 on:

http://www.ibm.com/software/htp/cics/support

OTE threadsafe and DB2 accounting

Using OTE in a DB2 environment also changes the way DB2 accounting times are reported.

- Before this enhancement, only DB2 calls were executed under a TCB that was created and managed by the CICS-DB2 attachment facility. All the other work the application performed was executed under the QR TCB. The result of that was the DB2 Class 1(in application + in DB2) and DB2 Class 2 (in DB2) times were almost identical. Since the actual application code was executed under a different TCB (QR), it could not be reflected in the TCB times of the TCB that did the DB2 work.
- When running as a threadsafe application starting in CICS TS 2.2, Class 1 and Class 2 times will be different. Class 2 time will be the same as before, since it reflects the elapsed time and CPU time spent inside DB2, but the Class 1 time will now include all the elapsed time spent from the first SQL call (switch to the OTE TCB) until the thread ends.

DB2 accounting in a CICS-OTE environment will be more in sync with the behavior of other transaction monitors.

11.7 Rebinding plans and packages

Before attempting any exotic tuning, determine if a rebind will change the access path.

> **Note:** If the application program was not bound with EXPLAIN(YES), or if the PLAN_TABLE output is gone, it is very difficult to be certain of the current access path.

It is best not to rebind a production program without knowing in advance what the new access path will be. However, a BIND done in a development environment may not pick the same access path as a BIND in a production environment, even if the catalog statistics are identical. Subtle differences between environments such as DSNZPARM settings, the number of processors and their speed, buffer pool, sort pool and RID pool sizes, as well as maintenance level, can all influence access path selection.

The best way without affecting the current production access path is to rebind the package into a new (dummy) collection using EXPLAIN(YES). However, the bind process acquires locks on the catalog, so be careful not to perform binds at a time when DDL is also executed, otherwise, lock contention may result. In the case of a plan, the same applies, since you can generate the bind instructions and give the plan a DUMMY name.

For dynamic SQL, in DB2 V8 you can flush the dynamic statement cache to force a reoptimization using RUNSTATS TABLESPACE xxxx TABLE yyy REPORT NO UPDATE NONE. It flushes the statements in the cache that have a dependency on table yyy without gathering statistics.

11.7.1 Review bind parameters

When looking at rebinding the plan or package, also have a look at the bind options the plan or package is currently using, because some of these options may have influence on the access path. Some of the options that you may want to have a look at include:

- ISOLATION
- CURRENTDATA
- RELEASE
- VALIDATE

For more details about the impact of these options on concurrency and performance, see “Database BIND options” on page 290.

11.8 The role of catalog statistics

DB2 is a relational database, the access path selection is one of the added values of the database. The system determines the most efficient way the data can be accessed and not the application itself. DB2 uses a cost-based optimizer to determine the best access path for an SQL statement.

It is, therefore, vital to keep the catalog statistics up to date.

During access path selection, DB2 needs to know the size and state of your tables and indexes in order to determine a good access path. The more sophisticated the optimizer has become, the more it relies on accurate statistics.

11.8.1 Getting the proper statistics

To obtain catalog statistics, you generally use the RUNSTATS utility. RUNSTATS can be run against a table space or an index. RUNSTATS has many options. Describing all of them is beyond the scope of this book.

The minimum is RUNSTATS TABLESPACE INDEX. TABLE and the column distribution statistics introduced by DB2 V8 are desirable for performance-sensitive applications.

Using RUNSTATS is not only important for the access path selection, but also for the correct behavior of the dynamic allocation of utilities that use the catalog information to determine the allocation size. RID list processing also exploits the catalog statistics to determine when to abort the RID gathering process. Thresholds of the REORG utility execution also exploit the information in the catalog.

With real-time statistics (RTS), DB2 introduced the ability to get information about the behavior of a table and the associated indexes without having to execute RUNSTATS. The information is gathered while transactions are running. The RTS information is stored in the RTS tables SYSIBM.TABLESPACESTATS and SYSIBM.INDEXSPACESTATS. The information in these tables can help you to determine when RUNSTATS, REORG, and COPY need to be performed. The DB2 utilities will not exploit the RTS tables at this point, but the DB2 Tools can use the RTS information to determine which utility needs to be executed. Currently, this information is not used during access path selection, and is focused on object maintenance (although this is likely to change in the future).

When do I gather statistics?

What is the best frequency to execute RUNSTATS?

The short and easy answer would be to run RUNSTATS when your data has changed significantly. But then the next question would be, “How do I know if my data has changed?” The answer would be then RTS. RTS gives a very good indication of the amount of changes that occurred to the data and that may justify a new RUNSTATS execution.

Another option may be to regularly run RUNSTATS with the UPDATE NONE HISTORY ALL option. This way, you gather runstats information, but only store the information in the statistics history tables in the DB2 catalog. The history tables are not used during access path selection, so SQL statements will not be impacted. This way, you can perform some trend analysis on the history information and decide to run a RUNSTATS UPDATE ALL when the data has changed enough to justify updating the catalog statistics that are used by the optimizer.

However, be aware that once the catalog statistics have been updated, dynamic SQL will start using these new statistics on the next PREPARE. For static SQL, you have more control because the new catalog statistics will only be used when the plan or package is rebound. In order to have all static plans and packages take advantage of the new statistics, you could run a REBIND PLAN(*) and REBIND PACKAGE (*.*). However, since your REBIND might not work if the package is being used, you need to schedule your REBIND activity to avoid contention with active packages. Furthermore, a massive rebind causes catalog contention and may also introduce the risk of picking up an access path that is worse than the existing one.

Generally, it is better to run the BIND in a controlled process where first the Explains are checked, and then a list from the catalog is run preparing the execution of binds separated by application grouping.

Additional considerations about this topic are at “Performing REBIND” on page 364.

Just looking at the total number of rows, or changes in the number of rows in a table may not be enough criteria to determine when RUNSTATS is required. Let us assume that we are using dynamic SQL or using REOPT, and your table receives 100,000 inserts, and 100,000 deletes per week. Do you need to execute RUNSTATS? Probably not, since the total size did not change. But what if your SQL is searching for rows based on a column with a value larger than the current date minus one week. In this case, the filter factor calculation of the range predicate will be driven by the nonuniform distribution statistics and the values in SYSCOLUMNS. If your last RUNSTATS was performed a month ago, DB2 wrongfully assumes that no rows qualify the range predicate. In reality, all rows qualify when the table only contains the orders of the last week.

Some tables have a very volatile nature; sometimes there are only a few rows in the table, and a few hours later there are a million rows. In this case, the biggest concern usually becomes the effective use of indexes in the access path selection in order to avoid locking problems. You can use the VOLATILE table attribute to indicate to the optimizer that the number of rows in the table is volatile, and that you would prefer a matching index access path on this table whenever possible. The VOLATILE attribute is also explained in “Volatile tables” on page 23.

Even for tables defined with the VOLATILE attribute, it is vital to have correct statistics, especially using the volatile table in joins with other tables. Therefore, make sure that the statistics are gathered when the table is full and properly organized. Once that is done, you should exclude the volatile table from subsequent RUNSTATS cycles unless the size of the table at peak volume starts to vary significantly.

We just established that the catalog information is a main driver for the access path selection. This means that it is not enough by itself to reflect the current status of the object. You should also worry about the “state” of the object at the time RUNSTATS is performed. Consider

issuing a REORG of the object and make the statistics gathering part of that process. This way, DB2 not only knows about the new size and information in the table, but also assumes the indexes and table are correctly organized, which has a positive impact on access path selection.

Statistics Advisor tool

Statistics Advisor tool (SA) is a Visual Explain plug-in. It can help you cut down the effort in determining what kind of statistics you need to give the optimizer the best chances of determining the best access path for a certain query. SA sets up the necessary RUNSTATS control statements to gather missing or correct inconsistent statistics. The main purpose of SA is to help you identify the RUNSTATS executions that help your SQL. Sometimes, it is not simple to understand which statistics can help your queries. SA builds the RUNSTATS statements for the correct set of statistics and gives you consistent and accurate information. You can continue to run the SA until it no longer provides suggestions.

SA analyzes the predicates of your query, column (groups) used as predicates, type of predicates, performs statistical analysis, checks missing statistics (default), looks for conflicting statistics, missing appropriate correlations, and skewed statistics. Based on all of that information, SA gives suggestions on the RUNSTATS to execute. Furthermore, it gives you an explanation why it thinks the statistics need to be gathered, and a conflict report. For more information about the SA tool, refer to the redbook, *DB2 UDB for z/OS Version 8 Performance Topics*, SG24-6465.

On what portion of the data

The granularity of RUNSTATS is either the entire table or a range of partitions. If you load daily 5% of new data and that data is spread randomly over the entire table, being selective regarding the amount of data about which you gather statistics can be quite cumbersome. You may consider clustering the data in a different way such that the rows are added in a limited number of partitions, but RUNSTATS would not be a good reason to change your design. Another option is to just gather statistics on the indexes that are most affected by the loading of the 5% new data and leave the table unchanged until a larger percentage of loaded data justifies the effort of doing a full RUNSTATS.

Do you really need to process all the rows to get a representative sample of the data? DB2 provides the ability to sample the table when gathering statistics. If the object is large enough, the added value of scanning all 200 million rows might be marginal compared to the additional CPU consumption. Consider using the SAMPLE keyword to reduce the CPU cost of running RUNSTATS. The default sampling percentage of 25% is a good balance between CPU savings and still looking at enough data to be able to collect statistics that are equivalent to those if all data was processed.

A possible strategy to implement sampling could be to do a RUNSTATS on all the data first, and EXPLAIN any queries with critical performance requirements. Then, introduce sampling at the default of 25%, REBIND those queries into a different collection, and analyze the access path differences. If there are no access path changes, consider further reductions in sampling percentage to maximize your potential CPU saving. If access path changes do result, you should run the queries concerned and check if the performance is acceptable.

At what level of detail

The RUNSTATS utility provides different levels of analysis that should be in line with the use the optimizer is making from the information in the catalog.

The main question is whether you need to use the COLGROUP, KEYCARD, NUMCOLS, and FREQVAL options.

The COLGROUP option enables RUNSTATS to collect cardinality statistics on the specified group of columns at the table level. When you specify the COLGROUP keyword, RUNSTATS collects correlation statistics for the specified column group. If you want RUNSTATS to also collect distribution statistics, specify the FREQVAL option with COLGROUP.

If the columns on which you want to gather statistics are leading index columns, you can also gather cardinality and frequency statistics using the RUNSTATS INDEX statement. The KEYCARD keyword indicates that you want to gather cardinality statistics. The FREQVAL option, when specified with the INDEX option, activates the collection of distribution statistics. This keyword must be followed by the NUMCOLS and COUNT keywords. NUMCOLS *n* indicates that you want to gather statistics on the first *n* leading index columns. COUNT m indicates how many (m) frequently occurring values you want DB2 to collect as frequency statistics. Note that you can collect the m frequently occurring values (MOST), the m least frequently occurring values (LEAST), or both (BOTH). See *DB2 UDB for z/OS Version 8 Utility Guide and Reference,* SC18-7427-02, for full details about the options.

You need to consider that by adding the frequency values in the SYSIBM.SYSCOLDIST catalog table, you may increase bind/prepare cost. High CPU consumption may occur for dynamic prepare when more than 100 frequent values are present.

Tip: PTF UK00296 for APAR PQ94147 is meant to reduce such large bind costs.

Whether or not to use the FREQVAL and COLGROUP options depends on the profile and nature of your SQL.

A full static SQL environment, not using literals, and with no use of REOPT(ONCE) or REOPTS(VARS), will not benefit from the extra cost of gathering nonuniform distribution and column correlation information.

Manipulating catalog statistics manually

With proper authority, you can influence access path selection by using an SQL UPDATE or INSERT statement to change statistical values in the DB2 catalog. Although updating catalog statistics can help a certain query, other queries can be affected adversely. Also, there are relations across values to be taken into account, and any manual update must be repeated after the standard RUNSTATS cycle resets the catalog values. Be careful and document what was done. After having updated the statistics, rebind the plan to the desired access path, and reestablish the correct statistics. Any program maintenance and rebind will need the process repeated.

Using OPTHINT, if at all possible, is a better alternative to updating the catalog statistics because it localizes the effect of the "influencing".

Recommendation: Avoid manipulating catalog statistics. It is an error prone process, and cumbersome to maintain. Do it only temporarily, to bypass an acute problem until a more permanent solution is available, or as a last resort.

11.9 System parameters

In this section, we list a number of DSNZPARMs that can have a significant impact on your application performance and concurrency. Although you may not be able to change their setting because your application could benefit from a certain value, it is important to know that they exist and what their value is, so you can take the behavior that results from a certain value into consideration when designing your application. For example, if you know that the

maximum number of locks per user (NUMLKUS) is 1,000, you have to make sure that your application will not acquire more than 1,000 locks, because it will get an SQLCODE -904.

11.9.1 System parameters that affect locking

The following system level parameters control the locking behavior of DB2. These parameters should be tuned to maintain the overall health of your DB2 subsystem. Therefore, it is unlikely that you will want to change them based upon the needs of your particular application. However, it is advisable to be aware of how these parameters are set and understand how they affect the locking behavior of your application.

IRLM PROC parameter DEADLOK: Deadlock detection cycle

This value, which is the DEADLOCK TIME on installation panel DSNTIPJ, controls the length of the deadlock detection cycles. Values can be set from 100 to 5,000 milliseconds.

While this is not a DSNZPARM, it is worthwhile to know how long the IRLM will wait between checks for deadlocks. Deadlocks do not go away. Once two threads are deadlocked, they can only wait for the deadlock to be detected, and one of the threads is forced to release its locks.

IRLMRWT DSNZPARM: Timeout value

This DSNZPARM specifies the minimum length of time a DB2 thread will wait for a lock before it times out. Every time the deadlock detection cycle runs, it will check each thread that is waiting for a lock. Any threads that have been waiting longer than this value will be timed out. This means that the maximum length of time a DB2 thread will wait for a lock before it times out should be the IRLMRWT DSNZPARM value plus the DEADLOK IRLM PROC value.

UTIMOUT DSNZPARM: Utility timeout multiplier

This DSNZPARM controls the length of time that a utility or utility command will wait for a lock or a claim to be released. The time is calculated by multiplying the IRLMRWT value by the UTIMOUT value, which allows utilities to wait longer than SQL applications to access a resource.

NUMLKTS DSNZPARM: Maximum number of locks per table space

This DSNZPARM sets the system level value of the LOCKMAX table space parameter. When a table space specifies LOCKMAX SYSTEM, this value is the maximum number of page or row locks that can be taken on that table space by a single thread before lock escalation occurs. This is discussed in more detail in the section entitled “The LOCKSIZE parameter” on page 333.

NUMLKUS DSNZPARM: Maximum number of locks per user

This DSNZPARM sets the maximum number of page or row locks that can be held concurrently by a single thread. If you specify 0, there is no limit.

Prior to DB2 Version 8, it was important to limit the number of locks per user because of storage constraints. With DB2 Version 8, the IRLM lock information is moved above the 2 GB bar, so storage constraints are less of an issue (if you have enough real storage on your machine). However, implementing NUMLKUS is a good way to implement a “reality check” so that applications that are taking too many locks and not committing frequently enough are identified and corrected.

EVALUNC DSNZPARM: Evaluate Uncommitted

This DSNZPARM controls the locking behavior of stage 1 predicate evaluation using isolation levels CS and RS. This option applies only to table access, stage 1 predicate evaluation in the index is not affected.

The default value of **NO** means that a lock must be acquired before stage 1 predicate evaluation can occur.

If **YES** is specified, predicate evaluation occurs before a lock is requested. If the row or page does not qualify, then no lock request is necessary.

Selecting YES for this DSNZPARM can reduce lock requests and improve application throughput with minimal data integrity exposure.

The data integrity exposure of using EVALUNC YES is that allowing predicate evaluation to read a locked page may disqualify an updated row based upon the new value. If the update is rolled back to a value that would have qualified the row, then the row will appear to have been excluded from the cursor in error.

The opposite is not true. If a value in an updated row causes the row to qualify on the predicate, then a lock request is issued and the predicate will be reevaluated before the data is returned to the application. If the update is rolled back to a value that no longer qualifies, the row will not be returned.

RELCURHL DSNZPARM: Release lock on CURSOR WITH HOLD

This DSNZPARM controls whether the lock on the current positioned row/page is retained when a COMMIT is issued.

The default of **YES** will cause any row or page lock on the positioned row or page to be released at COMMIT.

A value of **NO** will hold any lock on the positioned row or page to be held across a COMMIT.

The only reason to hold the lock on the current row in a cursor is if that data is going to be used subsequently within the unit of work. If the application is designed properly, the COMMIT will occur at the end of a unit of work, which means that the data from the currently positioned row has already been processed.

RRULOCK DSNZPARM: Acquire U lock for RR and RS

This DSNZPARM controls the behavior of cursors coded with the FOR UPDATE clause when using isolation levels of RR or RS.

The default value is **NO**, which means that an S lock is acquired on each qualifying row or page. If the row or page is updated, the lock is promoted to an X lock.

YES indicates that a U lock will be acquired on each qualifying row or page. Subsequently, the U lock will be demoted to an S lock on the next fetch if the row or page is not updated, or promoted to an X lock if the row or page is updated.

The advantage of using the YES option to acquire the U lock is that it reduces the chances of deadlock. The cost of the YES option is the additional IRLM request to demote the lock to an S if the row is not updated.

SKIPUNCI DSNZPARM: Skip uncommitted insert

This DSNZPARM controls the behavior of predicates that might qualify rows that have been inserted, but not committed. It was introduced with APARs PQ82390 and PQ92833. Note that

SKIPUNCI only applies to row level locking. Like EVALUNC, it applies only to isolation levels CS and RS.

If you specify the default value of **NO**, uncommitted inserts will be evaluated according to the behavior specified by the EVALUNC DSNZPARM.

YES means that uncommitted inserts will be treated as if they have not yet arrived.

> **Important:** Setting this DSNZPARM to YES before APAR PQ92833 has been applied may lead to unpredictable results.

When using row level locking, this DSNZPARM provides a way to allow queries to ignore newly inserted rows rather than having to wait for a commit. This can improve the performance of queries running concurrently with mass insert applications.

XLKUPDLT DSNZPARM: Acquire X lock on stage 1 qualified rows

This DSNZPARM controls the locking behavior of searched UPDATE and DELETE statements that have both stage 1 and non-stage 1 predicates. XLKUPDLT can assume three values:

- The default of **NO** will acquire an S or U lock on the row or page when it has been qualified based upon the stage 1 predicates and then promote the lock to an X if the row still qualifies after all remaining predicates have been applied.
- **YES** will acquire X locks on the row or page if it has been qualified by the stage 1 predicates.
- A third option of **TARGET** was introduced with APAR PQ98172. Like YES, TARGET will acquire X locks on the row or page if it has been qualified by the stage 1 predicate. The advantage of TARGET is that it will only acquire the lock on the table which is the target of the UPDATE or DELETE.

The implication of the third option is that specifying YES can cause tables other than the target table, such as a table specified in a subquery, to be locked. For this reason, we recommend that you only consider values of NO or TARGET.

The advantage of using the TARGET option to acquire an X lock as soon as the row is qualified using the stage 1 predicates is the elimination of an IRLM request to promote the lock when the row qualifies on all predicates. The cost is the potential of excessive locks, or rows that are locked exclusively that were not updated when the row failed to qualify on the non-stage 1 predicates.

11.10 Considerations when using Java to access DB2

In this section, we discuss a few considerations for users of Java applications and users of the IBM DB2 Universal Driver for SQLJ and JDBC when accessing DB2 for z/OS data.

11.10.1 Which Java programming interface should I use?

When using Java, and you want to talk to a database, there are two programming interfaces; JDBC and SQLJ. There are many differences between JDBC and SQLJ, too many to discuss here in great detail. When looking at the performance aspects of both, the JDBC versus SQLJ discussion is very similar to the dynamic SQL versus static SQL discussion, where JDBC is dynamic SQL, and SQLJ is static SQL.

From a performance and manageability perspective, SQLJ is your best option. The advantages of SQLJ are:

- SQLJ provides static SQL performance for Java applications. Programs are bound into packages prior to execution.
- Since prebound SQL is used, performance is more consistent. The access path is stored in a package, so it does not change without a REBIND or BIND REPLACE.
- SQLJ uses the static SQL authorization model. The user only needs execute on the package, not specific privileges on all the objects referenced in the package. Therefore, SQLJ provides Java with a stronger authorization model.
- since we are using packages with SQLJ, each application will have its own set of packages, unlike JDBC where everybody uses the same JDBC packages. Using separate packages allows for better monitoring and accounting.
- And, as a nice advantage on the side, writing SQLJ requires fewer lines of code, resulting in code that is easier to maintain, and more productive programmers.

For more information about JDBC versus SQLJ, see:

- *DB2 for z/OS and OS/390: Ready for Java*, SG24-6435
- *DB2 for z/OS and WebSphere: The Perfect Couple,* SG24-6319
- *DB2 UDB for z/OS V8 Application Programming Guide and Reference for Java*, SC18-7414-02
- Managing Enterprise Java Applications for DB2 UDB for z/OS at:

  http://www.ibm.com/support/docview.wss?rs=64&uid=swg27005333

11.10.2 Which driver type to use and when

There are two types of Universal Drivers, a Type 2 driver and a Type 4 driver (T4), each with its own performance characteristics. You use the same Universal Driver for DB2 for z/OS that is used for DB2 for Linux, UNIX, and Windows. In this discussion, we only discuss DB2 for z/OS.

From a functionality point of view, driver types are equivalent. The key difference between the two driver types is the way they interact with the database.

Type 2 versus Type 4

The Type 2 Driver is a local driver and talks directly to the database via a local attachment (RRS).

The Type 4 Driver implements a DRDA Application Requester and talks via DRDA (over a network) to DB2. For DB2 for z/OS, a Type 4 connection looks the same as any other distributed connection using a DBAT.

Therefore, from a pure performance point of view, a Type 2 (local) connection outperforms a Type 4 connection (that has to go out on the network; even if that network is a HiperSocket, there is still the extra overhead of the TCP/IP and DDF). However, in real life, not everything is just about performance. There are usually many other requirements that have to be taken into consideration when deciding whether or not to use a Type 2 or a Type 4 connection.

Type 4 with or without DB2 Connect

In this discussion, we completely ignore the fact that you must have a DB2 Connect license in order to be able to use a T4 Driver talking to DB2 for z/OS. We strictly limit ourselves to a technical discussion.

Since the Type 4 Driver has a built-in DRDA AR, it is not required to go via DB2 Connect, the traditional DRDA AR, to talk to DB2 for z/OS. So the question is, “Is there a reason to still use DB2 Connect?” or “Can I cut out “the middle man”?”
As almost always, the answer is, “it depends”:

- We recommend you use a direct Type 4 connection (so, without DB2 Connect) when connections are already pooled by some other means. For example, if you are using WebSphere on the distributed platform, connections are being pooled (highly recommended) and reused inside WebSphere. If that is the case, DB2 Connect features such as connection pooling and concentration do not provide much added value in terms of performance.
- On the other hand, when you have a large amount of direct client connections, for example, Applets connecting to your DB2 for z/OS system, this large number of connections come in through DB2 Connect and use DB2 Connect connection pooling and connection concentration to reduce the number of threads that are required on the DB2 for z/OS subsystem to support all of these remote clients.
- Note that the Universal Driver also supports DB2 data sharing connection distribution via the sysplex distributor and the use of dynamic VIPA. Enhanced data sharing support with connection concentrator and transaction workload balancing is scheduled for DB2 for Linux, UNIX, and Windows V8.2 FixPak 10.

11.10.3 Java application coding considerations

In this section, we examine several Java coding best practices.

Using the correct Java data types

Data type conversion is an expensive operation in Java. Therefore, it is important that your DB2 data types and your Java data types match, or be as “close as possible”.

You can find a list of DB2 data types and their Java equivalents in “Consider the data types of the application programming language” on page 6.

Encoding scheme

Besides the data type, you must also consider using a matching encoding scheme for the client, the Java programming language, the Driver, and the DB2 data. If there are differences somewhere along the line, the system will take care of it, but at the expense of performance, because these differences require additional conversions.

In general, Java applications always execute in Unicode. Character data types stored in the database as EBCDIC or ASCII, require a conversion. See also “Choosing an encoding scheme” on page 23.

The Universal Driver uses UTF8 when sending character data, and UTF16 to send graphic data. Consider storing user data in Unicode tables if the majority of the applications are implemented in Java to reduce codepage conversion costs.

UTF8 uses the same amount of storage as EBCDIC or ASCII, but UTF16 can use 50 to 100% more storage. However, when data is compressed by DB2, UTF16 gets a better compression ratio, such that the extra storage required becomes only 30 to 60%. Vendor products tend to use UTF16 with DB2 data compression to minimize the storage overhead.

Note that Unicode to and from EBCDIC and to and from ASCII conversion only applies to character data, not to numeric data. Numeric data does not depend on the encoding scheme.

Only select and update the columns that you need

Selecting and updating just the columns that you need have been general recommendations for many years, but these recommendations are especially true for Java applications.
For each retrieved column, this creates a Java object, and object creation is an expensive operation in Java. In addition, as mentioned above, strings for character columns may need to be converted between Unicode (Java) and EBCDIC/ASCII (DB2 engine). The more columns you retrieve, the more conversions.

Turn off autocommit feature

Defaulting to autocommit was not unique to Java. It was probably introduced by ODBC or CLI, and when JDBC and SQLJ came along, they decided use the same (bad) default commit behavior. By default, ODBC, JDBC, and SQLJ use autocommit. This means that DB2 forces a commit after every single SQL statement. This may sound tempting to use, because it make sure that locks are released as soon as possible, but it completely wipes out the concept of a unit of work. When using autocommit, every statement is its own unit of work, so a withdrawal and a payment would become two separate units of work; a guarantee for disaster. But not only is the functional impact usually not acceptable, using autocommit also has a big impact on performance.

Therefore, we highly recommend you switch off autocommit in every environment. In a Java environment, you can do this via a property of the connection, conn.setAutoCommit(false).

Use DB2 built-in functions

DB2 comes with many, useful built-in functions that are more efficient than their Java counterparts. For example, when retrieving a fixed-width character data column, you may want to get rid of the trailing blanks that DB2 appends whenever the value is shorter than the column's length. You can use the Java String.trim() method, such as this:

```
#sql { SELECT JOB INTO :job FROM DSN8710.EMP ... };
job = job.trim();
```

However, it is more efficient (and easier) to use the DB2 TRIM function, since you do not have to create an intermediate String object.

```
#sql { SELECT TRIM(JOB) INTO :job FROM DSN8710.EMP ... };
```

Using batched updates

A batch update is a set of multiple update statements that is submitted to the database for processing as a batch. Sending multiple update statements to the database together as a unit can, in some situations, be much more efficient than sending each update statement separately. This ability to send updates as a unit, referred to as the batch update facility, is part of the JDBC 2.0 API, and is supported by the DB2 Universal Driver for SQLJ and JDBC.

Note that autocommit should be turned off when using batch updates, and that the *Universal Driver always batches non-atomically*, and returns update counts for batched statements.

The DB2 Universal Driver also provides a *DB2-only interface* that lets you perform batch queries on a homogeneous batch. With the DB2PreparedStatement interface, you can execute a single SQL statement with multiple sets of input parameters. You can use the `PreparedStatement.addBatch()` method for creating a batch of parameters, so that a single statement can be executed multiple times in a batch, with a different set of parameters for each execution. You can use the `DB2PreparedStatement.executeDB2QueryBatch()` method to perform the batch query, as you see in Example 11-2 on page 321.

Example 11-2 Performing a batch query

```
try {
  ...
     PreparedStatement prepStmt = con.prepareStatement(
            "SELECT EMPNO FROM EMPLOYEE WHERE EMPNO=?");
     prepStmt.setString(1,empnum1);
     prepStmt.addBatch();

     prepStmt.setString(1,empnum2);
     prepStmt.addBatch();

     ((com.ibm.db2.jcc.DB2PreparedStatement)prepStmt).executeDB2QueryBatch();
    } catch(BatchUpdateException b)
       {
         // Add code to process the BatchUpdateException
       }
```

Use of deferred prepare

As with “ordinary” distributed dynamic SQL statements, deferring the prepare if possible also saves network messages when using JDBC with a Type 4 connectivity. The Universal Driver provides a connection property called `deferPrepares`. When enabled, and it is enabled by default, server prepare requests are deferred until execution time. This allows the prepare and execute to flow in a single network message to the server, reducing network delays.

However, the deferral of the prepare means that the Driver works without the benefit of described parameter or result set metadata. So undescribed input data is sent "as is" to the server without any data type cross-conversion of the inputs.

Support for cross-conversion of input types is an extension to JDBC, that is supported by the Universal Driver. Therefore, if a statement execution fails when deferPrepares is enabled, the execution is retried by the Driver with described input. This statement retry logic is done by the Driver, and is seamless to the application. However, if an application enables deferPrepares, we strongly recommend that input data types match the database column types as required by the JDBC specification. If an SQL error occurs because of a data type mismatch (-301), the Driver has to do cross-conversion of the input data and resend the execute. For example, if your DB2 column is a SMALLINT, you should use a PreparedStatement.setShort() setter for optimum performance.

If deferPrepares is disabled, then the Driver always requests describe information when the statement is logically prepared, and, therefore, has described input parameter metadata available, and cross-conversion of input types is supported for all PreparedStatement setter methods without requiring internal driver statement retry logic.

Internal statement retry logic is not performed for batched executes. Therefore, input data types must match the described column types for all batched parameters. The result will be an SQLException.

Internal statement retry logic is not performed by the Driver when any input parameter is a stream.

Sending data As-Is

The Driver provides cross-conversion to make up for data type mismatch. Certain “anomalies” occur because of this. If you experience such an anomaly, you can use the `sendDataAsIs` connection property. If this property is set to “true”, then the Driver is going to make an assumption about the type based on the setXXX method used and will no do cross-conversion.

In conclusion, JDBC applications should call prepared statement setters that match the column type at the server. This helps especially in cases where deferred prepare is on (the default), and the driver has to guess the data type to send to the server. If the driver guesses incorrectly, an error may be returned from the server, or in some cases, the driver attempts a retry of the statement using a different data type. This results in two network flows and defeats the advantage of having deferred prepare enabled in the Driver.

Release resources

Another thing that developers sometimes forget to do is to close and release resources when they are no longer using the resources. The JDBC driver maintains its own links to resources, and the resources are only released when the resources are closed, or when the connection is closed. For this reason:

- Close ResultSets when the application is done with them. If you do not do this, JVM™ garbage collection cannot reclaim the objects, and eventually the application may run out of JDBC resources or, even worse, run out of memory.
- Close PreparedStatements that have ResultSets as soon as they are no longer being used. Closing the ResultSet is not enough to release the underlying cursor resource. If you do not close the PreparedStatement, the cursor resource is tied up for the life of the PreparedStatement.
- Close CallableStatements when you are finished with them, too, or else the application may run out of call sections.
- Be sure to release resources even in the case of failure. The Java try/finally construct is well-suited to achieve this.

SQLJ makes things a bit easier for developers than JDBC, because the SQLJ translator automatically generates the code to release statements. However, you still have to close iterators yourself.

Using the built-in system monitor

The Universal Driver has a built-in system monitor that can be really useful. It can help you to isolate performance problems with your applications. The DB2 Universal Driver for SQLJ and JDBC provides a proprietary API (DB2SystemMonitor class) to enable application monitoring.

The driver collects the following timing information, as depicted in Figure 11-1 on page 323:

- Server time (the time spent in DB2 itself)
- Network I/O time (the time used to flow the DRDA protocol stream across the network)
- Core driver time (the time spent in the driver; this includes network I/O time and server time)
- Application time (the time between the start() and stop() calls)

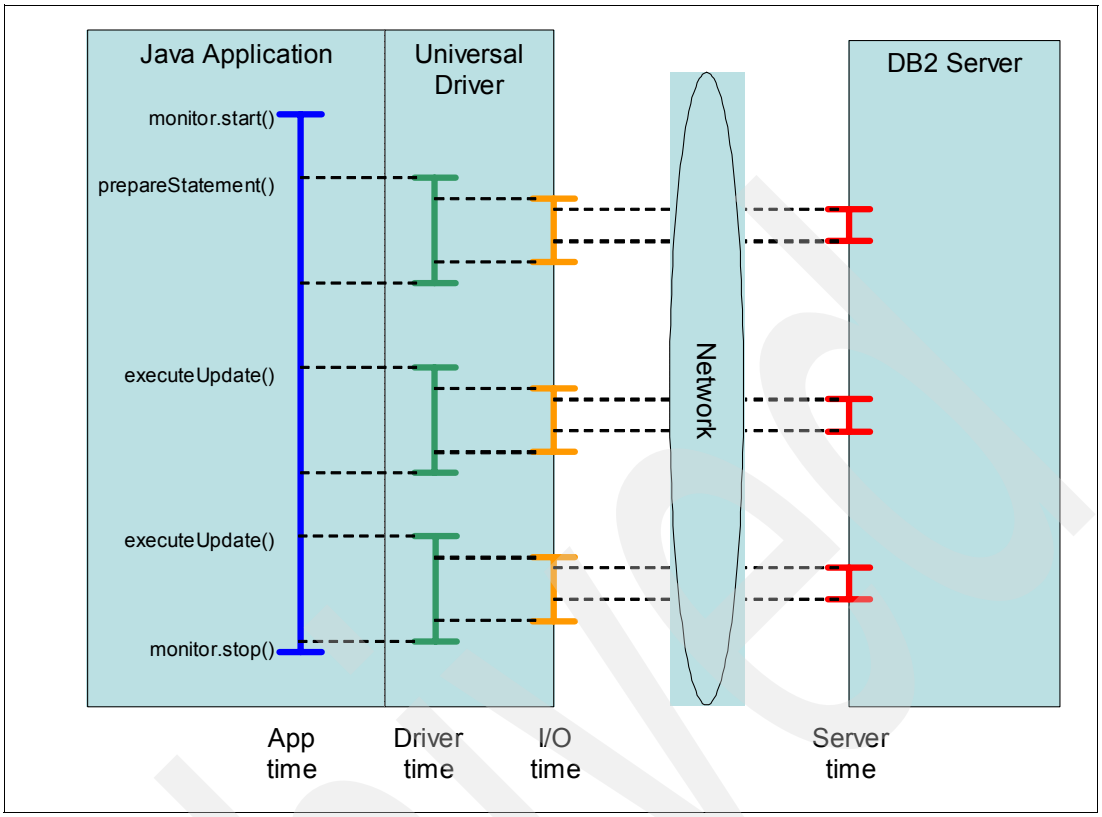


Figure 11-1 DB2SystemMonitor

The ServerTime is based on the DRDA server elapsed time. As a consequence, DRDA has to be involved in the transaction to get the ServerTime. If you are running your application locally to DB2 (connecting using Type 2 connectivity), you always get 0 (zero) from the getServerTimeMicros method. When run as a Type 4 Driver, the Universal Driver always uses DRDA to connect to the server.

Archived

Archived

Part 4

Designing for availability and performance

In this part, we provide background information about locking, we discuss techniques for achieving high application availability, and then we provide criteria for determining the application quality.

This part contains the following three chapters:

- Chapter 12, “What you need to know about locking” on page 327
- Chapter 13, “Achieving 24x7” on page 359
- Chapter 14, “Determining the application quality” on page 379

© Copyright IBM Corp. 2005. All rights reserved.

Archived

12

What you need to know about locking

The term “locking” refers to the collective set of serialization techniques used by DB2 to ensure the integrity of data in the database. This chapter covers locking topics that can affect application performance. While this chapter is not intended to be a tutorial about how DB2 locking works, since that could be an entire book in itself, this chapter covers some locking fundamentals.

Locking performance is affected by a number of factors, such as:

- Bind parameters

  Parameters, such as ACQUIRE, RELEASE, ISOLATION, and CURRENTDATA, affect how and when locks are taken for a specific plan or package.

- Table space parameters

  Parameters, such as LOCKSIZE and LOCKMAX, control the method of locking that is used for each table space. The MAXROWS parameter can be used to influence the granularity of locking on a table space by reducing the number of rows on a page.

- IRLM subsystem parameters

  Parameters at the IRLM level control the amount of time that DB2 threads will wait for locked resources and the frequency with which they will be checked for deadlock and timeout conditions.

- DB2 subsystem parameters

  DSNZPARMs, such as NUMLKTS, NUMLKUS, EVALUNC, RELCURHL, RRULOCK, SKIPUNCI, and XLKUPDLT, control locking behavior at a DB2 subsystem level.

- SQL coding

  The use of SQL clauses, such as WITH HOLD, WITH isolation-level, FOR UPDATE OF, and FOR FETCH ONLY, communicate to DB2 the locking requirements of a specific SQL statement.

© Copyright IBM Corp. 2005. All rights reserved.

- The order of SQL statement execution

  The order of execution of SQL statements and their placement within the logical unit of work can affect the duration of locks and the likelihood of deadlocks.

- The frequency of commits

  For DB2 processes that perform more than one logical unit of work, the frequency of commits has a significant effect on the locking behavior of the process and all other processes accessing the same DB2 objects concurrently.

In this chapter, we investigate how each of the above factors affect locking, and we provide recommendations about the following topics:

- Take advantage of lock avoidance and lock reduction DSNZPARMs parameters.
- Use ISOLATION(CS) CURRENTDATA(NO) for the default isolation level and only use others as needed.
- Define table spaces with LOCKMAX 0 to prevent lock escalations.
- Do not leave your locking decisions to DB2 by using LOCKSIZE ANY.
- Row level locking does not always improve concurrency.
- Issue commits in all long-running processes.
- Use different commit frequency depending upon the circumstances.
- Do not hold a cursor across commit if it can be easily and cheaply repositioned.
- Design applications to reduce hot spots.

Archived

12.1 Serialization mechanisms used by DB2

DB2 uses serialization mechanisms to maintain the integrity of the data in DB2 databases. The primary techniques are:

- Latches

  DB2 uses an internal lock called a *latch* in order to serialize access to physical pages of both indexes and table spaces within a single DB2 subsystem. A latch merely prevents two different processes from walking over each other, or reading a page that has been only partially updated.

  Latches are only held for a short duration while the process is accessing the page.

- Locks

  Locks control access to a row, page, table, partition, or table space within a single DB2 subsystem and are managed by the IRLM address space. While it is true that locks are not taken on indexes in DB2, an index-only access may result in locks on the table space to ensure data integrity. A quick refresher on locks can be found in “Locks” on page 329.

- Global locks

  Global locks are used in a data sharing environment to track and negotiate locks between multiple data sharing DB2 subsystems.

  Because the performance of global locks is specific to data sharing environments, it is discussed in Chapter 15, “Data sharing specifics” on page 403.

- Claims and drains

  DB2 uses the claim and drain mechanism to track threads that are accessing the physical data sets of table spaces and index spaces. The claim and drain process enables DB2 to control concurrency across SQL, utilities, and DB2 commands. In “Claims and drains” on page 333, we provide a brief overview of how claims and drains work.

12.1.1 Locks

Not considering LOBs, which we deal with at “LOB locks” on page 332, there are four levels of DB2 locking:

- Table space/partition level
- Table level
- Page level
- Row level

However, because the lowest level is either page or row, and partitioned table spaces can only contain one table, there are either two or three levels of hierarchy, depending upon the type of table space being used. Figure 12-1 on page 330 shows the locking hierarchy for each type of table space. Note that the use of simple table spaces is very limited because of their several disadvantages, not related to locking, when compared to segmented and partitioned table spaces and are listed here only for completeness.

To acquire a lock at any level in the hierarchy, it is necessary to first hold a lock at each higher level. For example, in order for a thread to acquire a lock on a page of a segmented table space, it must first have successfully acquired a lock on the table, and before that, a lock on the table space. The only exception to this rule is the mass delete lock that is taken at the table level when a dirty read occurs on a segmented table space.

Different application processes accessing a DB2 object can use different levels of locking granularity. For example, one process can be reading and locking a table at the page level,

holding an IS lock on the table space and acquiring individual S locks on each page, while another process can issue a LOCK TABLE IN SHARE MODE statement to acquire and hold a single S lock on the entire table space.

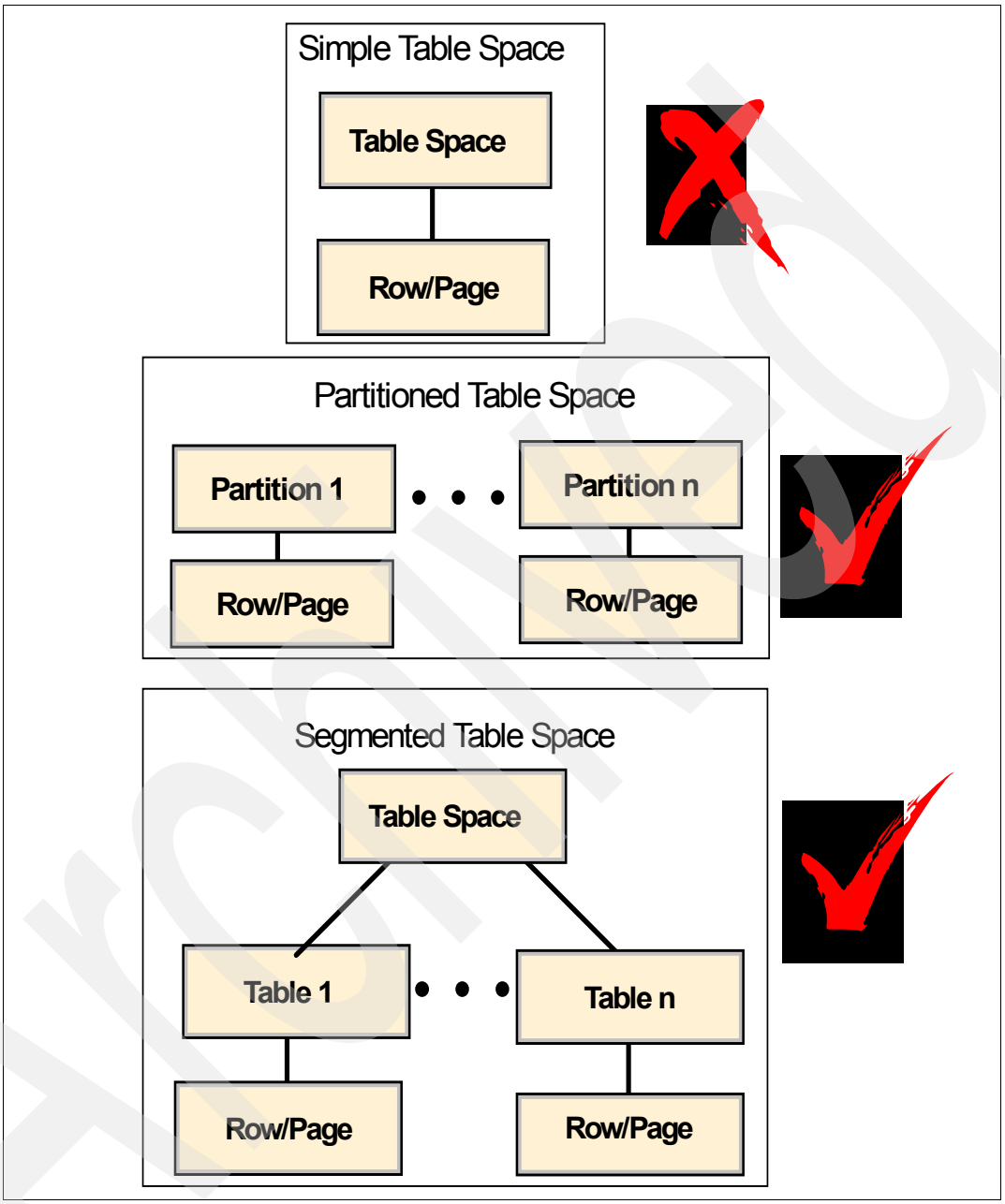


Figure 12-1 Locking hierarchy for the different types of table spaces

Acquiring locks is more expensive than acquiring latches and claims. Lock requests are processed by the IRLM address space, and each lock request requires about 400 instructions for the cross-memory call. You can reduce the number of locks acquired by your application by using various lock avoidance techniques. The simplest is locking at the highest possible level. This is a fine recommendation for a read-only table, but locking at a higher level reduces concurrency when you must support simultaneous read and update access. This is the general tradeoff to keep in mind when designing the application.

Table space, partition, and table locks

A lock is acquired at this level whenever a DB2 table or index is accessed by application SQL, even with ISOLATION UR or index-only access. Table 12-1 lists the possible values of these locks. Each lock type communicates the access intent of the lock holder.

Table 12-1 Table space, partition, and table lock types

Lock	Description
S	Share
U	Update
X	Exclusive
IS	Intent share
IX	Intent exclusive
SIX	Share/Intent exclusive

Page and row locks

At the page and row level there are only three types of locks to communicate the level of access of the lock holder. Table 12-2 shows the possible values.

Table 12-2 Page and row lock types

Lock	Description
S	Share
U	Update
X	Exclusive

Lock compatibility

Besides communicating the access intent of the lock holder, each of these locks also controls the level of concurrent access that is allowed. Table 12-3 shows the lock compatibility matrix for each of the different table space, table, and partition lock types.

The mass delete lock is not shown on these tables. It is a special lock that is taken by threads performing uncommitted reads, or isolation UR access, that is compatible with all other lock types, but will prevent a mass delete from occurring while the table is being read. A mass delete is a DELETE statement without a WHERE clause.

Table 12-3 Table space, table, and partition lock compatibility matrix

	IS	IX	S	U	SIX	X
IS	Yes	Yes	Yes	Yes	Yes	No
IX	Yes	Yes	No	No	No	No
S	Yes	No	Yes	Yes	No	No
U	Yes	No	Yes	No	No	No
SIX	Yes	No	No	No	No	No
X	No	No	No	No	No	No

Archived

Table 12-4 shows the lock compatibility matrix for each of the page and row lock types.

Table 12-4 Page and row lock compatibility matrix

	S	**U**	**X**
S	Yes	Yes	No
U	Yes	No	No
X	No	No	No

LOB locks

The concurrency of LOB table spaces is primarily controlled through the normal transaction locks that are taken on the base table. When locks are not acquired on the base table, such as when accessing the table using uncommitted read, DB2 maintains data consistency by using locks on the LOB table space and the LOB values stored within.

The primary purpose of the locks acquired on the LOB table space and the LOB values are:

- To determine whether space from a deleted LOB can be reused by an inserted or updated LOB.

  The space of a deleted LOB is not reused until there are no more readers (including held locators) on the LOB and the delete operation has been committed.

- To prevent deallocating space for a LOB that is currently being read.

  A LOB can be deleted from the point of view of one application process while another application process is reading the LOB. The reader continues reading the LOB because all readers, including those readers that are using uncommitted read, acquire S locks on the LOB to prevent the storage for the LOB they are reading from being deallocated.

Table 12-5 shows the table space and LOB locks acquired for each different operation that can be performed on LOBs. LOB locks have the same compatibility as page and row locks.

Table 12-5 Locks acquired for operations on LOBs

LOB operation	**LOB TS lock**	**LOB lock**	**Comment**
Read (including UR)	IS	S	Prevents storage from being reused while the LOB is being read.
Insert	IX	X	Prevents others from seeing a partial LOB.
Delete	IS	S	Holds the space in case the delete is rolled back. The base table row or page is locked with an X.
Update	IS → IX	Two LOB locks: An S lock for the delete and an X lock for the insert	Operates as a delete followed by an insert.
Update the LOB to null or a length of zero.	IS	S	Behaves like a delete.
Update a null or zero-length LOB to a value.	IX	X	Behaves line an insert.

12.1.2 Claims and drains

The first access to a table space or index by a DB2 thread causes a claim to be registered with DB2. Claims are acquired at the partition level for partitioned table spaces and indexes. Each claim has a claim class associated with it that is assigned based on the type of access being requested, much like a lock type. Table 12-6 shows the three possible claim classes.

Table 12-6 Claim classes

Claim class	Definition
CS	Read with ISOLATION(CS)
RR	Read with ISOLATION(RR)
WRITE	Delete, Insert, or Update activity

These claims are used to keep track of which threads are accessing each DB2 resource. When a utility needs to prevent certain types of access to a resource, it issues a drain of that claim class. A drain will wait for all existing claims of that type on the resource to be released, and will prevent new claims of that type from being acquired on the resource until it has successfully completed or it has timed out.

Not all utilities require drains, and not all claimers are applications. All of the SHRLEVEL(CHANGE) utilities are claimers, and, of them, only REORG requires a drain during the last iteration of the log apply and the switch phase. SHRLEVEL(REFERENCE) utilities will drain the WRITE claim class for the duration of the utility execution and SHRLEVEL(NONE) utilities will drain all claim classes.

12.1.3 The effect of table space parameters on locking

A great deal of DB2's locking behavior is controlled by parameters that are defined on the table space. In this section, we discuss the LOCKSIZE and LOCKMAX parameters and how they affect locking. We also look at how the MAXROWS table space parameter can be used to improve concurrency.

The LOCKSIZE parameter

The locksize parameter of the table space identifies the lowest level lock possible on the table space. Figure 12-1 on page 330 shows the locking hierarchy of simple, segmented, and partitioned table spaces.

LOCKSIZE TABLE and TABLE SPACE

For certain table spaces and applications the total number of locks taken can be reduced by specifying that a single lock should be taken on the table or table space. LOCKSIZE TABLE can only be specified for segmented table spaces. The use of LOCKSIZE TABLESPACE for a segmented table space depends on the number of tables and their use. If there is only one table in a segmented table space, then LOCKSIZE TABLE and LOCKSIZE TABLESPACE are effectively the same thing. But generally a segmented table space has a locking hierarchy at the table space level and it is not always advisable to lock at the table space level when there are multiple tables in the table space.

LOCKSIZE TABLE or TABLESPACE should be considered for read-only tables, small code tables that rarely change, and tables in which the rows represent a set of data that is always updated simultaneously.

If the table is truly read-only, and the application uses isolation CS with CURRENTDATA(NO), using LOCKSIZE TABLE or TABLESPACE will not actually save any locking since all locks will

be prevented anyway by lock avoidance. However, defining the table space in this way might protect you from applications that use a different isolation level. It might also bring any updaters to your immediate attention as the table, not truly read-only, will likely suffer from concurrency problems.

LOCKPART

In a partitioned table space, locks are obtained at the partition level. Individual partitions are locked as they are accessed. This locking behavior enables greater concurrency because gross locks (S, U, or X) can be obtained on individual partitions instead of on the entire partitioned table space. Partition locks are always acquired, even if the table space was defined in a version of DB2 prior to Version 8 with the LOCKPART NO clause. For LOCKPART NO table spaces, DB2 no longer locks the entire table space with one lock when any partition of the table space is accessed.

LOCKSIZE PAGE

Specifying locksize page indicates that DB2 should perform locking at the page level. This is typically the best choice for performance and concurrency, and is our general recommendation.

Recommendation: Use LOCKSIZE PAGE as the default and only consider LOCKSIZE TABLE, TABLE SPACE, or ROW as needed.

LOCKSIZE ROW: Row-level locking

Locking at the row level can improve the concurrency of a properly designed application, but it is also the most expensive level of locking in DB2.

If locks cannot be avoided, the volume of locks at the row level can be significantly higher than at the page level. In addition, in a data sharing environment, the physical consistency on a page is guaranteed through the use of a page P-lock, the data sharing latch. Multiple members accessing and updating at the row level on the same page will cause increased cross-invalidation of the page. To achieve the concurrency of row-level locking without the increased data sharing overhead, you should consider the use of MAXROWS 1 to force each row to its own page as described in “ISOLATION levels” on page 336.

The concurrency benefits from row-level locking are achieved when multiple processes need to access different rows on the same page. There is also a benefit of improved clustering when multiple insert processes are able to insert rows simultaneously into a target page with sufficient free space. However, if your application is suffering from a row- level hot spot or deadlocks caused by accessing the data in a different order in two or more processes, using row-level locking may only incur additional locking overhead without improving concurrency.

LOCKSIZE ANY

Specifying LOCKSIZE ANY lets DB2 decide at which level to take locks. In most cases, DB2 will choose to take locks at the page level, but in some cases specific access paths may cause DB2 to choose to lock at the next higher level, which is at the table, table space, or table space partition level, depending upon the type of table space. DB2 never chooses row-level locking. The question is, “Why would you want to let DB2 make your locking decisions for you?”

Recommendation: Do not define table spaces with LOCKSIZE ANY. Set the locksize of each table space to a level that is appropriate for most of your accesses. If a higher level of locking would be beneficial for a specific application, code a LOCK TABLE statement within that application.

The MAXROWS parameter

The MAXROWS parameter allows you to specify the maximum number of rows that can be placed on a single page of this table space. *Unlike PCTFREE, this parameter is honored by all DB2 processes that insert data, including INSERT statements, LOAD utilities, and REORG utilities.*

Using MAXROWS to reduce the number of rows on each page reduces the likelihood of lock contention on that page. In particular, MAXROWS 1 ensures that only one row is placed on each page, which effectively results in row-level locking with a LOCKSIZE of PAGE.

Be sure to consider the effect of using MAXROWS on the size and space management of your table space because reducing the number of rows per page increases the number of pages required to store the data. Also consider that the use of MAXROWS 1 eliminates all benefit of compression.

Tip: If you specify MAXROWS 1, there is only overhead in specifying COMPRESS YES.

The LOCKMAX parameter and lock escalation

The LOCKMAX parameter specifies the maximum number of locks that can be taken on this table space by a single process before lock escalation occurs. LOCKMAX SYSTEM means that the value specified in the NUMLKTS DSNZPARM should dictate the maximum number of locks before escalation occurs for this table space. LOCKMAX 0 specifies that lock escalation should not occur for this table space.

When lock escalation occurs, DB2 requests a single lock at the next higher level in the lock hierarchy of the table space. The lock requested is equal to the most restrictive lock held by the process on the table space. If this lock is acquired, then all of the lower level locks that were held by the process are released. If the higher level lock cannot be acquired before the timeout value has been reached, then the process times out, all updates are rolled back and all locks are released.

Referring back to Figure 12-1 on page 330, remember that there is no locking hierarchy at the table space level for a partitioned table space. The highest level of locking is at the partition level. Therefore, when the LOCKMAX threshold is reached, all partitions on which your process currently holds locks are escalated to a gross lock. Any unlocked partitions that are subsequently accessed will also be locked with a gross partition level lock.

Lock escalation is a very expensive process, both in terms of lock management and concurrency. In today's high availability environment, there is no reason to allow lock escalations to occur. The benefit of lock escalation is a safeguard by DB2 in reducing a large number of locks held in the IRLM on behalf of poorly designed applications that are not releasing their locks.

Lock escalation also prevents poorly designed applications from reaching the maximum number of locks per user specified by the NUMLKUS DSNZPARM. From an individual application perspective, this may seem like a good thing, but from an overall availability and performance standpoint, it is merely rewarding bad behavior, or at a minimum failing to apply proper discipline. The focus now should be on identifying and correcting those poorly designed applications that take excessive numbers of locks. Besides the recommendations in this section, we show several ways to improve these applications in "Unit of work" on page 347.

See also "Locking (F)" on page 393 for information about how to monitor your system for lock escalations.

Recommendation: Always use LOCKMAX 0 to disable lock escalation. If you can afford to lock the whole table or partition, specify a LOCKSIZE of TABLE or TABLESPACE, or issue a LOCK TABLE statement in the application.

LOB considerations

An application working with LOBs may be more likely to accumulate a large number of locks, because locks are put on each LOB column in a logical row. Therefore, an insert of one logical row that contains three LOB columns will acquire four table space locks, three LOB locks, and one page or row lock on the base table. In addition, LOB locks are taken even when using an isolation level of UR.

For these reasons, be careful to commit frequently when using LOBs to avoid reaching the NUMLKUS limit and prevent lock escalations.

12.1.4 ISOLATION levels

Every application process accessing DB2 has an isolation level. The isolation level is controlled by the ISOLATION parameter on the BIND of the plan or package as described in “ISOLATION” on page 292. The ISOLATION can also be overridden at the statement level through the use of the WITH clause as you see in Example 12-1.

Example 12-1 Overriding the isolation level of a single SQL statement

```
SELECT FIRSTNME,
       LASTNAME,
       WORKDEPT
  INTO :HV-FIRSTNME,
       :HV-LASTNAME,
       :HV-WORKDEPT
  FROM EMP
 WHERE EMPNO = :HV-EMPNO
  WITH UR
```

The isolation level tells DB2 the level of data stability that your application requires for each particular access, which has a significant effect on the locking that will occur. Specifying the inappropriate isolation level can cause unnecessary problems ranging from data inconsistency to deadlocks and timeouts. This having been said, the majority of application processes can achieve high performance and availability using a single isolation level.

Remember that rows or pages that have been modified using DELETE, INSERT, or UPDATE will always be locked with an X lock regardless of the isolation level. Therefore, the isolation level is only controlling the behavior of locking when issuing SELECT statements.

As a general rule, best overall performance and availability can be achieved using ISOLATION(CS) and CURRENTDATA(NO) on the bind. The other isolation levels should be used only as needed, and should be specified using the WITH clause on the individual SQL statement. Binding an entire program with one of the other isolation levels introduces the likelihood of more restrictive locking or more chance of data inconsistency than necessary. The following sections discuss each of the possible isolation levels in detail, and provide examples of when the possible isolation levels might be useful.

There are four values that can be specified in the ISOLATION parameter of the bind or in the WITH clause of an SQL statement: UR, CS, RS, and RR. Before each of these are discussed in detail, it is necessary to point out there is an additional bind parameter that affects the locking of the process call: CURRENTDATA. This parameter only applies to an ISOLATION of

CS, and it can have a value of YES or NO. In effect, this adds a fifth isolation level. Unfortunately, the CURRENTDATA parameter *cannot* be overridden at the SQL statement level.

UR: Uncommitted read

Uncommitted read is the least restrictive of the isolation level. Processes accessing DB2 data with this isolation level can read data from pages locked by other processes that has been changed but not yet committed. While DB2 uses latching to ensure that your process does not read data that is inconsistent across the page, there is no guarantee that values read from the data page match the key values in the index or that the data you read will ever be committed.

This introduces the possibility of two types of data inconsistency.

- Figure 12-2 shows the inconsistency that can occur when reading an updated row that is subsequently rolled back.

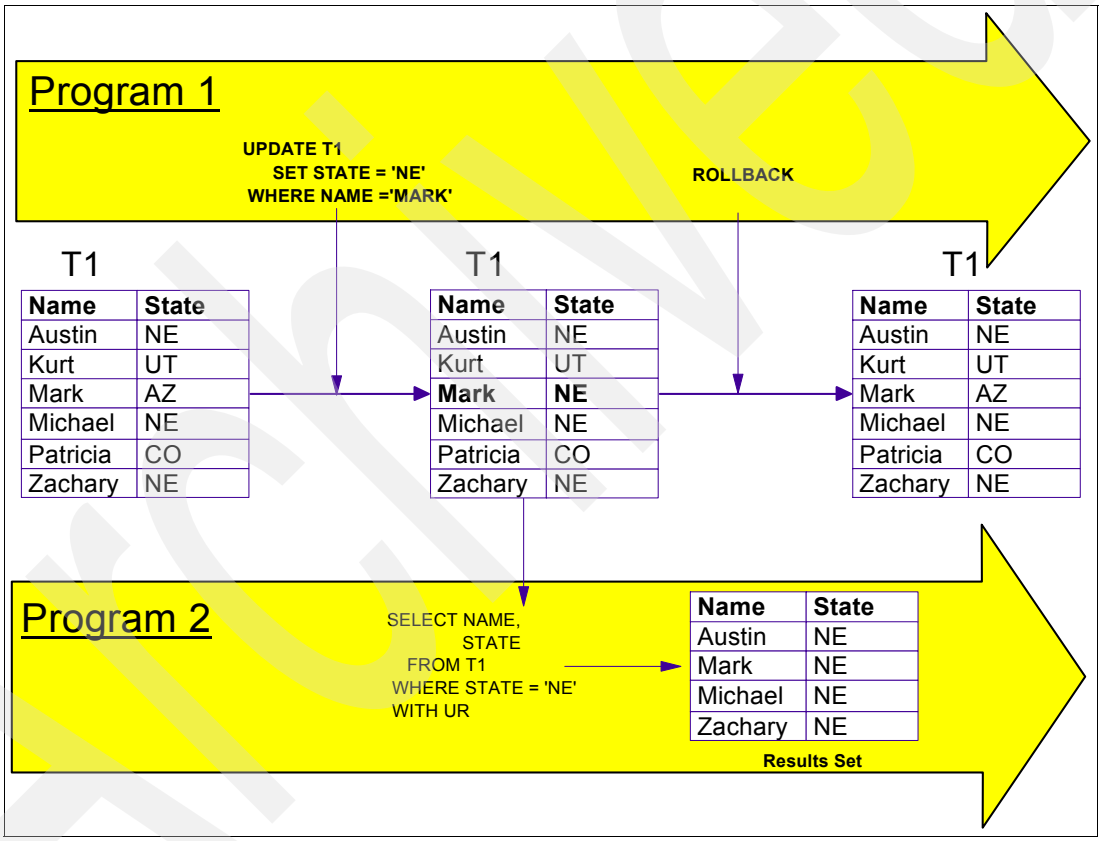


Figure 12-2 Data inconsistency caused by rollback

- Figure 12-3 on page 338 show an inconsistency that can occur when a row is qualified through the index using an updated column.

  The index contains the new value of the column, but the data page still contains the old value. The query will return a DEPTNO value of 10 even though the query has a predicate that specifies that only rows with DEPTNO = 20 should be returned.

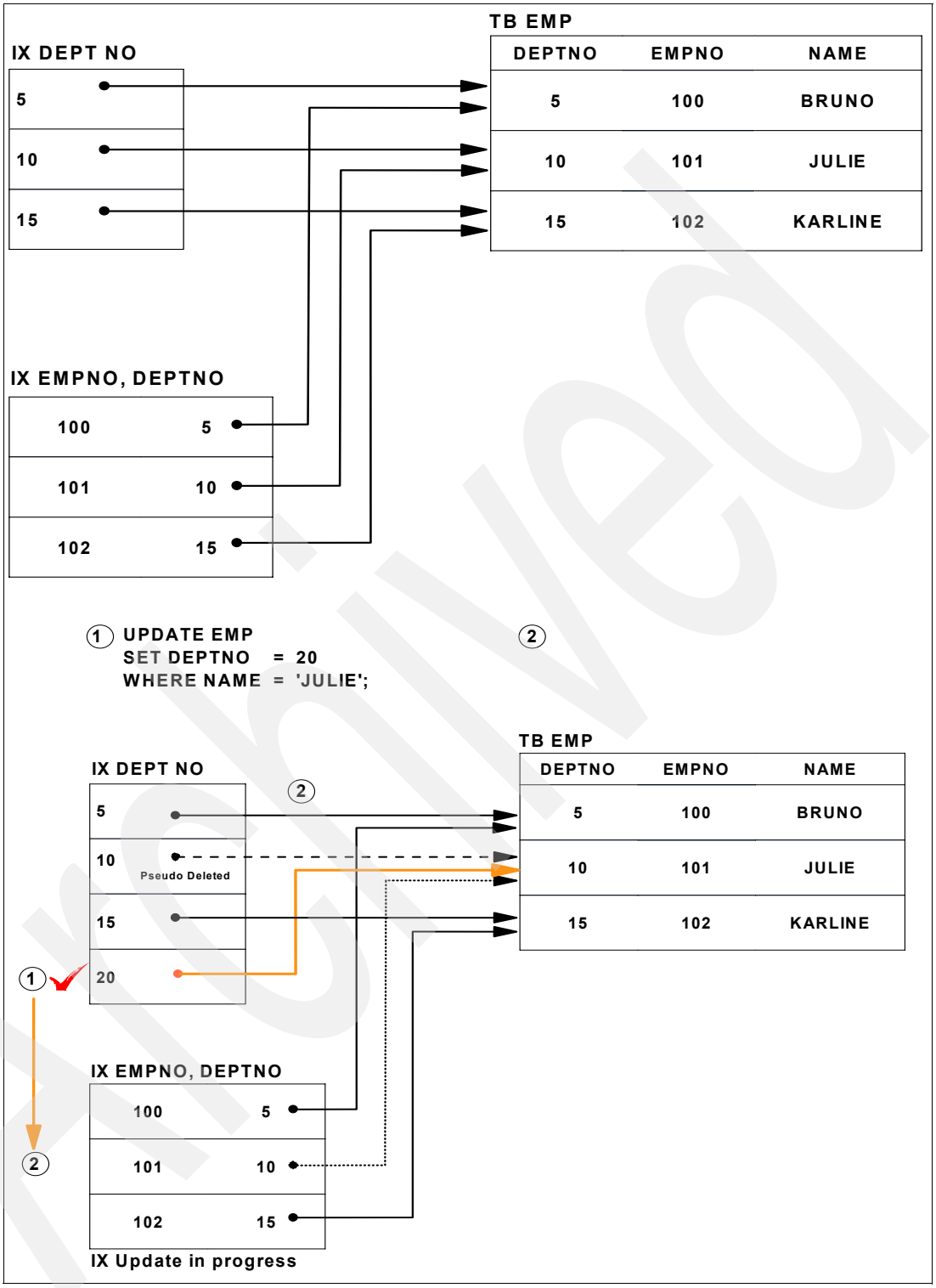


Figure 12-3 Data inconsistency between index and data page

Because of these potential data inconsistencies, you should only use uncommitted read when the inconsistencies will not adversely affect the integrity of your application. In particular, be wary of selects that are qualifying data using indexed columns that can be updated.

While UR can significantly improve throughput for applications that are experiencing a lot of lock contention, in the majority of cases there is little or no performance improvement using UR over using CS with CURRENTDATA(NO) with a large amount of lock avoidance.

Our recommendation is to only use UR on individual statements that are known to experience lock wait time and are not excluded from using UR by any of the data inconsistency issues described above.

CS with CURRENTDATA(NO): No dirty data

Specifying CURRENTDATA(NO) with ISOLATION(CS) tells DB2 that you do not want a read-only cursor to read dirty or uncommitted data, but you also do not want to take any locks. With this combination, DB2 will only request an S lock on a page or row if lock avoidance fails. Once your lock has been granted, you can safely read the page or row. As soon as your cursor moves off of the row or page the lock is released. This process of only taking a lock when it is required is referred to as lock avoidance, and is discussed in more detail in 12.1.5, “Lock avoidance” on page 340.

ISOLATION(CS) and CURRENTDATA(NO) is the recommended isolation level for most applications. All data read will be consistent and committed, and locks are not taken unless they are needed for serialization on the row or page.

CS with CURRENTDATA(YES): Cursor stability

Specifying CURRENTDATA(YES) with ISOLATION(CS) informs DB2 that you want to lock the current row or page being referenced by the cursor with an S lock. In this case, DB2 will always request a lock each row or page as it is fetched into the cursor. As soon as the cursor moves off of the row or page the lock is released.

By holding an S lock on the current row of the cursor you are preventing other processes from updating it. This is a desirable behavior if you are going to use the data to perform an insert or update in the same unit of work. However, a better way to achieve the same result is to code the FOR UPDATE OF clause on the cursor.

When coded with the FOR UPDATE OF clause, the locking behavior of CS with CURRENTDATA(NO) and CS with CURRENTDATA(YES) is identical.

The recommendation is to use ISOLATION(CS) with CURRENTDATA(NO), and code FOR UPDATE OF on cursors that need to prevent updates to the currently positioned row or page.

RS: Read stability

An isolation level of RS causes DB2 to acquire an S lock on each row or page containing a row that qualifies for your result set based on stage 1 predicates. All rows or pages containing rows for which stage 2 predicates must be evaluated will be locked even if the row is rejected by the stage 2 predicate. These locks will be held until commit so that no other process can change the rows that have been fetched into the application.

The use of ISOLATION(RS) greatly increases overall lock duration both in terms of the number of locks acquired and the length of time that they are held. Only use RS at the cursor level and when it is necessary to lock the contents of an entire result set. Place these cursors as late in the unit of work as possible, and commit as soon afterwards as possible to reduce the amount of time the locks are held.

RR: Repeatable read

When using an isolation level of RR DB2 will acquire a row or page lock on all accessed pages or rows, qualifying or not. In addition, it will prevent the insert of new rows that would

qualify for the cursor. These locks will be held until commit, and will be released at commit even if the WITH HOLD clause is specified on the cursor.

ISOLATION(RR) is the most restrictive of the isolation levels, and can have several nasty side effects. When used to read a table space with LOCKSIZE ANY specified, RR sometimes causes the optimizer to choose table or table space level locking at bind time. Used improperly, ISOLATION(RR) can increase the potential for deadlocks.

12.1.5 Lock avoidance

Lock avoidance is a process by which DB2 internally tries to avoid acquiring locks on the condition that it is not needed to ensure data correctness (committed data) when delivering data to the application. Traditionally, lock avoidance has been used for unambiguously read-only cursors that are bound with ISOLATION(CS) and CURRENTDATA(NO), but new lock avoidance techniques have been applied to other isolation levels.

For an unambiguously read-only cursor operating with ISOLATION(CS) and CURRENTDATA(NO) locks are avoided by only requesting an S lock on a qualifying row or page if that row or page contains potentially uncommitted data. Figure 12-4 shows a visual depiction of the lock avoidance process.

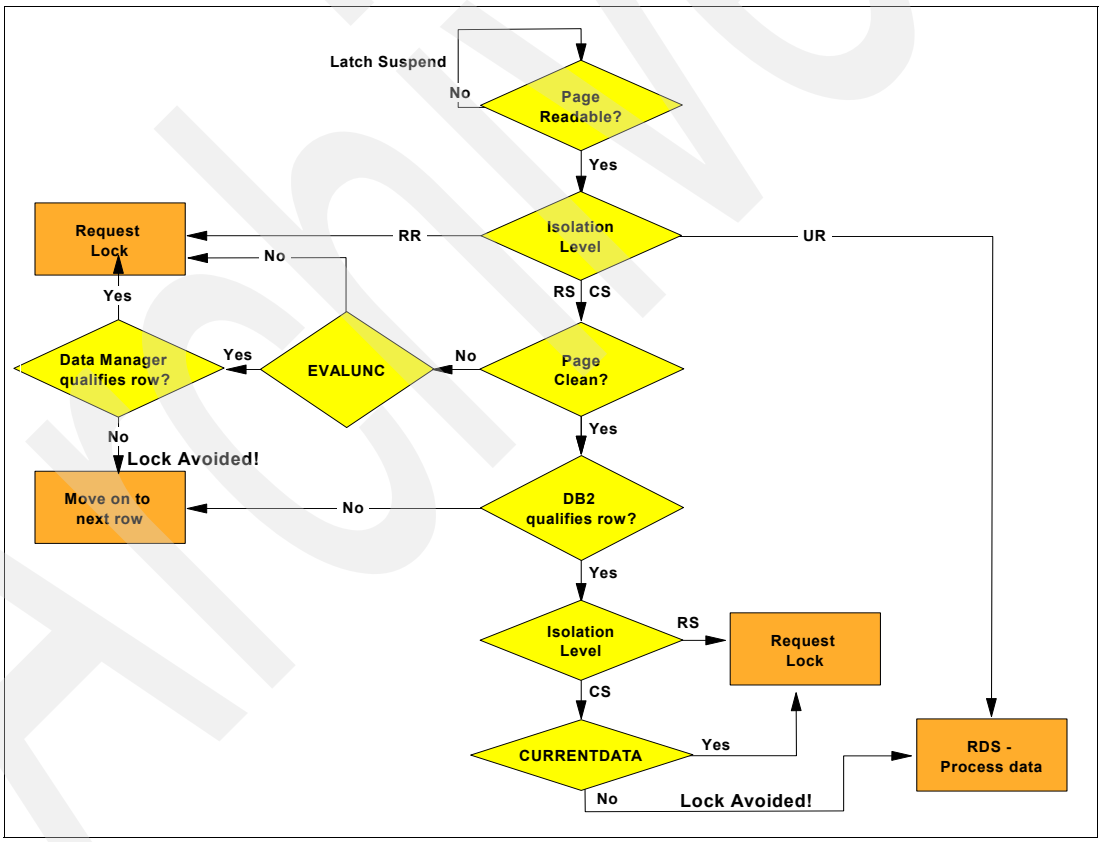


Figure 12-4 Lock avoidance

With the introduction of the EVALUNC DSNZPARM in DB2 Version 7 it is possible to avoid locks by evaluating the stage 1 predicates against locked rows or pages. If no qualifying rows are found, the lock can be avoided. This lock avoidance technique benefits isolation levels of CS and RS regardless of the CURRENTDATA value. For more information, refer to “EVALUNC DSNZPARM: Evaluate Uncommitted” on page 316.

In DB2 V8, lock avoidance was expanded to singleton selects with ISOLATION(CS) and CURRENTDATA(YES).

In order to get the benefit of lock avoidance, it is important to code cursors unambiguously. This is especially true for dynamic SQL. Coding a cursor unambiguously means explicitly specifying either FOR FETCH ONLY for read-only cursors or FOR UPDATE OF for cursors whose data will be used to perform subsequent updates.

For more information about the effect of lock avoidance in a data sharing environment, see “Maximize lock avoidance” on page 409.

12.2 Lock suspensions, timeouts, and deadlocks

An application process is suspended when it requests a lock on a resource that is already locked by another process with an incompatible lock type. The suspended process temporarily stops running. Deadlocks and timeouts are necessary processes to control lock suspensions. While its true that we never like to terminate a DB2 thread and force a rollback, it is a far better option than letting a process wait indefinitely for a locked resource.

It is important to monitor the amount of lock suspensions, timeouts and deadlocks that are occurring in your system. See “Locking (F)” on page 393 for information about how to monitor this activity in your system.

12.2.1 Timeouts

The amount of time that a thread will wait for a specific lock before timing out is controlled by the IRLMRWT DSNZPARM, which is described in “IRLMRWT DSNZPARM: Timeout value” on page 315. Unlike a deadlock, it is important to allow your applications to wait a certain amount of time before declaring a timeout. However, there can only be *one timeout value* specified for the DB2 subsystem. This value typically needs to be low enough so that the most critical applications will not wait too long before timing out. All other applications in the subsystem must be good citizens and adjust their unit of work scope and commit frequency accordingly, see “Unit of work” on page 347.

Because a timeout is only detected on a single lock wait it is possible that a process could have lock wait time that is much longer than the timeout value without timing out when multiple locks are requested.

12.2.2 Deadlocks

A deadlock occurs when two or more processes suspend with each holding the lock requested by the other. Figure 12-5 on page 342 shows an example of a deadlock condition between two threads.

It is important to detect deadlocks as soon as possible, as a deadlock situation will never resolve itself. If your system is experiencing a high volume of deadlocks, consider reducing the interval between deadlock detection cycles. However, the deadlock detection process can cause latch suspensions, so it shouldn’t run as often for systems that are not experiencing many deadlocks. The deadlock detection cycle is controlled by the IRLM PROC parameter DEADLOK which is described in “IRLMRWT DSNZPARM: Timeout value” on page 315.

It is possible for two processes to be running on separate DB2 subsystems in a data sharing group, each trying to access a resource held at the other location. In that case, neither

subsystem can detect that the two processes are in deadlock; the situation is resolved only when one process times out.

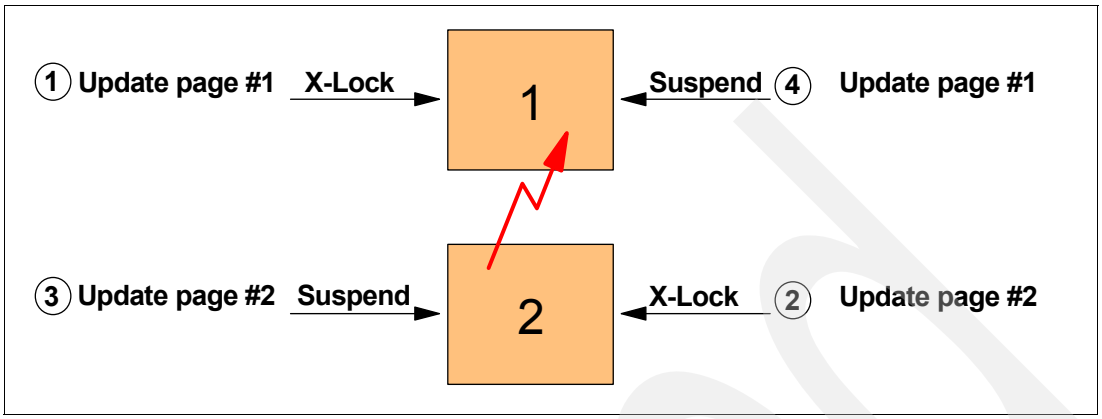


Figure 12-5 Deadlock situation

Utility contention

Deadlocks between application programs and utilities will not be detected by IRLM. Typically, applications receive SQLCODE -911 (lock timeout). Utilities could end with RC = 8 and reason-code 00C200EA.

12.2.3 Deadlock avoidance

Deadlocks tend to occur in hot spots. First, know where hot spots occur and consider cooling down the hot spots to avoid deadlocks. See “Hot spots” on page 353 for a discussion of hot pages and rows.

As a general rule, consider the following procedures to avoid deadlocks:

- Reduce lock duration using the recommendations at “Lock duration” on page 345.
- Commit as frequently as possible using the recommendations at “Commit frequency” on page 350.
- Use ordered updates on different kinds of resources that cannot detect each other's deadlock situations, for example, between DB2 and IMS.
- Use ordered updates within the same kind of resources, for example, in a DB2 subsystem, although care should be taken in multi-table SQL statements in concurrent programs where the join order can change between binds.
- Consider row-level locking if different applications are accessing different rows.
- Code cursors unambiguously specifically stating FOR FETCH ONLY when appropriate therefore acquiring less restrictive locks, per “Lock avoidance” on page 340.

Code FOR FETCH ONLY for read-only cursors

If you know that this cursor is used only for reading, notify DB2, so that it can take advantage of the situation. Do not code your cursor to be ambiguous if it does not need to be.

Code FOR UPDATE OF for updatable cursors

Specifying FOR UPDATE OF in a cursor communications to DB2 your intent to update or delete the rows that you read from the cursor. Figure 12-6 on page 343 shows how coding the FOR UPDATE OF clause can avoid a deadlock situation.

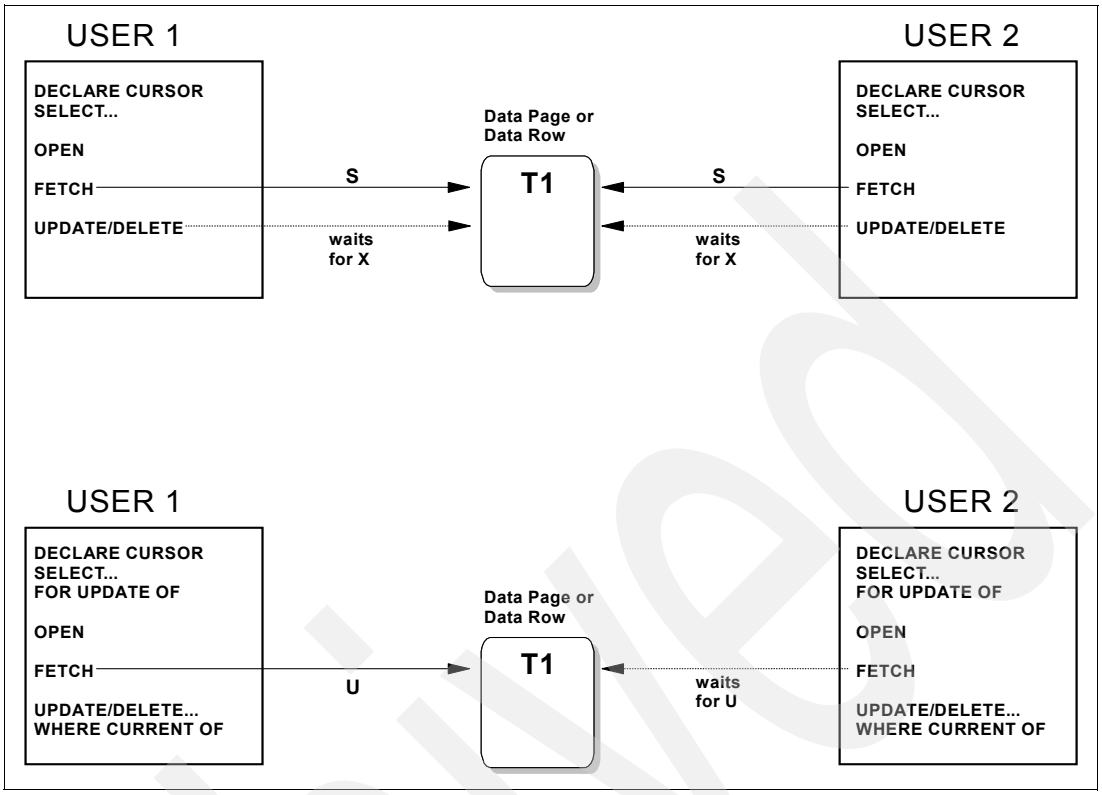


Figure 12-6 Using FOR UPDATE OF to acquire a U lock

The cursor in the top half of Figure 12-6 is coded without the FOR UPDATE OF clause. Both USER 1 and USER 2 fetch a row (which acquires an S lock) and then update or delete the row.

This creates a deadlock because:

- USER 1
  - Has an S lock on T1 (data page or row)
  - Waits for an X lock on T1 (data page or row)
- USER 2
  - Has an S lock on T1 (data page or row)
  - Waits for a X lock on T1 (data page or row)

The cursor in the bottom half of Figure 12-6 has the FOR UPDATE OF clause coded. Both USER 1 and USER 2 fetch a row and then update or delete the row.

There is no deadlock situation because U lock is used:

- User 1
  - Has a U lock on T1 (data page or row)
- User 2
  - Waits for a U lock on T1 (data page or row)

Restriction: This technique cannot be used when ordering is needed because the FOR UPDATE OF and the ORDER BY clauses cannot be specified in the same cursor.

Archived

Deadlocks due to unordered updates

When different application processes perform update or delete operations in different sequence at the same time it is likely that deadlocks occurs. Figure 12-7 shows two deadlocks caused by applications that:

- Update tables in a different order
- Update rows inside one table in a different order

These deadlocks can occur with both page level locking and row level locking.

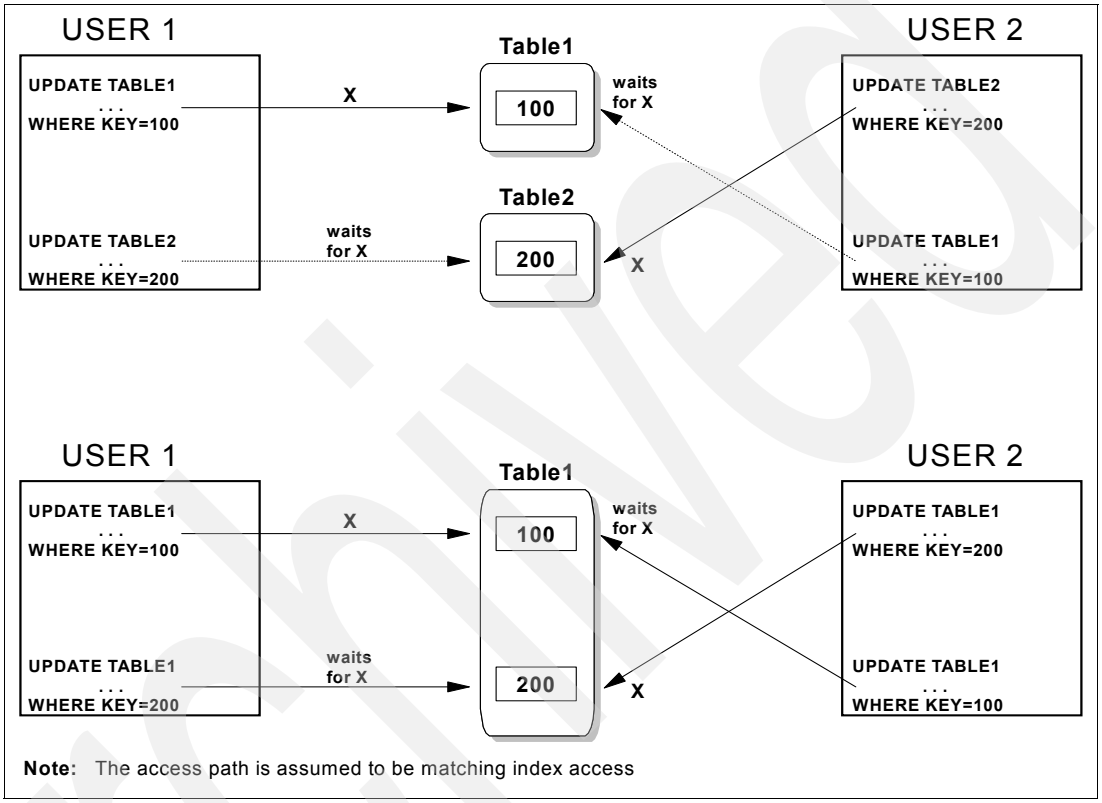


Figure 12-7 Deadlocks due to unordered updates

Deadlocks due to unordered updates and deletes can be avoided by:

- Ordered access to the different tables

  For parent and dependent tables that will both be updated or deleted in an application process the dependent table should be processed first. This is because DB2 enforced RI will require that the child rows are deleted first in the delete process.

  In other cases always using an arbitrary order, such as alphabetical order, might be appropriate.

- Ordered access to the rows within a table

  Always performing updates and deletes in the order of the clustering index is one way to reduce the potential for deadlock and possibly achieve better performance from dynamic prefetch.

12.2.4 Application action after a deadlock or timeout

The typical SQLCODE returned to the victim of a deadlock or timeout is a -911, which indicates that the current unit of work has been rolled back to the last commit point and all

locks have been released. In some environments, it is possible for the victim of a deadlock or timeout to receive an SQLCODE of -913, which indicates that no rollback has been performed. In “Intelligent exception handling” on page 251 we discuss how your application should handle these and other error situations.

12.3 Lock duration

The primary objective of tuning application locking is to reduce overall lock duration while still maintaining the integrity of the data in the database and your application. By reducing the durations of locks held by one process it is possible to reduce the amount of lock suspensions of all other processes accessing the same DB2 objects. There are four components to overall lock duration, we discuss them individually.

12.3.1 When will the lock be acquired?

The acquisition of DB2 locks when executing a DBRM that is bound into a plan is controlled by the ACQUIRE bind parameter. If ACQUIRE(ALLOCATE) is specified all table and table space level locks required for SQL statements in the plan are acquired at thread allocation. If ACQUIRE(USE) is specified the locks will be acquired at the time the statement that requires the lock is executed. For more information about the ACQUIRE bind parameter, see “ACQUIRE” on page 290.

For packages, DB2 always acquires locks at the execution of the statement that requires the lock. Therefore, the only thing your application can do to control the acquisition of locks is to position the SQL statements in your application in an order that acquires the most restrictive locks as late as possible within each unit of work.

As an example, Figure 12-8 shows the same series of statements being executed in two different sequences.

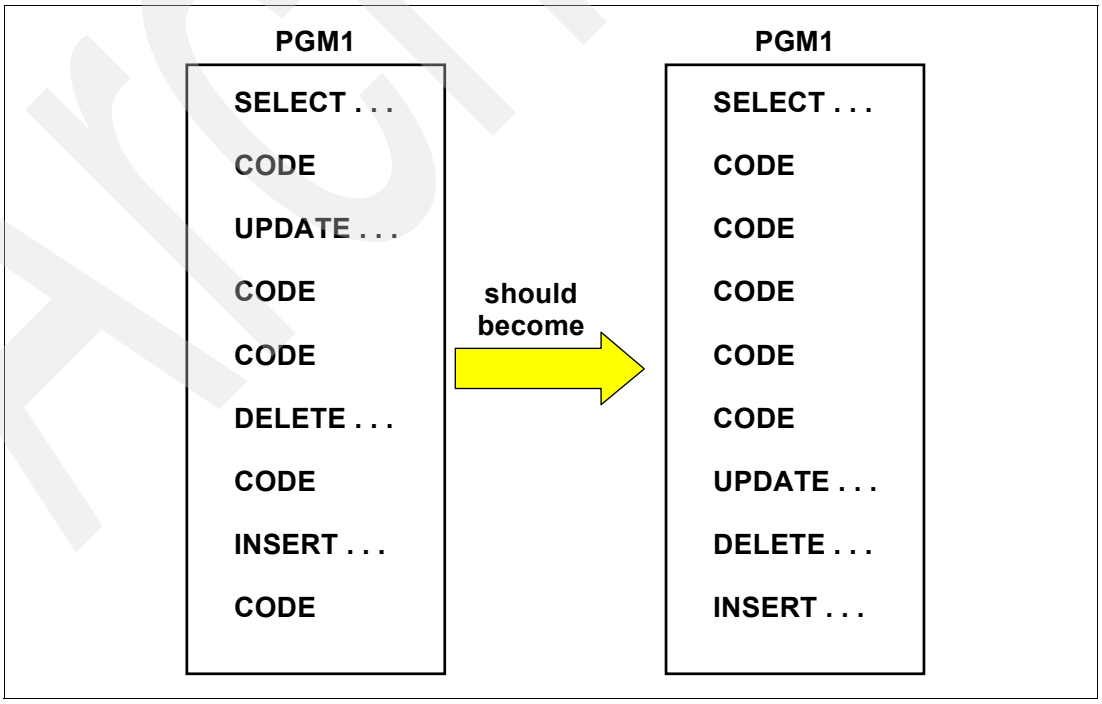


Figure 12-8 Changing processing sequence to reduce lock duration

By arranging the sequence of statements so that the DB2 updates are performed at the end lock duration can be reduced.

An additional benefit of this processing order might be that no database updates occur until all edits have been completed, which eliminates the need for a rollback in the event of an edit failure.

12.3.2 Will a lock be required?

A lock is always required for DELETE, INSERT and UPDATE statements, so this question is only for SELECT activity.

- Can this application afford to read uncommitted data?

  If your application can afford to read uncommitted data (data that has been updated that will potentially be rolled back) then you can avoid taking locks for read activity by using an isolation level of UR. The benefits and dangers of using isolation UR are discussed in the paragraph entitled “UR: Uncommitted read” on page 337.

- Can this application afford to have the row just read immediately updated?

  Assuming that your application requires that each row read must be clean and committed, you next must consider whether your application cares if the row you just read is updated within your unit of work.

  If the data read from the row is going to be used for a subsequent update, then you will want to tell DB2 to lock the page or row to prevent another process from updating the row in the middle of your unit of work. The recommended way to accomplish this is by using the SQL clause FOR UPDATE OF. Another way is using isolation of CS with the CURRENTDATA(YES) option specified as described in the paragraph entitled “CS with CURRENTDATA(YES): Cursor stability” on page 339.

12.3.3 How restrictive will the lock be?

In other words, how much of the table will you lock?

- Can this application afford to have new rows inserted that qualify for the result set while it is being read?

  If not, an isolation level of RR, or repeatable read, will lock all of the rows that qualify for the cursor and will not allow new rows that would qualify to be inserted. In “RR: Repeatable read” on page 339 we discuss this isolation level, but if your application really needs this level of locking consider issuing a LOCK TABLE IN SHARE MODE before issuing your query.

- Can this application afford to have any part of the result set updated or deleted while it is being read?

  If not, an isolation level of RS, or read stability, will lock all of the rows that qualify for the cursor. Before using this isolation level consider the options discussed in paragraph “RS: Read stability” on page 339.

If neither of these cases apply, then an isolation level of CS, or cursor stability, will probably satisfy your applications locking requirements. See “ISOLATION levels” on page 336.

12.3.4 When will the lock be released?

Exclusive locks will always be held until a COMMIT is issued, at which time they will always be released. S and U page and row locks taken by SELECT statements will also be released at COMMIT unless the RELCURHL DSNZPARM is set to NO. However, if an isolation level of

CS is used S and U locks that were acquired when a row was fetched from a cursor are released as soon as the cursor moves to the next row or page.

The duration of table, table space and partition locks are controlled by the RELEASE parameter on the bind. If RELEASE(COMMIT) is specified, then the locks are released at commit unless there are held cursors or held locators on the table, table space or partition. If RELEASE(DEALLOCATE) is specified the table, table space and partition level locks are released at thread termination. For more information about the RELEASE bind parameter see "RELEASE" on page 291.

Recommendation: To reduce lock duration, favor the use of ISOLATION(CS) with CURRENTDATA(NO), RELEASE(COMMIT) and issue COMMIT statements at appropriate intervals.

12.4 Unit of work

A logical unit of work in an application is a set of SQL statements that make up a business process. At any point during the logical unit of work a failure will require the rollback of all previous updates in that logical unit of work in order to keep the database in a state that is consistent with the business rules. At the end of the logical unit of work the data manipulated by the business process should be in a state that is consistent with the business rules.

In DB2 a unit of recovery is the set of UPDATE, DELETE and/or INSERT statements that are performed from one point of consistency to another by a single application process. A unit of recovery begins with the first change to the data after the beginning of the job or following the last point of consistency and ends at a commit point.

In order to ensure that the data in a database remains in a state that is consistent with the business rules the logical unit of work should be no larger than the DB2 unit of recovery. However, in some cases the logical unit of work cannot be completed within a single DB2 unit of recovery. An example would be a process that requires three online transactions to complete a business function that is considered a single logical unit of work. Because each online transaction ends with a commit, the rows that have been added or altered in the first transaction are exposed to other processes without being locked. This may also be the case when multiple batch job steps or even multiple batch jobs are required to complete a logical unit of work. In either of these cases there is also the dilemma of how to handle an abend in a subsequent unit of recovery after previous units of recovery have already been committed.

One way of solving the problem is by adding application-managed locking information to the table which indicates that the current data is inconsistent. This solution has two disadvantages:

- Every SQL statement that uses the table must verify the status of the application lock even when just retrieving the data
- It does not solve the problem of the abend situation, because you do not have a before image of the data.

A better solution is to use the techniques described in "Work in progress tables" on page 249 and "Restricting an application's data access" on page 260 to control access to the data until the logical unit of work completes.

For best performance it may be advisable to group multiple logical units of work into a single DB2 unit of recovery. When this is done, savepoints (which are discussed in "Savepoints" on page 352) can be used to allow the application to rollback a single logical unit of work.

The diagram in Figure 12-9 shows the relationship between a DB2 unit of recovery and a logical unit of work.

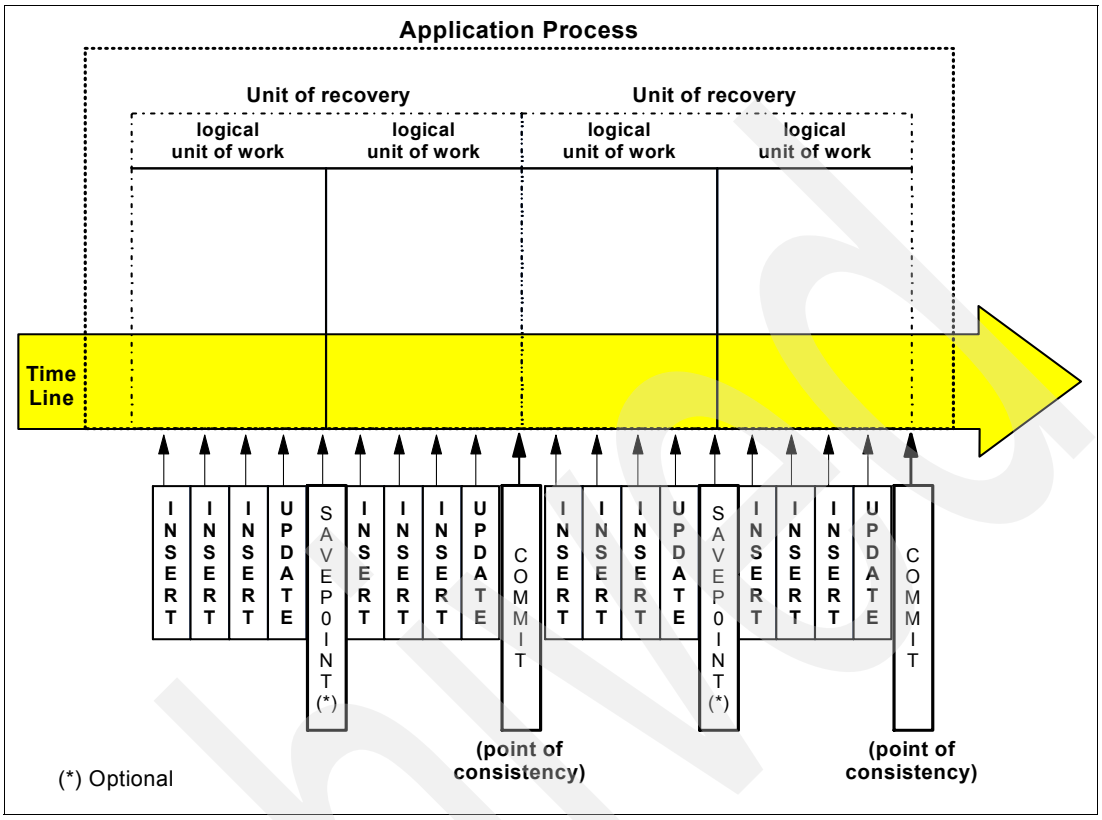


Figure 12-9 Logical units of work and DB2 units of recovery

12.4.1 Commit

All batch application processes, as well as any other application processes that acquire locks and run for more than a few seconds should perform commits periodically. The following points summarize the reasons for performing commits:

- Any processes that are updating DB2 are acquiring locks on objects that may prevent other processes from accessing these objects. If these locks are held for an un-necessarily long period of time, concurrency will suffer. Excessive long lock duration leads to deadlock and timeout.
- Any processes that are accessing DB2, including read-only applications, are acquiring claims on objects that may prevent drains from completing. Drains are typically issued by utilities that must be run on the database objects in order to keep them continuously available.
- In a well designed environment most read-only applications do not acquire page or row level locks because of lock avoidance. An increase in the number or duration of X locks on pages and rows in the system will reduce the amount of lock avoidance. Acquiring additional locks will add to the elapsed time of read-only transactions and increase the workload in the IRLM address space leading to overall system performance degradation.
- Whenever a batch program abends or a system failure occurs DB2 backs out data changes that have not been committed. If no commits are taken then the amount of data to be backed out may be large and may take a long time. In addition, upon restart the

program will have to reapply all of the updates. More frequent commits reduces the amount of data that must be rolled back and then reapplied on restart.

Committing in a batch process requires that the application be designed for restart. See “Restart using sequential input and output files” on page 283 for detailed information regarding how to design an application for restart.

- When updating a table space that has a LOCKMAX other than 0 specified, it is important to commit before reaching the maximum number of locks on the table space to avoid lock escalation. You also want to commit before reaching the NUMLKUS threshold, which is the maximum number of locks per user.

Issuing a COMMIT statement ends a unit of recovery and commits all database updates that were made in that unit of recovery. DB2 performs the following functions when a commit occurs:

- All page and row locks are released, except those held by an open cursor that was declared WITH HOLD
- If the RELEASE(COMMIT) bind option was specified, all table, table space and partition locks will be released, except for those held by an open cursor that was declared WITH HOLD
- All claims are released, except those held by an open cursor that was declared WITH HOLD
- If the RELEASE(COMMIT) bind option was specified, the statistics used by the index look-aside and sequential detection processes are discarded
- If the RELEASE(COMMIT) bind option was specified, any IPROCs or UPROCs that had been created by the thread are discarded
- A write is forced out of the log buffer to both active logs

  The length of time required for the immediate write of the log buffers depends upon the amount of data that has been updated since the last write of the log buffer. Spreading updates throughout a unit of recovery increases the chances that your log data has been already written before you commit.

  While saving all update, insert and delete activity until right before you commit can reduce lock duration, it can also increase the amount of log data that must be written at commit and therefore may cause the commit to take longer.

For more information about the effect of the use of cursors defined WITH HOLD on locking see “Cursors WITH HOLD” on page 350.

The duration of commit varies. The primary factor is physical log write I/Os. The bind option RELEASE resource at COMMIT causes more overhead as more resources need to be freed. As more distinct SQL statements are executed within a commit interval, more resources have to be freed, taking longer time. Data sharing commit additionally includes GBP writes and page P-lock unlocks.

The physical log write I/O of a commit is two synchronous I/O operations for the write of the log buffer to both copies of the active log. In addition to any IPROCs or UPROCs, commit throws away the information used for index look-aside and sequential detection. See “Repetitive SQL procedures” on page 429 for details about xPROCs.

DB2 commits updates when a COMMIT is issued either explicitly or implicitly as a result of a termination of a unit of work. The type of DB2 attachment influences when a unit of work is terminated. See Table 12-7 on page 350.

Table 12-7 Termination of a unit of recovery

Attachment	Termination of a unit of recovery
CAF	The program issues a COMMIT/ROLLBACK SQL statement The program terminates
RRSAF	The program issues a SRRCMIT/SRRBACK statement The program terminates
IMS	The program issues a CKPT, ROLB or SYNC call The program issues a GU for the I/O PCB for single-mode transactions The program terminates
CICS	The program issues a CICS SYNCPOINT command explicitly The program terminates as signaled by CICS RETURN The program issues a DL/I checkpoint or TERM call command

When designing an application that is going to issue COMMIT statements be sure to place the COMMIT in the program that controls the logical unit of work. Issuing a COMMIT statement in a sub-program may lead to inadvertently committing in the middle of a logical unit of work. If data access modules are used, the COMMIT statement can be placed in a module that only performs the COMMIT statement, and it should be called only by the program that controls the scope of the logical unit of work.

12.4.2 Commit frequency

The commit frequency of an application is different depending upon the purpose of the commits. We do not advise setting the commit frequency of all applications to a single value, because the commit frequency that satisfies the requirements of the most highly accessed and critical DB2 objects is probably too frequent for the majority of other processes.

The primary reason for read-only programs to commit is to release the claims that may prevent utilities from draining. The maximum time that a utility will wait for a claim is calculated my multiplying the IRLMRWT DSNZPARM value times the UTIMOUT DSNZPARM value. An appropriate commit frequency for these processes would be half of that time.

For update processes that need to allow concurrent read activity at the row or page level a general commit frequency would be 25% of the IRLMRWT DSNZPARM value. However, for update processes that access critical, highly accessed rows or pages, a lower commit frequency might be required.

Recommendation: When determining your application's commit frequency, take into consideration the IRLMRWT timeout value and the ability of concurrent processes to wait.

A general rule is to commit no more than three times a second, but at least every three to five seconds.

For information about how to design your application to have variable commit frequencies, see "Error handling control table" on page 258.

12.4.3 Cursors WITH HOLD

The WITH HOLD clause of a DECLARE CURSOR statement allows the cursor to remain open across commit boundaries. From an application's point of view, this option simplifies the use of commits since cursor repositioning is not necessary. However, as mentioned, using WITH HOLD *may* prevent locks and claims from being released at commit.

Whether or not the page or rows locks of a cursor defined WITH HOLD are held across a commit is controlled by the RELCURHL DSNZPARM. If a value of YES is specified, then the lock on the currently positioned row is released at commit. If a value of NO is specified, then the lock on the currently positioned row is held across commit. However, if the lock held on the currently positioned row is an X or U lock, the lock is demoted to an S lock.

While either value of RELCURHL may be acceptable to achieve application concurrency and batch restartability, neither option will affect the releasing of claims held by the cursor. However, whether or not the cursor materializes will affect the releasing of claims held by the cursor.

The effect of materialization on cursors WITH HOLD

If the cursor does not materialize, the claims on the base DB2 objects held by the cursor WITH HOLD will be held until the first commit after the cursor is closed.

If the cursor does materialize, the claims on the base DB2 objects held by the cursor WITH HOLD will be released by the first commit following materialization (OPEN CURSOR). This is because the data from the base DB2 objects is read into a work file when the OPEN CURSOR is executed. The cursor will continue to hold claims on the work file, but all locks and claims on the base objects are released. Since we never run utilities on the work files, these claims to not threaten our concurrency and availability requirements.

To hold or not to hold

The following criteria should be used to decide when it is appropriate to code WITH HOLD on a cursor:

- If you are certain that the cursor will always materialize, you should leave the WITH HOLD clause on the cursor.

  By definition, the materialized cursor has already read all of the qualifying rows and placed them in a work table. Removing the WITH HOLD clause and reopening the cursor after each commit would result in the unnecessary overhead of repeatedly materializing all or part of the result set.

  If you want to guarantee materialization you might consider the use of a static scrollable cursor. However, in most cases it is not possible to materialize very large result sets because of work file storage limitations.
- If the cursor does not materialize, and is easy and inexpensive to reposition, you should consider removing the WITH HOLD from the cursor and reopening it after each commit.
- If the cursor does not materialize, but the repositioning logic is expensive, consider the following techniques to improve the performance of the repositioning so that the WITH HOLD clause can be removed.

  The typical cause of poor performance in the repositioning of a non-materialized cursor is a high number of columns used in the ORDER BY to uniquely identify each row and a low number of matching columns (usually one matching column) on the index.

  The first technique to improve repositioning performance is to create a separate cursor for each subset of rows that can be returned. The number of cursors required will be equal to the number of columns used in the order by to uniquely identify a row in the result set. This technique is described in detail in “Cursor repositioning using multiple cursors” on page 225.

  Another option to improve repositioning performance is to commit only after a change in the value of the high order column in the index. This will improve the performance of the repositioning, because the values in the non-high order repositioning columns should be blank, and the first row qualified through the index should be the first row processed on restart.

This technique is only recommended if the maximum number of qualifying rows for each value of the high order column can be processed within a reasonable commit scope.

- If the cost of the cursor repositioning cannot be reduced to an acceptable level, the WITH HOLD should be left on the cursor to maintain position.

  Be aware of the effect of the WITH HOLD clause on lock and claim retention, and schedule the process at a time when it will not be in contention with utilities that require drains or other applications.

- In a distributed environment, using a WITH HOLD cursor will prevent DB2 from separating the connection from the DBAT at COMMIT time when using CMTSTAT=INACTIVE. This will prevent efficient thread pooling in a distributed environment. See “Thread reuse in a distributed environment” on page 300 for more information.

12.4.4 Savepoints

A savepoint represents the state of data at some point in time during a unit of work. Savepoints are set by the application through the use of the SAVEPOINT statement. If subsequent logic dictates that only a portion of the unit of recovery should be rolled back, the application can perform a rollback to the savepoint. Figure 12-10 illustrates the concept of rolling back to a savepoint. In this example, all DB2 changes made between time T2 and time T4 would be rolled back.

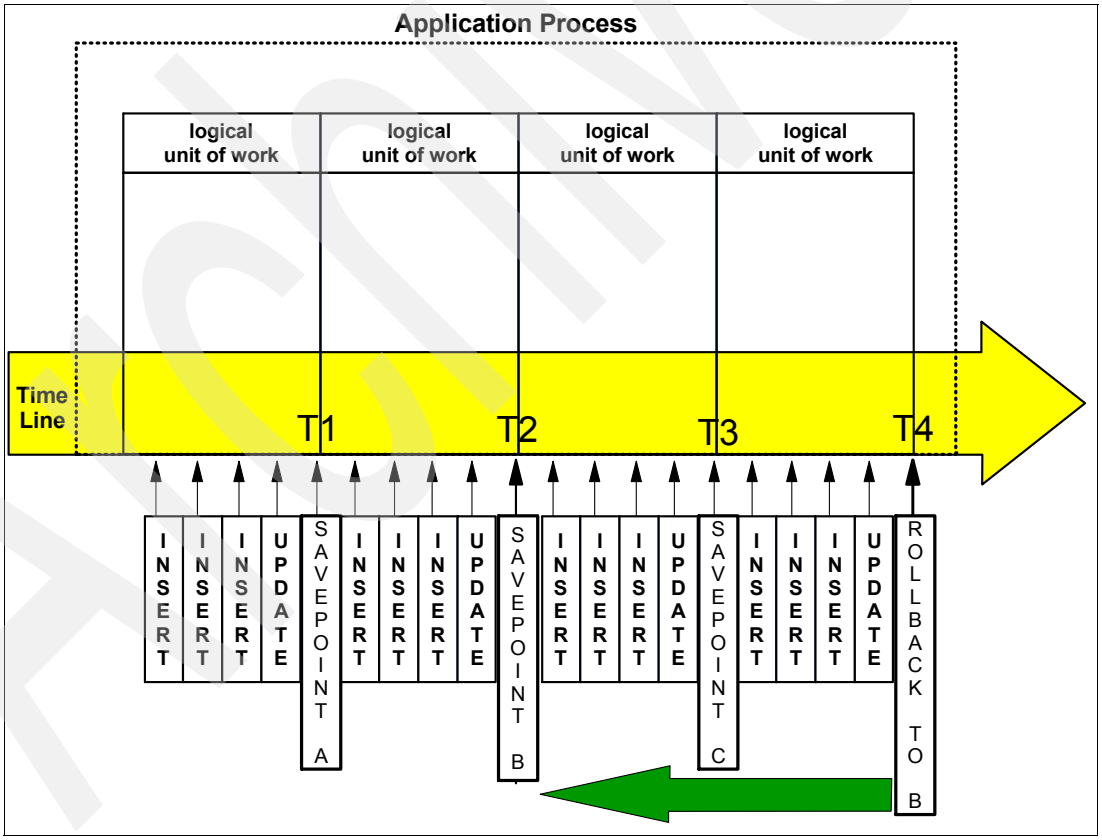


Figure 12-10 Rollback to a savepoint

Savepoints provide applications with a tool to achieve a more granular level of data recoverability without the expense of frequent commits. Issuing a SAVEPOINT command is much less expensive than issuing a COMMIT, because there is no forced write of the log buffer.

The ROLLBACK TO SAVEPOINT command performs a partial rollback of the unit of recovery. If a savepoint name is not specified, rollback is to the last active savepoint. If a rollback to a named savepoint occurs, all savepoints set after the named savepoint are released. The savepoint to which rollback is performed is not released.

A rollback to a specific savepoint backs out all DB2 changes that were made after the savepoint was set. Changes that were made to created temporary tables are not logged and are not backed out, but changes to declared temporary tables are logged and rolled back. A rollback to savepoint has no effect on the opening or closing of cursors, cursor positioning, the acquisition or release of any locks or claims, and the caching of SQL statements.

Restriction: A rollback to a SAVEPOINT only backs out changes made to DB2 objects. It has no effect on changes made to data in any other databases or file management systems, such as VSAM, MQ queues, or flat files.

Use SAVEPOINT rather than COMMIT to create a point to which your application can rollback for logic purposes. However, be aware that neither SAVEPOINT nor ROLLBACK TO SAVEPOINT end a unit of recovery. All locks and claims that are acquired during the unit of recovery are held until commit.

12.5 Hot spots

Hot spots are portions of a table or index that are continuously and repeatedly updated by the applications, for example, a counter or a total column. As a result, contention is likely to occur when accessing that data, and updates must serialize.

To avoid hot spots, you have to spread updates across the table or index by reducing the counter or total scope. For example, if you have a sales total in your table, every sale transaction will update this total, which becomes a hot spot. You can spread the updates by using departmental sales total, branch sales total, or even employee sales total.

You experience a hot page, when concurrent processes try to lock the same page, even if they are not processing the same row. You experience a hot row, when concurrent processes try to lock the same row. Figure 12-11 on page 354 shows the difference between a hot page and a hot row. You typically experience a hot row in an application that keeps track of the last number used in a DB2 table. Hot rows and hot pages can occur in any type of program that tries to acquire locks that are incompatible with locks acquired by concurrent programs or SQL statements. The approaches to solving these problems differ.

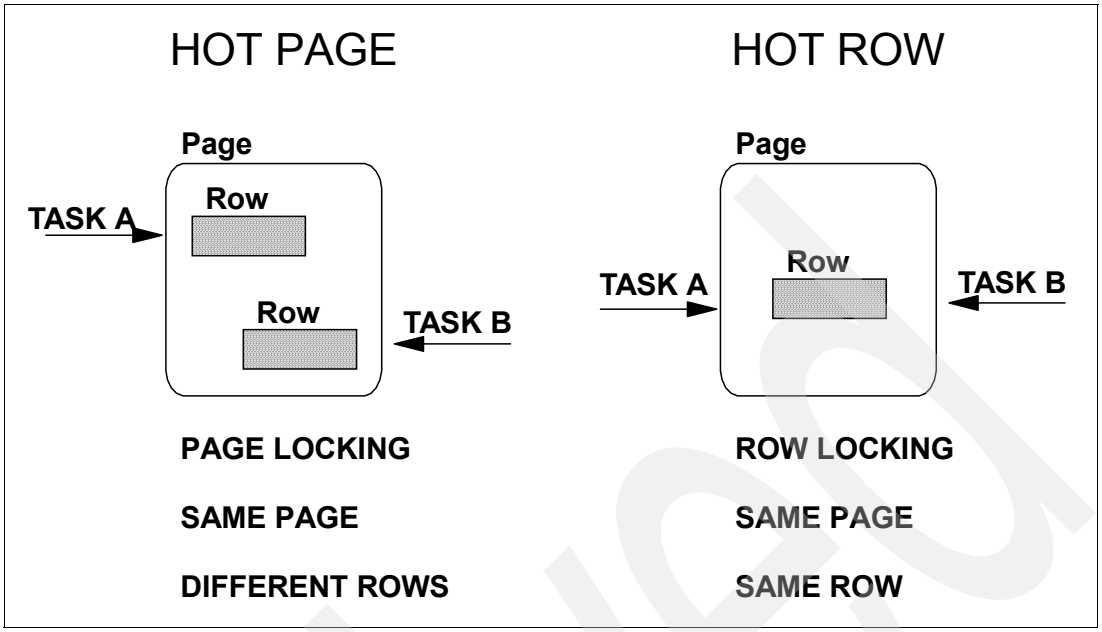


Figure 12-11 A hot page and a hot row

12.5.1 Hot data page problem

If you use page locking, a hot spot can occur on a data page. In this case, concurrent programs do not always process the same row. The hot data page problem typically occurs in the following cases:

- Small table

  The probability of using the same page is very high since the number of pages is limited. The use of a table space scan, because the table space size is too small is another factor that increases the probability of lock contention. Consider defining small tables with the VOLATILE attribute to ensure index access.

- Processing pattern

  Although there may be many pages in the table, concurrent programs end up using the same limited set of pages.

- Data clustering sequence

  If the clustering index correlates with the processing pattern, you increase the probability of collision.

The possible solutions to this problem are:

- Use smaller lock size or place fewer rows on each page

  Row level locking can be used to reduce lock contention when different processes are attempting to lock different rows on the same page.

  The MAXROWS parameter can be used to reduce the number of rows on each page, and, with a value of 1, can obtain virtual row locking with LOCKSIZE PAGE.

  See “LOCKSIZE ROW: Row-level locking” on page 334 and “LOCKSIZE ANY” on page 334 for more about these topics.

Archived

- Change the processing pattern or data clustering sequence.

  If you can modify your program logic to access DB2 data without concentrating on the same range, the contention decreases. You can also change your program's access pattern by altering the data clustering sequence, that is, redesigning the clustering index. Be aware of the impact of such a change on prefetch in your other applications.

12.5.2 Hot row problem

In the case of a hot row, concurrent processes try to access the same row, so you cannot resolve this problem by using a smaller lock size. Consider the following alternatives in resolving the hot row problem.

Reduce lock duration

Attempt to reduce the duration of each lock on the hot row through the use of the techniques described throughout this chapter, such as:

- Avoid locks wherever possible.
- Acquire locks as late in the unit of work as possible.
- Release locks as soon as possible with commit.

Evaluate the design

Hot rows are often created by a denormalization or an aggregation in the physical design of the tables. Review the design of the tables and the application to determine if a change might reduce or eliminate the hot row problem. The following case study describes how this can be done.

Assume your application needs to use a summary table to reduce the execution cost for read transactions to summarize the data, and the summary table is concurrently updated by transactions. Given the degree of concurrency, lock contention problems may occur.

Assume that you have an order history table, ORDER, and an order summary table, ORDER_SUMMARY, that is updated every time a transaction updates the ORDER table. In this case, ordering transactions update both the ORDER table and ORDER_SUMMARY table. Each insert, update, or delete in the ORDER table must be reflected in the totals of the ORDER_SUMMARY table to guarantee data consistency. The ORDER_SUMMARY table contains only one row, so you end up with a hot row problem. See Figure 12-12 on page 356.

Archived

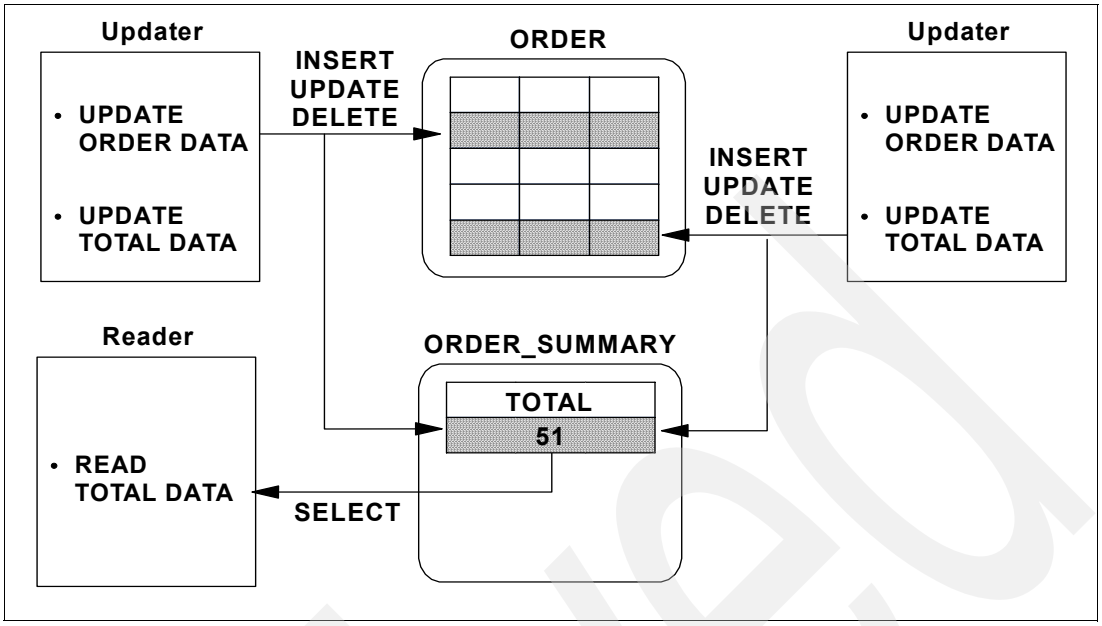


Figure 12-12 Updating summary table

If you cannot reduce the application requirements, a solution to this problem may be to spread the updates among a larger number of rows in the ORDER_SUMMARY table by adding another column to the primary key to achieve a more granular level of summarization, as you see in Figure 12-13.

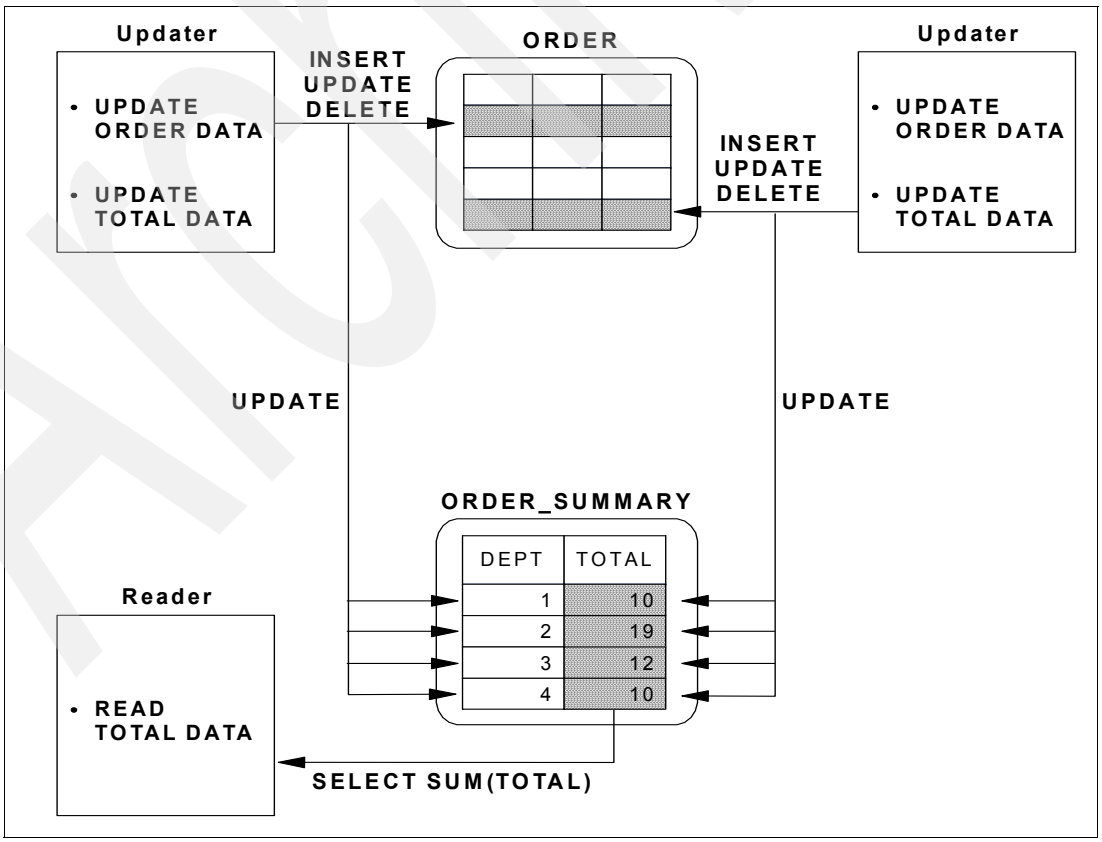


Figure 12-13 Spreading updates in summary table

To improve concurrency, you can combine this technique of spreading updates with other techniques, such as a smaller lock size or reducing the number of rows per page.

This discussion applies, if the updates on both tables are to be done in the same unit of work. However, if your business allows a delay in the propagation of the changes from the ORDER table to the ORDER_SUMMARY table, a better solution is to use an asynchronous update technique such as DPropR. Using DPropR, you can propagate updates asynchronously to the ORDER_SUMMARY table. The performance of the transactions that change the ORDER table is not affected. There is no need for any additional programming to accomplish the replication.

12.5.3 Sequential numbers in DB2 tables

Another potential hot-spot scenario is a table used to assign the next sequential number. Assume that you have a COUNTER_TBL table, which contains only one row with a counter value in the SEQ_NBR column. You fetch the row with a cursor to protect the row after the counter has been retrieved. To avoid a deadlock problem, define the cursor with the FOR UPDATE OF clause. Example 12-2 shows the coding that you can use.

Example 12-2 Sequential numbers in DB2 table: Case one

```
1 DECLARE C1 CURSOR FOR
   SELECT SEQ_NBR
     FROM COUNTER_TBL
      FOR UPDATE OF SEQ_NBR

2 OPEN C1

3 FETCH SEQ_NBR INTO :SEQ_NBR

4 UPDATE COUNTER_TBL
     SET SEQ_NBR = SEQ_NBR + 1
   WHERE CURRENT OF C1

5 CLOSE C1

6 Application processing logic

7 COMMIT (end of the transaction)
```

The page or row containing the counter is a hot spot and may cause contention. When updating the counter, consider committing before the end of the transaction to free the X lock on the data page or row as soon as possible. Example 12-3 shows how to code this.

Example 12-3 Sequential numbers in DB2 table: Case two

```
1 DECLARE C1 CURSOR FOR
   SELECT SEQ_NBR
     FROM COUNTER_TBL
      FOR UPDATE OF SEQ_NBR

2 OPEN C1

3 FETCH SEQ_NBR INTO :SEQ_NBR

4 UPDATE COUNTER_TBL
     SET SEQ_NBR = SEQ_NBR + 1
   WHERE CURRENT OF C1
```

```
5 CLOSE C1

6 COMMIT

7 Application processing logic

8 COMMIT (end of the transaction)
```

However, Case two has the following disadvantages:

- If a transaction abends in the second unit of work (after the first commit), a hole is created in the sequence. You have to be sure that no other updates are committed; for example, updates performed earlier in another subprogram that called this program.
- Commit requires additional resources.

If you do not need sequential numbers, you can take the approach shown in Example 12-4.

Example 12-4 Sequential numbers in DB2 table: Case three

```
1 UPDATE COUNTER_TBL
     SET SEQ_NBR = SEQ_NBR + 1

2 SELECT SEQ_NBR FROM COUNTER_TBL

3 COMMIT

4 Application processing logic

5 COMMIT
```

Case three has the same disadvantages as Case two.

If you are considering having more than one counter, you should have only one counter in each table in a segmented table space. Be aware of the following repercussions, if you do not follow this guideline.

- If the counters are placed in the same row, they are always locked together.
- If the counters are placed in different rows in the same table, they can be in the same page. The rows are also locked together if page locking occurs.
- If the rows are forced to different pages of the same table (for example, by using MAXROWS 1), it is still possible that lock contention will occur.

  When, for example, row 2 in page 2 is accessed, a table space scan can occur. The scan stops to wait at page 1, if page 1 is locked by another transaction. The situation is the same when you use row locking. You should, therefore, avoid using a table space scan.
- If you define an index to avoid a table space scan, the index may not be used if the number of pages in the table space is low. In this case, use the VOLATILE table attribute to influence the use of index access.

See “Generating consecutive numbers” on page 144 for other techniques to generate sequential and unique numbers.

Archived

13

Achieving 24x7

In this chapter, we provide several guidelines about designing applications for continuous availability.

Building a system that provides users with 24x7 availability is not an easy task. The cost of providing the availability can be significant and is not linear, because every percentage point that brings you closer to the goal of 100% 24x7 availability has an increased cost.

The first activity for your organization is to set a realistic goal for the availability of your system providing a realistic return on investment. A risk assessment must analyze the criticality of each process and the impact of unavailability of that process on your business. You can set different availability goals, depending on your industry and even the criticality of your business processes. Some processes allow for a downtime of one day, where other processes can be unavailable only for hours, minutes, or even seconds.

The user perception of non-availability covers a broad range of issues including hardware availability, system software defects, application software defects, database maintenance, application design practices, procedures for application life cycle management, and recovery procedures. If you want to achieve the maximum availability goal, you need to evaluate, manage, and reduce the impact of all of these potential points of failure.

Hardware and system software are constantly improving and providing new and better mechanisms and features which help you build the infrastructure delivering the availability. The data sharing architecture enables you to dramatically improve the availability of your system components, where dual and remote copy contribute to the overall availability of the I/O subsystem and disaster recovery.

Application software defects can be very disruptive because they can cause data inconsistencies. Deploying server side constraints such as RI or table check constraints can protect you against some of these issues, but they do not cover the full scope of application defects. Setting up proper testing procedures helps to reduce the probability of moving erroneous programs into production. Once the damage is done, it is important to limit its impact, so you need verification procedures and utilities that enable you to back out the changes and restore both the data and the programs to a consistent state.

The impact of database maintenance depends on the database design and the nature of your applications. The frequency of reorganization is a good example to illustrate that concept. In

© Copyright IBM Corp. 2005. All rights reserved.

an R/O environment, the frequency is close to zero, but in a highly volatile environment, the frequency could be as high as daily. Database design and insert patterns can heavily affect the frequency of database maintenance. The business itself can influence insert patterns by deciding to turn on a feature or take in new streams of transactions.

DB2 provides features, such as partition independence and online REORG and LOAD with inline image copy, that limit the scope or even eliminate the unavailability.

Some application design techniques sacrifice availability in order to reduce the elapsed processing time. If you unload a table in a sequential file, the table should be made unavailable for updates until the file is reloaded in the table. Failing to do so causes the changes, which are applied to the table after the unload, to be lost. Achieving the continuous availability goal could mean that some of the design techniques are no longer applicable.

This chapter contains the following topics:

- Maintaining the schema
- Managing application programs
- Designing for application parallelism
- Managing application data

Archived

13.1 Maintaining the schema

When we take 24x7 availability to the extreme, the data should never be unavailable and the applications should always be available. This is a stretch goal, and it is likely that we are going to have some level of unavailability at some point in time. With every version of DB2, new online schema options are introduced, but even though they help with database modifications, a fundamental issue remains: synchronization between the database schema and the application logic.

Being able to simply add a new column is not solving all the issues; the application logic probably needs additional coding to deal with the new information. It is, however, possible to implement the schema changes in such a way that the changes are transparent to the existing code and that newly deployed code introduces the processing of the new schema. In some cases, the two DDL options live in parallel, the example in "Using automatic table switching options" on page 376 explains a case where two indexing models are used concurrently.

13.1.1 Issuing DDL

Issuing DDL against existing objects requires special attention, because we need to think about lock concurrency and performance after the change. Table 13-1 gives you an overview of the locks acquired by various SQL statements. The numbers in parentheses refer to the footnotes in the table.

Table 13-1 Overview of locks acquired by various SQL statements

Process	Catalog table spaces	Skeleton tables (SKCT and SKPT)	Database descriptor (DBD) (1)	Target table space (2)
Transaction with static SQL	IS(3)	S	n/a (4)	Depends on SQL
Query with dynamic SQL	IS(5)	S	S (8)	Depends on SQL
BIND process	IX	X	S	n/a
CREATE TABLE	IX	n/a	X	n/a
ALTER TABLE	IX	X(6)	X	n/a
ALTER TABLESPACE	IX	X(7)	X	n/a

(1) In a lock trace, these locks usually appear as locks on the DBD.
(2) The target table space is one of the following table spaces:
- Accessed and locked by an application process
- Processed by a utility
- Designated in the data definition statement

(3) The lock is held briefly to check EXECUTE authority.
(4) If the required DBD is not already in the EDM DBD cache, locks are acquired on table space DBD01, which effectively locks the DBD.
(5) Except while checking EXECUTE authority, IS locks on catalog tables are held until a commit point.
(6) The plan or package using the SKCT or SKPT is marked invalid if a referential constraint (such as a new primary key or foreign key) is added or changed, or the AUDIT attribute is added or changed for a table.
(7) These locks are not held when ALTER TABLESPACE is changing the following options: PRIQTY, SECQTY, PCTFREE, FREEPAGE, CLOSE, and ERASE.
(8) DBD lock is obtained when using dynamic SQL unless the statement is cached in the dynamic statement cache.

Archived

Implementing schema changes

Online schema evolution offers much more availability and flexibility while changing DB2 objects. In order to support online schema evolution, DB2 has implemented a new change management architecture, called *versioning*.

Versioning allows you to track an object's definitions at different times during its life by using versions. Altering existing objects results in a new format for tables, table spaces, or indexes and versioning indicates how the data should be stored and used for each change. All the data for an object and its image copies cannot be changed immediately to match the format of the latest version, so support for migrating the data over time is implemented by using versions of tables and indexes. This allows data access, index access, recovery to current, and recovery to a point-in-time while maximizing data availability.

Online schema evolution is a great help, but you still need to follow a set of steps to complete the changes. In summary, the steps are:

1. Change the applications.

   Make sure to assess which programs need to be changed and how. Host variables, for example, may need to be extended to accept the extra length.

2. Alter data definitions.

   Execute the DDL. Consider changing data and indexes in the same unit of work.

3. REORG and REBUILD

   To minimize any performance degradation, schedule a REORG as soon as possible after ALTER because it may also avoid some of the row relocation problems. This reestablishes fast column processing and eliminates the conversion from the old row format to the new one when retrieved by the application. Rebuild any affected indexes.

4. RUNSTATS

   Schedule RUNSTATS to repopulate the catalog with accurate column and index statistics.

5. REBIND

   Rebind plans and packages. Otherwise, DB2 may pick an inefficient access path during automatic rebind because the best suited index is currently not available.

Availability considerations

When an index is in Rebuild Pending (RBDP), the optimizer is unaware of this and can still pick that index during access path selection when you are executing a query. In Version 8, DB2 tries to avoid using indexes in RBDP for dynamically prepared queries. This may cause significant overhead, due to the selection of an inefficient access path because DB2 bypasses indexes in Rebuild Pending (index is ignored) in the following cases:

- For all DELETE processing
- For all nonunique indexes for UPDATEs and INSERTs

A unique index in RBDP cannot be avoided for UPDATE or INSERT, because DB2 needs the index in order to enforce uniqueness.

Note: Check APAR PK06185 for DELETE improvements with RBDP indexes.

The DB2 optimizer treats RBDP indexes as follows:

- Dynamic PREPARE:
  - Indexes in RBDP are avoided.

- Cached statements:
  - If the statement is cached, the PREPARE is bypassed. But if the index is used by the statement that is cached, it is invalidated by the ALTER.
  - Invalidation occurs during ALTER.
- Static BIND does not avoid an index that is in RBDP:
  - Indexes in RBDP can still be chosen, and will get a resource unavailable condition at execution time.

13.2 Managing application programs

The tables and indexes are not our only concern when looking at the high availability of our DB2 system. Every active program is locking the package or plan, when using a DBRM bound into a plan. So bringing a new version of a modified program into DB2 would cause contention with the active version, unless we make it look like a new object or unless we avoid the whole process.

You should also consider the sequence of your program preparation and make sure that it is correct. The correct sequence is to perform the BIND first, before the link-edit. You need to BIND your package before you link-edit the modified module in the production library, because the new module might be picked up by a new program load before the BIND is completed.

13.2.1 Avoid the BIND if the SQL did not change

Why perform a BIND of the SQL, if the program did not change? The precompiler is involved whenever SQL is present in the application program. In some cases, in particular in testing environments, programmers tend to change the application logic without making any change to the SQL in the application. The precompiler will create a new DBRM with a new consistency token which will be used to do run time consistency checking between the package/plan and the load module. A mismatch in the consistency token will result in a -805 SQLCODE. There are solutions on the market, or you can code for yourself a DBRM compare that verifies the DBRM before and after the precompile and avoids the BIND if the consistency token is the only change in the DBRM. It is more complicated than just changing the consistency token, because the STMTNO can also change if new code was introduced in the application program.

13.2.2 Use packages with versions

Package versioning is a great technique that solves the locking issue of a BIND on an active package. DB2 uses the version as part of the primary key of the package, which means that a BIND of a new version is an INSERT instead of an UPDATE in the catalog.

A positive side effect is the simplicity of the rollback in case you want to revert back to the old load module. Given the appropriate package cleaning approach, you are able to use more than one version of the code side by side.

There is no automatic cleanup of the versions of the packages in DB2, and the FREE PACKAGE command does not have any indication of the number of versions that need to be kept. The FREE PACKAGE command either frees all the versions, or otherwise just the version that you specify. The best technique is to parse the accounting record to detect which package versions are still active and free any packages that have been inactive for a certain period of time. You also may want to keep at least two versions of each package because there might be some packages that are only used once per quarter or even once per year. You also need a way to allow certain packages to have more than two versions available as

some modules might be linked statically, which means that a load module can be using an old version for a significant amount of time.

Using stored procedures requires additional attention when multiple versions need to be supported. DB2 searches SYSIBM.SYSROUTINES using the name of the stored procedure and the schema of the stored procedures. The SCHEMA can be passed either as a BIND parameter of the caller, or at run time with SET CURRENT PATH statement within the caller, or by fully qualifying the schema on the call. We do not recommend to fully qualify the stored procedures, since that makes the management of multiple environments very difficult. See Figure 13-1 for a possible multi-version environment using a multiple schema implementation.

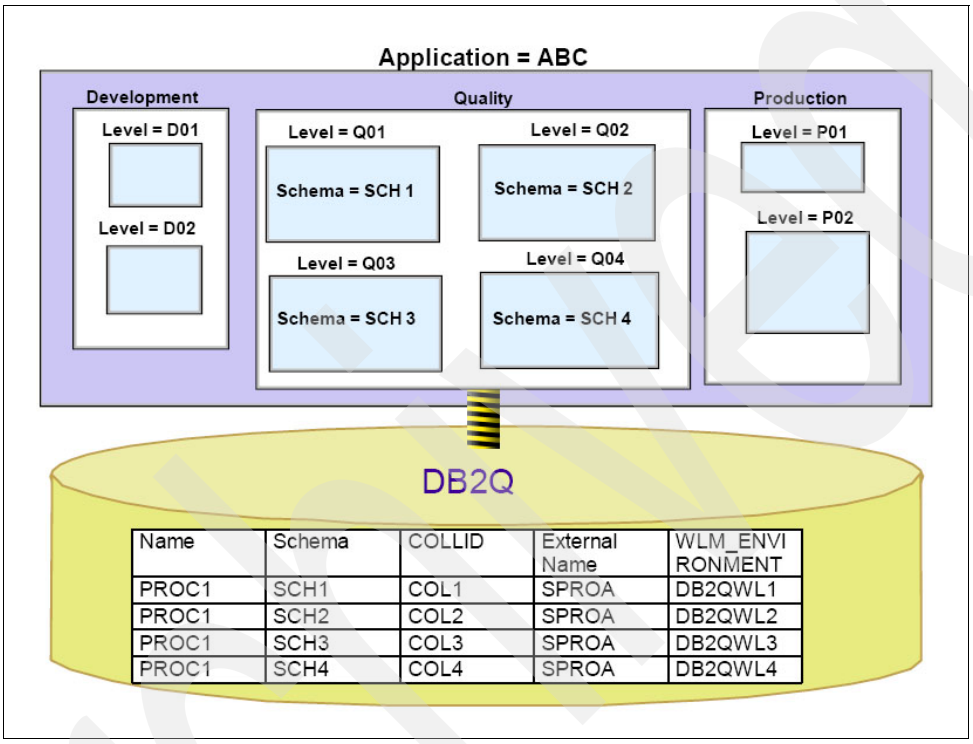


Figure 13-1 Sample versioning of stored procedures in a DB2 subsystem

13.2.3 Performing REBIND

Rebinding a package of an application that is currently active is impossible due to the X lock on the SKPT of the active package. So how do you cater for REBIND of an active package? There is no easy answer to this problem, since it is requires you to cater for this in the application program. A possible solution is to use multiple collections and put the collecting selection logic in the application. This way you can perform the REBIND on the collection that is not active and then after the REBIND, inform the active application that it can switch to the new access path. This means, however, a significant overhead on every program to include this collection switching logic for every program that might potentially need to be rebound while being active.

The other approach would be to just try the REBIND in a separate collection and when a significant difference in the access path occurs, you verify that the gains of the new access path justify the temporary unavailability of the program.

13.3 Designing for application parallelism

Batch environments often consist of a series of programs, where the execution of one program depends on the results from a previous program, often controlled by a scheduling product. The individual batch programs are then serialized to a certain degree. In general, these programs must run at a particular time and must complete within a given batch window.

In most sites, two trends can impact this serialized process:

- The amount of data grows (DB2 tables become larger), so the elapsed time required to run batch programs increases.
- The online period is expanded, which leaves a smaller window for batch programs.

These trends often cause the batch window to become too short compared to the elapsed time required for batch programs. An increased degree of parallelism is a natural solution to the problem, but often the existing design does not support this.

A prerequisite for running batch jobs in parallel is to implement the checkpoint and restart procedures discussed in “Batch process design” on page 282. The main reason is that commits are needed to free page locks. In this section, we discuss different aspects of designing parallel batch job streams. The term *parallel jobs* in this section describes the ability to run several batch programs concurrently, each one performing functionally the same processing as the single batch job, but on different parts of data. The parallel jobs access the same DB2 tables, but accessing different ranges of data, at least in some of the tables.

A single job would have to process all pages in the table. With three parallel jobs, each job would have to process only one-third of the table. In practice, you need an index to set up the appropriate ranges of data. The rows are then accessed by key range. The table for which the key range applies is often referred to as the *main table*. Other tables are accessed when needed, but not always in the order of this specific key range.

There are two main reasons for running parallel jobs:

- Batch window limitations

  Sometimes batch jobs require longer elapsed times than provided by the batch window. Not all batch operations can run concurrently with the online system. Splitting the batch job into several jobs and running them in parallel often significantly reduces the elapsed time.

- Resource utilization

  Given a hardware configuration, the goal is often to maximize resource utilization. Parallelism can be a mechanism to achieve this goal.

Running a single batch job requires a certain amount of CPU cycles and I/O operations. The typical situation is that the job at a given point in time is using either CPU or I/O resources, but not both. Therefore, it cannot fully utilize either the CPU or the I/O capacity.

Scheduling several parallel jobs generates a need for a larger amount of CPU cycles and I/O operations per second. In theory, this amount is the product of multiplying the number of scheduled jobs by the number of requests per second generated by a single job.

In practice, however, the sum of the elapsed times of the individual jobs tends to be longer in parallel compared to nonparallel job execution because of:

- CPU contention
- I/O contention
- Locking and latching in DB2 (data contention)

The efficiency of running batch jobs in parallel depends on how CPU-bound the jobs are and the number of multiple processors used to run the jobs:

- Single-CPU processor

  CPU-bound jobs are restricted mainly by the amount of CPU resources available. Therefore, splitting the job into several smaller jobs with the same CPU-to-I/O ratio does not reduce the elapsed time. The jobs spend time waiting for CPU.

  I/O-bound jobs are characterized by high I/O elapsed time and correspondingly low CPU usage. If the original job is I/O-bound, it is a candidate for being split and run in parallel. The CPU utilization on the processor is then increased, if no other bottleneck exists.

- Multi-CPU processor

  In this case, a CPU-bound job can benefit, because the job can be divided into several smaller jobs and executed in parallel. Each job can then utilize one of the CPUs. The worst situation from a CPU utilization point of view is to have a single I/O-bound job executing on a multi-CPU processor. The job cannot even utilize a single processor. It is the typical candidate for parallel execution, if I/O bottlenecks can be avoided. A general requirement for taking advantage of more parallelism is that no other bottleneck is introduced into the system. For table and table space design regarding application parallelism, see also “Parallelism exploitation” on page 280.

13.3.1 Locking considerations

See Chapter 12, “What you need to know about locking” on page 327 for an introduction to locking and locking options. This section provides an overview of the locking considerations that are related to the 24x7 availability. The design of parallel jobs must consider the effects that jobs running concurrently have on locking:

LOCK TABLE statement

With parallel jobs, LOCK TABLE must be used with care. Several jobs can run in parallel if they acquire a SHARE lock on the table space. In that case, they cannot perform updates to the table space. If the program takes an EXCLUSIVE lock on the table space, parallelism is impossible on that table. Page level locking should be used instead for the table space getting updated.

Lock escalation

If some of the updated tables are defined with LOCKSIZE=ANY, DB2 system parameters NUMLKTS and NUMLKUS may have to be increased to match the commit frequency. They should not conflict with the number of page locks required between two commit points for a given table space and for a given user. Lock escalation should not be allowed to occur for any of the parallel jobs.

IRLM timeout value

When determining the commit frequency for a batch job running in parallel (possibly with the online system), you should consider if other jobs or users could request access to pages updated by this batch job. Those jobs or users could wait for as long as the time between two commits of this batch job.

You must then consider synchronizing the commit frequency and the IRLM timeout value (which is a DB2 system-wide parameter), so that the IRLM timeout value is larger than the expected time between two commits. Otherwise, the jobs or users waiting for an updated page will time out.

Order of tables processed

The first point to observe is table access order. Deadlocks and timeouts are likely to happen if one application updates a row in small table A and then updates a row in small table B, while another application updates a row in small table B and then updates a row in small table A.

Order of locks acquired

The next critical characteristic is the order in which parallel jobs acquire individual pages. If, for example, all locks are acquired in ascending order, no deadlocks should occur. Timeouts are still possible. However, in many cases, the locks are not acquired in any significant sequence, because data pages in tables that have a clustering sequence different from the main processing order are also accessed more or less randomly.

Parallel jobs can take advantage of a design where the tables are accessed in the same (clustering) sequence. This design avoids most locking problems and is also the most efficient design from the I/O point of view.

Application retry

A common requirement in an active batch environment is to control application retry when deadlocks or timeouts are detected. The application must be coded to retry when deadlocks or timeouts are detected. For suggestions about application retry logic, see “Retry logic” on page 256.

13.3.2 Conclusions

When accessing large tables or handling a large amount of data, you have to carefully design the batch processes to get the best performance. The significant points are:

- Process the data in the clustering sequence to benefit from index look-aside and sequential prefetch or detection.
- Sort the input files in the clustering sequence of the tables that are referenced.
- Use files to avoid repetitive accesses to the tables and to transmit information between processes.
- Split your process into as many steps as there are large tables with different clustering sequences to access.
- Use the clustering index to establish different data ranges to be used for each parallel subprocess.
- Keep synchronous I/Os to a minimum.
- Use external sort.

Techniques, that you can use for sequential processing, provide the best performance. However, maintaining data consistency is complex because the data remains in an inconsistent state for long periods because the logical unit of work spreads between processes. As a consequence, data recovery to a previous point in time becomes complex.

Although programming for random accesses is easier, do not use random access in large batch processes unless a very small amount of data is involved.

Archived

13.4 Managing application data

The application data is your most important resource and it must be managed from a life cycle point of view and maintained from the recoverability and performance points of view. In some cases, the housekeeping required to keep the table and indexes in good shape can be performed in the maintenance window, but the windows tend to become smaller as time progresses and the tables just keep getting larger. In some cases, the maintenance window is close to nonexistent.

13.4.1 Supporting multiple versions of the data

Bringing a new version of your data online without stopping the activity can be quite a challenge, especially when dealing with large volumes.

It is possible to keep track of the final version of the data in a separate table. In this case, you can coding a select to obtain the latest value of the flag that indicates the current version. A solution is the usage of UNION IN VIEW to make the switch transparent to the selectors.

In Example 13-1, you can find the DDL required to define the table TMYDATA that holds the actual user data. The table has two partitions, one for current data and one for future data. All indexes on the table are defined as DPSI indexes. This is not a problem because every SQL statement knows the value of the partitioning key, which is the FL column. The Control table keeps track of which partition contains the current and which partition contains the future data. The initial current data is in partition 1, and the future data is in partition 2. The application can flip/flop the current and future by inverting the values of the flag in the two records in the control table. Updating the control table to contain ((‘A’,’F’),(‘B’,’C’) inverts the definition of current and future.

Example 13-1 Table DDL for base table and control table

```
CREATE TABLE TMYDATA
        ( FL CHAR(1) NOT NULL ,
          other columns
        )
PARTITION BY (FL) ( PART 1 VALUES(‘A’) ,
                    PART 2 VALUES(‘B’));

CREATE TABLE CONTROL
        ( FL CHAR(1) NOT NULL ,
          VERSION CHAR(1) NOT NULL
        )

INSERT INTO CONTROL VALUES(‘A’,’C’);
INSERT INTO CONTROL VALUES(‘B’,’F’);
```

Example 13-2 lists the views required to access the future and current rows.

Example 13-2 View definitions for current and future view

```
CREATE VIEW TABACTIVE AS
(SELECT A.all columns FROM TMYDATA AS A , CONTROL AS CON
 WHERE A.FL = CON.FL AND CON.FL = ‘A’ AND CON.VERSION = ‘C’
   UNION ALL
 SELECT A.all columns FROM TMYDATA AS B, CONTROL AS CON
 WHERE CON.FL = ‘B’ AND CON.VERSION = ‘C’
) ;
CREATE VIEW TABFUTURE AS
(SELECT A.all columns FROM TMYDATA AS A , CONTROL AS CON
```

```
 WHERE A.FL = CON.FL AND CON.FL = 'A' AND CON.VERSION = 'F'
   UNION ALL
 SELECT A.all columns FROM TMYDATA AS B, CONTROL AS CON
 WHERE CON.FL = 'B' AND CON.VERSION = 'F'
) ;
```

The application can now use one of the two views to read either current or future information as you see in Example 13-3.

Example 13-3 Two cursor definitions to read future and current data

```
EXEC SQL DECLARE CURSOR C1 FOR
         SELECT COLS
         FROM TABACTIVE
         ;

EXEC SQL DECLARE CURSOR C2 FOR
         SELECT COLS
         FROM TABFUTURE
         ;
```

There are considerations when using this kind of design:

- How do the updaters switch between current and future?

  The application needs to manage the inserts, updates, and deletes in the application until DB2 supports INSTEAD OF TRIGGERs or lifts the restriction on updatable views. This means that the program should read the control table before making any changes to determine which value to use on the SQL. This introduces the problem of consistency and cache coherence. If the application caches the value of the control tables to avoid reading it for every row processed, then the program might assume A is still future, while A has become current. This is probably not the case as the updater is likely to be the process that eventually performs the update of the control table.

- What is the impact on RI?

  The FL column becomes part of the primary key of the table; this means that it also gets pushed down into the foreign keys that refer to the parent. The design of future and current has to be pushed down into the children of the parent that supports this kind of versioning. The children use the same control table as the parent to guarantee consistency.

13.4.2 Keeping the data in good shape

Tables and indexes are a bit like your desk, at some point you might want to do some cleaning if you want to be able to continue working. This does not apply to everybody, since some people work very neatly anyway and do not need to invest time in a cleanup.

The need for reorganization is very much dependent on the nature of your inserts and the way you decide to cluster the table. Indexes are different: Each index has its own clustering sequence determined by the mix of columns on which you defined the index. So, when deciding on the clustering sequence, the insert algorithm of each index, and, therefore, each potential table cluster sequence needs to be evaluated and weighed against the data access advantages and the resulting REORG frequency.

There is a direct link between the insert algorithm, the REORG frequency, and the clustering sequence. In some cases, the combination of the three elements results in an unimplementable configuration. An example is the requirement to insert 5,000 new rows per second in a sequence that is the invoice number. The 5,000 inserts all belong to a random selection of customers. This insert rate needs to be maintained for at least 3 hours. These

requirements do not fit very well together, and are, in fact, contradictory. You would need to define an infinite amount of free space and even doing this would most likely lead to an unacceptable amount of synchronous I/Os on the insert. As a result, you decide to cluster the table by invoice number, therefore, resolving the free space issue in the table. You soon realize that the problem is solved completely because the table might be fine but the index on customer number still faces the same challenges. Any index is always faced with its column sequence as an implicit CLUSTER clause. Solving the index problem is much harder than the table problem because your options are more limited.

When the physical design is performed, not all objects will end up in an insert at the end scheme. This means that at some point, the objects will use up all the free space added by the REORG or LOAD. The only exception are of course read-only objects or objects that are only maintained by means of utilities. In all other cases, where data is added by means of INSERT, the object will deteriorate. The result of the deterioration is an increase in the time it takes to add new rows. See Figure 13-2 for an example of a stacked chart indicating the number of getpages required to do a single insert plotted over time. Every time you see a drop in the number of getpages, a REORG was performed.

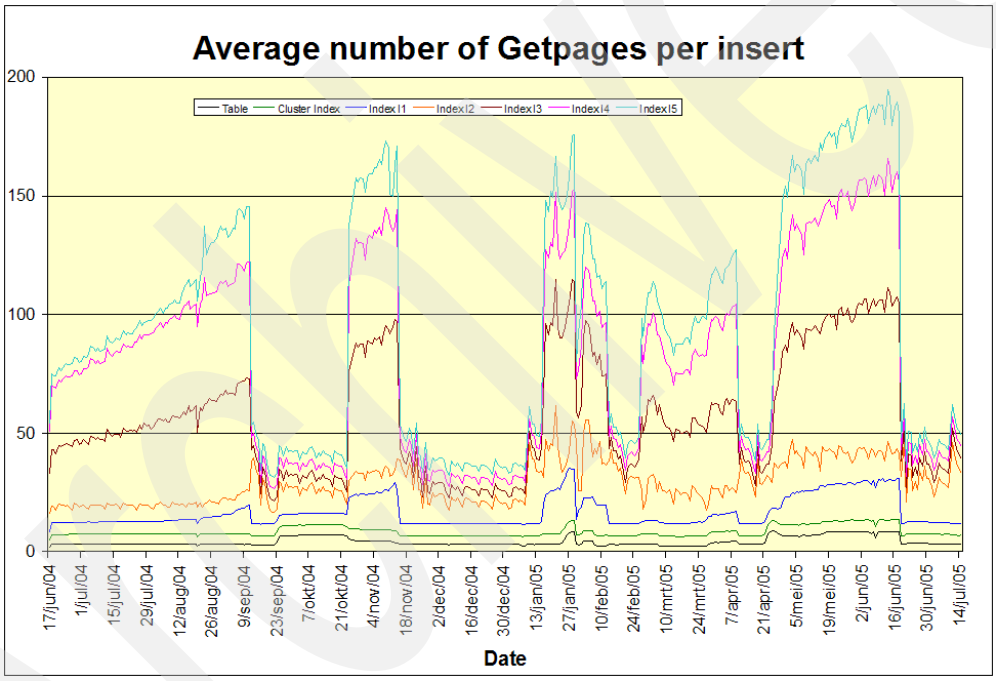


Figure 13-2 Number of getpages per insert in relationship to REORG

An additional consideration is the impact of making the table larger.

These two are not parallel. Ten times the rows means roughly 10 times more scan time. Ten times more rows might be only one more level in the index or even 0. Some differentiation is needed, as noted next, but not connected.

An increased number of rows is translated into a proportionally increased time to scan the table or index. If the SQL is executed randomly, there may also be an increase in the index levels required to perform an index probe. The total number of data pages, index leaf pages, and number of index levels are important performance metrics. Is there any merit in making a distinction between the active operational data and the active, but nonoperational data and consider offloading the nonoperational part? See “Horizontal table split” on page 43.

Offloading old data

At some point, data needs to be archived from the operational table and moved to either offline storage or to an online archive table (see Figure 13-3 for a schematic representation of the process).

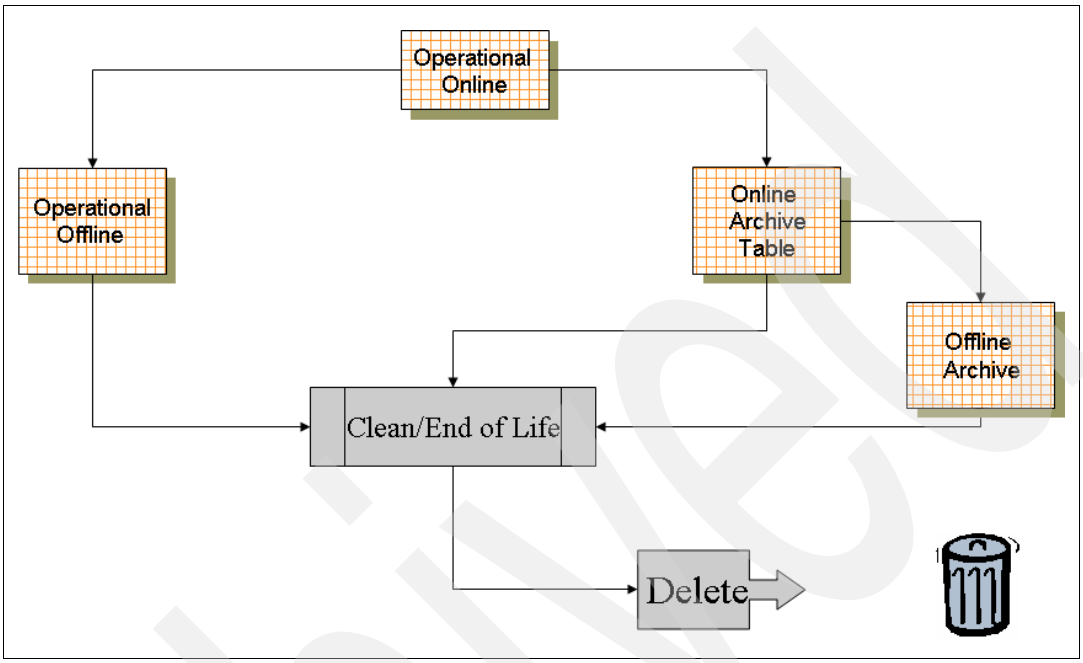


Figure 13-3 Life cycle of data

As more companies face stricter laws by regulatory bodies, the concept of data archival becomes much more than just “how to get old data out of the way”. Now, you need to concern yourself with the legal ramifications of data removal.

An important first step in data archival management is to identify whether the DB2 system team or the application design team can determine what needs to happen at which point in the life cycle. The retention and placement attributes of data need to be defined at the logical model level and that probably depends on the business domain your company deals with. The middleware team should deliver the tooling required to manage the archival and destaging of the information, so every application does not need to deal with the problem individually.

At a second level and within the boundaries of the legal constraints, you need to concern yourself with the performance of the application and the operational impact of dealing with larger objects. There may not be a need for the application from a logical point of view to move the end-of-life rows to a separate location. The impact of having the old rows in the same object with the new rows may create unacceptable overhead and may lead to poor performance or unacceptable manageability of the tables.

So there are really two levels at which you need to deal with offloading: The legal level that determines when something can be removed and the operational level that determines when data needs to be moved to another object.

The process of moving data from the operational table to the secondary location can be fairly disruptive if you do not plan the process carefully. Several approaches exist for moving the data:

- Use SQL to DELETE and INSERT the rows
- Use an UNLOAD and reLOAD
- Use (ONLINE) REORG with DISCARD and LOAD

The most appropriate technique depends on several parameters:

- The ratio of the part that needs to be removed versus the total amount of information.

  If all the data needs to be moved at once, a different approach needs to be used than if we just need to remove 2% of the data.

- The location to where the rows need to be moved.

  If all the rows that need to be removed are grouped together, the cost is quite different than a case where all the data is spread throughout the table. Utilities tend to be more efficient, if the impacted area can be contained within a limited area identifiable by the utility.

- The fact that the data does not only need to be deleted, but also needs to be moved to another object.

  It is much simpler if you only need to perform the delete. If the data needs to be unloaded and moved to another object, the process gets more complicated because we need to deal with unloading files and the impact of additional fetches and non-DB2 resources to store intermediate results. The alternative is to use an online REORG with DISCARD to solve the problem.

Online REORG on smaller parts

Using partitioning and performing the REORG of a single or limited set of partitions is a way to reduce the scope of the utility. There is, however, a potential major concern when using this type approach. This is the use of any NPIs defined on the partitioned table space. The efficiency of the utility tends to degrade significantly and the cost and overhead of maintaining the NPI can be in the order of magnitude of 10 to 100 times less efficient.

You need to weigh the gains against the inefficiencies that you incur when running a partition level utility with NPIs defined on the object. There is a tradeoff between the efficiency of maintaining the logical NPI parts versus reorganizing the entire table space.

Let us look at the example where you have a table space with 100 partitions and your application is inserting rows in a single partition. The inserts do not arrive in clustering sequence, and this results in a bad clustering and inefficient page utilization on the data partition. Possibly this may lead to increased insert costs or unacceptable response times of your Selects. You now have two options, REORG the entire table space or REORG just the partition that has the new inserts. If you REORG all partitions, you do not suffer from the NPI update process, but you need to reorganize 99 partitions that have not changed. If you REORG just the changed partition, you spend a lot of time updating the NPI. At some point, updating the NPI is going to be more expensive than doing a REORG of the entire table space.

The other aspect of the equation is the space required to perform the REORG. If you perform a full table REORG, the space required is at least double the current size because the shadow data sets need to be defined. Using a REORG on a single partition only requires the space for the table partition, part of the partitioned index, and the space for the data set to store the keys for each NPI that needs to be fixed in the BUILD2 phase.

13.4.3 What if your insert rate is not compatible with your select requirements?

The insert rate in your table can be so aggressive that the number of indexes that can be defined comes under pressure. You have a conflict of interest, if at the same time that the inserts take place, the selects require a large number of indexes.

One option is to tune the object in such a way that insert performance is the primary goal, while still maintaining the SLA of the select statement. This, however, may not always be possible.

An important question is the latency requirement of the select statement. Duplicating the table may be an option, if the select does not need up-to-the-second accurate information. Another alternative, which can be deployed if the latency is described in terms of hours, is the creation of multiple tables. Figure 13-4 shows two instances of the same table T1 (a) and (b) that would flip/flop after a predefined time.

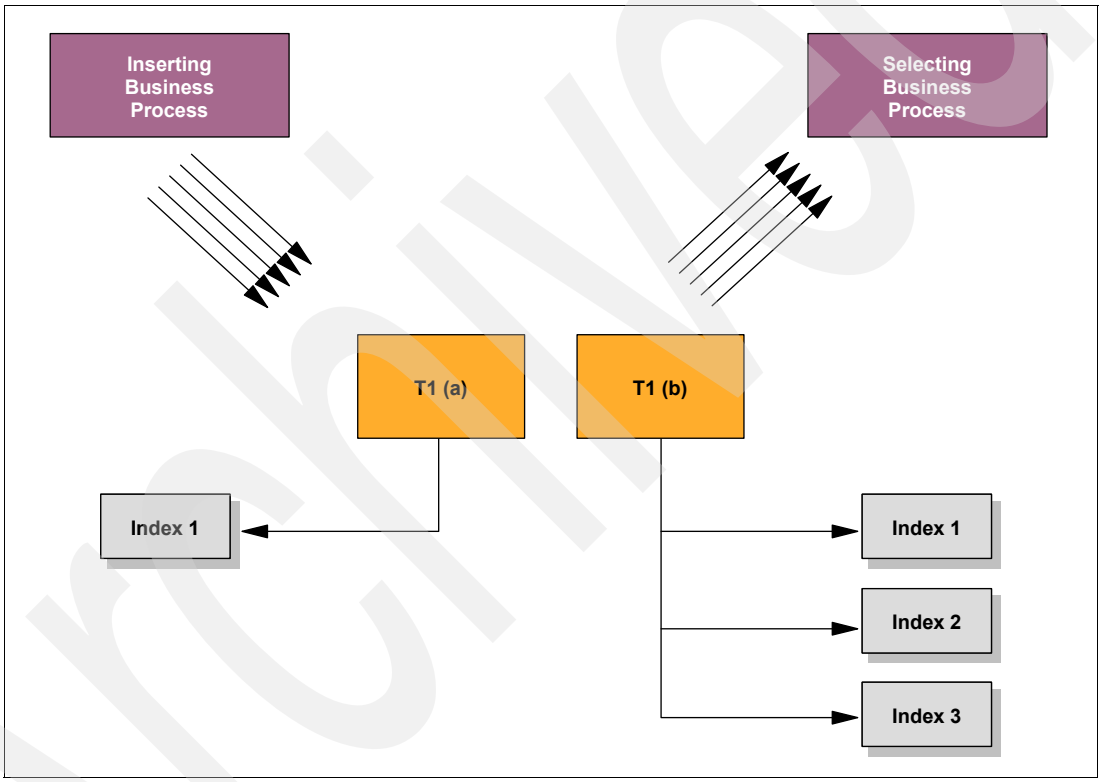


Figure 13-4 Two table implementation for concurrent insert and select

Let us look into these options in more detail.

Low latency solutions

It is virtually impossible to switch between two or more objects if the latency of the select versus the insert time is very short. The real bottleneck is going to be the performance of the insert process and the degree of parallelism that can be introduced at that level. If the cost of extra indexes is unacceptable for the insert process, simply because it would introduce too much I/O wait, then moving the insert to a secondary asynchronous process may solve the problem. Note it will be required for the asynchronous process to increase the degree of parallelism or to perform the inserts in a more efficient way. Failing to improve the efficiency of degree of parallelism will cause the latency to increase by a percentage that is the ratio between the arrival rate and the insert frequency.

There are several ways to implement the synchronism, but they all increase in one way or another the overall resource consumption. We can rule out triggers as a solution. This would only make the original process slower due to the synchronous nature of the trigger.

You can use WebSphere MQ to pass the insert information on to the second process. See Figure 13-5. This will not save a lot of CPU in the inserting process, because it will require an extra PUT call and it introduces a second resource manager. The CPU consumption of the PUT is very similar to that of a DB2 INSERT with a limited amount of indexes. The major difference is likely the avoidance of any synchronous I/O since the PUT just needs to add the message to the QUEUE. Do not ignore the impact on the cost of COMMIT, since a second resource manager is now involved and RRS may be required to manage the two-phase commit. The asynchronous process needs to issue a GET and has to perform the INSERT into the second table. Comparing the cost of the two solutions, consider the dual storage cost, the cost of the extra GET and PUT, and the cost of the two inserts.

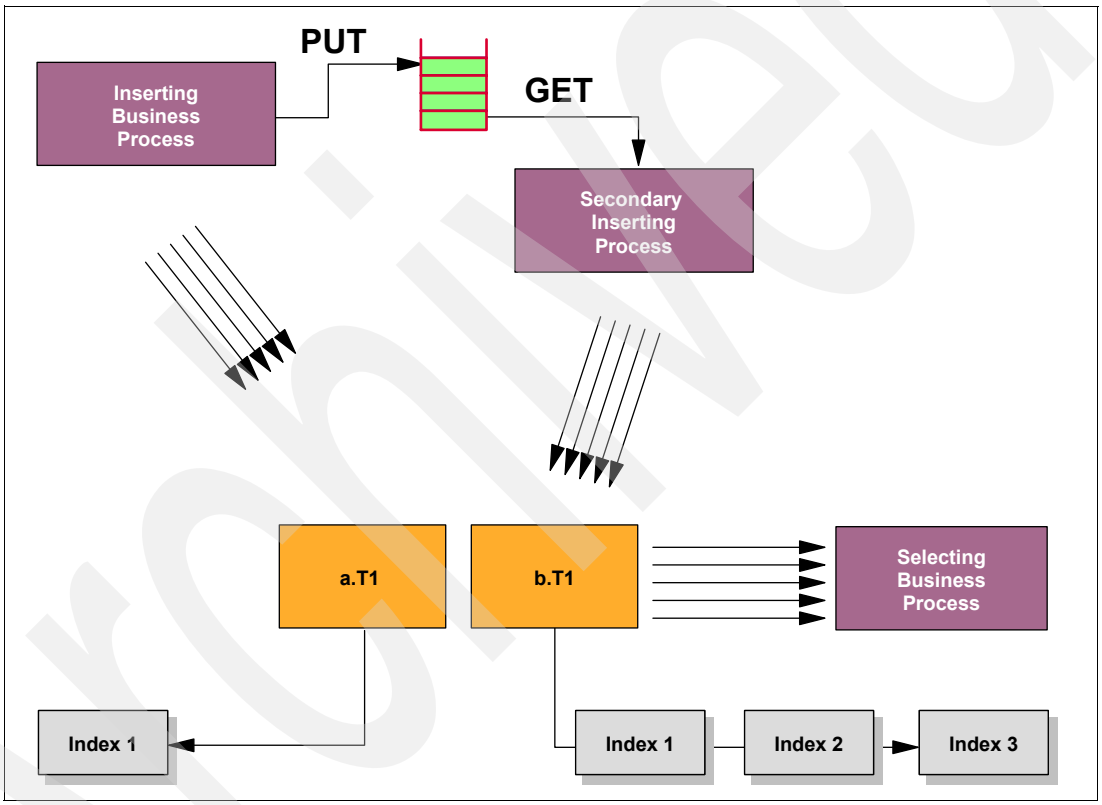


Figure 13-5 Using a queue to keep two tables in line

Another alternative is to use a log capture-based solution, since this has no impact on the first process that does the initial insert. The extra activity is performed by the log capture process that scans the log, inserts the data into a queue or in a DB2 table so that it can be picked up by the apply side which will put the data in the target table. It looks like the *build your own* option is better, but using WebSphere MQ-based replication is more efficient than building your own version of the queue-based replication because there is no impact on the process that performs the insert on the first table. See Figure 13-6 on page 375 for an overview of the design. Just stating that the replication solution is more efficient is incorrect because there are additional costs associated with using log-based capturing, such as extra logging, the need to scan the entire log and not just the inserts for which we are looking. The extra logging does not apply to the insert but any subsequent updates would need to log the entire record and not just the columns that changed. The cost of processing the log can be substantial, if you are already a heavy logger and just need to copy a small percentage of the data.

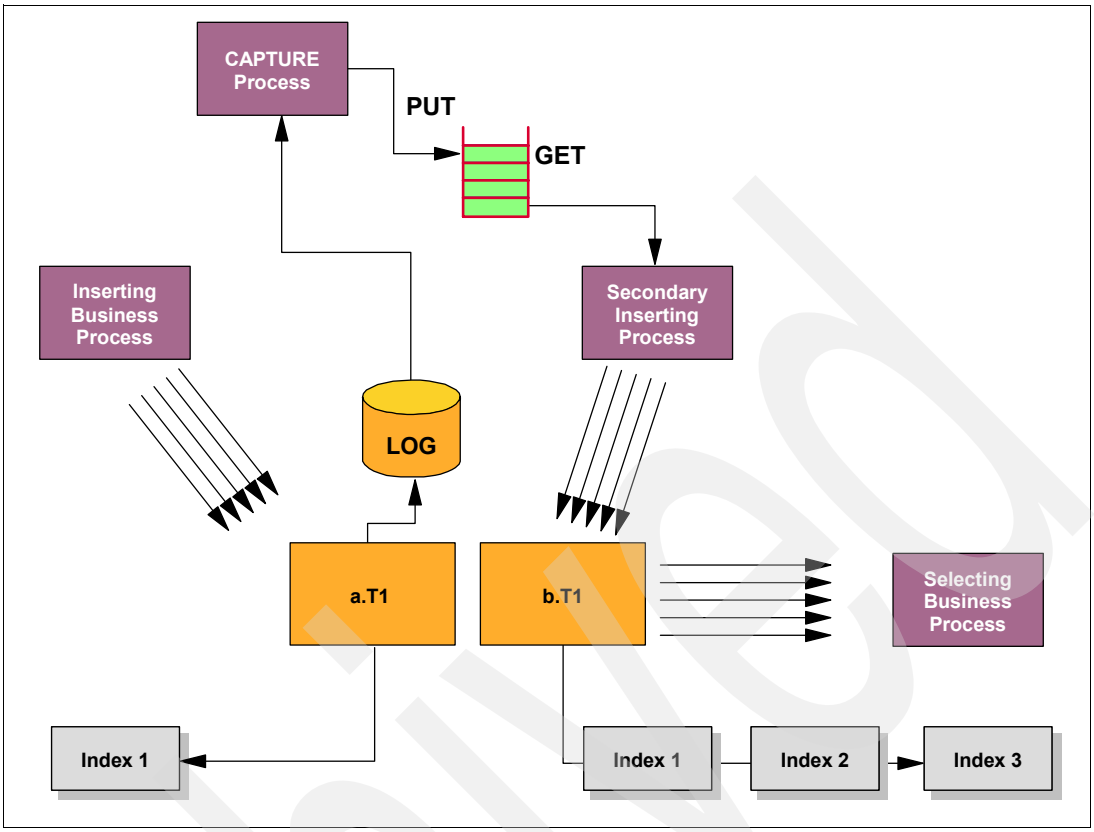


Figure 13-6 Using a Q-Replication to keep two tables in line

High latency solutions

If it takes a significant amount of time before the inserted data needs to be selected, there are additional alternatives that do not involve doing the second insert but that simply separate the data that is used by the insert from the data that is used by the select. This also means that there will be DDL involved since we switch from one cycle to another to add the indexes necessary for the select process and to remove the added indexes before the new insert cycle starts using the object. DDL concurrency is an important consideration when using this type of approach because you do not want to interrupt the insert flow.

An example of high latency table switching is tracking invoices for a mobile phone company. All new calls need to be registered for invoicing, but the actually invoicing is only done at a later stage. The insert of every phone call needs to be as fast as possible and probably does not involve any indexing. The invoicing cycle may need several indexes to get the required throughput and performance. So, a solution is to define two tables where the data can be stored, one without indexes where you do all the inserts, and one with indexes where the old data resides that has to be invoiced. As soon as the inserted data ages and becomes part of the invoicing window, the required indexes need to be added.

Using partitioning does not solve this problem because DB2 does not allow you to define indexes on a limited number of partitions or set of column values. Using a partition for inserted and a partition for invoiced will not change the index issue. You need a multi-table approach to solve the multiple indexing scheme problem.

Using automatic table switching options

How can we use an efficient technique to switch between the different objects? Copy A of the table is being used for the inserts and copy B contains the inserts of the previous interval that is now being processed by the select process. See Figure 13-7 for a schematic representation.

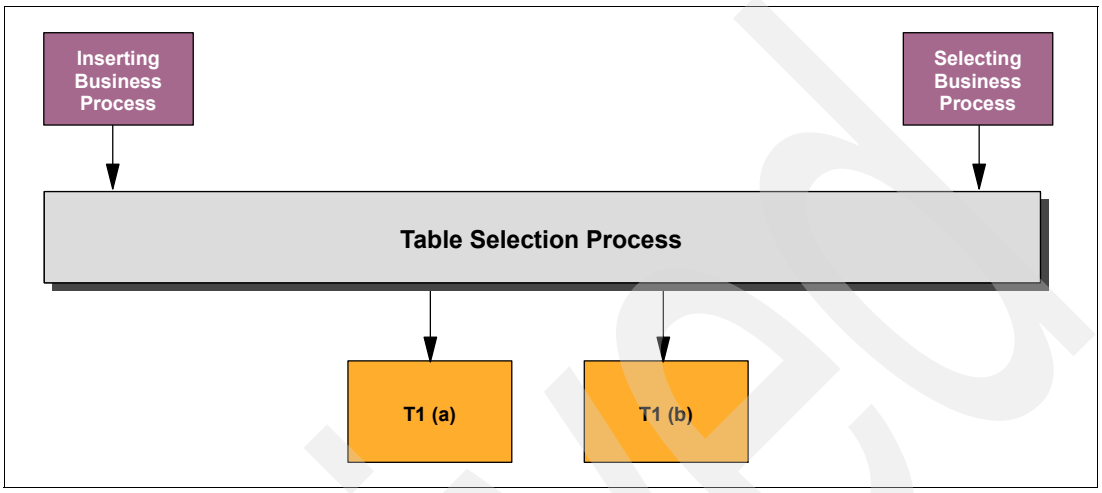


Figure 13-7 Table select process when using multiple tables

Package switching

Using multiple qualifiers is one way of providing different indexing models in parallel. The special register CURRENT PACKAGESET allows a program to switch fast and without additional coding between two or more copies of the table. The main advantage of package switching is that the client code only needs to deal with setting the CURRENT PACKAGESET and that mapping to the correct set of tables is dealt with at package bind time.

Making the process work requires you to BIND the application package in two or more collections and varying table qualifiers. See Figure 13-8 on page 377 for an overview of the components.

Archived

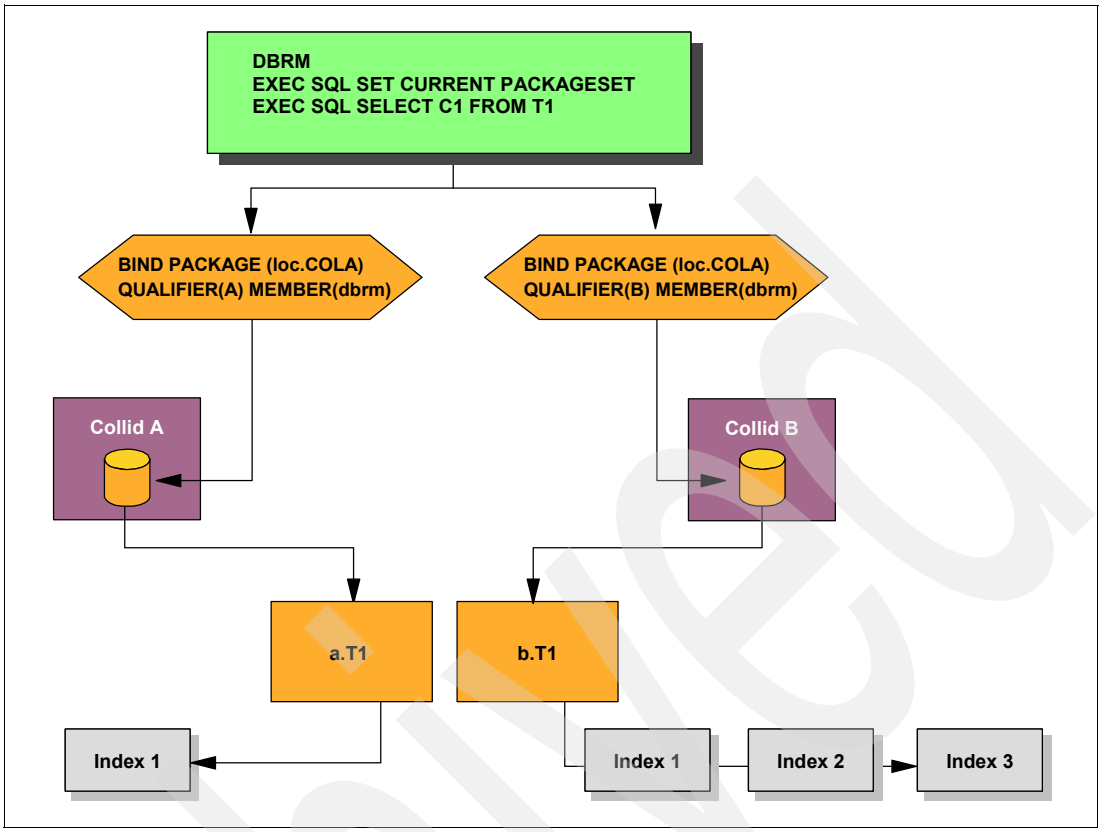


Figure 13-8 Bind DBRM into multiple collections with different QUALIFIER

Let us have a look at the example of the mobile phone company.

On day one the inserts are taking place into copy A of the table until midnight when the inserting applications switch to table copy B by setting the package set to the B collection. Now the preparatory steps are performed to get the invoicing processing running against the A copy of the table.

The steps required prior to the execution of the selection process are:

1. Define index 2 and index 3, use the DEFER option.
2. REBUILD index 2 and index 3 with inline statistics.
3. Optionally restore statistics on index 2 and index 3 through catalog update if inline statistics were not used.
4. REBIND the packages in collection A used by the select process.

The selection process can now be run without any issue and can use the newly defined indexes. Once the selection process is completed, one might decide to move the processed data into a historical staging area. So an REORG DISCARD, which discards all the rows, can move the data in copy A to a file which can subsequently be added by means of a LOAD into the copy C of that table, a table that holds the history.

Now table A needs to be prepared to receive the new inserts once the interval expires. The steps are:

1. DROP the index 2 and index 3.
2. REBIND the packages in collection A used by the insert process.

On the next cycle the whole process repeats itself, the only difference is that copy A becomes copy B and vice versa.

13.4.4 Backing up the data

Recoverability of the data is a key issue as you need to cater for application problems and you should not forget that things can break at any point. Successful recoverability starts often at the conception of the application and the design of the tables, table spaces, and indexes.

Your systems programmer should worry about the dimension of losing a table or index because of a disk failure, but every application program needs to contribute in the effort of keeping a good SLA on the recovery of a disk loss. A key element for every application is the commit frequency since it will determine how much logging needs to be processed in the event of a ROLLBACK. An application that does not perform frequent commits is difficult to deal with if the system programmer wants to get some level of consistency.

If the application is performing changes in concurrence with other tasks and if the scope of the work is bigger than the DB2 unit of work, you probably cannot rely on any system job to deal with recoverability. See “Unit of work” on page 347 for an explanation of the unit of work concept. Using a system type of backup and restore will simply lead to missing updates and incorrect data, since an application might have taken a decision on data that is reset by performing a restore. This means that the only way the application can recover from a logical error is to use the “*the fix and go forward*” approach. You cannot rely on the system being able to restore the state of the object prior to the execution as other units of work have probably touched the data that was changed by your application. The DB2 ROLLBACK only has the ability to undo the actions in the last unit of work. The RECOVER utility has the ability to put objects into a prior point of time but that would not be able to isolate a specific unit of work. There are tools in the market that allow for a more restrictive rollback or redo implementation, but that does not deal with the whole issue of other units of work that may have taken business decisions on the values that you put in the table.

In cases where you still have the traditional batch design involving monostream data access as shown in Table 13-2, it is possible to restore the initial backup and start again as though no concurrent updates were performed. This is not an option in most cases today.

Table 13-2 Flow of nonconcurrent batch chain

Step	Description
1	Back up the tables that are about to be updated using SHREVEL REFERENCE.
2	Take an optional QUIESCE point of all the table spaces involved.
3	Perform the batch jobs in a single stream.
4	Backup the tables that have been updated using SHREVEL REFERENCE.
5	Take an optional QUIESCE point of all the table spaces involved.
6	Start the table spaces in RW.

In today’s online parallel world, there is not a silver bullet solution because anything can happen to the rows you change after they are committed. It is up to the application to define what happens to your data if it is committed by DB2 but if your business transaction has not completed yet. Not performing the COMMIT is not an option because there are too many ramifications. The only answer is to introduce some level of logical locking that flags the data as logically incomplete. This means that the reading application must recognize this case as DB2 locking will no longer deal with the concurrency checking.

14

Determining the application quality

In this chapter, we discuss various approaches you can use to verify the efficiency and performance of your application. We describe multiple options you have prior to running the application, during the execution of the application, and afterward.

We also give you guidance for analyzing a DB2 accounting report, pointing out the significant areas, and alerting you to things to watch out for.

This chapter contains the following topics:

- Performance expectations
- Verifying during program preparation
- Verifying at run time
- Verifying after the fact

© Copyright IBM Corp. 2005. All rights reserved.

14.1 Performance expectations

We have already seen many examples throughout this redbook of how mileage can vary with similar statements under different circumstances. We have also seen that there are different elements that influence the performance of your application. They includes environmental aspects, such as disk response times, CPU utilization, WLM classification rules, CF utilization, CF link activity, and many more. Often these aspects are outside the sphere of influence of the programmer, or even the DBA.

With machines getting faster and faster, inefficiencies can go unnoticed. You can look at this in a philosophical way and assume that something is only a problem when someone complains, or you can worry and try to fix the problem. The latter is the better approach since more and more companies switch to a usage-based pricing model where the user response time is no longer your only concern. If an online transaction takes 100 µseconds more that it should, that will not be noticed by the user. You, however, might have an increased CPU consumption of 5% which will not go unnoticed at the end of the month when you report your system usage to IBM.

You need to have some kind of metric to determine if you are within reasonable boundaries, given the type of workload that is performed. Having the programmer indicate the expected performance can be a starting point, but it does not solve all the problems because the physical implementation can change over time.

14.2 Verifying during program preparation

If you are not using a tool to do the job, such as Visual Explain, you can perform SQL quality control by accessing the data provided by DB2 Explain in the PLAN_TABLE, the DSN_STATEMNT_TABLE, and the DSN_FUNCTION_TABLE. These tables give you an indication of the selected access path and the amount of resources DB2 *thinks* are necessary to perform the SQL.

EXPLAIN is the function that produces information about the following:

- A plan, package, or SQL statement when it is bound.

  The output appears in a table that you create called PLAN_TABLE (also called plan table). Experienced users can use the PLAN_TABLE to give optimization hints to DB2.
- An estimated cost of executing an SQL SELECT, INSERT, UPDATE, or DELETE statement.

  The output appears in a table that you create called DSN_STATEMNT_TABLE (also called statement table).
- User-defined functions referred to in the statement, including the specific name and schema.

  The output appears in a table that you create called DSN_FUNCTION_TABLE (also called function table).

The information contained in PLAN_TABLE can help you to:

- Design databases, indexes, and application programs
- Determine when to rebind an application
- Determine the access path chosen for a query

For each access to a single table, EXPLAIN tells you if an index access or table space scan is used. If indexes are used, EXPLAIN tells you how many indexes and index columns are used

and what I/O methods are used to read the pages. For joins of tables, EXPLAIN tells you which join method and type are used, the order in which DB2 joins the tables, and when and why it sorts any rows. The primary use of EXPLAIN is to observe the access paths for the SELECT parts of your statements. For UPDATE, DELETE WHERE CURRENT OF, and INSERT, you receive somewhat less information in your plan table. And some accesses EXPLAIN does not describe: for example, the access to LOB values, which are stored separately from the base table, and access to parent or dependent tables needed to enforce referential constraints. The access paths shown for the example queries in this chapter are intended only to illustrate those examples. If you execute the queries in this chapter on your system, the access paths chosen can be different.

Use the following overall steps to obtain information from EXPLAIN:

- Have appropriate access to a plan table.
- Populate the table with the information you want.
- Select the information you want from the table.

Although this is very useful information, there are several additional aspects that need to be considered when validating the quality of the SQL and the program logic.

- You need to consider frequency of execution.

  A reasonably innocent SQL statement that performs a matching index scan may still end up using minutes of CPU when executed too frequently. There is no indication of frequency of execution prior to run time.

- No insight in the efficiency of I/O (for instance, dynamic prefetch).

  You might see a list prefetch which gives you a hint of possible I/O inefficiencies in the tables but any SQL can cause excessive I/O wait time.

- Information can be blinded because of values of host variables.

  You may have a matching index scan but the value of the host variable at run time may be a very common value or the content of the host variable on the LIKE could start with a “%” character.

14.3 Verifying at run time

It is possible to *intercept* the SQL while it is running and stop it from using excessive amounts of resources. The Resource Limit Facility (RLF) is a DB2 function, defined at DB2 install time, which provides the ability to prevent SQL from starting, which we call *preemptive governing*, or it can stop activity after a certain amount of resources are consumed, which we call *reactive governing*. The RLF can also issue warning SQLCODEs to allow the application to decide to run the SQL or adapt it.

The usability of the governing feature depends on the kind of application. It is difficult to set the values for transactional systems, because they should typically contain well behaving SQL that has been validated prior to run time. Experience tells us that most shops just tend to increase the limits every time an online application hits the limit. The RLF option is, however, very useful in BI-type workloads where the resource consumption can sometimes be inappropriate for the time of the day.

14.3.1 Using RLF

RLF (or governor) lets you perform the following activities:

- Set warning and error thresholds by which the governor can inform users (via your application programs) that a certain processing limit might be exceeded for a particular

dynamic SELECT, INSERT, UPDATE, or DELETE statement. This is the predictive governing.

- Stop a currently executing dynamic SQL statement (SELECT, INSERT, UPDATE, or DELETE) that exceeds the processor limit that you have specified for that statement. This is often called reactive governing to differentiate its function from that of predictive governing. The RLF does not control static SQL statements whether they are executed locally or remotely.
- Restrict bind and rebind activities to avoid performance impacts on production data.
- Restrict particular parallelism modes for dynamic queries.

You have the following options to control governing:

- Reactively govern dynamic SELECT, INSERT, UPDATE, or DELETE statements by plan name.
- Reactively govern bind operations.
- Reactively govern dynamic SELECT, INSERT, UPDATE, or DELETE statements by package or collection name.
- Disable query I/O parallelism.
- Disable query CP parallelism.
- Disable Sysplex query parallelism.
- Predictively govern dynamic SELECT, INSERT, UPDATE, or DELETE statements by plan name.
- Predictively governs dynamic SELECT, INSERT, UPDATE, or DELETE statements by package or collection name.

The predictive governing capability has an advantage over the reactive governor in that it avoids wasting processing resources by giving you the ability to prevent a query from running when it appears that it exceeds processing limits.

With the reactive governor, those resources are already used before the query is stopped. See Figure 14-1 on page 383 for an overview of how predictive governing works.

At prepare time for a dynamic SELECT, INSERT UPDATE, or DELETE statement, DB2 searches the active resource limit specification table (RLST) to determine if the processor cost estimate exceeds the error or warning threshold that you set in the RLFASUWARN and RLFASUERR columns for that statement.

DB2 compares the cost estimate for a statement to the thresholds you set, and the following actions occur:

- If the cost estimate is in cost category A and the error threshold is exceeded, DB2 returns a -495 SQLCODE to the application, and the statement is not prepared or run.
- If the estimate is in cost category A and the warning threshold is exceeded, a +495 SQLCODE is returned at prepare time. The prepare is completed, and the application or user decides whether to run the statement or not.
- If the estimate is in cost category B, DB2 takes the action you specify in the RLF_CATEGORY_B column; that is, it either prepares and executes the statement, does not prepare or execute the statement, or returns a warning SQLCODE, which lets the application decide what to do.
- If the estimate is in cost category B and the warning threshold is exceeded, a +495 SQLCODE is returned at prepare time. The prepare is completed, and the application or user decides whether to run the statement or not.

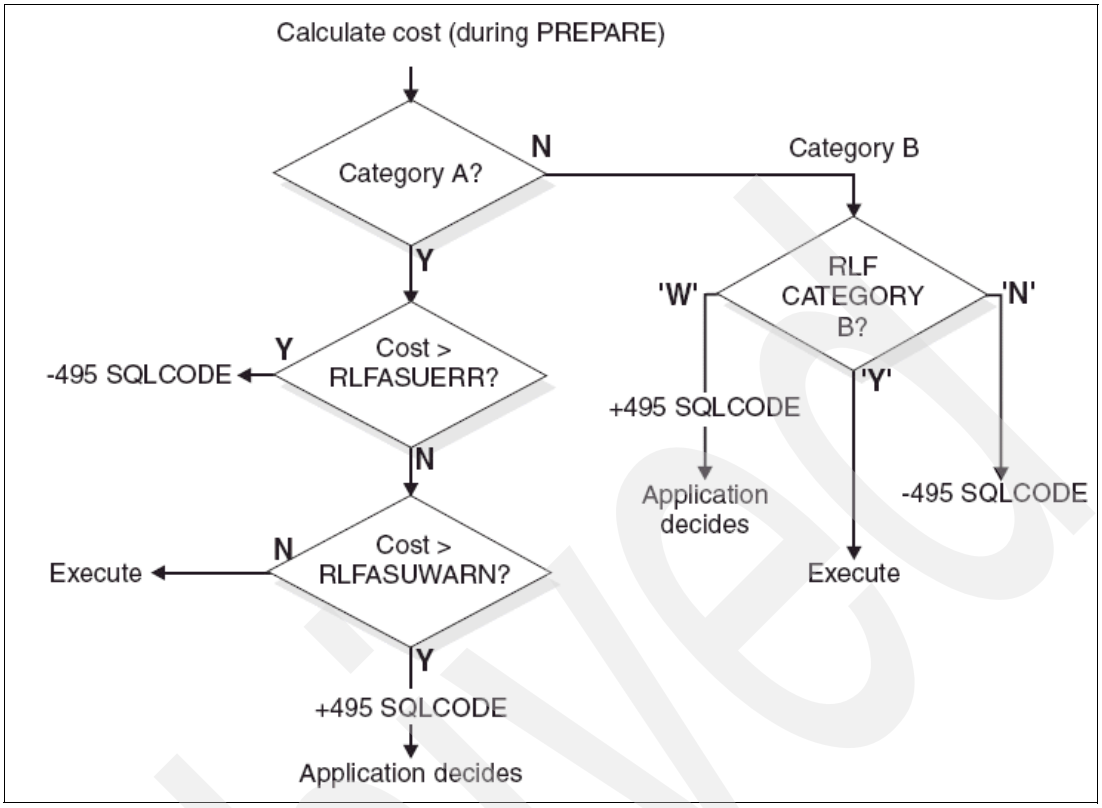


Figure 14-1 Processing for predictive governing

14.3.2 Using DSNZPARM options

DB2 has the capability to track applications that do not behave according to the guidelines. The system programmer can set thresholds after which a console message and/or trace record can be created. This helps the system programmer to track badly behaving applications without having to scan huge amounts of trace information. It might be a good approach to set the values more aggressively in your test systems because the volumes tend to be smaller in these environments.

Lock escalation

Lock escalation is the act of releasing a large number of page, row or LOB locks, held by an application process on a single table or table space, to acquire a higher level table or table space lock, or a set of partition locks, of mode S or X instead. When it occurs, DB2 issues message DSNI031I. This identifies the table space for which lock escalation occurred and the plan or package active when the escalation occurred. See Example 14-1 on page 384.

Example 14-1 DSNI031I message after lock escalation

```
DSNI031I -DB3 DSNILKES - LOCK ESCALATION HAS
          OCCURRED FOR
          RESOURCE NAME = DB2.B0TW0AA
          LOCK STATE = X
          PLAN NAME : PACKAGE NAME = INMYPLA : RWMYPA
          STATEMENT NUMBER = 000666
          CORRELATION-ID = 0016FDEHF904
          CONNECTION-ID = IPLC
          LUW-ID = MYCON004.EDBPL3.BD20898F2CDF
          THREAD-INFO = AVUCC : * : *
```

Not committed for *n* checkpoints

You can specify the number of checkpoint cycles that are to complete before DB2 issues a warning message to the console and the instrumentation facility for an uncommitted unit of recovery (UR). If you use this option, specify a value that is based on how often a checkpoint occurs in your system and how much time you can allow for a restart or shutdown. For example, if your site's checkpoint interval is 5 minutes and the standard limit for issuing commits with units of recovery is 20 minutes, divide 20 by 5 to determine the best value for your system.

The value is set by means of the URCHKTH DSNZPARM option. When DB2 detects that a UR exceeds the specified value, a console message (see Example 14-2) is issued and a trace record is written when the statistics class 3 trace is active. The IFCID 313 signals the long-running URs (inflight URs and indoubt URs) that exist during a checkpoint, or after the threshold for log records written without a COMMIT is exceeded.

This option is disabled by default. This option does not affect performance.

Example 14-2 DSNR035I output after not committing during four checkpoints

```
DSNR035I . DSNRPBCW WARNING - UNCOMMITTED UR AFTER 4
          CHECKPOINTS -
          CORRELATION NAME = CTHDCORID001
          CONNECTION ID = TEST0001
          LUWID = USIBMSY.SYEC1DB2.B066C0A407D0 = 10
          PLAN NAME = JRSSQL1
          AUTHID = SYSADM
          END USER ID = *
          TRANSACTION NAME = *
          WORKSTATION NAME = *
DSNR036I . DSNRPBCW WARNING - UNRESOLVED UR
          ENCOUNTERED DURING CHECKPOINT -
          CORRELATION NAME = CTHDCORID001
          CONNECTION ID = TEST0001
          LUWID = USIBMSY.SYEC1DB2.B066C0A407D0
          PLAN NAME = JRSSQL1
          AUTHID = SYSADM
```

Not committed in a RO UOW

You can specify the number of minutes that a read claim can be held by an agent before DB2 issues a warning message to report it as a long-running reader by means of the LRDRTHLD DSNZPARM. If you specify a value of 0, DB2 will not report long-running readers.

Notice that even *read-only* transactions should commit regularly. This is to allow utilities (that need to drain the object) to take control of the object. For example, during the switch phase of

online REORG, the utility needs full control of the object to switch between the shadow data set and the actual data set used by DB2. For parallel operations, DB2 records this information only for the originating task. DB2 writes an IFCID 313 record to report when an application reaches this threshold.

Not committed for *n* log records

You can specify the number of log records that are to be written by an uncommitted unit of recovery (UR) before DB2 issues a warning message to the console by means of the URLGWTH DSNZPARM. The purpose of this option is to provide notification of a long-running UR. Long-running URs might result in a lengthy DB2 restart or a lengthy recovery situation for critical tables. A value of 0 indicates that no write check is to be performed.

14.4 Verifying after the fact

DB2 traces provide you with a wealth of information that gives you a better insight of the behavior of the individual applications and the overall system. The entire z/OS infrastructure has an excellent monitoring and reporting function that can report on the behavior of the system after the fact. All measuring (active traces) is going to impact the performance of the original task at one point or another. So you have to find the delicate balance between the overhead of measuring and the risk of running without monitoring, and, not knowing why things went wrong.

14.4.1 Buffer pool efficiency

Buffer pool efficiency directly relates to SQL execution time. When looking at the information that you can get from the system in terms of buffer pool efficiency, there are two categories of sources, the DB2 traces and the SMF/RMF traces. Figure 14-2 gives you an overview of all the potential traces that might help you when looking at buffer pool efficiency.

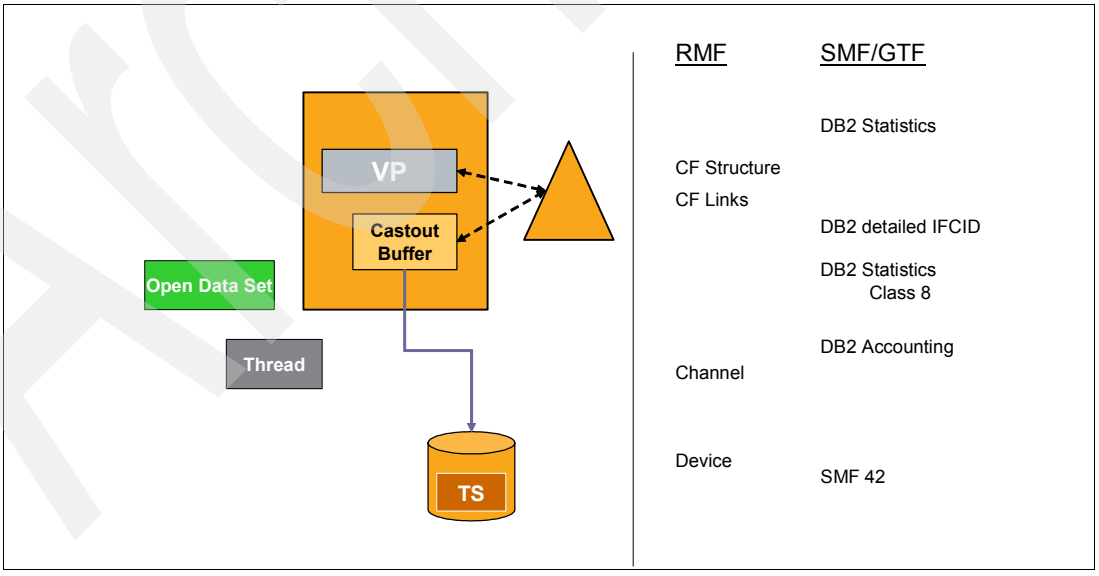


Figure 14-2 Buffer pool monitoring options

14.4.2 Accounting and statistics

When looking at DB2 performance, you can use the Instrumentation Facility of DB2. This facility provides accounting, statistics, and record traces. The accounting trace gives a task-related view of the workload. In “Accounting traces” on page 387, we analyze in some detail, the different elements that you can find on the report. It is worth mentioning that DB2 does not have any interval accounting records, other than using the IFI interface. This means that the trace contains the start and end times, but it does not tell you how the resources were consumed within that interval. This can be an issue when dealing with applications that have a significant amount of thread reuse and that do not create accounting records when the thread is reused. While not cutting the trace record might save some overhead, it could create a record that reflects 12 hours of activity. This can make the accounting record an insufficient source to explain the behavior of spiking activity.

Accounting records require a trade-off between information detail and the amount of records that are being created. In most cases, the accounting records are written to SMF, which is a shared resource in z/OS. Large quantities of accounting records do not always provide a lot of added value, because they are simply describing a large quantity of small well behaving transactions.

There are ways to reduce the number of records that are written for CICS regions and DDF connections.

CICS thread reuse is a very good way to reduce resource consumption and reduce the number of SMF records. There are some additional considerations for CICS. If the CICS application is doing an EXEC SQL SET CURRENT PACKAGESET, this command alters the setting of the CURRENT PACKAGESET special register. At transaction termination, if a special register is not in its initial state, CICS will do a partial sign-on to DB2 to reset the special register. This partial sign-on causes DB2 to cut an accounting record. You need to change the application to do a second EXEC SQL SET CURRENT PACKAGESET to set the special register back to its initial state. This change can significantly decrease the number of DB2 accounting records.

With WebSphere, WebLogic, and other e-business application servers or distributed applications that connect to DB2 for z/OS via DDF, to optimize the use of DB2’s resources, we recommend you use CMTSTAT=INACTIVE (DSNZPARM). This normally allows DB2 to separate the DBAT from the connection (at commit time) and reuse the DBAT to process somebody else’s work while your connection is inactive. This is called DB2 inactive connection support (sometimes inaccurately called 'type 2 inactive threads).

A side effect of using CMTSTAT=INACTIVE is that DB2 cuts an accounting record when the connection becomes inactive, normally on every COMMIT or ROLLBACK. If you are running high volume OLTP in this environment, the large volume of accounting records can be a problem, since you can end up flooding SMF, compromising your ability to do chargeback, or do performance monitoring and tuning.

When using the RRS attach, you can run into a similar problem. WebSphere on z/OS drives the RRS sign-on interface on each new transaction, and DB2 cuts an accounting record when this sign-on happens. Some of these transactions are very short (for example, just one SELECT statement followed by a COMMIT), but they still result in a DB2 accounting record being produced.

The other type of record is the statistics record. See “Statistics trace” on page 397. This type of record uses a view that includes all applications and provides general counters independent of the applications. This kind of record is time- based and the interval size can be determined by setting the DSNZPARM. Beware that it is not very likely to have the

statistics match up with the aggregation of the accounting records over the same period of time due to the nature of the accounting trace.

14.4.3 Accounting traces

The DB2 accounting trace is a direct way to determine the resource consumption of an application. The level of detail that you obtain is related to the trace classes that are activated. The highest level of tracing, class 1, is the standard accounting time. The class 2 level of tracing gives an insight in the time that was spent inside the SQL processing. You get the elapsed time and the CPU consumption within DB2. If time is spend waiting in DB2, you need to activate class 3 to determine the cause of the wait. Figure 14-3 gives a schematic overview of the breakdown by the different trace classes. Classes 1, 2, and 3 provide the information by plan. Classes 7 and 8 are the equivalent of Classes 2 and 3 at the level of package or DBRM.

It is obvious that tracking all the events requires CPU cycles, and that activating additional classes has an impact on the overall resource consumption.

Recommendation: Activate accounting trace and use classes 1, 2, 3, 7, and 8.

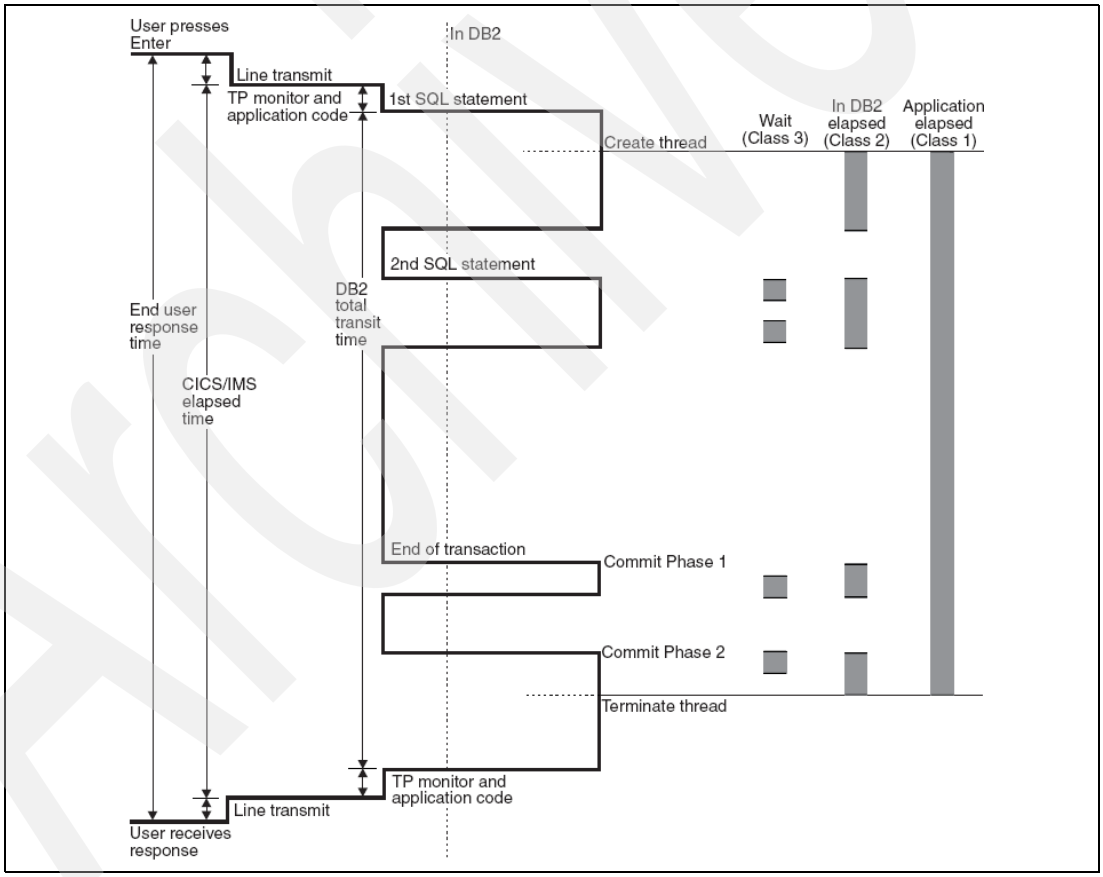


Figure 14-3 Schematic overview of accounting classes

The number of accounting records that are produced daily can be extremely significant and most likely requires some kind of automation to determine anomalies. We strongly recommend that you routinely investigate the accounting details and verify that the applications are still behaving efficiently. There are several ways to search for problem areas.

One way is by using *key performance indicators* (KPI). Every installation should have a set of KPIs that are used to monitor application efficiency. This can be done by first establishing a baseline from your currently *correctly* behaving applications. You need to calibrate your own environment, because the cost of the SQL depends on the complexity of your physical model, whether you are using data sharing, the type of processing (BI or OLTP), and so on.

Some of the KPIs you can use are:

- CPU per SQL statement

  This ratio gives a very good indication of the efficiency of the statement as inefficient access paths and processing tend to cause high CPU consumption.

- I/O wait time per SQL statement

  This ratio is an indication that you are either processing the rows in an inefficient sequence or that there is something wrong with the access path.

- Elapsed time per SQL statement

  This ratio is a first indication of efficiency. The underlying cause could still be anything, but it should be the starting point to trigger further investigation.

- GETPAGEs per SQL statement

  This ratio gives a very good indication of the amount of pages that are required to find the results. Higher ratios are not uncommon in a BI environment since these queries tend to do a high level of aggregation, but OLTP workloads should have a low number of GETPAGEs per SQL.

Figure 14-4 gives a sample report of the KPIs of one plan and the underlying packages.

DB2 FOOTPRINT							
DB2 Cpu consumption	0.101681		SELECT	56			
Application Cpu consumption	0.120863		INSERT	41			
DB2 Elapsed	0.194379		UPDATE	24			
Application Elapsed	4.303569		DELETE	1			
Number of packages used	27		OPEN	9			
DB2 pages referenced per SQL stmt	2.28		FETCH	14			
CPU cost per stmt	0.66		CLOSE	9			
Total number of DB2 pages referenced	352		SQLTMTS	155			
CLASS	METHOD	PACKAGE	ELAPSED_STMT	CPU_STMT	ELAPSED	CPU	SQLSTMT:
DALI_ACCOUNTS_WITH_LENT	EXECUTE	MDDVT02	1.18833	0.5855	7.13	3.513	6
DAL_SETTLEMENT_INSTRUCTION	SET_STATUS	MDDSN07	0.5157	0.4464	5.157	4.464	10
BUILD_TESTING	CICS_TEST_HARNESS	MDBLD02	2.67757	0.61371	18.743	4.296	7
DAL_REIMBURSEMENT_INSTR	IS_REIMB_INSTR_CREATED	MDDQR05	9.779	6.175	9.779	6.175	1
DAL_REIMBURSEMENT_INSTR	CREATE_FROM_DATA	MDDQR01	0.88	0.875	1.76	1.75	2

Figure 14-4 Table with overview of plan and package efficiency

Let us have a look at a detailed accounting report that was produced by the DB2 Performance Expert V2. We describe the meaning of the fields on this report. For convenience, we split the report in two parts. Figure 14-5 on page 389 shows the first part of the report. The *DB2 UDB for z/OS Version 8 Administration Guide,* SC18-7413-02, in Chapter 24. Analyzing performance data, section Reading accounting reports from DB2 Performance Expert gives more information about this topic.

```
ACCT TSTAMP: 07/22/05 23:22:40.76    PLANNAME: DSNESPCS        WLM SCL: 'BLANK'          CICS NET: N/A
 BEGIN TIME : 07/22/05 23:22:40.48    PROD ID : N/P                                       CICS LUN: N/A
 END TIME   : 07/22/05 23:22:40.76    PROD VER: N/P             LUW NET: USIBMSC          CICS INS: N/A
 REQUESTER  : DB8F                    CORRNAME: PAOLOR2         LUW LUN: SCPD8F1
 MAINPACK   : DSNESM68                CORRNMBR: 'BLANK'         LUW INS: BD595A0537EB     ENDUSER : 'BLANK'
 PRIMAUTH   : PAOLOR2                 CONNTYPE: TSO             LUW SEQ:             1    TRANSACT: 'BLANK'
 ORIGAUTH   : PAOLOR2                 CONNECT : TSO                                       WSNAME  : 'BLANK'

ELAPSED TIME DISTRIBUTION                                      CLASS 2 TIME DISTRIBUTION
 ------------------------------------------------------------   ------------------------------------------------------------
 APPL   |============> 24%                                      CPU    |===> 6%
 DB2    |===========> 23%                                       NOTACC |============> 24%
 SUSP   |==========================> 52%                        SUSP   |===================================> 69%

 TIMES/EVENTS  APPL(CL.1)  DB2 (CL.2)  IFI (CL.5)
 ------------  ----------  ----------  ----------
 ELAPSED TIME    0.276563    0.209139         N/P
  NONNESTED      0.276563    0.209139         N/A
  STORED PROC    0.000000    0.000000         N/A
  UDF            0.000000    0.000000         N/A
  TRIGGER        0.000000    0.000000         N/A
                   A
 CPU TIME        0.022709    0.012992         N/P
  AGENT          0.022709    0.012992         N/A
   NONNESTED     0.022709    0.012992         N/P
   STORED PRC    0.000000    0.000000         N/A
   UDF           0.000000    0.000000         N/A
   TRIGGER       0.000000    0.000000         N/A
  PAR.TASKS      0.000000    0.000000         N/A

 SUSPEND TIME    0.000000    0.144975         N/A
  AGENT               N/A    0.144975         N/A
  PAR.TASKS           N/A    0.000000         N/A
  STORED PROC    0.000000         N/A         N/A
  UDF            0.000000         N/A         N/A

 NOT ACCOUNT.         N/A    0.051172         N/P
 DB2 ENT/EXIT         N/A          98         N/A
 EN/EX-STPROC         N/A           0         N/A
 EN/EX-UDF            N/A           0         N/A
 DCAPT.DESCR.         N/A         N/A         N/P
 LOG EXTRACT.         N/A         N/A         N/P

 CLASS 3 SUSPENSIONS   ELAPSED TIME   EVENTS
 -------------------   ------------   --------
 LOCK/LATCH(DB2+IRLM)      0.000000        0
 SYNCHRON. I/O             0.074456       42
  DATABASE I/O             0.000000        0
  LOG WRITE I/O            0.074456       42
 OTHER READ I/O            0.000000        0
 OTHER WRTE I/O            0.000000        0
 SER.TASK SWTCH            0.004303        1
  UPDATE COMMIT            0.004303        1
  OPEN/CLOSE               0.000000        0
  SYSLGRNG REC             0.000000        0
  EXT/DEL/DEF              0.000000        0
  OTHER SERVICE            0.000000        0
 ARC.LOG(QUIES)            0.000000        0
 ARC.LOG READ              0.000000        0
 DRAIN LOCK                0.000197        4
 CLAIM RELEASE             0.000000        0
 PAGE LATCH                0.000000        0
 NOTIFY MSGS               0.000000        0
 GLOBAL CONTENTION         0.002139       11
 COMMIT PH1 WRITE I/O      0.000000        0
 ASYNCH CF REQUESTS        0.063880       45
 TOTAL CLASS 3             0.144975      103
                B

 HIGHLIGHTS
 ------------------------
 THREAD TYPE   : ALLIED
 TERM.CONDITION: NORMAL
 INVOKE REASON : DEALLOC
 COMMITS       :        1
 ROLLBACK      :        0
 SVPT REQUESTS :        0
 SVPT RELEASE  :        0
 SVPT ROLLBACK :        0
 INCREM.BINDS  :        0
 UPDATE/COMMIT :     0.00
 SYNCH I/O AVG.: 0.001773
 PROGRAMS      :        1
 MAX CASCADE   :        0
 PARALLELISM   : NO

 GLOBAL        CONTENTION       L-LOCKS  ELAPSED TIME   EVENTS
 ----------------------------------------  ------------  --------
 L-LOCKS                          C          0.002139        11
  PARENT (DB,TS,TAB,PART)                    0.001530        10
  CHILD  (PAGE,ROW)                          0.000000         0
  OTHER                                      0.000609         1

 GLOBAL        CONTENTION       P-LOCKS  ELAPSED TIME   EVENTS
 ----------------------------------------  ------------  --------
 P-LOCKS                          D          0.000000         0
  PAGESET/PARTITION                          0.000000         0
  PAGE                                       0.000000         0
  OTHER                                      0.000000         0

 SQL DML     TOTAL   SQL DCL       TOTAL   SQL DDL     CREATE  DROP  ALTER
 -------- --------   ----------  --------  ----------  ------ ------ ------
 SELECT          0   LOCK TABLE         0  TABLE            0      0      0
 INSERT          0   GRANT              0  CRT TTABLE       0    N/A    N/A
 UPDATE          0   REVOKE             0  DCL TTABLE       0    N/A    N/A
 DELETE          0   SET SQLID          0  AUX TABLE        0    N/A    N/A
                     SET H.VAR.         0  INDEX            0      0      0
 DESCRIBE        0   SET DEGREE         0  TABLESPACE       0      0      0
 DESC.TBL        0   SET RULES          0  DATABASE         0      0      0
 PREPARE         1   SET PATH           0  STOGROUP         0      0      0
 OPEN            1   SET PREC.          0  SYNONYM          0      0    N/A
 FETCH          43   CONNECT 1          0  VIEW             0      0    N/A
 CLOSE           1   CONNECT 2          0  ALIAS            0      0    N/A
                     SET CONNEC         0  PACKAGE        N/A      0    N/A
                     RELEASE            0  PROCEDURE        0      0      0
 DML-ALL        46   CALL               0  FUNCTION         0      0      0
                     ASSOC LOC.         0  TRIGGER          0      0    N/A
                     ALLOC CUR.         0  DIST TYPE        0      0    N/A
                     HOLD LOC.          0  SEQUENCE         0      0      0
                     FREE LOC.          0
        E            DCL-ALL            0  TOTAL            0      0      0
                                           RENAME TBL       0
                                           COMMENT ON       0
                                           LABEL ON         0

 LOCKING                   TOTAL
 ------------------------ --------
 TIMEOUTS                        0
 DEADLOCKS                       0
 ESCAL.(SHAR)                    0
 ESCAL.(EXCL)                    0
 MAX PG/ROW LCK HELD             2
 LOCK REQUEST                   45
 UNLOCK REQST                   26
 QUERY REQST                     0
 CHANGE REQST                    3
 OTHER REQST                     0
 LOCK SUSPENS.                   0
 IRLM LATCH SUSPENS.             0
 OTHER SUSPENS.                  0
 TOTAL SUSPENS.                  0
                F

 DATA SHARING     TOTAL
 ------------ --------
 L-LOCKS  (%)       33
 P-LOCK REQ         42
 P-UNLOCK REQ       42
 P-CHANGE REQ        0
 LOCK - XES         57
 UNLOCK-XES         46
 CHANGE-XES          0
 SUSP - IRLM         0
 SUSP - XES          0
 SUSP - FALSE      N/A
 INCOMP.LOCK         0
 NOTIFY SENT         0
                G
```

Figure 14-5 DB2 accounting report: Part 1

Class 1 and class 2 (A)

Class 1 elapsed time:

Compare this with the CICS, IMS, WebSphere, or distributed application transit times:

- In CICS, you can use CICS Performance Analyzer to find the attach and detach times; use this time as the transit time.
- In IMS, use the PROGRAM EXECUTION time reported by IMS Performance Analyzer.

Differences between the CICS, IMS, WebSphere, or distributed application times and the DB2 accounting times arise mainly because the DB2 times do not include:

- Time before the first SQL statement, which for a distributed application includes the inbound network delivery time
- DB2 create thread
- DB2 terminate thread
- For a distributed application, the time to deliver the response to a commit, if database access thread pooling is used

Non-OTE CICS class 1 is about the same as class 2. It is all in DB2 time.

Differences can also arise from thread reuse in CICS, IMS, WebSphere, or distributed application processing or through multiple commits in CICS. If the class 1 elapsed time is significantly less than the CICS or IMS time, check the report from Tivoli® Decision Support for OS/390, IMS Performance Analyzer, or an equivalent reporting tool to find out why. Elapsed time can occur:

- In DB2, during sign-on, create, or terminate thread
- Outside DB2, during CICS or IMS processing

The problem might be caused by DB2 or IRLM traces, a change in access paths, or an increase in the number of users. In the DB2 Performance Expert accounting report, DB2 processor resource consumption is indicated in the field for class 2 CPU TIME (C).

- Not Accounted

This is time calculated as the difference between the class 1 and the class 2 elapsed time. It is time spent outside of DB2, but within the DB2 accounting interval. A lengthy time can be caused by thread reuse, which can increase class 1 elapsed time, or a problem in the application program, CICS, IMS, or the overall system. For a distributed application, not-in-DB2 time is calculated with this formula, reported in Chapter 24 of the *DB2 UDB for z/OS Version 8 Administration Guide,* SC18-7413-02:

```
Not_in_DB2 time = A - (B + C + (D - E))
```

Where the variables have the following values:

- A, Class 1 elapsed time
- B, Class 2 nonnested elapsed time
- C, Class 1 nonnested time of any stored procedures, user-defined functions, or triggers
- D, Class 1 nonnested CPU time
- E, Class 2 nonnested CPU time

The calculated not-in-DB2 time might be zero. Furthermore, this time calculation is only an estimate. A primary factor that is not included in the equation is the amount of time that requests wait for CPU resources while executing within the DDF address space. To determine how long requests wait for CPU resources, look at the NOT ACCOUNT field. The NOT ACCOUNT field shows the time that requests wait for CPU resources while a distributed task is inside DB2.

There is DB2 accounting class 2 elapsed time that is not recorded as class 2 CPU time or class 3 suspensions. The most common contributors to this category are:

- z/OS paging
- Processor wait time

- On DB2 requester systems, the amount of time waiting for requests to be returned from either VTAM® or TCP/IP, including time spent on the network and time spent handling the request in the target or server systems
- Time spent waiting for parallel tasks to complete (when query parallelism is used for the query)
- Some online performance monitoring
- Data sharing duplexed write to structure

Class 3 (B)

We have already explained that the class 3 times are a breakdown of the wait times. In some cases, multiple wait events can occur at the same time. The wait will only be reported once when this happens.

Other Read

The accumulated wait time for read I/O done under a thread other than this one. It includes time for:

- Sequential prefetch
- List prefetch
- Sequential detection
- Synchronous read I/O performed by a thread other than the one being reported

Generally, an asynchronous read I/O for sequential prefetch or sequential detection takes about 0.1 to 1 millisecond per page. List prefetch takes about 0.2 to 2 milliseconds per page.

Having a large Other Read wait time could be an indication that your package could benefit from introducing parallelism, and you might consider a REBIND with a DEGREE(ANY).

Service Task suspension

This is the accumulated wait time from switching synchronous execution units, by which DB2 switches from one execution unit to another. The most common contributors to service task suspensions are:

- Wait for phase 2 commit processing for updates, inserts, and deletes (UPDATE COMMIT). You can reduce this wait time by allocating the DB2 primary log on a faster disk. You can also help to reduce the wait time by reducing the number of commits per unit of work.
- Wait for OPEN/CLOSE service task (including HSM recall). You can minimize this wait time by using two strategies. If DSMAX is frequently reached, increase DSMAX. If DSMAX is not frequently reached, change CLOSE YES to CLOSE NO on data sets that are used by critical applications.
- Wait for SYSLGRNG recording service task.
- Wait for data set extend/delete/define service task (EXT/DEL/DEF). You can minimize this wait time by defining larger primary and secondary disk space allocation for the table space.
- Wait for other service tasks (OTHER SERVICE).

Lock/Latch

LOCK + LATCH TIME/ # of events should be considerably lower than the timeout value, because you might be close to getting a timeout instead of a simple suspend/resume.

Other Write

This is the accumulated wait time for write I/O done under a thread other than this one. It includes time for:

- Asynchronous write I/O
- Synchronous write I/O performed by a thread other than the one being reported

As a guideline, an asynchronous write I/O takes 0.1 to 2 milliseconds per page.

Sync I/O

This is the total application wait time for synchronous I/Os. It is the total of database I/O and log write I/O. If the number of synchronous read or write I/Os is higher than expected, check for:

- A change in the access path to data. If you have data from accounting trace class 8, the number of synchronous and asynchronous read I/Os is available for individual packages. Determine which package or packages have unacceptable counts for synchronous and asynchronous read I/Os. If you suspect that your application has an access path problem, see “Verifying during program preparation” on page 380.
- A lower than expected buffer pool hit ratio. You need to validate that you are the only application that is experiencing a bad ratio for this pool. This can be done by looking at the statistics report or more specific buffer pool reports.
- Pages might be out of order so that sequential detection is not used, or data might have been moved to other pages.
- A RID pool failure. See “RID processing failure (A)” on page 394 for more details.

If I/O time is greater than expected, and not caused by more read I/Os, check for:

- Synchronous write I/Os.
- I/O contention. In general, each synchronous read I/O typically takes from 5 to 20 milliseconds, depending on the disk device. This estimate assumes that there are no prefetch or deferred write I/Os on the same device as the synchronous I/Os.

Drain lock suspension

This is the accumulated wait time the thread was suspended while waiting for a drain lock. You might have a concurrency problem with utilities running in parallel with the application.

Page latch suspension

This field shows the accumulated wait time because of page latch contention. As an example, when the RUNSTATS and COPY utilities are run with the SHRLEVEL(CHANGE) option, they use a page latch to serialize the collection of statistics or the copying of a page. The page latch is a short duration lock. In a data sharing environment, high page latch contention could occur in a multi-threaded application that runs on multiple members and requires many inserts. See “MEMBER CLUSTER” on page 411 for more details.

Global contention (C) and (D)

Gives more details about the kind of objects that are at the origin of lock wait times in data sharing.

Archived

SQL(E)

This block gives a detailed overview of the frequency of each the SQL statements that were executed. This can be used to do some simple quality checks to verify for example the ratio of the number of fetches per open cursor.

$$100 - \left(\frac{100 \times (FETCH - OPEN)}{FETCH} \right)$$

This formula returns the percentage of fetches that did not return data, assuming none of the cursors stopped fetching before reaching SQLCODE 100. A ratio that is close to 100 indicates that most of the fetches were just SQLCODE 100.

The values in this block can also be used to calculate the KPI that were mentioned in the beginning of this chapter. Do not forget that there are events inside DB2 that can influence the results, such as conditional triggers that increase the number of SELECT statements without actually executing a SELECT. Also, notice that multi-row FETCH and INSERT do not reflect the number of rows that were inserted or fetched, but merely the number of times that statement was executed. This could make it more difficult to verify the SQL KPI.

Locking (F)

The locking section gives a breakdown of the locking behavior in the application. The class 3 section already gives an indication of the amount of time the application was waiting for locks and latches. The section also provides details, such as the maximum number of locks that were acquired by the application. Make sure this value stays significantly below the number of locks per user because failing to do so will cause negative SQLCODES. You also need to make sure that the application does not get any lock escalation. It might be better to issue a LOCK TABLE or LOCK PART statement at the beginning of the application, if the application needs to lock a large number of rows or pages.

Data sharing (G)

There are many fields describing the data sharing locks that were taken in this section. You might want to pay special attention to the fields with the suspend reasons because they indicate global contention. For sections (C) and (D), see “Global contention (C) and (D)” on page 392, which describes the type of objects that were involved in the suspension.

Let us have a look at part 2 of the report (Figure 14-6 on page 394).

Archived

```
RID LIST                TOTAL
---------------        --------
USED                        0
FAIL-NO STORAGE             0
FAIL-LIMIT EXC.             0
```

A

```
ROWID                   TOTAL
----------            --------
DIR ACCESS                  0
INDEX USED                  0
TS SCAN                     0
```

B

```
STORED PROC.          TOTAL     UDF              TOTAL     TRIGGERS            TOTAL
------------        --------    ---------       --------    ------------       --------
CALL STMTS                0     EXECUTED            0     STMT TRIGGER            0
ABENDED                   0     ABENDED             0     ROW TRIGGER            70
TIMED OUT                 0     TIMED OUT           0     SQL ERROR               0
REJECTED                  0     REJECTED            0
```

C

```
QUERY PARALLEL.             TOTAL
-------------------        --------
MAXIMUM MEMBERS              N/P
MAXIMUM DEGREE                 0
GROUPS EXECUTED                0
 RAN AS PLANNED                0
 RAN REDUCED                   0
 ONE DB2 COOR=N                0
 ONE DB2 ISOLAT                0
 ONE DB2 DCL TTABLE            0
 SEQ - CURSOR                  0
 SEQ - NO ESA                  0
 SEQ - NO BUF                  0
 SEQ - ENCL.SER                0
MEMB SKIPPED(%)                0
DISABLED BY RLF          NO
REFORM PARAL-CONFIG            0
REFORM PARAL-NO BUF            0
```

D

```
DATA CAPTURE          TOTAL
------------        --------
IFI CALLS              N/P
REC.CAPTURED           N/P
LOG REC.READ           N/P
ROWS RETURN            N/P
RECORDS RET.           N/P
DATA DES.RET           N/P
TABLES RET.            N/P
DESCRIBES              N/P
```

E

```
SERVICE UNITS          CLASS 1          CLASS 2
-------------        --------------   --------------
CPU                          378              136
 AGENT                       378              136
  NONNESTED                  303               61
  STORED PRC                   0                0
  UDF                          0                0
  TRIGGER                     75               75
 PAR.TASKS                     0                0
```

F

```
DYNAMIC SQL STMT            TOTAL
--------------------       --------
REOPTIMIZATION                 0
NOT FOUND IN CACHE             0
FOUND IN CACHE                 1
IMPLICIT PREPARES              0
PREPARES AVOIDED               0
CACHE_LIMIT_EXCEEDED           0
PREP_STMT_PURGED               0
```

G

```
DRAIN/CLAIM           TOTAL
------------        --------
DRAIN REQST                0
DRAIN FAILED               0
CLAIM REQST                5
CLAIM FAILED               0
```

H

```
LOGGING                   TOTAL
-----------------        --------
LOG RECS WRITTEN            158
TOT BYTES WRITTEN         29105
```

I

```
MISCELLANEOUS            TOTAL
---------------        --------
MAX STOR VALUES             0
```

J

```
---- RESOURCE LIMIT FACILITY ----------------------------------------------------------------------------------------------
TYPE: N/P                    TABLE ID: N/P    SERV.UNITS:      N/P    CPU SECONDS: 0.000000    MAX CPU SEC:      N/P
```

K

```
BP0    BPOOL ACTIVITY      TOTAL  BP1    BPOOL ACTIVITY      TOTAL  BP2    BPOOL ACTIVITY      TOTAL  BP7    BPOOL ACTIVITY      TOTAL
---------------------    --------  ---------------------    --------  ---------------------    --------  ---------------------    --------
BPOOL HIT RATIO (%)            50  BPOOL HIT RATIO (%)           100  BPOOL HIT RATIO (%)           100  BPOOL HIT RATIO (%)           100
GETPAGES                       12  GETPAGES                         3  GETPAGES                         3  GETPAGES                        12
GETPAGES-FAILED                 0  GETPAGES-FAILED                 0  GETPAGES-FAILED                 0  GETPAGES-FAILED                 0
BUFFER UPDATES                152  BUFFER UPDATES                  0  BUFFER UPDATES                  0  BUFFER UPDATES                 10
SYNCHRONOUS WRITE               0  SYNCHRONOUS WRITE               0  SYNCHRONOUS WRITE               0  SYNCHRONOUS WRITE               0
SYNCHRONOUS READ                6  SYNCHRONOUS READ                0  SYNCHRONOUS READ                0  SYNCHRONOUS READ                0
SEQ. PREFETCH REQS              0  SEQ. PREFETCH REQS              0  SEQ. PREFETCH REQS              0  SEQ. PREFETCH REQS              1
LIST PREFETCH REQS              0  LIST PREFETCH REQS              0  LIST PREFETCH REQS              0  LIST PREFETCH REQS              0
DYN. PREFETCH REQS              0  DYN. PREFETCH REQS              0  DYN. PREFETCH REQS              0  DYN. PREFETCH REQS              0
PAGES READ ASYNCHR.             0  PAGES READ ASYNCHR.             0  PAGES READ ASYNCHR.             0  PAGES READ ASYNCHR.             0
```

L

Figure 14-6 DB2 accounting report: Part 2

RID processing failure (A)

RID processing is a technique used for access paths such as multiple index access, hybrid join, or as an access path enhancer for basic index access paths. There are run-time limits that control the RID processing. The maximum number of RIDs that can be addressed from a single RID map is approximately 26 million RIDs. When RID list processing is requested, you will always get at least one RID list, which can store around 6,530 RIDs. So if fewer that 6,530 RIDs qualify, list prefetch will not fail.

During execution, DB2 ends list prefetching if more than 25% of the rows in the table (with a minimum of 6,530 number of RIDs in a single RID list) must be accessed. The decision is made based on the catalog statistics information that was available at bind time. This will show up in the report as RDS-limit exceeded.

It is also possible that too many concurrent RID processes took place, which will exhaust the RID pool. This will show up as FAIL-NO STORAGE in the report. See “List prefetch” on page 67 for more details about RID processing. If you want more details about RID processing, it is possible to do list prefetch monitoring.

List prefetch monitoring

IFCID 125 in the performance trace class 8, mapped by macro DSNDQW01, provides the details about whether list prefetch was used successfully or not, for each index processed by list prefetch.

You can format the IFCID 125 with DB2 Performance Expert. The trace contains the statement and the package that executed the statement. Dealing with dynamic SQL coming from a CLI or ODBC might be a challenge because the package would refer the CLI/ODBC driver package. So you may want to include an IFCID 22, 63 trace to capture the access path and the SQL statement. See Figure 14-7 for an example of a pivot table that was built with the output of the trace.

Plan	Program	Stmt	DBIDIx	ODIDIx	Succes full	Failure	Number of Executions	Total RidsRead	Max of RidsinList
ZKEP2209	ZKEP2209	748	311	270	Y	0	1	14	14
				359	Y	0	20	361	31
ZKEP2900	ZKEP2900	943	311	270	Y	0	1	1173	1173
GGMP81	GGMP80	1543	262	1071	Y	0	34	0	0
GGMPV0	GGMPV0	1406	262	436	Y	0	340	412	12
				931	Y	0	340	-340	12
GGRP01	GGMA12	24654	262	1008	Y	0	1	0	0
GGRP1002	GGRP1002	4847	262	794	Y	0	5	0	0
		4938	296	152	Y	0	5	0	0
		4948	296	152	Y	0	5	0	0
		6481	262	794	Y	0	5	0	0
		6497	262	1008	Y	0	5	0	0
		6563	296	152	Y	0	5	0	0
ILKP1010	ILKP1010	799	296	180	Y	0	5	0	0
		818	296	154	Y	0	5	0	0
ILKP15	ILKP15	1110	296	176	Y	0	34	0	0
DFZS03	DFZS03	1301	318	167	Y	0	10	11	2
DFZS1005	DFZS1005	921	318	261	Y	0	4	0	0
DFZS1006	DFZS1006	769	318	261	Y	0	2	64	64
LVZS01	LVZS01	2815	301	77	Y	0	102	0	0
		2934	301	77	Y	0	102	0	0
LVZS03	LVZS21	1399	301	399	Y	0	9	6	1
LVZS1010	GGRP1007	1931	262	462	Y	0	105	1155	21
SARP2900	SARP2900	386	305	67	Y	0	1	1173	1173
SQL35NB1	SWDB2M6M	863	305	69	N	Max Limit	8	-24	-2
					Y	0	2	23918	11959
			313	604	Y	0	4	681	463

Figure 14-7 Pivot table based on IFCID 125 records

Direct row access (B)

If you use that ROWID value in the search condition of a SELECT, UPDATE, or DELETE, DB2 can choose to navigate directly to that row without using an index access or a table space scan to find the row. This access method is called *direct row access*, and is a very efficient access path. However, if you expect direct row access to be used, but for some reason the row has moved (this is usually the case if a REORG has been executed on the particular table space in the meantime), DB2 decides not to use direct row access at execution time. This may lead to serious performance problems because an alternate access path is used. The INDEX USED field indicates that DB2 fell back to using the index defined on the ROWID column. If no such index exists DB2 needs to fall back to a table space scan which is indicated by the TS SCAN field. Both failure fields should always be zero since having them defeats the whole process of using ROWID.

Stored procedures, UDFs, and triggers (C)

The block on server side processes gives an overview of the frequency of execution and the number of failures that occurred. You do not get many details on the actual cost in this block because that is already recorded in the nested activity entry in the class 1 and class 2

reporting block. When package level accounting is active, you should also find corresponding entries for the packages that represent the nested activity.

Query parallelism (D)

When a plan/package was bound with DEGREE(ANY), parallel processing can be selected. This block will show if DB2 ran the SQL in parallel and if the planned degree was actually used. It is possible that the query gets a reduced degree at run time. The different reasons are detailed in the accounting flags. Parallel groups can fall back to sequential operation because of:

- A cursor that can be used for update or delete
- A lack of ESA sort support
- Storage shortage or contention on the buffer pool

If this field is not zero, increase the size of the current buffer pool by using the alter buffer pool command, or use the alter table space statement to assign table spaces accessed by this query to a different buffer pool.

Data capture (E)

The data capture block is not likely to be filled in because it is used for applications that read the log through the IFI interface. This kind of process is used by tools such as Data Propagator.

Service units consumed (F)

The service units block gives the resource consumption in service units. The advantage of Service units is this unit of measure is independent of the machine model and can also be used in the Resource Limit Facility.

Dynamic statement cache (G)

The dynamic statement cache is an important tuning mechanism when using dynamic SQL. This block gives you an idea how successful your SQL was using the cache. You have to be aware of the basic principles of the cache. The global cache will calculate a hashing key based on the SQL statement as it appears in the prepare. This means that a subsequent SQL statement in which we inject a blank in the middle will not hash to the same key and will not be found in the DSC. Your application program must make sure to use the same SQL in order to get a cache hit. One way of increasing the probability of a cache hit is to exploit parameter marker, because that takes the value of the host variables out of the hash key. The problem with the DSC is that it is not possible to exclude SQL from going into the cache. So an application performing a large quantity of inserts without using parameter markers will have a highly negative impact on the cache efficiency.

> **Tip:** Make sure you use parameter markers on your SQL to increase the efficiency of the DSC.

DRAIN/CLAIM (H)

This block gives you an idea of the number of CLAIM/DRAIN requests and the amount of failures that occurred on these requests.

Logging (I)

The DB2 log is a shared resource that has a limited bandwidth. This bandwidth is very high for most customers but you might want to keep an eye on the amount of logging an application is generating. The second metric is the amount of log records that are produced per commit. We have already explained the DSNZPARM that creates the message if an

application created too many log records without committing. This field allows you to track this metric at an application level.

Memory consumption (J)

LOB processing can be very memory-intensive if you do not manage the locator variables efficiently. The *max stor* field gives you an indication of your storage consumption.

RLF (K)

The resource limit facility is described in “Using RLF” on page 381. This block gives you an overview of the resources consumed and which limits are applied to the execution. You can use this block to calibrate your RLF control table.

Buffer pool (L)

The block on buffer pool is an important part of the report since it allows you to better understand some of the class 3 components and gives you an indication of both the type of GETPAGEs that were executed and the I/O activity that was caused by this workload. It might be simple or complicated to relate the numbers you find in this block to individual statements, depending on how many tables and indexes are used by your application, and are sharing the same pool.

A bad hit ratio can be caused by the application not processing in an efficient sequence, but it could also be caused by an inefficient pool. You need to consult with the system DBA and probably look at the statistics and even performance traces.

Sometimes, the buffer pool hit ratio is not a problem, but the number of GETPAGEs per SQL might be a problem. This should show up in the CPU consumption per SQL ratio.

Tip: Page updates on a R/O application are an indication of work file creation.

14.4.4 Statistics trace

The statistics trace provides the transversal view of the DB2 subsystem without looking at individual applications. The advantage of the statistics trace is that this is really interval driven and does not suffer from the same problem accounting does. You can easily correlate all the different blocks in the statistics trace. We are not going to cover all the fields in the statistics trace, because it is more a system programmer’s tuning task to look into this kind of report.

Just as with accounting, the statistics trace has different classes. The difference with statistics is that the classes provide different kinds of trace records, rather than more detailed information on the same trace record.

The statistics trace provides the following classes:

- Class 1 – Basic statistics data *
- Class 3 – Deadlock, group buffer pool, data set extension information, indications of long-running URs, and active log space shortages
- Class 4 – DB2 exceptional conditions
- Class 5 – Data sharing statistics
- Class 6 – Storage usage
- Class 8 – Dataset I/O statistics

Tip one: Always activate Classes 1, 3, 4, 6, and 8. Class 5 should be activated when running data sharing.

Tip two: APAR PQ99658 adds IFCID 225 (previously started only by class 6) to class 1, so activating class 6 is no longer necessary.

Class 3 deadlock information

Activating Statistics class 3 provides a key element in determining the full scope of deadlock and timeout situations. You will find deadlock and timeout information in the master console, but that will only cover two units of recovery involved in the process. It is possible that the locking problem involves more than two units of recovery and the easiest way to find that out is by activating the class 3 statistics trace.

14.4.5 SMF 30 and job output

You can get a lot of information by simply looking at the jobstep information in your job output of your batch applications. The SMF type 30 report gives you a very good idea of the resource consumption in terms of CPU and EXCPs per step. Note that SMF type 30 records are interval records and are also cut at step termination. It might, therefore, be necessary to combine multiple records to cover the entire duration of a thread.

Figure 14-8 shows an extract of the columns you can get from an SMF 30 report.

JOB NAME	PST16A01	PST16A01	PST16A01	PST16A01
Program Name	JD229	JD229	JD229	JD229
Avg Working Set Size	1.594	1.593	1.609	2.072
Step TCB+SRB CPU Time	1	0	0	0
Step DASD I/O Count	271	255	48	42
Step DASD I/O Wait Time	0	0	0	0
Step Enclave CPU Time	-	-	-	-
Step Execution Time	4.965	6.253	4.137	1.186
IO Service Units	90	190	30	175
Step Residency Time	153	121	30	10
Step SRB CPU Time	0	0	-	-
Step TCB CPU Time	1	0	0	0
Pseudo Elapsed (occupancy) Time	153	121	30	10

Figure 14-8 Spreadsheet format of SMF 30 record

You can see the evolution of the working set size of the job. This gives you a good indication of the memory footprint the application has in its own address space. However, it does not take into consideration the memory that is used in the DB2 address space. The memory footprint of the application is influenced by the size of the load module and the additional memory that is being GETMAINed. One of the reasons for an increased memory footprint can be the use of multi-row processing with a large rowset size. Do not abuse the rowset size, because the increased memory footprint could negatively impact performance.

14.4.6 Real-time statistics

Real-time statistics (RTS) is a very good way to track the activity on an object, because it provides the object dimension that is missing from accounting and statistics traces.

Tip: The overhead of RTS is negligible compared to the added value of the information that is obtained. You should therefore activate RTS.

The counters in the RTS tables are always overwritten and DB2 does not provide a mechanism to keep track of history. Also, note that you can define *triggers* on the RTS tables, but they will not be fired. You might therefore consider a process that saves the information of the RTS tables in a history table in order to do trend analysis.

You can use RTS to better understand your SQL workload but you can also use the information to determine if you need to run utilities to keep your objects in good shape. You could also come to the conclusion that the DDL options such as PCTFREE and FREEPAGE are set incorrectly for certain tables or indexes.

Example 14-3 provides a sample query to determine if a table space needs to be REORGed.

Example 14-3 Sample query to determine if a table space needs to be REORGed

```
SELECT DBNAME, NAME, PARTITION
FROM SYSIBM.TABLESPACESTATS
WHERE REORGLASTTIME IS NULL
OR REORGUNCLUSTINS/TOTALROWS > 0.1
OR REORGDISORGLOB/TOTALROWS > 0.1
OR (REORGNEARINDREF+REORGFARINDREF)/TOTALROWS > 0.1
OR REORGMASSDELETE > 0
OR REORGDELETE/REORGINSERT > 1.5
OR EXTENTS > 50
```

Example 14-4 provides a sample query to determine if an index needs to be REORGed.

Example 14-4 Sample query to determine if an index needs to be REORGed or rebuilt

```
SELECT DBNAME, NAME, PARTITION
FROM SYSIBM.INDEXSPACESTATS
WHERE REORGLASTTIME IS NULL
OR REORGPSEUDODELETES/TOTALENTRIES > 0.1
OR REORGLEAFFAR/NPAGES > 0.1
OR REORGMASSDELETE > ?
OR REORGNUMLEVELS > ?
OR REORGDELETE/REORGINSERT > ?
OR EXTENTS > ?
```

Free space

Specifying appropriate free space is not only important for the data, but also for the index. The same parameters PCTFREE and FREEPAGE exist for indexes to specify the initial percentage of free space to keep per page, and after how many pages an empty page is to be provided.

As for data free space allocation, index free space allocation is determined by the insert patterns used by the processes accessing the table.

If the index is the clustering index and the insert pattern is always inserting rows at the end, your index only grows at the end and no free space needs to be provided inside the page, or after *n* number of pages.

However, if the insert/update pattern is not according to the clustering index, or the index is not the clustering index, you need to provide adequate free space. If not, the newly allocated pages, after a full index page is split, will be at the physical end of the index file, and DB2 may have to perform extra synchronous I/Os to retrieve those pages when doing a nonmatching index scan, or index screening.

Archived

You can only determine indirectly if your free space is still adequate. The indicator that allows you to determine how well your index is physically organized is the LEAFFAR column in the catalog table SYSIBM.SYSINDEXPART. LEAFFAR counts the number of index splits which resulted in the new index leaf "far from the original leaf page". The information is gathered by RUNSTATS.

We recommend that you consider executing a REORG INDEX if LEAFFAR/NLEAF > 10%. The comparison should be made at table space level by summing up partition level statistics where necessary. There is a trade-off between free space and reorganization frequency. For example, if the insert pattern matches the clustering index (continuous ascending inserts), you probably will not have to reorganize the table. In this case, you should have a PCTFREE = 0.

When old rows are removed from the beginning of the table, REORG will be required to reclaim the freed space in both the table and the index. If the insert pattern does not match the clustering index, you must allocate free space in all pages in the table. In this case, you must select the PCTFREE value according to your table growth rate predictions and REORG frequency.

Archived

Part 5

Data sharing

In this part, we gather information that is useful when developing applications in a data sharing environment.

This part contains one chapter:

- Chapter 15, “Data sharing specifics” on page 403

© Copyright IBM Corp. 2005. All rights reserved.

Archived

15

Data sharing specifics

In this chapter, we cover several of the application and database design issues that influence the efficiency and overhead in a data sharing environment.

This chapter contains the following topics:

- Data sharing goals
- Data sharing overhead
- Reducing data sharing overhead
- IMMEDWRITE
- DML considerations

For more information about this topic, see:

- *DB2 for z/OS Version 8 Data Sharing: Planning and Administration*, SC18-7417-02
- *DB2 for MVS/ESA Version 4 Data Sharing Performance Topics,* SG24-4611
- *DB2 for MVS/ESA Version 4 Data Sharing Implementation,* SG24-4791
- *DB2 for OS/390 Version 5 Performance Topics,* SG24-6465

© Copyright IBM Corp. 2005. All rights reserved.

15.1 Data sharing goals

Before we go into the details of data sharing, we need to determine the reasons for using data sharing. When done correctly, DB2 data sharing provides improved availability, and scalability of your system and application. Depending on your goals, you might choose different design and coding options. Although there is no need to change your applications to use data sharing, you may have to do some tuning for optimal performance and for minimizing overhead. In general, it is safe to assume that an application, behaving correctly in a non-data sharing environment and having implemented all the typical locking optimization hints and tips, will work fine in data sharing without any major modification. However, some applications may have significant overhead once they are moved into a data sharing configuration.

15.1.1 Provide improved availability

More and more DB2 users are demanding access to DB2 data every hour of the day, every day of the year. By improving availability for both planned and unplanned outages, DB2 data sharing is a key element in meeting this service objective.

Unplanned downtime

You can achieve a higher level of continuous availability, provided your design does not rely too heavily on the degree of system *affinity*. Affinity is when you code your application in such a way that it has to run on a specific member. That member becomes your single point of failure. Therefore, to maintain a high level of availability, you should limit the degree of affinity in your design. However, you can find a compromise if you can clearly define your service objectives. You could decide not to provide continuous availability for all application components. This would mean that parts of your application are at risk of being unavailable, which might not be an issue if your SLA allows you to buffer the input and if you have the spare capacity to process the backlog once the system is available.

Since the weakest link overall determines the strength of the chain, a global approach to system availability should be taken. You have to deal with the setup of your job scheduling, TP monitors, network definition, disk resilience, and many more: all elements of your computing environment have to participate in the high availability exercise. You must also consider that data sharing adds an additional critical component, the coupling facility (CF), and its structures. The CF needs to be considered in a disaster recovery scenario. The design and the implementation of the CF and CF structures have a significant impact on the data sharing overhead. See “Reducing data sharing overhead” on page 409 for more details.

Planned downtime

Utilizing multiple LPARs and multiple members of the data sharing group can help reduce significantly the amount of required downtime. The Parallel Sysplex and data sharing introduce the ability to do rolling maintenance and perform system maintenance without bringing the entire complex down. The net result might be a reduced capacity for a short period of time, but the users will still have the impression their service is available. The redbook, *Achieving the Highest Levels of Parallel Sysplex Availability,* SG24-6061, covers the subject of reducing downtime in great detail.

15.1.2 Provide scalability

IBM announces, on a regular basis, new machines that have more MIPS (which translates into more processing capacity), more memory, and more I/O bandwidth. As a net result, most heavy customer workloads can now be run on a single machine. So, are there any reasons from a scalability point of view to move to data sharing?

Move system boundaries

The impact of putting multiple processors under control of a single LPAR is one of the considerations of utilizing an increased number of processors. Some installations prefer to use two LPARs with 6 processors rather than one huge LPAR with 12 processors. Each z990 box can accommodate up to 32 central processors. An IBM @server® zSeries z990 model 2084-324, which has 24 processors on board, provides the equivalent of 15.39 single engine machines in a mixed workload environment.

The more recent IBM System z9™ 109 S54 can have 54 Processor Units. Although adding further capacity can degrade performance as the processors compete for shared resources (the law of diminishing returns), there are many customers running with 16 processors and several other customers are moving beyond 16. The *Large Systems Performance Reference*, SC28-1187, is the source for IBM's assessment of relative processor capacity and can help in identifying the point beyond which horizontal scaling can come into play with the Parallel Sysplex.

Move DB2 boundaries

The DB2 active log is a shared resource for all applications connected to the same DB2 subsystem. Web applications can create tremendous sudden workloads with log activity spikes. With LOB (LOG YES) and the introduction of XML data types, we may also see an increased demand for logging throughput. The DB2 active log is a single VSAM data set, which has two copies when using dual logging, that has a limited bandwidth. The log write bandwidth is related to the number of I/Os that are performed per second. This is related to the number of log force write events, which is related to the number of CIs that have to be written per second. High total logwrite volumes have required the use of VSAM striping techniques and the use of FICON channels. But there still is a practical limit of about 40 MB per second. If your application volume needs more than this bandwidth, data sharing can increase this value by adding additional DB2 members, each one of them with the 40 Mbyte bandwidth.

15.2 Data sharing overhead

Ideally, a DB2 subsystem using many *identical* processors would generate a throughput that can be defined as:

```
Throughput = (number of DB2 members) x (transaction rate of each member)
```

The definition assumes that the response time of each transaction remains unchanged. However, combining multiple processors creates an overhead that results from the additional processing cost due to the introduction of inter-DB2 data coherency control and global locking. The design and implementation of DB2 data sharing can help mitigate the overhead.

The overhead depends on the percentage of total application processing capacity against DB2 data sharing in read-write (R/W) or write-write (W/W) mode, the intensity of CF access (accesses per second of CPU time), technical configuration, workload distribution, and volume of lock contention. Data sharing overhead should be very small for read-only (R/O) OLTP applications with thread reuse and effective lock avoidance. Data sharing overhead may be relatively high for an update-intensive batch application. Typical overhead is 5% to 15%, with a lot of variation at the individual application level.

15.2.1 Infrastructure impact

Data sharing overhead is not only dependent on DB2 options and application behavior, but also on infrastructure entities, such as intersystem distance and internal CF considerations. We examine them in more detail.

Intersystem distance

As a rule of thumb, you should add 10 µsec per km of distance between the LPAR and the CF to the response times of the CF requests.

Tests with Coarse Wave Division Multiplexing (CWDM) at 18 km distance using four z900 2064-213 connected to one external CF machine (z900) and ICF CF LPAR gave the following results for the most accessed CF structure (the IMS IRLM structure). Before CWDM, all the requests were synchronous requests (40 µsec). Once CWDM was implemented, those requests became asynchronous (+150 µsec to 170 µsec). A small supplementary cost in the XCF address space was also noted for the conversion of synchronous to asynchronous requests (2% CP on a 2064-213).

A second set of tests, conducted using two z990 machines (2084-314) and two external CF machines (2085-304) at a distance of 1.5 km, showed the following results:

- All requests to the CFs were already asynchronous before CWDM implementation.
- After CWDM implementation, at a distance of 11 km, an increase of 100 µsecs was noted in CF response times.

These test results show that distance has an impact on the elapsed time of the request. The operating system will make sure that the CPU cost does not increase in a linear fashion, by switching to an asynchronous mode. This switch is not free, but it would be worse for the requesting CPU to keep spinning, waiting on a synchronous request.

Important:
Make sure you have the following APARs in your maintenance level:

- z/OS: OA06729
- DB2 PE: PK03878 and PK01759

They avoid the reporting of synchronous to asynchronous switch as a false contention.

Internal CF considerations

It is possible to assign one or multiple processors in a zSeries box as CF processors. This CF implementation is called an Internal Coupling Facility (ICF). When implementing a CF as an ICF, you should be aware that the CF runs Coupling Facility Control Code (CFCC) which is a real-time OS. A real-time OS always runs at a 100% CPU busy. For financial reasons, customers sometimes want to take shortcuts when defining multiple sysplexes on a limited number of boxes. A common action is to try to avoid buying extra processors for the CF. An example of multiple v(virtual)CFs assigned to a single ICF processor is shown in Figure 15-1 on page 407. If a customer would try to implement this, what would be the result?
There are two options for trying to make this configuration work:

- **Basic time slicing**

  Each vCF gets a fixed number of milliseconds after which control is passed on to the next vCF. It takes milliseconds before the next vCF gets dispatched. Therefore unacceptable service times are experienced on the CF request and almost all of the synchronous requests are switched to asynchronous requests.

- **Enhanced time slicing**

  Put the non-critical vCF in DYNDISP YES mode. This means that the vCF frees up the processor if no request is received, thereby passing the processor to the next vCF and not using up the allotted time slice. This increases the slice for the vCF with DYNDISP NO. If you have vCF1 and vCF2 with DYNDISP YES and vCF3 with DYNDISP NO and no significant traffic occurs for vCF1 and vCF2, vCF3 would get close to 99% of the ICF. Changing vCF2 to DYNDISP NO would give vCF2 and vCF3 close to 50%, therefore, defeating the whole purpose of the exercise.

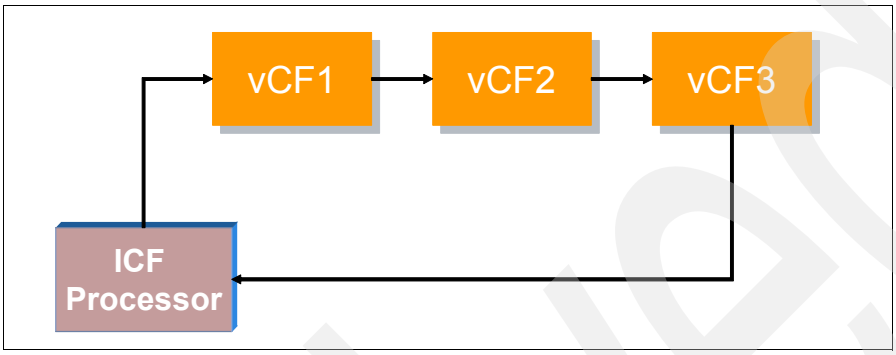


Figure 15-1 Putting multiple CF instances on a single ICF

The obvious conclusion is that you cannot have more than one CF defined on a single production sysplex processor.

15.2.2 Inter-DB2 data coherency control cost

The level of inter-DB2 R/W interest is an indication of the degree and type of data sharing used in your application. The degree of data sharing can be defined on the basis of the ratio of SQL processing against shared data to other processing. This degree is secondary to the type of processing and the objects on which the processing occurs. The data sharing overhead is very small if your processing is either read-only or the objects referenced have a strong member affinity.

The inter-DB2 R/W interest is tracked through a mechanism called page set partition physical locks (P-locks). Each member acquires one page set P-lock for each open table space or index, or for any partition of a partitioned table space or partitioned index. These P-locks are owned by a data sharing member and are, therefore, independent of the transactions. The mode of a page set P-lock for a given page set or partition in a given member is determined by the activities on the page set or partition as shown in Table 15-1.

Table 15-1 P-lock modes

Activity	Interest
Physical open	R/O
Physical close	None
First update or pseudo-open	R/W
R/O switching or pseudo-close	R/O

Switching of a page set or partition between R/O and R/W affects the mode of the page set P-locks. That mode is the indication to the DB2 members of the level of inter-DB2 R/W interest, also called GBP-dependency. This switching is caused by R/O switching and by updates to the page set or partition. Installation parameter options PCLOSET and PCLOSEN

allow the specification of time interval and number of checkpoints to initiate the check for R/O switching. Low values for the PCLOSET and PCLOSEN parameters may cause frequent switching in and out of GBP-dependency. However, setting these parameters too high could mean GBP-dependent processing for a longer period than is really needed. Every situation must be analyzed individually, based on workload and access profiles, to establish a good balance between R/O switching and the data sharing overhead.

Page set P-locks are represented in the CF lock structure for each page set or partition that is open on each member. However, this volume of page set P-lock activity is not a primary performance concern.

You have much more leverage optimizing the mode of the page set P-lock acquired by the different members in the data sharing group. The compatibility of the lock modes used by the different members determines the level of data coherency control, possibly causing a page set to become GBP-dependent. When a page set becomes GBP-dependent, all the read pages belonging to that page set that are cached in the reading DB2 subsystems' local buffer pools must be registered in the CF, and all updated pages by the updating DB2 subsystems must be written to the CF using a process called FORCE AT COMMIT. Note that read pages cached in updating DB2s in a W/W (Write/Write) scenario must also be registered in the CF. These actions are necessary to support the additional processing of cross-invalidation, and page validity checking of locally cached pages, when a page set is GBP-dependent.

You can affect the degree of inter-DB2 R/W interest by managing the affinity of processes to certain page sets. However, be aware that creating member affinity may jeopardize your goal of increased availability, because you are likely to introduce a single point of failure. See “Create member affinity” on page 412, for more information about creating member affinity.

15.2.3 Global locking cost

For data consistency in a data sharing environment, locks must be known and respected among all members. DB2 data sharing uses global locks to ensure that each member is aware of locks on other members.

A *local lock* guarantees serialization for one member only. A *global lock* is one that guarantees serialization across the members in a data sharing group. The IRLM keeps track of lock modes associated with multiple concurrent transactions running against a particular member and propagates only those locks that are necessary for global concurrency to the CF lock structure.

P-locks are global locks and have two purposes:

- Ensure inter-DB2 data coherency by tracking the inter-DB2 R/W interest, using page set P-locks (see “Inter-DB2 data coherency control cost” on page 407).
- Ensure the data coherency of a page, using page P-locks. Page P-locks are used to provide write serialization between members. They can be thought of as inter-member page latches.

Transaction locks are logical locks (L-locks). DB2 uses L-locks for inter-transaction concurrency control within a data sharing group. L-locks are owned by a transaction, but DB2 will only move transaction lock information necessary for global serialization into the CF to minimize the cost.

The L-lock at the table space level is used to verify the access compatibility between different processes within a DB2 subsystem. In a data sharing environment, all members need to know these locks, and therefore, they are propagated to the CF. On the basis of these parent L-locks, DB2 determines the concurrency of processes. With explicit hierarchical level

locking, DB2 propagates only the most restrictive locks, the parent L-locks, to the CF initially. DB2 propagates the child L-locks only when there is inter-DB2 R/W interest at the parent level (table space or partition). Therefore, you can optimize performance by reducing the number of L-locks that are propagated. The efficiency of the explicit hierarchical locking is determined by the lock mode on the parent acquired by the different members of the data sharing group. If two members have an IX lock on a parent, all modes of child locks acquired by any of those members must be propagated to the CF. In an attempt to further optimize the amount of information that is passed to the CF, DB2 propagates only the most restrictive states per resource to the CF. Five transactions on one member that have an S lock on a table space page propagate only one lock to the CF.

z/OS Cross System Extended Services® (z/OS XES) does not track the same granularity of lock modes as the IRLM. The z/OS XES tracks only two lock modes, share and exclusive. IRLM locking supports many additional lock modes. When the z/OS XES component detects a contention because of incompatible lock modes for the same resource, that contention is not necessarily a real contention by IRLM standards. For example, the IRLM finds the IX-mode compatible with the IS-mode, but since the z/OS XES component only knows about S and X, it finds these locks incompatible. The additional processing required to determine if the contention is real is reduced in DB2 Version 8, which registers IX-mode locks as an S lock in XES. This new locking mechanism is known as protocol level 2.

15.3 Reducing data sharing overhead

DB2 is designed to reduce the cost of data sharing enablement as much as possible. Various optimization techniques and algorithms have been implemented to enable you to take full advantage of your Parallel Sysplex environment. However, the data sharing enablement overhead is sensitive to the system and application environment used. Several design and configuration options enable you to further reduce the cost.

15.3.1 Reduce locking cost

The path length of acquiring a lock from the IRLM compared to acquiring a latch is considerably higher. The elapsed time of an interaction with the CF is even longer than a request to the IRLM even though most of the processing is performed synchronously, assuming the CF structure response time is adequate. Propagating a lock to the CF is more expensive than latching a page. Every lock you can avoid improves the overall resource consumption in a data sharing environment. The impact of lock avoidance is therefore more important than in a non-data-sharing environment.

Every time you acquire a lock, you could have contention on the object. This contention can be due to the granularity of the lock modes of z/OS XES. Resolving this contention can be a lengthy process that might have to be repeated if the lock released is acquired again. A way of reducing the overhead is by hanging on to that lock for a longer period of time, but that could be contradicting your concurrency requirement.

Maximize lock avoidance

The number of child locks that need to be propagated to the CF affects the cost of global locking. If the L-lock on the parent is IX, even S locks on children must be propagated to the CF if there is inter-DB2 R/W interest. Processing a large number of qualifying rows could cause a considerable overhead. Lock avoidance reduces the impact of child lock propagation by reducing the number of S locks. Lock avoidance, if successful, eliminates calls to the IRLM to request a page lock or row lock.

The efficiency of lock avoidance depends on three parameters:

- The CURRENTDATA value
- The ISOLATION level of your plan and package
- The highest committed global log record sequence number (GCLSN)

CURRENTDATA and ISOLATION

RR isolation does not allow lock avoidance, because it always makes sure that the data remains unchanged, where lock avoidance allows the flexibility of change even if you are positioned on the row. RS isolation allows the use of lock avoidance for nonqualifying rows. The definition of a nonqualifying row in the case of RS isolation is any row that does not qualify in stage 1. RS isolation locks all rows for which stage 2 predicates have to be evaluated. These rows are locked even if they are rejected after the stage 2 predicate evaluation. The number of rows that are locked is, therefore, not necessarily equal to the number of rows fetched by the application.

The CURRENTDATA bind option enables you to determine which rows can be subject to lock avoidance when using CS isolation. Specifying a value of YES indicates to DB2 that you want to guarantee the contents of the row on which you are positioned. DB2 must make sure that the column values that have been retrieved are not changed while the application remains positioned on the row. That guarantee is implemented through the use of an S lock on the row or page. Using CURRENTDATA(YES) effectively disables lock avoidance for qualifying rows. Protecting a qualifying row with an S lock may not be required, provided your application has no intention to update or delete the row on which you are positioned. CURRENTDATA(NO) can expose existing SQL coding. Two examples are:

- Application RI check against a parent table row if the open SQL cursor on the parent does not have FOR UPDATE OF... defined
- Noncursor update of row retrieved using OPEN cursor with ORDER BY

You must try to use CURRENTDATA(NO) and ISOLATION(CS) as much as possible to maximize the effect of lock avoidance.

Global commit log sequence number

For objects without inter-DB2 R/W interest, lock avoidance depends on the highest *committed log record sequence number* (CLSN) within the DB2 member. For objects with inter-DB2 R/W interest, lock avoidance depends on the *global highest committed log record sequence number* (GCLSN), because it would be inefficient and expensive to propagate the CLSN of each page set into the CF. Instead the GCLSN is used, which is the oldest CLSN of all page sets with inter-DB2 R/W interest per member. Each member puts its value in the CF, and the oldest value is used as the GCLSN in the entire data sharing group.

A GCLSN that reflects a recent point in time increases the efficiency of lock avoidance, because the number of pages that are changed before the GCLSN is more important. The GCLSN is based on the GBP-dependent page sets. It is, therefore, important that your applications issue commits at frequent intervals to ensure that the GCLSN stays current. Failing to do so deteriorates the lock avoidance efficiency of the GBP-dependent page sets.

Maximize thread reuse

With DB2 V8, the need for binding plans with RELEASE(DEALLOCATE) to avoid XES contention has been significantly reduced. In DB2 Version 8, IX level table space intent locks, and page set P-locks, will be registered in the XES as being an S lock. The net effect is a significant reduction of the number of invocations of the contention exit because of the avoidance of XES contention. You might decide to continue to use RELEASE(DEALLOCATE), but be aware of the potential side effects.

There may be a trade-off in concurrency and EDM pool requirements. Using RELEASE(DEALLOCATE) can increase EDM pool storage requirements as you hold on to your plans and packages for a longer period of time. This is especially the case if one plan is used for a large number of packages. For example, using just one plan for all transactions in one CICS results in a large number of packages using the same thread. Unplanned lock escalation has a dramatic impact, because with RELEASE(DEALLOCATE) the escalation will stay until thread deallocation, which might take a long time and could prohibit application concurrency. Problems can arise if you have processes that require exclusive access, such as vendor utilities or batch processes. They can break into long-running SQL threads through drain or by issuing a -STOP DB... AT COMMIT.

Use uncommitted read isolation

UR isolation has the lowest locking cost, since it does not acquire parent or child L-locks. It only acquires a mass delete lock. If your application can manage the potential inconsistency or if your data is guaranteed to be consistent, you should exploit UR isolation to reduce the cost of locking.UR isolation is available at the package/plan level and also at the statement level.

Use row locking with caution

Row locking can have a considerable impact in a data sharing environment. The cost of row locking can be significantly higher than page locking if DB2 cannot use lock avoidance efficiently and you process a number of rows per page.

Independent of the cost, the degree of concurrency might not immediately benefit from row locking in a data sharing environment. The physical consistency on a page is guaranteed through the use of a page P-lock, the data sharing latch. Even if you use row locking, you cannot simultaneously update two distinct rows in the same page, since there is no such thing as a row P-lock. It might be better for the overall throughput if you use the MAXROWS 1 option in combination with page locking, which is the equivalent of a row lock. MAXROWS N does not have to be MAXROWS 1. Simply reducing the number of rows per page can be sufficient to avoid the peak of the contention curve.

Consider the MAXROWS option if the number of rows in the table is limited, and to help the situation if you experience a high degree of contention on a page without it.

15.3.2 Avoid overhead on system pages in your user objects

We have already seen that, in the case of row level locking, page P-locks are needed in addition to L locks to provide page consistency.

There are additional examples of sub-page concurrency overhead that data sharing must do, such as handling space map page updates, and index page updates. There are DDL options that reduce this overhead.

MEMBER CLUSTER

If your application does heavy sequential insert processing from multiple members of the group, consider putting the data in a table space that is defined with the MEMBER CLUSTER option. The MEMBER CLUSTER option causes DB2 to insert the data based on available space rather then respecting the clustering index. For sequential insert applications, specifying MEMBER CLUSTER can reduce P-lock contention and give better insert performance at the cost of possibly longer query times.

TRACKMOD NO

This option specifies whether DB2 tracks modified pages in the space map pages of the table space or partition. When an application is rapidly updating a single table space, contention can exist on the space map pages because DB2 tracks which pages have changed. DB2 tracks these changes to help improve the speed of incremental image copies. With TRACKMOD NO, DB2 does not track change, and therefore avoids contention on the space map. *We do not recommend this option if you depend on fast incremental image copies for your backup strategy.* If your application does heavy sequential inserts, consider also using the MEMBER CLUSTER option.

15.3.3 Create member affinity

Creating member affinity is a complex process that requires intervention on the job scheduling, DDL, and application design. Creating member affinity is beneficial only if it enables you to reduce the number of GBP-dependent page sets. The technique is not relevant in a read-only environment because you need at least one updating member to become GBP-dependent. If you fail to control all of the parameters, you might end up with reduced availability without the performance benefits.

Job scheduling

You have two basic options for creating member affinity: Isolating a set of tables by routing all work for a set of specific tables to one member or isolating the work to a partition.

Isolating a set of tables

Isolating the work to a specific table is an option, if you can isolate sets of transactions and batch programs that use a subset of your tables. If your accounting application has its own set of tables, that work can be routed to a single member. It is possible to restrict access to data to a single member. For example, if you want access to a table space named NOSHARE limited only to DB2A, you could assign NOSHARE to a previously unused buffer pool, such as BP25, using the ALTER TABLESPACE command. Do not define a GBP that corresponds to BP25, and assign BP25 a size of zero on any other DB2 in the group. This prevents the other members of the group from attempting to use this buffer pool, and therefore from accessing table space NOSHARE. This technique is applicable only if the number and size of tables that you want to group is sufficiently large. If not, you probably do not want to dedicate a buffer pool because that would increase your demand for virtual storage without any real benefit. The decrease in virtual storage efficiency is probably more important than the data sharing overhead of the occasional access to the data from another member. Another consideration is the impact on your error handling routines and availability. A job could be executed on the wrong member, either planned or as the result of an operational error. Your contingency scenario should be prepared to intervene on your buffer pool configuration in case you want to reroute the work of other members.

Isolating the work to a partition (partition level)

If you cannot restrict the use of a table to a member, it might still be possible to isolate the activity to a partition. For batch jobs, it can be quite easy to determine the key range involved in the execution. If a batch program is split over multiple members for performance reasons, you need to partition in such a way that your partition ranges match the boundaries of your jobs. The requirement to use different page sets for each of the members cannot be applied to any nonpartitioning index, because there is only one page set for the entire table space.

Determining the range in a transaction environment can be much more complicated or even impossible, because the range might depend on the processing logic. You could be halfway through the transaction before you know which partition is used.

Having one transaction out of 1,000 crossing the line and updating a row in a partition assigned to another member might in fact be worse than a full data sharing solution with random scheduling of the transactions. Maintaining the GBP-dependency has a cost, but going in and out of this state is probably worse. If an update is performed to the partition of the other member, you have a R/W interest by both members on the same page set. Since you now have inter-DB2 R/W interest and the page set has become GBP-dependent, DB2 needs to register in the CF all pages of the page set that reside in the local buffer pool. Provided that the update was a one off, remember that only one transaction in 10,000 crosses the line, DB2 will declare the page set non-GBP-dependent some time after the members that are not supposed to update the partition perform the R/O switch, assuming the page set is defined as CLOSE YES. DB2 physically closes the page set after it has been R/O for a certain amount of time if there is one R/W member and the R/O members do not use the page set. The worst scenario is the second transaction again crossing the line after you have made the R/O switch, because the entire process is repeated.

15.4 IMMEDWRITE

Consider the situation where one transaction updates DB2 data using INSERT, UPDATE, or DELETE, and then, before completing (phase 2 of) commit, spawns a second transaction that is dependent on the updates that were made by the first transaction. This type of relationship is referred to as “ordered dependency” between transactions.

Now consider the following scenario.

We have a two way data sharing group DB0G with members DB1G and DB2G. Transaction T1, running on member DB1G, makes an update to a page. Transaction T2, spawned by T1 and dependent on the updates made by T1, runs on member DB2G. If transaction T2 is not bound with isolation repeatable read (RR), and the updated page (on DB1G) has been used previously by DB2G and is still in its local buffer pool, there is a chance, due to lock avoidance, that T2 uses an old copy of the same page in the virtual buffer pool of DB2G if T1 still has not committed the update.

Here are some possible “work arounds” for this problem:

- Execute the two transactions on the same member
- Bind transaction T2 with ISOLATION(RR)
- T2 does not use ambiguous cursors
- Make T1 commit before spawning T2
- Use IMMEDWRITE

IMMEDWRITE allows the user to specify when DB2 should write updated GBP dependent buffers to the Coupling Facility. You can either wait until commit or rollback or you can have DB2 write the changed page as soon as it was updated (without waiting on the commit). The IMMEDWRITE parameter can either be specified at BIND time or using the IMMEDWRI DSNZPARM parameter.

The default processing writes changed pages during phase 1 of commit processing. If you specify “NO”' or “PH1”, the behavior will be identical, changed pages are written during phase 1 of commit processing. The “PH1” option exists for compatibility reasons, but its usage should be discouraged. The DSNZPARM IMMEDWRI parameter will no longer accept a value of “PH1”. With this change, pages are either written at (the latest at) phase 1, never at phase 2 of the commit processing.

Be careful with IMMEDWRITE YES because changed pages are written to the group buffer pool as soon as the buffer updates are complete (so definitely before committing). Specifying this option may impact performance.

See Figure 15-2 for an illustration of what might happen when a batch program does a massive update of a table with the IMMEDWRI parameter set to YES. The program caused the spike of 3 million page writes per interval. Changing the DSNZPARM to NO caused a dramatic drop in the number of requests to the CF and has a positive impact on the elapsed time of the application.

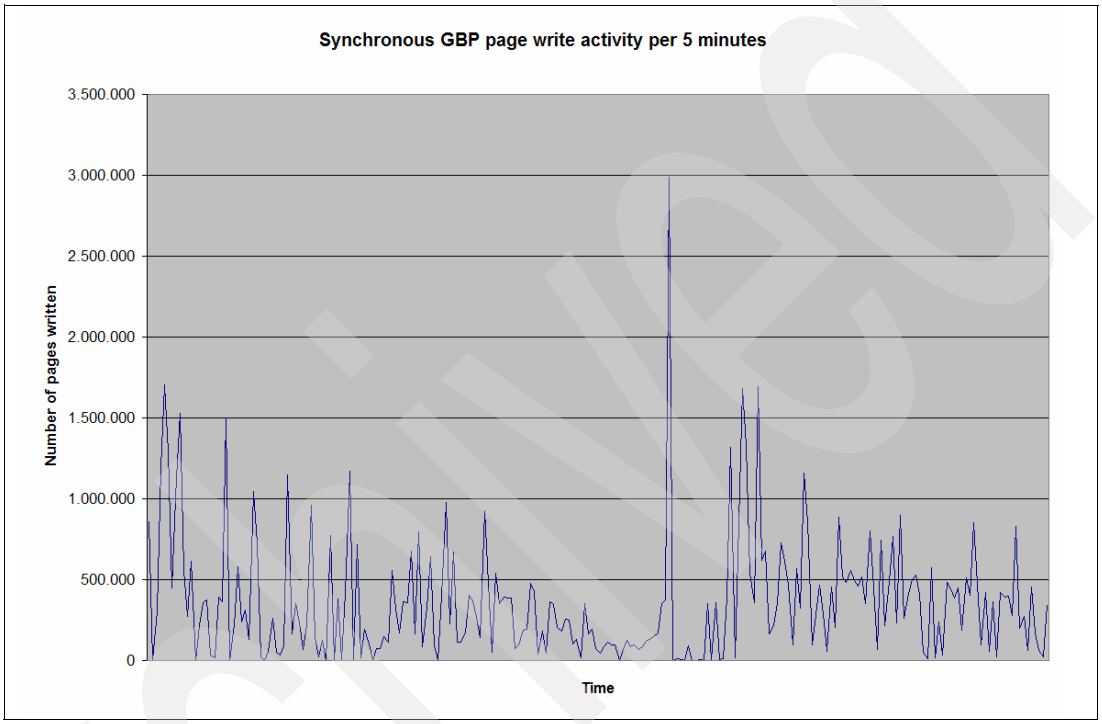


Figure 15-2 Synchronous GBP page write activity per five minutes

So IMMWRITE NO is solving the problem when the spawned transaction starts at COMMIT time. You are still exposed to the data inconsistency when the spawned transaction starts before the commit. We recommend that you use the BIND option of the spawning transaction and not use the DSNZPARM option because that has an impact on all plans and not just the one that is causing the problem.

Tip: Avoid using the IMMEDWRI DSNZPARM and use the IMMWRITE BIND option only when required.

15.5 DML considerations

Some SQL features have a special behavior when running in data sharing mode. You should be aware of these considerations when designing data sharing enabled applications.

15.5.1 Sequences and identity columns

Section “Identity columns and sequences” on page 141 gives you a comprehensive overview of all the options you have when using sequences and identity columns.

The CACHE option of the CREATE SEQUENCE statement is a performance and tuning option which directs DB2 to preallocate a specified number of sequential values in memory. Every time a cache is allocated, the SYSIBM.SYSSEQUENCES table is updated and a forced log record is written for the update. Using ORDER or NO CACHE will result in a SYSIBM.SYSSEQUENCES table update and the associated forced log write every time a new value is generated by DB2.

Specifying CACHE provides faster access to the sequence since values are assigned from the cache and reduces the number of SYSIBM.SYSSEQUENCES updates and the associated forced log records. The SYSIBM.SYSSEQ table space is defined with the MAXROW 1 attribute, which removes any contention among sequences.

In a data sharing environment, if ORDER is specified, then NO CACHE is implicit even if CACHE *n* is specified. In a nondata sharing environment, the numbers are always assigned in strict numeric order even if NO ORDER was specified. So specifying NO CACHE is not a good idea. There are opposing considerations when deciding on the CACHE (or ORDER) option:

- Using a CACHE > 0 can result in generating values for the sequence across multiple DB2 members which may not be in strict numeric order.
- Using a NO CACHE or ORDER can result in excessive log force writes and cross invalidation of the SYSIBM.SYSSEQUENCES table pages.

Using CACHE > 0

DB2 always generates sequence numbers in order of request. However, when a sequence is shared across multiple members of a data sharing group, each DB2 member allocates its own cache of unique consecutive numbers for the sequence. Therefore, in situations where transactions from different members are requesting the next sequence number from the same sequence, values assigned for the sequence across multiple DB2 members may not be in strict numeric order.

For instance, assume that members DB2A and DB2B share a sequence named SEQ1 that starts with 1, increments by 1, and has cache 20. If the transaction that is associated with DB2A makes the first request for a sequence number, DB2A allocates a cache of 20 values (from 1 to 20) and the value of 1 is provided to the application. If the transaction that is associated with DB2B makes the next request for a sequence number, DB2B allocates its own cache of 20 values (from 21 to 40) and the value of 21 is provided to the application. Assuming that sequence number requests continue to arrive from the transactions that are associated with members DB2A and DB2B in this manner (one from DB2A and then one from DB2B), the values assigned for that sequence are 1, 21, 2, 22, 3, 23, and so on. Although the numbers are in sequence within each DB2, the numbers assigned across multiple DB2 members are not in strict numeric sequence.

Using NO CACHE

The NO CACHE option will avoid the creation of sequence numbers or identity column values out of sequence. However, this guaranteed sequence will come at a premium cost. Without data sharing, the recoverability of the sequence/identity column is guaranteed by the log write ahead protocol.

With data sharing, we need to consider the recoverability and the inter-system consistency. DB2 uses log write force each time the entry in SYSIBM.SYSSEQUENCES is updated. This can lead to an excessive amount of force log write I/Os which in turn reduces the log write bandwidth dramatically. What is the throughput of an ONLINE LOAD or an INSERT INTO SELECT using a sequence or identity column? Their insert rate is limited by the number of I/Os to the active log that can be performed per second. At an average response time of 3

msec per I/O, only 333 inserts per second can be achieved. This will not only impact the inserter, but also all other activity taking place in this data sharing member. Using the CACHE x option would reduce the number of log force writes by a factor of x. This behavior is also present when SYSIBM.SYSSEQUENCES is not group buffer pool dependent.

We illustrate this behavior by defining a SEQUENCE with NO CACHE. See Example 15-1 for the SQL we used to create the sequence.

Example 15-1 Creating MYSEQ1 with NO CACHE

```
CREATE SEQUENCE MYSEQ1 NO CACHE;
```

We now use a simple SELECT from the employee table to drive the NEXT VALUE from the sequence. See Example 15-2 for the SELECT of the employee table.

Example 15-2 Selecting the next 42 values of MYSEQ1

```
SELECT NEXT VALUE FOR MYSEQ1 FROM DSN8810.EMP;
```

In the accounting report that corresponds to the execution of the SELECT, see Example 15-3, you notice in bold the high elapsed time which is caused by the wait for log write. The 42 synchronous I/O events are cause by the 42 NEXT VALUE in the SELECT. The full accounting report can be found in “Sequences in data sharing” on page 435.

Example 15-3 Account extract for NEXT VALUE with NO CACHE

```
TIMES/EVENTS  DB2 (CL.2)    CLASS 3 SUSPENSIONS   ELAPSED TIME    EVENTS
  ------------ ----------   --------------------  ------------   --------
  ELAPSED TIME    0.209139   SYNCHRON. I/O            0.074456         42
   NONNESTED      0.209139    DATABASE I/O            0.000000          0
                               LOG WRITE I/O          0.074456         42
  CPU TIME        0.012992   SER.TASK SWTCH           0.004303          1
   AGENT          0.012992    UPDATE COMMIT           0.004303          1
                              DRAIN LOCK               0.000197          4
                              GLOBAL CONTENTION        0.002139         11
  SUSPEND TIME    0.144975   ASYNCH CF REQUESTS       0.063880         45
   AGENT          0.144975   TOTAL CLASS 3            0.144975        103

   SQL DML     TOTAL    DATA SHARING    TOTAL
   -------- --------    ------------ --------
   OPEN            1    L-LOCKS  (%)       33
   FETCH          43    P-LOCK REQ        42
   CLOSE           1    P-UNLOCK REQ      42
                        P-CHANGE REQ       0
                        LOCK - XES        57
   DML-ALL        46    UNLOCK-XES        46
                        CHANGE-XES         0
                        SUSP - IRLM        0
                        SUSP - XES         0
                        SUSP - FALSE     N/A
                        INCOMP.LOCK        0
                        NOTIFY SENT        0
```

We changed the CACHE of the sequence to 50 (see Example 15-4 for the S) and reran the same test from Example 15-2.

Example 15-4 Changing the sequence to CACHE 50

```
ALTER SEQUENCE MYSEQ1 CACHE 50;
```

You notice in the extract of the accounting report in Example 15-5 that the number of log write events dropped to 0. The net result is a dramatic reduction of the elapsed time. The full accounting report can be found in Appendix B, “Detailed performance reports” on page 431.

Example 15-5 Account extract for NEXT VALUE with CACHE 50

```
TIMES/EVENTS    DB2 (CL.2)        CLASS 3 SUSPENSIONS   ELAPSED TIME EVENTS
------------    ----------        --------------------  ------------ ------
ELAPSED TIME      0.006704        SYNCHRON. I/O             0.000000      0
 NONNESTED        0.006704         DATABASE I/O             0.000000      0
                                   LOG WRITE I/O            0.000000      0
CPU TIME          0.001805        SER.TASK SWTCH            0.000035      1
 AGENT            0.001805         UPDATE COMMIT            0.000035      1
                                  DRAIN LOCK                 0.001463      2
SUSPEND TIME      0.003747        GLOBAL CONTENTION          0.002249      5
 AGENT            0.003747        ASYNCH CF REQUESTS        0.000000      0
                                  TOTAL CLASS 3             0.003747      8

SQL DML     TOTAL                  DATA SHARING    TOTAL
-------- --------                 ------------ --------
OPEN            1                  L-LOCKS  (%)       30
FETCH          43                  P-LOCK REQ          0
CLOSE           1                  P-UNLOCK REQ        0
                                   P-CHANGE REQ        0
                                  LOCK - XES           7
                                  UNLOCK-XES           2
                                  CHANGE-XES           0
                                  SUSP - IRLM          0
                                  SUSP - XES           0
                                  SUSP - FALSE       N/A
                                  INCOMP.LOCK          0
                                  NOTIFY SENT          0
```

Example 15-3 gives you an overview of the different elapsed time components for both the NO CACHE and the CACHE 50 test.

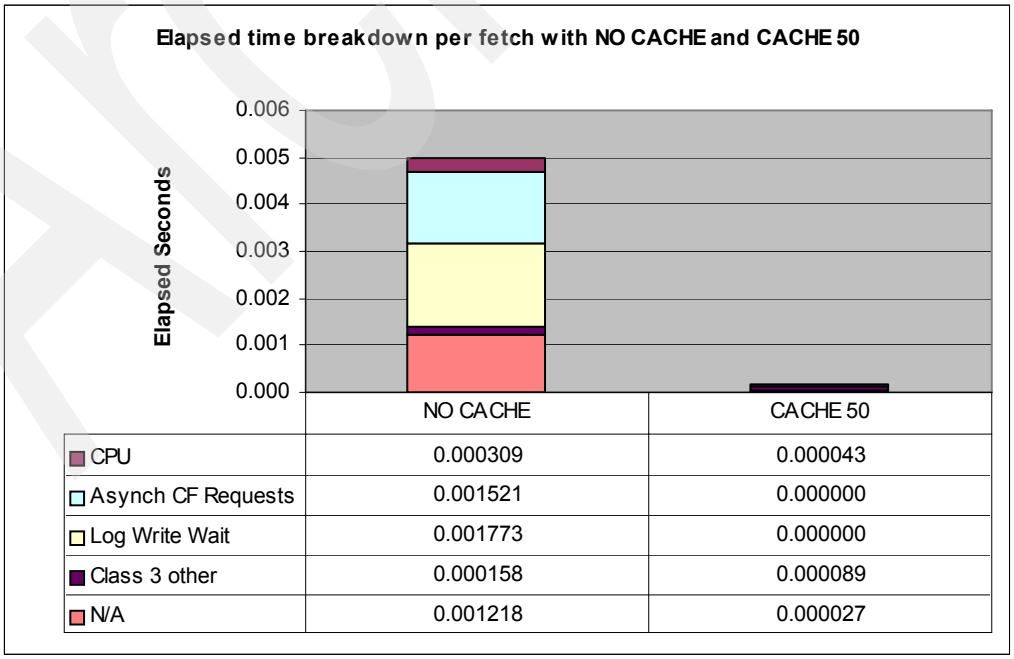


Figure 15-3 Elapsed time breakdown per FETCH with NO CACHE and CACHE 50

We notice that it takes almost 50 milliseconds of elapsed time to perform the FETCH of the NEXT VALUE of the sequence when there is no cache. Most of the time is spent on the log write wait, and the second largest consumers are the coupling facility requests.

Tip: Avoid using a small CACHE size in data sharing if your application can tolerate rows out of sequence if rows are inserted in more than one member, or if sequence values are obtained on more than one member.

Important: The extra logging is performed, even if you are in single member data sharing.

Archived

Part 6

Appendixes

This part contains the following appendixes:

- DB2 access techniques
- Detailed performance reports
- Additional material
- Abbreviations and acronyms
- Related publications

Archived

© Copyright IBM Corp. 2005. All rights reserved.

Archived

A

DB2 access techniques

In this appendix, we provide a brief description of:

- Data access methods
- Prefetch mechanisms
- Sequential detection
- Index look-aside
- Repetitive SQL procedures

First, we introduce the various access methods DB2 uses when retrieving data from disk storage. Then, we describe the types of prefetching DB2 uses when scanning data and indexes.

Finally, we document three instances of access techniques DB2 uses in order to optimize SQL execution.

Sequential detection and index look-aside reduce I/O activity and contribute to improved elapsed time and CPU time.

The repetitive SQL procedures are structures that DB2 can reuse when executing sequences of the same SQL statement, cutting down CPU time. DB2 directly uses all three techniques, based on thresholds defined by the DB2 code. Their settings can change with DB2 maintenance. There is no external parameter to activate or deactivate these functions. There are however recommendations about verifying and taking full advantage of these techniques.

© Copyright IBM Corp. 2005. All rights reserved.

A.1 Data access methods

Depending on how you code your SQL statement, DB2 has to make the decision how to look up the data you have requested. If you only look for one qualifying row it will be bad to scan the entire table to find the qualifying row. DB2 uses the following techniques to find the data you are looking for:

- Table space scan: Nonsegmented (simple) table space

  DB2 reads and examines every page in the table space, regardless to which table the page belongs. It might also read pages that have been left as free space and space not yet reclaimed after deleting data.

- Table space scan: Segmented table spaces

  DB2 determines which segments need to be read. It then reads only those segments in the table space that contain rows of the accessed table.

- Table space scan: Partitioned table space

  Partitioned tables are nonsegmented. DB2 can limit a scan of data in a partitioned table space to one or more partitions. The method is also called a *page range screening* or *limited partition scan*.

- Matching index scan: Index only

  In a matching index scan, predicates are specified for either the leading or all of the index key columns. These predicates provide filtering, only specific index pages need to be accessed. There is no access on data pages because a row can be qualified only looking at the index. Only columns contained in the index are returned by DB2.

- Matching index scan: Index and data

  In a matching index scan, predicates are specified for either the leading or all of the index key columns. These predicates provide filtering; only specific index pages and data pages need to be accessed. DB2 has to access data pages for either checking if a row qualifies or for returning columns which are not part of the index.

- Nonmatching index scan: Index only

  In a nonmatching index scan, no matching columns are in the index. All index keys must be examined. There is no access on data pages because a row can be qualified only looking at the index. Only columns contained in the index are returned by DB2.

- Nonmatching index scan: Index and data

  In a nonmatching index scan, no matching columns are in the index. All index keys must be examined. DB2 has to access data pages for either checking if a row qualifies or for returning columns which are not part of the index.

- Index screening

  In index screening, predicates are specified on index key columns but are not part of the matching columns. Those predicates improve the index access by reducing the number of rows that qualify while searching the index. To enable index screening, a predicate must be stage 1.

  Assume an index on TB1 exists with columns COLb, COLc, COLd, and COLe.

  Example A-1 on page 423 provides a simple example of index screening predicates.

Example: A-1 Index screening predicate

```
SELECT COLa FROM TB1
 WHERE COLb = 1
   AND COLc > 0
   AND COLd < 2;
```

The predicates can be applied to the index, but they are not all matching predicates. COLc and COLd are index screening predicates. There is one matching column in the example, which is COLb.

- Multiple index access

  Multiple index access uses more than one index to access a table. It is a good access path when no single index provides efficient access and a combination of index accesses provides efficient access. RID lists are constructed for each index involved. RIDs qualifying in all indexes are sorted and used for data access using list prefetch.

- Index look-aside

  DB2 checks whether the required entry is in the leaf page accessed by the previous call and checks against the lowest an highest key of the leaf page. If the entry is found, DB2 can avoid the getpage and traversal of the index tree.

  If the entry is not within the cached range, DB2 checks the parent nonleaf page's lowest and highest key. If the entry is found in the parent nonleaf range, DB2 has to perform a getpage, but can avoid a full traversal of the index tree.

  In case an entry is not found within the parent nonleaf page, DB2 starts an index probe from the index root page. See also "Index look-aside" on page 426.

- Direct row access

  Allows DB2 to navigate directly to that row without using an index access or a table space scan to find the row.

  You can find more details about direct row access in "Data access using ROWID" on page 276.

A.2 Prefetch mechanisms

For table space scans, DB2 generally uses sequential prefetch. Sequential prefetch reads a sequential set of pages into the buffer pool with one asynchronous I/O. Usually the maximum number of pages is 32 for a base table.

Sometimes sequential prefetch is also used for index scans:

- If index cluster ratio greater than 0.8
- For index only, if the number of qualified leaf pages > 8
- For index and data, if the number of qualified clustered data pages > 8

If DB2 is not able to choose sequential prefetch at bind time, it may still utilize it at run time:

- DB2 determines further if data access is sequential or nearly sequential. This kind of sequential detection is also known as *dynamic prefetch*.

  Data access is declared sequential if more than four out of the last eight pages are page-sequential (not valid for DB2 internal referential integrity checks).

- Tracking is continuous, allowing access to slip into and out of sequential data access.

- Runtime usage can be determined by your system DBA using:
  - IFCID 0003 in accounting traces
  - IFCID 0006 in performance traces

DB2 also supports list prefetch, which differs from sequential prefetch:

- List prefetch reads a set of pages into the buffer pool with one synchronous I/O.
- The set of pages is determined by a list of RIDs taken from an index.
- Currently the maximum number of pages prefetched is 32 within 180 pages swath (list prefetch can skip pages).

List prefetch is generally used with:

- Index scan when cluster ratio is less than 0.8
- Accessing data from the inner table during a hybrid join
- Multiple index access
- When direct row access is not possible (FOR UPDATE OF...)
- With high CLUSTERRATIO, if the number of qualified pages is between 1 and the number of pages required for sequential prefetch

For estimating the time a certain query takes to complete, see “Quick upper bound estimate” on page 120.

A.3 Sequential detection

Prefetch is a mechanism for reading a set of pages, usually 32, into the buffer pool with only one asynchronous I/O operation. Prefetch can allow substantial savings in both processor cycles and I/O costs. A plan table can indicate the use of three kinds of prefetch:

- Sequential prefetch (PREFETCH=S)
- Dynamic prefetch (PREFETCH=D)
- List prefetch (PREFETCH=L)

Even if DB2 does not choose prefetch at bind time, it can sometimes use it at execution time, the method is called “sequential detection at execution time”.

Sequential detection was introduced in DB2 V2.3. It enables sequential prefetch I/O at run time when the optimizer does not choose this option at bind time. For example, a single SQL SELECT statement is not usually optimized with sequential prefetch I/O at bind time. However, the sequential detection function tracks and monitors the SQL execution at run time. If the SELECT statement is in a loop, for example, and requires pages continuously, sequential prefetch is invoked dynamically at run time. Sequential prefetch stops when the function detects that prefetch is no longer needed.

In loop cases sequential detection dramatically reduces the I/O time, because it can trigger sequential prefetch, which exploits asynchronous I/O processing instead of synchronous I/O processing.

Starting with V4, DB2 extended the use of sequential detection to data pages in INSERT, and to data pages in UPDATE and DELETE when there is an index-only access during retrieval.

DB2 can use sequential detection for both index leaf pages and data pages. It is most commonly used on the inner table of a nested loop join, if the data is accessed sequentially. If a table is accessed repeatedly using the same statement (for example, DELETE in a do-while

loop), the data or index leaf pages of the table can be accessed sequentially. This is common in a batch processing environment.

Sequential detection can then be used if access is through:

- SELECT or FETCH statements
- UPDATE and DELETE statements
- INSERT statements

When existing data pages are accessed sequentially, DB2 can use sequential detection if it did not choose sequential prefetch at bind time because of an inaccurate estimate of the number of pages to be accessed.

Sequential detection is not used for an SQL statement that is subject to referential constraints.

You can determine whether sequential detection was used from record IFCID 0003 in the accounting trace or record IFCID 0006 in the performance trace.

The pattern of accessing a page is tracked when the application scans DB2 data through an index. Tracking is done to detect situations where the access pattern that develops is sequential or nearly sequential. The most recent eight pages are tracked. A page is considered page-sequential, if it is within P/2 advancing pages of the current page, where P is the prefetch quantity. P is usually 32. If a page is page-sequential, DB2 determines further if data access is sequential or nearly sequential. Data access is declared sequential if more than four out of the last eight pages are page-sequential; this is also true for index-only access. The tracking is continuous, allowing access to slip into and out of data access sequential. When data access is first declared sequential, which is called *initial data access sequential*, three page ranges are calculated as follows:

- Let A be the page being requested. RUN1 is defined as the page range of length P/2 pages starting at A.
- Let B be page A + P/2. RUN2 is defined as the page range of length P/2 pages starting at B.
- Let C be page B + P/2. RUN3 is defined as the page range of length P pages starting at C.

For example, assume that page A is 10. Figure A-1 on page 426 illustrates the page ranges that DB2 calculates.

For initial data access sequential, prefetch is requested starting at page A for P pages (RUN1 and RUN2). The prefetch quantity is always P pages. For subsequent page requests where the page is 1) page sequential and 2) data access sequential is still in effect, prefetch is requested as follows:

- If the desired page is in RUN1, no prefetch is triggered because it was already triggered when data access sequential was first declared.
- If the desired page is in RUN2, prefetch for RUN3 is triggered and RUN2 becomes RUN1, RUN3 becomes RUN2, and RUN3 becomes the page range starting at C+P for a length of P pages.

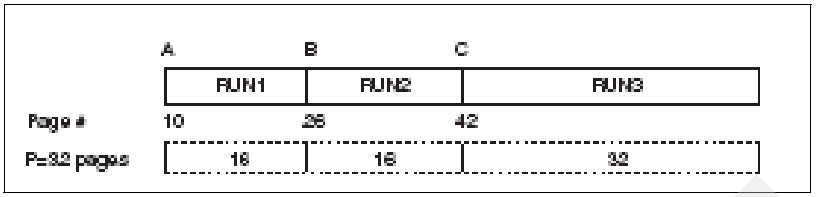


Figure A-1 Initial page ranges to determine when to use prefetch

If a data access pattern develops such that data access sequential is no longer in effect, and, thereafter, a new pattern develops that is sequential, then initial data access sequential is declared again and handled accordingly.

Because, at bind time, the number of pages to be accessed can only be estimated, sequential detection acts as a safety net and is employed when the data is being accessed sequentially.

In extreme situations, when certain buffer pool thresholds are reached, sequential prefetch can be disabled.

The benefits of sequential detection for update, insert, and delete operations are due to the fact that expensive individual synchronous I/O operations are replaced by dynamic prefetch activity, producing significant improvement in elapsed time.

Note: For plans bound with RELEASE(COMMIT), the cache used for sequential detection is released at commit time. This can adversely impact the activation of dynamic prefetch.

A.4 Index look-aside

The objective of the index look-aside technique is to minimize the number of getpage operations that are generated when an individual SQL statement or DB2 process is executed repeatedly and makes reference to the same or nearby pages. Index look-aside results in a significant reduction in the number of index and data page getpage requests when an index is accessed in a sequential, skip-sequential, or *hot spot* pattern.

A DB2 index is a set of key data organized into a "B-tree" structure. Figure A-2 on page 427 shows an example of a three-level DB2 index. At the bottom of the tree are the leaf pages of the index. Each leaf page contains a number of index entries consisting of the index key itself and a pointer, known as a record identifier (RID), which are used to locate the indexed data rows. Each entry in the intermediate nonleaf index page (LEVEL 2) identifies the highest key of a dependent leaf page along with a pointer to the leaf page's location. At the top of the tree, a single page, called the *root page*, provides the initial entry point into the index tree structure. Just like nonleaf pages, the root page contains one entry for each dependent LEVEL 2 page.

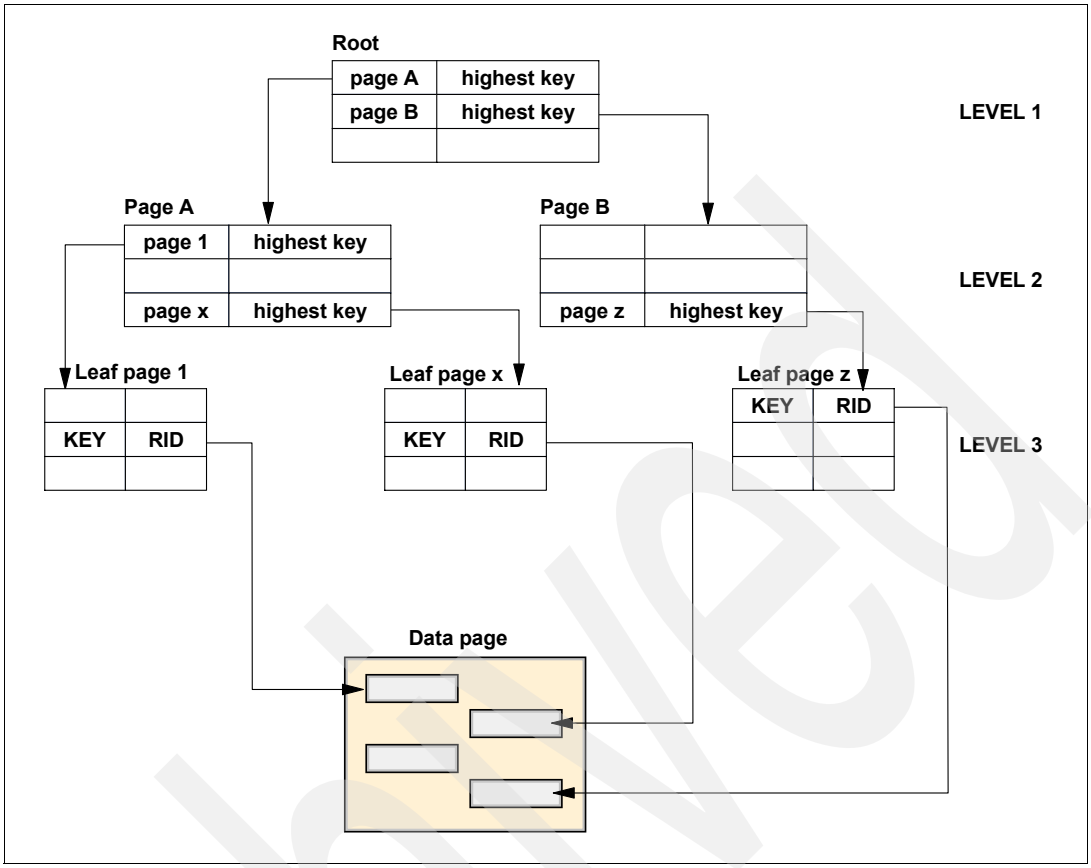


Figure A-2 DB2 index structure

Index look-aside consists of DB2 checking whether the required entry is in the leaf page that was accessed by the previous call. If the entry is not found, DB2 checks the range of pages that are addressed by the parent nonleaf page. If the entry is not found there either, DB2 goes to the top of the index tree structure and establishes an absolute position on the required leaf page.

For example, assume your table has the data and index structure shown in Figure A-3 on page 428.

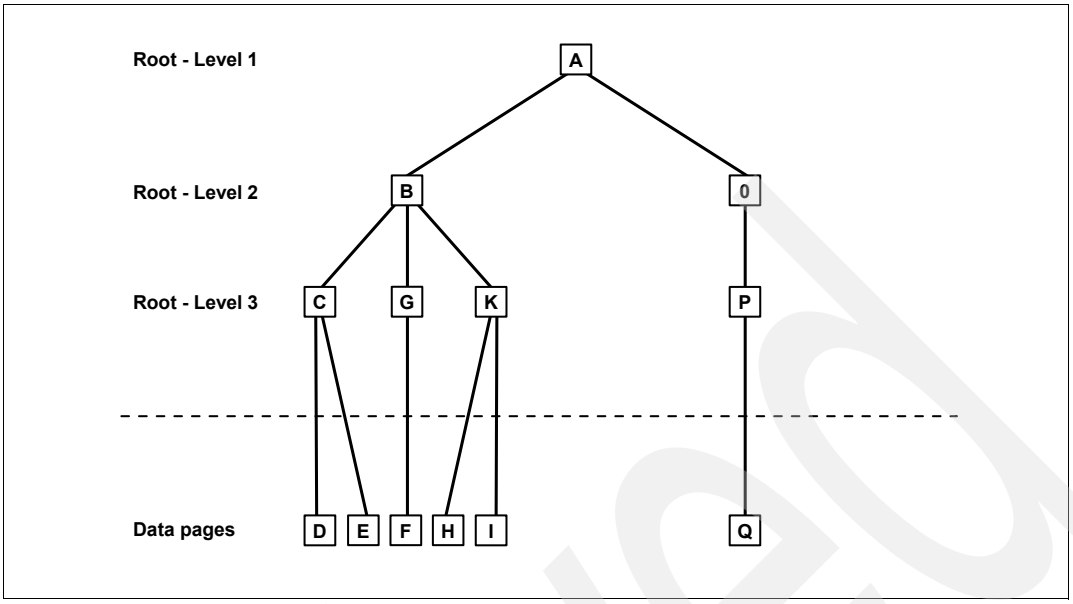


Figure A-3 DB2 table structure

Assume also that your program reads from a sorted input file the key of the rows you want to access and processes as the pseudocode shows in Example A-2.

Example: A-2 Sample sequential processing program

```
READ INPUT FILE
DO WHILE (RECORDS_TO_PROCESS)
SELECT * FROM TABLE
WHERE KEY = INPUT_KEY
UPDATE TABLE
READ INPUT FILE
END DO;
```

Assuming the input file is sorted in the clustering index sequence and the index is clustered, when the program executes, the sequence of getpages is:

1. For the first select to a row in D: getpages A, B, C, D
2. For the remaining selects to rows in D: no getpages
3. For first select to a row in E: getpage E
4. For the remaining selects to rows in E: no getpages

...

5. For the first select to a row in H: getpages K, H

...

6. For the first select to a row in Q: getpages A, O, P, Q

...

Compare the number of getpages required for random accesses in the example table. This is usually four per select (one per index level plus one per data page).

A reduced number of getpages leads to reduced CPU time.

Note: For plans bound with RELEASE(COMMIT), the cache used for index look-aside is released at commit time. This can adversely impact the avoidance on index probing.

A.5 Repetitive SQL procedures

In order to reduce and optimize SQL path length, DB2 detects that some SQL statements are repetitive, and builds internal procedures. Basically, DB2 monitors the SQL execution, and based on internal thresholds related to the number of times the same SQL statements are being issued against the same object, decides to build an executable procedure (internal structure) for processing column definitions. This accelerates the following executions. The more columns and rows are involved, the greater the benefit will be.

The procedures are related to SELECT (SPROC), INSERT (IPROC), and UPDATE (UPROC).

There is no “user” external parameter to affect the triggering of these procedures, the activation is automatic, and based on values chosen and set only by DB2.

However, your commit strategy (both frequency of commits and RELEASE COMMIT or DEALLOCATE) affects the persistence of any active procedure during the program execution. After an SQL COMMIT, the threshold is reset and must be reached again.

This is not true if RELEASE DEALLOCATE is specified at bind time or the cursors declared use is the WITH HOLD option.

Furthermore, maintenance or code incompatibilities can have the effect of disabling the SPROC altogether. Generally a REBIND removes the restrictions caused by maintenance issues.

Changes to data type (online schema) also disable the fast path for column processing at ALTER time until re-enabled at REORG.

You can verify the number of disablements of SPROCs by checking the QISTCOLS field of IFCID 2 (statistics), reported by DB2 PE as BYPASS COL. If it is not zero, consider activating IFCID 224 to identify plans and packages to be rebound.

INSERT procedure

For multiple inserts, starting with Version 3, DB2 builds the optimized code for the insert operation at the third insert and uses the optimized code from the fourth insert on. If RELEASE DEALLOCATE is specified, DB2 retains the optimized code beyond commit points.

INSERT loop, INSERT subselect, and LOAD are the main beneficiaries of this enhancement.

UPDATE procedure

The DB2 update procedure (UPROC) was introduced with DB2 Version 4. Prior versions of DB2 processed each column definition for every row updated. V4 improved the performance of update operations by building the update procedure for more than three rows in a table. This improvement significantly reduces the cost of updating more than four rows.

The use of UPROC significantly reduces the cost of updating a large number of rows.

Archived

B

Detailed performance reports

This Appendix contains performance measurement reports related to Chapter 6, “Database side programming” on page 151, where we discuss several options DB2 provides in terms of server side programming techniques.

We show the detailed reports we used for:

- Language Environment storage
- Trigger performance
- Sequences in data sharing

© Copyright IBM Corp. 2005. All rights reserved.

B.1 Language Environment storage

The memory that is used by LE is dependent on your installation defaults. It may be useful to validate how much memory is used and adapt the LE runtime options. The best approach is to activate the LE storage report for a short period of time and then disable the storage report and adapt the run-time options based on the report output.

The storage report produced by run-time option RPTSTG(ON) can be found in Example B-1.

See “Calibrate LE options” on page 160 for an analysis of the report.

Example: B-1 Storage report produced by run-time option RPTSTG(ON)

```
Storage Report for Enclave main 07/23/05 8:23:20 PM
Language Environment V01 R06.00

STACK statistics:
   Initial size:                                          4096
   Increment size:                                        4096
   Maximum used by all concurrent threads:                7488
   Largest used by any thread:                            7488
   Number of segments allocated:                             2
   Number of segments freed:                                 0
THREADSTACK statistics:
   Initial size:                                          4096
   Increment size:                                        4096
   Maximum used by all concurrent threads:                3352
   Largest used by any thread:                            3352
   Number of segments allocated:                             6
   Number of segments freed:                                 0
LIBSTACK statistics:
   Initial size:                                          4096
   Increment size:                                        4096
   Maximum used by all concurrent threads:                   0
   Largest used by any thread:                               0
   Number of segments allocated:                             0
   Number of segments freed:                                 0
THREADHEAP statistics:
   Initial size:                                          4096
   Increment size:                                        4096
   Maximum used by all concurrent threads:                   0
   Largest used by any thread:                               0
   Successful Get Heap requests:                             0
   Successful Free Heap requests:                            0
   Number of segments allocated:                             0
   Number of segments freed:                                 0
HEAP statistics:
   Initial size:                                         49152
   Increment size:                                       16384
   Total heap storage used (sugg. initial size):         29112
   Successful Get Heap requests:                           251
   Successful Free Heap requests:                          218
   Number of segments allocated:                             1
   Number of segments freed:                                 0
HEAP24 statistics:
   Initial size:                                          8192
   Increment size:                                        4096
   Total heap storage used (sugg. initial size):             0
   Successful Get Heap requests:                             0
   Successful Free Heap requests:                            0
```

```
   Number of segments allocated:                                    0
   Number of segments freed:                                        0
ANYHEAP statistics:
   Initial size:                                                32768
   Increment size:                                              16384
   Total heap storage used (sugg. initial size):               104696
   Successful Get Heap requests:                                   28
   Successful Free Heap requests:                                  15
   Number of segments allocated:                                    6
   Number of segments freed:                                        5
BELOWHEAP statistics:
   Initial size:                                                 8192
   Increment size:                                               8192
   Total heap storage used (sugg. initial size):                    0
   Successful Get Heap requests:                                    0
   Successful Free Heap requests:                                   0
   Number of segments allocated:                                    0
   Number of segments freed:                                        0
Additional Heap statistics:
   Successful Create Heap requests:                                 1
   Successful Discard Heap requests:                                1
   Total heap storage used:                                      4912
   Successful Get Heap requests:                                    3
   Successful Free Heap requests:                                   3
   Number of segments allocated:                                    2
   Number of segments freed:                                        2
   Largest number of threads concurrently active:                   2
End of Storage
Report
```

B.2 Trigger performance

In “Using triggers” on page 168, we discuss triggers. In this Appendix, we provide the accounting detail report used to compare the performance of a row trigger versus a statement trigger on an insert with a very large number of rows. The scenario and the analysis are described in “Deciding between after row and after statement triggers” on page 169. The full accounting report for the row trigger is Example B-2.

Example: B-2 Row trigger accounting

```
ACCT TSTAMP: 07/22/05 17:36:48.56    PLANNAME: DSNESPCS        WLM SCL: 'BLANK'         CICS NET: N/A
 BEGIN TIME : 07/22/05 17:35:55.93    PROD ID : N/P                                      CICS LUN: N/A
 END TIME   : 07/22/05 17:36:48.56    PROD VER: N/P             LUW NET: USIBMSC         CICS INS: N/A
 REQUESTER  : DB8A                    CORRNAME: PAOLOR2         LUW LUN: SCPDB8A
 MAINPACK   : DSNESM68                CORRNMBR: 'BLANK'         LUW INS: BD590C847391    ENDUSER : 'BLANK'
 PRIMAUTH   : PAOLOR2                 CONNTYPE: TSO             LUW SEQ:            1    TRANSACT: 'BLANK'
 ORIGAUTH   : PAOLOR2                 CONNECT : TSO                                      WSNAME  : 'BLANK'
ELAPSED TIME DISTRIBUTION                                         CLASS 2 TIME DISTRIBUTION
 ----------------------------------------------------------------   ----------------------------------------------------------------
 APPL   |                                                          CPU    |======> 12%
 DB2    |=======> 14%                                              NOTACC |=> 2%
 SUSP   |==========================================> 86%          SUSP   |==========================================> 86%

 TIMES/EVENTS  APPL(CL.1)  DB2 (CL.2)  IFI (CL.5)    CLASS 3 SUSPENSIONS   ELAPSED TIME    EVENTS    HIGHLIGHTS
 ------------  ----------  ----------  ----------    --------------------  ------------  --------    --------------------------
 ELAPSED TIME   52.625410   52.551982         N/P    LOCK/LATCH(DB2+IRLM)      0.001424         3    THREAD TYPE   : ALLIED
  NONNESTED     28.868211   28.794784         N/A    SYNCHRON. I/O            44.148001     10090    TERM.CONDITION: NORMAL
  STORED PROC    0.000000    0.000000         N/A     DATABASE I/O            42.671875     10089    INVOKE REASON : DEALLOC
  UDF            0.000000    0.000000         N/A     LOG WRITE I/O            1.476127         1    COMMITS       :          1
  TRIGGER       23.757199   23.757199         N/A    OTHER READ I/O            0.001937         1    ROLLBACK      :          0
                                                      OTHER WRTE I/O            1.147213         8    SVPT REQUESTS :          0
 CPU TIME        6.433509    6.423486         N/P    SER.TASK SWTCH            0.011687         1    SVPT RELEASE  :          0
  AGENT          6.433509    6.423486         N/A     UPDATE COMMIT            0.011687         1    SVPT ROLLBACK :          0
   NONNESTED     2.819868    2.809844         N/P     OPEN/CLOSE               0.000000         0    INCREM.BINDS  :          0
```

```
   STORED PRC    0.000000    0.000000         N/A     SYSLGRNG REC            0.000000         0    UPDATE/COMMIT :     165.8K
   UDF           0.000000    0.000000         N/A     EXT/DEL/DEF             0.000000         0    SYNCH I/O AVG.:   0.004375
   TRIGGER       3.613641    3.613641         N/A     OTHER SERVICE           0.000000         0    PROGRAMS      :          3
  PAR.TASKS      0.000000    0.000000         N/A    ARC.LOG(QUIES)           0.000000         0    MAX CASCADE   :          1
                                                      ARC.LOG READ             0.000000         0    PARALLELISM   : NO
 SUSPEND TIME    0.000000   45.310263         N/A    DRAIN LOCK               0.000000         0
  AGENT               N/A   45.310263         N/A    CLAIM RELEASE            0.000000         0
  PAR.TASKS           N/A    0.000000         N/A    PAGE LATCH               0.000000         0
  STORED PROC    0.000000         N/A         N/A    NOTIFY MSGS              0.000000         0
  UDF            0.000000         N/A         N/A    GLOBAL CONTENTION        0.000000         0
                                                      COMMIT PH1 WRITE I/O     0.000000         0
 NOT ACCOUNT.         N/A    0.818234         N/P    ASYNCH CF REQUESTS       0.000000         0
 DB2 ENT/EXIT         N/A          14         N/A    TOTAL CLASS 3           45.310263     10103
 EN/EX-STPROC         N/A           0         N/A
 EN/EX-UDF            N/A           0         N/A
 DCAPT.DESCR.         N/A         N/A         N/P
 LOG EXTRACT.         N/A         N/A         N/P

SQL DML     TOTAL    SQL DCL       TOTAL    SQL DDL    CREATE   DROP  ALTER    LOCKING                TOTAL    DATA SHARING    TOTAL
-------- --------    ---------- --------    ---------- ------ ------ ------    -------------------- --------    ------------ --------
 SELECT          0    LOCK TABLE        0    TABLE           0      0      0    TIMEOUTS                 0    L-LOCKS  (%)       N/P
 INSERT     165830    GRANT             0    CRT TTABLE      0    N/A    N/A    DEADLOCKS                0    P-LOCK REQ        N/P
 UPDATE          0    REVOKE            0    DCL TTABLE      0    N/A    N/A    ESCAL.(SHAR)             0    P-UNLOCK REQ      N/P
 DELETE          0    SET SQLID         0    AUX TABLE       0    N/A    N/A    ESCAL.(EXCL)             0    P-CHANGE REQ      N/P
                      SET H.VAR.        0    INDEX           0      0      0    MAX PG/ROW LCK HELD   5690    LOCK - XES        N/P
 DESCRIBE        0    SET DEGREE        0    TABLESPACE      0      0      0    LOCK REQUEST         11487    UNLOCK-XES        N/P
 DESC.TBL        0    SET RULES         0    DATABASE        0      0      0    UNLOCK REQST          5768    CHANGE-XES        N/P
 PREPARE         1    SET PATH          0    STOGROUP        0      0      0    QUERY REQST              0    SUSP - IRLM       N/P
 OPEN            0    SET PREC.         0    SYNONYM         0      0    N/A    CHANGE REQST             1    SUSP - XES        N/P
 FETCH           0    CONNECT 1         0    VIEW            0      0    N/A    OTHER REQST              0    SUSP - FALSE      N/A
 CLOSE           0    CONNECT 2         0    ALIAS           0      0    N/A    LOCK SUSPENS.            0    INCOMP.LOCK       N/P
                      SET CONNEC        0    PACKAGE       N/A      0    N/A    IRLM LATCH SUSPENS.      3    NOTIFY SENT       N/P
                      RELEASE           0    PROCEDURE       0      0      0    OTHER SUSPENS.           0
 DML-ALL    165831    CALL              0    FUNCTION        0      0      0    TOTAL SUSPENS.           3
                      ASSOC LOC.        0    TRIGGER         0      0    N/A
                      ALLOC CUR.        0    DIST TYPE       0      0    N/A
                      HOLD LOC.         0    SEQUENCE        0      0      0
                      FREE LOC.         0
                      DCL-ALL           0    TOTAL           0      0      0
                                             RENAME TBL      0
                                             COMMENT ON      0
                                             LABEL ON        0

 RID LIST            TOTAL    ROWID          TOTAL    STORED PROC.     TOTAL    UDF           TOTAL    TRIGGERS         TOTAL
 ---------------  --------    ---------- --------    ------------  --------    ---------  --------    ------------  --------
 USED                     0    DIR ACCESS        0    CALL STMTS           0    EXECUTED         0    STMT TRIGGER         0
 FAIL-NO STORAGE          0    INDEX USED        0    ABENDED              0    ABENDED          0    ROW TRIGGER     165829
 FAIL-LIMIT EXC.          0    TS SCAN           0    TIMED OUT            0    TIMED OUT        0    SQL ERROR            0
                                                      REJECTED             0    REJECTED         0
```

The report for the statement trigger is listed in Example B-3.

Example: B-3 Statement trigger accounting

```
ACCT TSTAMP: 07/22/05 17:32:30.70    PLANNAME: DSNESPCS        WLM SCL: 'BLANK'         CICS NET: N/A
 BEGIN TIME : 07/22/05 17:31:40.53    PROD ID : N/P                                       CICS LUN: N/A
 END TIME   : 07/22/05 17:32:30.70    PROD VER: N/P             LUW NET: USIBMSC          CICS INS: N/A
 REQUESTER  : DB8A                    CORRNAME: PAOLOR2         LUW LUN: SCPDB8A
 MAINPACK   : DSNESM68                CORRNMBR: 'BLANK'         LUW INS: BD590B90E0F3    ENDUSER : 'BLANK'
 PRIMAUTH   : PAOLOR2                 CONNTYPE: TSO             LUW SEQ:            1    TRANSACT: 'BLANK'
 ORIGAUTH   : PAOLOR2                 CONNECT : TSO                                       WSNAME  : 'BLANK'
ELAPSED TIME DISTRIBUTION                                          CLASS 2 TIME DISTRIBUTION
 ----------------------------------------------------------------   ----------------------------------------------------------------
 APPL   |                                                          CPU    |====> 9%
 DB2    |=====> 10%                                                NOTACC |> 1%
 SUSP   |==========================================> 89%           SUSP   |==========================================> 90%

 TIMES/EVENTS  APPL(CL.1)  DB2 (CL.2)  IFI (CL.5)    CLASS 3 SUSPENSIONS   ELAPSED TIME    EVENTS    HIGHLIGHTS
 ------------  ----------  ----------  ----------    --------------------  ------------  --------    --------------------------
 ELAPSED TIME   50.175786   50.111831         N/P    LOCK/LATCH(DB2+IRLM)      0.011247         5    THREAD TYPE   : ALLIED
  NONNESTED     28.336861   28.272906         N/A    SYNCHRON. I/O            44.804275     10436    TERM.CONDITION: NORMAL
  STORED PROC    0.000000    0.000000         N/A     DATABASE I/O            44.804275     10436    INVOKE REASON : DEALLOC
  UDF            0.000000    0.000000         N/A     LOG WRITE I/O            0.000000         0    COMMITS       :          1
  TRIGGER       21.838925   21.838925         N/A    OTHER READ I/O            0.000946         1    ROLLBACK      :          0
                                                      OTHER WRTE I/O            0.078738         6    SVPT REQUESTS :          0
 CPU TIME        4.695984    4.686395         N/P    SER.TASK SWTCH            0.011289         2    SVPT RELEASE  :          0
  AGENT          4.695984    4.686395         N/A     UPDATE COMMIT            0.007962         1    SVPT ROLLBACK :          0
   NONNESTED     2.823885    2.814296         N/P     OPEN/CLOSE               0.000000         0    INCREM.BINDS  :          0
   STORED PRC    0.000000    0.000000         N/A     SYSLGRNG REC             0.003327         1    UPDATE/COMMIT :       2.00
   UDF           0.000000    0.000000         N/A     EXT/DEL/DEF              0.000000         0    SYNCH I/O AVG.:   0.004293
   TRIGGER       1.872099    1.872099         N/A     OTHER SERVICE            0.000000         0    PROGRAMS      :          2
```

```
  PAR.TASKS      0.000000    0.000000         N/A    ARC.LOG(QUIES)            0.000000         0    MAX CASCADE   :          1
                                                      ARC.LOG READ              0.000000         0    PARALLELISM   : NO
 SUSPEND TIME    0.000000   44.906495         N/A    DRAIN LOCK                0.000000         0
  AGENT               N/A   44.906495         N/A    CLAIM RELEASE             0.000000         0
  PAR.TASKS           N/A    0.000000         N/A    PAGE LATCH                0.000000         0
  STORED PROC    0.000000         N/A         N/A    NOTIFY MSGS               0.000000         0
  UDF            0.000000         N/A         N/A    GLOBAL CONTENTION         0.000000         0
                                                      COMMIT PH1 WRITE I/O      0.000000         0
 NOT ACCOUNT.         N/A    0.518941         N/P    ASYNCH CF REQUESTS        0.000000         0
 DB2 ENT/EXIT         N/A          10         N/A    TOTAL CLASS 3            44.906495     10450
 EN/EX-STPROC         N/A           0         N/A
 EN/EX-UDF            N/A           0         N/A
 DCAPT.DESCR.         N/A         N/A         N/P
 LOG EXTRACT.         N/A         N/A         N/P

SQL DML     TOTAL    SQL DCL       TOTAL    SQL DDL    CREATE   DROP  ALTER    LOCKING                TOTAL    DATA SHARING    TOTAL
-------- --------    ---------- --------    ---------- ------ ------ ------    -------------------- --------    ------------ --------
 SELECT          0    LOCK TABLE        0    TABLE           0      0      0    TIMEOUTS                  0    L-LOCKS  (%)       N/P
 INSERT          2    GRANT             0    CRT TTABLE      0    N/A    N/A    DEADLOCKS                 0    P-LOCK REQ        N/P
 UPDATE          0    REVOKE            0    DCL TTABLE      0    N/A    N/A    ESCAL.(SHAR)              0    P-UNLOCK REQ      N/P
 DELETE          0    SET SQLID         0    AUX TABLE       0    N/A    N/A    ESCAL.(EXCL)              0    P-CHANGE REQ      N/P
                      SET H.VAR.        0    INDEX           0      0      0    MAX PG/ROW LCK HELD       2    LOCK - XES        N/P
 DESCRIBE        0    SET DEGREE        0    TABLESPACE      0      0      0    LOCK REQUEST           5726    UNLOCK-XES        N/P
 DESC.TBL        0    SET RULES         0    DATABASE        0      0      0    UNLOCK REQST           5697    CHANGE-XES        N/P
 PREPARE         1    SET PATH          0    STOGROUP        0      0      0    QUERY REQST               0    SUSP - IRLM       N/P
 OPEN            0    SET PREC.         0    SYNONYM         0      0    N/A    CHANGE REQST              1    SUSP - XES        N/P
 FETCH           0    CONNECT 1         0    VIEW            0      0    N/A    OTHER REQST               0    SUSP - FALSE      N/A
 CLOSE           0    CONNECT 2         0    ALIAS           0      0    N/A    LOCK SUSPENS.             0    INCOMP.LOCK       N/P
                      SET CONNEC        0    PACKAGE       N/A      0    N/A    IRLM LATCH SUSPENS.       1    NOTIFY SENT       N/P
                      RELEASE           0    PROCEDURE       0      0      0    OTHER SUSPENS.            0
 DML-ALL         3    CALL              0    FUNCTION        0      0      0    TOTAL SUSPENS.            1
                      ASSOC LOC.        0    TRIGGER         0      0    N/A
                      ALLOC CUR.        0    DIST TYPE       0      0    N/A
                      HOLD LOC.         0    SEQUENCE        0      0      0
                      FREE LOC.         0
                      DCL-ALL           0    TOTAL           0      0      0
                                             RENAME TBL      0
                                             COMMENT ON      0
                                             LABEL ON        0

 RID LIST            TOTAL    ROWID          TOTAL    STORED PROC.     TOTAL    UDF           TOTAL    TRIGGERS         TOTAL
 ---------------  --------    ----------  --------    ------------  --------    ----------  --------    ------------  --------
 USED                     0    DIR ACCESS         0    CALL STMTS           0    EXECUTED          0    STMT TRIGGER          1
 FAIL-NO STORAGE          0    INDEX USED         0    ABENDED              0    ABENDED           0    ROW TRIGGER           0
 FAIL-LIMIT EXC.          0    TS SCAN            0    TIMED OUT            0    TIMED OUT         0    SQL ERROR             0
                                                       REJECTED             0    REJECTED          0
```

B.3 Sequences in data sharing

We tested the performance of sequences with varying CACHE values. The full scenario and analysis can be found in “Sequences and identity columns” on page 414.

The accounting report for SELECT with NO CACHE is shown in Example B-4.

Example: B-4 Accounting report for SELECT with NO CACHE

```
ACCT TSTAMP: 07/22/05 23:22:40.76    PLANNAME: DSNESPCS        WLM SCL: 'BLANK'         CICS NET: N/A
 BEGIN TIME : 07/22/05 23:22:40.48    PROD ID : N/P                                       CICS LUN: N/A
 END TIME   : 07/22/05 23:22:40.76    PROD VER: N/P             LUW NET: USIBMSC          CICS INS: N/A
 REQUESTER  : DB8F                    CORRNAME: PAOLOR2         LUW LUN: SCPD8F1
 MAINPACK   : DSNESM68                CORRNMBR: 'BLANK'         LUW INS: BD595A0537EB     ENDUSER : 'BLANK'
 PRIMAUTH   : PAOLOR2                 CONNTYPE: TSO             LUW SEQ:            1     TRANSACT: 'BLANK'
 ORIGAUTH   : PAOLOR2                 CONNECT : TSO                                       WSNAME  : 'BLANK'

ELAPSED TIME DISTRIBUTION                                        CLASS 2 TIME DISTRIBUTION
 ------------------------------------------------------------     ------------------------------------------------------------
 APPL   |============> 24%                                        CPU    |===> 6%
 DB2    |===========> 23%                                         NOTACC |============> 24%
 SUSP   |=========================> 52%                            SUSP   |==================================> 69%

 TIMES/EVENTS  APPL(CL.1)  DB2 (CL.2)  IFI (CL.5)    CLASS 3 SUSPENSIONS   ELAPSED TIME    EVENTS    HIGHLIGHTS
 ------------  ----------  ----------  ----------    --------------------  ------------  --------    --------------------------
 ELAPSED TIME    0.276563    0.209139         N/P    LOCK/LATCH(DB2+IRLM)      0.000000         0    THREAD TYPE   : ALLIED
  NONNESTED      0.276563    0.209139         N/A    SYNCHRON. I/O             0.074456        42    TERM.CONDITION: NORMAL
  STORED PROC    0.000000    0.000000         N/A     DATABASE I/O             0.000000         0    INVOKE REASON : DEALLOC
  UDF            0.000000    0.000000         N/A     LOG WRITE I/O            0.074456        42    COMMITS       :          1
```

```
 TRIGGER        0.000000    0.000000         N/A    OTHER READ I/O            0.000000         0    ROLLBACK      :          0
                                                     OTHER WRTE I/O            0.000000         0    SVPT REQUESTS :          0
CPU TIME        0.022709    0.012992         N/P    SER.TASK SWTCH            0.004303         1    SVPT RELEASE  :          0
 AGENT          0.022709    0.012992         N/A     UPDATE COMMIT            0.004303         1    SVPT ROLLBACK :          0
  NONNESTED     0.022709    0.012992         N/P     OPEN/CLOSE               0.000000         0    INCREM.BINDS  :          0
  STORED PRC    0.000000    0.000000         N/A     SYSLGRNG REC             0.000000         0    UPDATE/COMMIT :       0.00
  UDF           0.000000    0.000000         N/A     EXT/DEL/DEF              0.000000         0    SYNCH I/O AVG.:   0.001773
  TRIGGER       0.000000    0.000000         N/A     OTHER SERVICE            0.000000         0    PROGRAMS      :          1
 PAR.TASKS      0.000000    0.000000         N/A    ARC.LOG(QUIES)            0.000000         0    MAX CASCADE   :          0
                                                     ARC.LOG READ              0.000000         0    PARALLELISM   : NO
SUSPEND TIME    0.000000    0.144975         N/A    DRAIN LOCK                0.000197         4
 AGENT               N/A    0.144975         N/A    CLAIM RELEASE             0.000000         0
 PAR.TASKS           N/A    0.000000         N/A    PAGE LATCH                0.000000         0
 STORED PROC    0.000000         N/A         N/A    NOTIFY MSGS               0.000000         0
 UDF            0.000000         N/A         N/A    GLOBAL CONTENTION         0.002139        11
                                                     COMMIT PH1 WRITE I/O      0.000000         0
NOT ACCOUNT.         N/A    0.051172         N/P    ASYNCH CF REQUESTS        0.063880        45
DB2 ENT/EXIT         N/A          98         N/A    TOTAL CLASS 3             0.144975       103
EN/EX-STPROC         N/A           0         N/A
EN/EX-UDF            N/A           0         N/A
DCAPT.DESCR.         N/A         N/A         N/P
LOG EXTRACT.         N/A         N/A         N/P

GLOBAL       CONTENTION       L-LOCKS  ELAPSED TIME    EVENTS    GLOBAL       CONTENTION       P-LOCKS  ELAPSED TIME    EVENTS
-----------------------------------  ------------  --------    -----------------------------------  ------------  --------
L-LOCKS                                  0.002139        11    P-LOCKS                                  0.000000         0
 PARENT (DB,TS,TAB,PART)                 0.001530        10     PAGESET/PARTITION                       0.000000         0
 CHILD  (PAGE,ROW)                       0.000000         0     PAGE                                    0.000000         0
 OTHER                                   0.000609         1     OTHER                                   0.000000         0

SQL DML     TOTAL    SQL DCL       TOTAL    SQL DDL    CREATE   DROP  ALTER    LOCKING                TOTAL    DATA SHARING    TOTAL
-------- --------    ---------- --------    ---------- ------ ------ ------    -------------------- --------    ------------ --------
SELECT          0    LOCK TABLE        0    TABLE           0      0      0    TIMEOUTS                    0    L-LOCKS  (%)       33
INSERT          0    GRANT             0    CRT TTABLE      0    N/A    N/A    DEADLOCKS                   0    P-LOCK REQ        42
UPDATE          0    REVOKE            0    DCL TTABLE      0    N/A    N/A    ESCAL.(SHAR)                0    P-UNLOCK REQ      42
DELETE          0    SET SQLID         0    AUX TABLE       0    N/A    N/A    ESCAL.(EXCL)                0    P-CHANGE REQ       0
                     SET H.VAR.        0    INDEX           0      0      0    MAX PG/ROW LCK HELD         2    LOCK - XES        57
DESCRIBE        0    SET DEGREE        0    TABLESPACE      0      0      0    LOCK REQUEST               45    UNLOCK-XES        46
DESC.TBL        0    SET RULES         0    DATABASE        0      0      0    UNLOCK REQST               26    CHANGE-XES         0
PREPARE         1    SET PATH          0    STOGROUP        0      0      0    QUERY REQST                 0    SUSP - IRLM        0
OPEN            1    SET PREC.         0    SYNONYM         0      0    N/A    CHANGE REQST                3    SUSP - XES         0
FETCH          43    CONNECT 1         0    VIEW            0      0    N/A    OTHER REQST                 0    SUSP - FALSE     N/A
CLOSE           1    CONNECT 2         0    ALIAS           0      0    N/A    LOCK SUSPENS.               0    INCOMP.LOCK        0
                     SET CONNEC        0    PACKAGE       N/A      0    N/A    IRLM LATCH SUSPENS.         0    NOTIFY SENT        0
                     RELEASE           0    PROCEDURE       0      0      0    OTHER SUSPENS.              0
DML-ALL        46    CALL              0    FUNCTION        0      0      0    TOTAL SUSPENS.              0
                     ASSOC LOC.        0    TRIGGER         0      0    N/A
                     ALLOC CUR.        0    DIST TYPE       0      0    N/A
                     HOLD LOC.         0    SEQUENCE        0      0      0
                     FREE LOC.         0
                     DCL-ALL           0    TOTAL           0      0      0
                                            RENAME TBL      0
                                            COMMENT ON      0
                                            LABEL ON        0
```

The accounting report for SELECT with CACHE 50 is shown in Example B-5.

Example: B-5 Accounting report for SELECT with CACHE 50

```
ACCT TSTAMP: 07/23/05 00:22:42.09    PLANNAME: DSNESPCS        WLM SCL: 'BLANK'         CICS NET: N/A
 BEGIN TIME : 07/23/05 00:22:42.02    PROD ID : N/P                                       CICS LUN: N/A
 END TIME   : 07/23/05 00:22:42.09    PROD VER: N/P             LUW NET: USIBMSC          CICS INS: N/A
 REQUESTER  : DB8F                    CORRNAME: PAOLOR2         LUW LUN: SCPD8F1
 MAINPACK   : DSNESM68                CORRNMBR: 'BLANK'         LUW INS: BD59676FEAC2     ENDUSER : 'BLANK'
 PRIMAUTH   : PAOLOR2                 CONNTYPE: TSO             LUW SEQ:            1     TRANSACT: 'BLANK'
 ORIGAUTH   : PAOLOR2                 CONNECT : TSO                                       WSNAME  : 'BLANK'

ELAPSED TIME DISTRIBUTION                                        CLASS 2 TIME DISTRIBUTION
 ----------------------------------------------------------------  ----------------------------------------------------------------
 APPL   |=========================================> 90%          CPU    |============> 27%
 DB2    |==> 4%                                                   NOTACC |========> 17%
 SUSP   |===> 6%                                                  SUSP   |============================> 56%

 TIMES/EVENTS  APPL(CL.1)  DB2 (CL.2)  IFI (CL.5)    CLASS 3 SUSPENSIONS   ELAPSED TIME    EVENTS    HIGHLIGHTS
 ------------  ----------  ----------  ----------    --------------------  ------------  --------    --------------------------
 ELAPSED TIME    0.066473    0.006704         N/P    LOCK/LATCH(DB2+IRLM)      0.000000         0    THREAD TYPE   : ALLIED
  NONNESTED      0.066473    0.006704         N/A    SYNCHRON. I/O             0.000000         0    TERM.CONDITION: NORMAL
  STORED PROC    0.000000    0.000000         N/A     DATABASE I/O             0.000000         0    INVOKE REASON : DEALLOC
  UDF            0.000000    0.000000         N/A     LOG WRITE I/O            0.000000         0    COMMITS       :          2
  TRIGGER        0.000000    0.000000         N/A    OTHER READ I/O            0.000000         0    ROLLBACK      :          0
                                                      OTHER WRTE I/O            0.000000         0    SVPT REQUESTS :          0
```

CPU TIME	**0.011425**	**0.001805**	N/P	SER.TASK SWTCH	**0.000035**	1	SVPT RELEASE :	0
AGENT	0.011425	0.001805	N/A	UPDATE COMMIT	0.000035	1	SVPT ROLLBACK :	0
NONNESTED	0.011425	0.001805	N/P	OPEN/CLOSE	0.000000	0	INCREM.BINDS :	0
STORED PRC	0.000000	0.000000	N/A	SYSLGRNG REC	0.000000	0	UPDATE/COMMIT :	0.00
UDF	0.000000	0.000000	N/A	EXT/DEL/DEF	0.000000	0	SYNCH I/O AVG.:	N/C
TRIGGER	0.000000	0.000000	N/A	OTHER SERVICE	0.000000	0	PROGRAMS :	1
PAR.TASKS	0.000000	0.000000	N/A	ARC.LOG(QUIES)	0.000000	0	MAX CASCADE :	0
				ARC.LOG READ	0.000000	0	PARALLELISM : NO	
SUSPEND TIME	**0.000000**	**0.003747**	N/A	DRAIN LOCK	0.001463	2		
AGENT	N/A	0.003747	N/A	CLAIM RELEASE	0.000000	0		
PAR.TASKS	N/A	0.000000	N/A	PAGE LATCH	0.000000	0		
STORED PROC	0.000000	N/A	N/A	NOTIFY MSGS	0.000000	0		
UDF	0.000000	N/A	N/A	GLOBAL CONTENTION	**0.002249**	5		
				COMMIT PH1 WRITE I/O	0.000000	0		
NOT ACCOUNT.	N/A	0.001152	N/P	ASYNCH CF REQUESTS	**0.000000**	0		
DB2 ENT/EXIT	N/A	98	N/A	TOTAL CLASS 3	0.003747	8		
EN/EX-STPROC	N/A	0	N/A					
EN/EX-UDF	N/A	0	N/A					
DCAPT.DESCR.	N/A	N/A	N/P					
LOG EXTRACT.	N/A	N/A	N/P					

GLOBAL CONTENTION	L-LOCKS	ELAPSED TIME	EVENTS	GLOBAL CONTENTION	P-LOCKS	ELAPSED TIME	EVENTS
L-LOCKS		0.002249	5	P-LOCKS		0.000000	0
PARENT (DB,TS,TAB,PART)		0.001858	4	PAGESET/PARTITION		0.000000	0
CHILD (PAGE,ROW)		0.000000	0	PAGE		0.000000	0
OTHER		0.000392	1	OTHER		0.000000	0

SQL DML	TOTAL	SQL DCL	TOTAL	SQL DDL	CREATE	DROP	ALTER	LOCKING	TOTAL	DATA SHARING	TOTAL
SELECT	0	LOCK TABLE	0	TABLE	0	0	0	TIMEOUTS	0	L-LOCKS (%)	30
INSERT	0	GRANT	0	CRT TTABLE	0	N/A	N/A	DEADLOCKS	0	**P-LOCK REQ**	**0**
UPDATE	0	REVOKE	0	DCL TTABLE	0	N/A	N/A	ESCAL.(SHAR)	0	P-UNLOCK REQ	0
DELETE	0	SET SQLID	0	AUX TABLE	0	N/A	N/A	ESCAL.(EXCL)	0	P-CHANGE REQ	0
		SET H.VAR.	0	INDEX	0	0	0	MAX PG/ROW LCK HELD	2	LOCK - XES	7
DESCRIBE	0	SET DEGREE	0	TABLESPACE	0	0	0	LOCK REQUEST	23	UNLOCK-XES	2
DESC.TBL	0	SET RULES	0	DATABASE	0	0	0	UNLOCK REQST	19	CHANGE-XES	0
PREPARE	1	SET PATH	0	STOGROUP	0	0	0	QUERY REQST	0	SUSP - IRLM	0
OPEN	1	SET PREC.	0	SYNONYM	0	0	N/A	CHANGE REQST	0	SUSP - XES	0
FETCH	43	CONNECT 1	0	VIEW	0	0	N/A	OTHER REQST	0	SUSP - FALSE	N/A
CLOSE	1	CONNECT 2	0	ALIAS	0	0	N/A	LOCK SUSPENS.	0	INCOMP.LOCK	0
		SET CONNEC	0	PACKAGE	N/A	0	N/A	IRLM LATCH SUSPENS.	0	NOTIFY SENT	0
		RELEASE	0	PROCEDURE	0	0	0	OTHER SUSPENS.	0		
DML-ALL	46	CALL	0	FUNCTION	0	0	0	TOTAL SUSPENS.	0		
		ASSOC LOC.	0	TRIGGER	0	0	N/A				
		ALLOC CUR.	0	DIST TYPE	0	0	N/A				
		HOLD LOC.	0	SEQUENCE	0	0	0				
		FREE LOC.	0								
		DCL-ALL	0	TOTAL	0	0	0				
				RENAME TBL	0						
				COMMENT ON	0						
				LABEL ON	0						

Archived

Archived

C

Additional material

This redbook refers to additional material that you can download from the Internet as described below.

Locating the Web material

The Web material associated with this redbook is available in softcopy on the Internet from the IBM Redbooks Web server. Point your Web browser to:

ftp://www.redbooks.ibm.com/redbooks/SG247134

Alternatively, you can go to the IBM Redbooks Web site at:

ibm.com/redbooks

Select the **Additional materials** and open the directory that corresponds with the redbook form number, SG24-7134.

Using the Web material

The additional Web material that accompanies this redbook includes the following file:

File name	*Description*
QuickPhyssizerV1R1.zip	Microsoft® Excel® Spreadsheet

System requirements for downloading the Web material

The following system configuration is recommended:

Hard disk space: 1 MB minimum
Operating System: Windows

© Copyright IBM Corp. 2005. All rights reserved.

How to use the Web material

Create a subdirectory (folder) on your workstation, and unzip the contents of the Web material zip file into this folder.

Archived

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

IBM Redbooks

For information on ordering these publications, see “How to get IBM Redbooks” on page 442. Note that some of the documents referenced here may be available in softcopy only.

- *Achieving the Highest Levels of Parallel Sysplex Availability,* SG24-6061
- *DB2 - Quick Upper Bound Estimate An Application Design Methodology*, SG24-2549
- *DB2 for MVS/ESA Version 4 Data Sharing Performance Topics,* SG24-4611
- *DB2 for MVS/ESA Version 4 Data Sharing Implementation,* SG24-4791
- *DB2 for OS/390 and Data Compression*, SG24-5261
- *DB2 for OS/390 Application Design Guidelines for High Performance,* SG24-2233
- *DB2 for OS/390 Version 5 Performance Topics,* SG24-2213
- *DB2 for z/OS and OS/390 Version 7 Selected Performance Topics,* SG24-6894
- *DB2 for z/OS and OS/390: Ready for Java*, SG24-6435
- *DB2 for z/OS and OS/390 Version 7 Performance Topics,* SG24-6129
- *DB2 for z/OS Application Programming Topics*, SG24-6300
- *DB2 for z/OS Stored Procedures: Through the CALL and Beyond,* SG24-7083
- *DB2 UDB for z/OS Version 8: Everything You Ever Wanted to Know, ... and More,* SG24-6079
- *DB2 UDB for z/OS Version 8 Performance Topics,* SG24-6465
- *DB2 UDB Server for OS/390 Version 6 Technical Update*, SG24-6108
- *Distributed Functions of DB2 for z/OS and OS/390*, SG24-6952
- *Large Objects with DB2 for z/OS and OS/390*, SG24-6571
- *Locking in DB2 for MVS/ESA Environment*, SG24-4725
- *WebSphere Information Integrator Q Replication: Fast Track Implementation Scenarios*, SG24-6487
- *z/OS Multilevel Security and DB2 Row-level Security Revealed,* SG24-6480

Other publications

These publications are also relevant as further information sources:

- *DB2 UDB for z/OS Version 8 Administration Guide,* SC18-7413-02
- *DB2 UDB for z/OS Version 8 Application Programming and SQL Guide,* SC18-7415-02
- *DB2 UDB for z/OS Version 8 Application Programming Guide and Reference for Java,* SC18-7414-02

© Copyright IBM Corp. 2005. All rights reserved.

- *DB2 UDB for z/OS Version 8 Command Reference,* SC18-7416-02
- *DB2 UDB for z/OS Version 8 Data Sharing: Planning and Administration,* SC18-7417-02
- *DB2 UDB for z/OS Version 8 Installation Guide,* GC18-7418-03
- *DB2 UDB for z/OS Version 8 Messages and Codes,* GC18-7422-02
- *DB2 UDB for z/OS Version 8 SQL Reference,* SC18-7426-02
- *DB2 UDB for z/OS Version 8 Utility Guide and Reference,* SC18-7427-02
- *z/OS V1R6 Language Environment Debugging Guide*, GA22-7560-05
- *z/OS V1R6 Language Environment Customization Guide,* SA22-7564-06
- *DB2 UDB for z/OS Version 8 Diagnostics Guide and Reference*, LY37-3201-02

  This is licensed material of IBM
- *CICS Transaction Server for z/OS V3.1: Application Programming Guide*, SC34-6433
- *Large Systems Performance Reference*, SC28-1187

Online resources

These Web sites are also relevant as further information sources:

- DB2 UDB for z/OS Version 8

  http://www.ibm.com/software/data/db2/zos/db2zosv8.html
- The DB2 Information Management Software Information Center

  http://publib.boulder.ibm.com/infocenter/dzichelp/index.jsp
- CICS family Web site

  http://www.ibm.com/software/htp/cics/support

How to get IBM Redbooks

You can search for, view, or download Redbooks, Redpapers, and Tips, draft publications and Additional materials, as well as order hardcopy Redbooks or CD-ROMs, at this Web site:

ibm.com/redbooks

Help from IBM

IBM Support and downloads

ibm.com/support

IBM Global Services

ibm.com/services

Archived

Abbreviations and acronyms

AC	autonomic computing
ACS	automatic class selection
AIX®	Advanced Interactive Executive from IBM
APAR	authorized program analysis report
API	application programming interface
AR	application requester
ARM	automatic restart manager
AS	application server
ASCII	American National Standard Code for Information Interchange
B2B	business-to-business
BCDS	DFSMShsm™ backup control data set
BCRS	business continuity recovery services
BI	Business Intelligence
BLOB	binary large objects
BPA	buffer pool analysis
BSDS	boot strap data set
CBU	Capacity BackUp
CCA	channel connection address
CCA	client configuration assistant
CCP	collect CPU parallel
CCSID	coded character set identifier
CD	compact disk
CDW	central data warehouse
CEC	central electronics complex
CF	coupling facility
CFCC	coupling facility control code
CFRM	coupling facility resource management
CICS	Customer Information Control System
CLI	call level interface
CLOB	character large object
CLP	command line processor
CMOS	complementary metal oxide semiconductor
CP	central processor
CPU	central processing unit
CRCR	conditional restart control record
CRD	collect report data
CRUD	create, retrieve, update, or delete
CSA	common storage area
CSF	Integrated Cryptographic Service Facility
CTE	common table expression
CTT	created temporary table
CUoD	Capacity Upgrade on Demand
DAC	discretionary access control
DASD	direct access storage device
DB	database
DB2	Database 2™
DB2 PE	DB2 Performance Expert
DBA	database administrator
DBAT	database access thread
DBCLOB	double-byte character large object
DBCS	double-byte character set
DBD	database descriptor
DBID	database identifier
DBM1	database master address space
DBRM	database request module
DCL	data control language
DDCS	distributed database connection services
DDF	distributed data facility
DDL	data definition language
DDL	data definition language
DES	Data Encryption Standard
DLL	dynamic load library manipulation language
DML	data manipulation language
DNS	domain name server
DPSI	data partitioning secondary index
DRDA	Distributed Relational Data Architecture
DSC	dynamic statement cache, local or global
DSNZPARMs	DB2's system configuration parameters
DSS	decision support systems
DTT	declared temporary tables

Archived

© Copyright IBM Corp. 2005. All rights reserved.

DWDM	dense wavelength division multiplexer
DWT	deferred write threshold
EA	extended addressability
EAI	enterprise application integration
EAS	Enterprise Application Solution
EBCDIC	extended binary coded decimal interchange code
ECS	enhanced catalog sharing
ECSA	extended common storage area
EDM	environmental descriptor manager
EJB™	Enterprise JavaBean
ELB	extended long busy
ENFM	enable-new-function mode
ERP	enterprise resource planning
ERP	error recovery procedure
ESA	Enterprise Systems Architecture
ESP	Enterprise Solution Package
ESS	Enterprise Storage Server®
ETR	external throughput rate, an elapsed time measure, focuses on system capacity
EWLC	Entry Workload License Charges
EWLM	Enterprise Workload Manager
FIFO	first in first out
FLA	fast log apply
FTD	functional track directory
FTP	File Transfer Program
GB	gigabyte (1,073,741,824 bytes)
GBP	group buffer pool
GDPS®	Geographically Dispersed Parallel Sysplex™
GLBA	Gramm-Leach-Bliley Act of 1999
GRS	global resource serialization
GUI	graphical user interface
HALDB	High Availability Large Databases
HPJ	high performance Java
HTTP	Hypertext Transfer Protocol
HW	hardware
I/O	input/output
IBM	International Business Machines Corporation
ICF	internal coupling facility
ICF	integrated catalog facility
ICMF	integrated coupling migration facility
ICSF	Integrated Cryptographic Service Facility
IDE	integrated development environments
IFCID	instrumentation facility component identifier
IFI	Instrumentation Facility Interface
IFL	Integrated Facility for Linux
IGS	IBM Global Services
IMS	Information Management System
IORP	I/O Request Priority
IPLA	IBM Program Licence Agreement
IRD	Intelligent Resource Director
IRLM	internal resource lock manager
IRWW	IBM Relational Warehouse Workload
ISPF	interactive system productivity facility
ISV	independent software vendor
IT	information technology
ITR	internal throughput rate, a processor time measure, focuses on processor capacity
ITSO	International Technical Support Organization
IVP	installation verification process
J2EE™	Java 2 Enterprise Edition
JDBC	Java Database Connectivity
JFS	journaled file systems
JNDI	Java Naming and Directory Interface
JTA	Java Transaction API
JTS	Java Transaction Service
JVM	Java Virtual Machine
KB	kilobyte (1,024 bytes)
LCU	Logical Control Unit
LDAP	Lightweight Directory Access Protocol
LOB	large object
LPAR	logical partition
LPL	logical page list
LRECL	logical record length
LRSN	log record sequence number
LRU	least recently used
LSS	logical subsystem
LUW	logical unit of work

Archived

LVM	logical volume manager
MAC	mandatory access control
MB	megabyte (1,048,576 bytes)
MBps	megabytes per second
MLS	multi-level security
MQT	materialized query table
MTBF	mean time between failures
MVS	Multiple Virtual Storage
NALC	New Application License Charge
NFM	new-function mode
NFS	Network File System
NPI	nonpartitioning index
NPSI	nonpartitioned secondary index
NVS	non volatile storage
ODB	object descriptor in DBD
ODBC	Open Database Connectivity
ODS	Operational Data Store
OLE	Object Link Embedded
OLTP	online transaction processing
OP	Online performance
OS/390	Operating System/390®
OSC	optimizer service center
PAV	parallel access volume
PCICA	Peripheral Component Interface Cryptographic Accelerator
PCICC	PCI Cryptographic Coprocessor
PDS	partitioned data set
PIB	parallel index build
PPRC	Peer-to-Peer Remote Copy
PR/SM™	Processor Resource/System Manager
PSID	pageset identifier
PSP	preventive service planning
PTF	program temporary fix
PUNC	possibly uncommitted
PWH	Performance Warehouse
QA	Quality Assurance
QMF™	Query Management Facility
QoS	Quality of Service
QPP	Quality Partnership Program
RACF®	Resource Access Control Facility
RAS	reliability, availability, and serviceability
RBA	relative byte address
RBLP	recovery base log point
RDBMS	relational database management system
RDS	relational data system
RECFM	record format
RI	Referential Integrity
RID	record identifier
ROI	return on investment
RPO	recovery point objective
RR	repeatable read
RRS	resource recovery services
RRSAF	resource recovery services attach facility
RS	read stability
RTO	recovery time objective
RTS	real-time statistics
SAN	storage area networks
SBCS	store single byte character set
SCUBA	self-contained underwater breathing apparatus
SDM	System Data Mover
SDP	Software Development Platform
SLA	service-level agreement
SMIT	System Management Interface Tool
SOA	service-oriented architecture
SOAP	Simple Object Access Protocol
SPL	selective partition locking
SQL	Structured Query Language
SQLJ	Structured Query Language for Java
SRM	Service Request Manager
SSL	Secure Sockets Layer
SU	Service Unit
TCO	total cost of ownership
TPF	Transaction Processing Facility
UA	Unit Addresses
UCB	Unit Control Block
UDB	Universal Database
UDF	user-defined functions
UDT	user-defined data types
UOW	unit of work
UR	unit of recovery
USS	UNIX System Services
vCF	virtual coupling facility
VIPA	Virtual IP Addressing
VLDB	very large database

Archived

VM	virtual machine
VSE	Virtual Storage Extended
VSIP	Visual Studio® Integrator Program
VWLC	Variable Workload License Charges
wizards	Web-based assistants
WLC	Workload License Charges
WLM	Workload Manager
WSDL	Web Services Definition Language
WTO	write to operator
XA	Extended Architecture
XML	Extensible Markup Language
XRC	Etended Remote Copy
z800	zSeries 800
z890	zSeries 890
z990	zSeries 990
zAAP	zSeries Application Assist Processor
zELC	zSeries Entry License Charge

Archived

Index

Symbols

+100 216
+222 238
+231 238
+252 268
+495 382

Numerics

0 value 14
00C90088 257
00C9008E 257
180 CIs 67
-222 238
-224 238
-225 239
-228 239
-243 239
-244 239
-249 241
-253 268
-254 268
-301 321
-305 17
32,672 255
32,767 241
32,767 rows 266
-399 140
-495 382
-803 141
-805 252, 363
-811 204
-904 256
-911 256, 344
-913 256, 345
-925 209
-926 209

A

absolute number 232
access frequency 30
access path 26, 47, 61–62, 65, 69, 71–72, 117–118, 121–123, 147–148, 157, 159, 167, 181, 192, 194, 198, 203–204, 212–213, 218–223, 228–230, 276–278, 281, 305, 307–308, 310–314, 318, 362, 364, 380, 388, 392, 394–395, 423
    efficiency 77
    matching columns 220
    significant difference 364
ACCESSTYPE 67, 70
accounting record 297, 299, 363, 386
accounting report 161, 174, 379, 394, 416–417
accounting report example 389
accounting trace
    class 1 387
    class 2 387
    class 3 387
    class 7 387
    class 8 387
ACQUIRE 327
ACQUIRE(ALLOCATE) 345
ACQUIRE(USE) 345
affinity 93, 100–101, 404, 407–408, 412
    isolating a set of tables 412
    isolating at partition level 412
aggregation 27
ALL 255
ambiguous cursor 290
anagram 87
AND 0 =
    H 228
AND NOT 224
application code 110, 153, 242, 290, 307, 310, 334
application development client 156
application life cycle 359
application logic 146, 153, 158, 162, 192, 204, 256, 258, 268–269, 280, 282–283, 361
application messaging interface 163
application parallelism 365
application process
    elapsed time 110
    passing data 210
    peak volume 1, 110
application program 131, 162, 166–167, 189, 192, 205–207, 210, 239, 255, 259, 272, 283, 285, 290, 293, 301, 310, 363–364, 378, 396
application retry 367
architecture 186, 362
ASENSITIVE 234, 236
Asynchronous messaging 168
ATOMIC 267–269
attachment 349
attributive table 38
AUTHCACH 303
authorization 179, 229, 293, 299, 303–304, 318
authorization checking 229
AUTOCOMMIT 301
automatic query rewrite 133
automatic rebind 362
auxiliary table 60
AVG 2, 75

B

B37 285
BACKUP 271, 277, 378, 412
backward browsing 226
backward index scan 62, 70, 118, 194, 239
balanced tree 60

Archived

© Copyright IBM Corp. 2005. All rights reserved.

base table 13, 33, 60, 134, 233–234, 237, 269, 332, 336, 381, 423
row 237, 332
batch and online processing 287
batch application 134, 282–283, 348, 405
batch job 291
batch processing 215, 279, 425
batch program 148, 158, 282, 348, 412, 414
batch window 282, 287, 365
before trigger 212
BIND 157, 193, 235, 260, 279–280, 290, 292–294, 296, 299, 303, 305–306, 309–310, 314, 318, 336, 340, 349, 361, 363, 376, 382, 394, 410, 413–414, 423–426, 429
bind 157, 193, 290, 292–294, 296, 310–311, 314, 345, 363, 376, 382, 413–414, 424
BIND parameter
ACQUIRE 345
RELEASE 347
bind parameters 327
BIND PLAN 262
bind time 293, 424
BOTH 314
browsing problem 217
BSAM files 284
buffer pool 42, 55, 57, 107, 111, 114–115, 119, 135, 232, 265, 278, 280–281, 310, 392, 397, 412–414, 423–424
contention 396
hit ratio 58, 111, 392, 397
buffer pool efficiency 385
buffer pool thresholds 426
buffer pools xxvi, 9, 91, 159, 408
BUILD2 99, 372
business intelligence 111
business rule 19, 132
byte counts 57

C

CACHE xxv, 141, 143, 281, 303, 306, 361, 415–417, 426, 429
CACHE 50 report 436
CACHE size 418
CACHESIZE 303
caching 353
caching SQL results 281
CAF 256, 282, 303, 350
candidate key 19–20
CARDINALITY 1, 23, 72, 81, 85, 136, 153, 167, 224–225, 314
cardinality 76, 136, 314
CASCADE 131
CASE expression 165, 275
CAST function 201
casting 197, 201
catalog statistics 311
catalog table 234, 314, 361
CATEGORY 162, 382, 390
CCSID 23, 294
CF 291, 380, 404–409, 413–414, 416–417
CF lock structure 408
Check Constraint 116, 135
check constraint 116, 134
check pending 131, 135–136
checkpoint and restart procedure 365
CICS 75, 136–137, 148–149, 154, 158, 162, 209, 239, 247, 251–252, 257, 292, 298–299, 303, 309, 350, 386, 389–390, 411
common work area 149
quasi-reentrant task control block 309
ROLBE=YES 257
sequential numbers 149
TCB switching 309
CICS thread reuse 386
claim classes 333
claims and drains 329
class 3 trace 384
CLOB 87, 164, 166, 178–179, 185–186
CLOB variable 164, 166
CLOSE NO 391
CLSN 410
CLUSTER 62, 68, 71, 76, 88, 106, 122
cluster 369
Clustering 19, 31, 45, 62–63, 73–76, 91–92, 95–98, 105, 107, 113–114, 119, 122, 209, 264–265, 270, 272–273, 276, 279, 283, 313, 334, 354, 367, 372, 399, 411, 428
clustering
impact on other indexes 95
impact on synchronous I/O 95
Clustering index 63, 74–76, 97, 106, 119, 264–265, 271, 273, 279, 354, 399
adequate free space 399
clustering index 88, 355, 367
changing 97
choice 90
clustering process 344
clustering sequence 90, 107, 209, 279, 369
clusterratio 63, 71, 89, 118, 423–424
CMTSTAT 300, 352, 386
code table 35, 132, 134–136
code tables 136
COLGROUP 74, 313
collating sequence 70, 98, 199
column correlation 73
column placement 9
performance 18
column value 224
COMMIT xxiv–xxv, 62, 114, 131, 146, 148, 155, 157, 159, 163, 192, 205, 207–209, 211, 213, 256, 258, 264, 268–270, 272–274, 276–277, 279, 281, 283, 286–287, 291–292, 297, 300, 316, 320, 336, 339, 344, 346–347, 350–353, 357–358, 361, 366, 374, 378, 384, 386, 390, 396, 408, 410–411, 413–414, 417, 426, 429
commit frequency 217, 328, 341–342, 348, 350
recommendation 350
variable 350
committed log record sequence number 410
common system area 149
compression 51, 112
compression dictionary 51
Concurrency 280
CONCURRENCY(THREADSAFE) 309

concurrent updates 245
CONDITION 255
conflict report 313
CONNECT xxiv, 253, 298, 301, 386
connection 157, 168, 255, 300, 303, 318, 320–321, 352, 384, 386
Consecutive numbers 280
constant value 204, 206–207
constraint 4, 129, 133, 273, 361
CONTOKEN 259
control table 258–260, 397
Coprocessor 207
COPY 192, 272, 308, 311, 359, 376–378, 392, 413
COUNT 75, 120, 131, 165, 192, 314
COUNT m 314
COUNT(*) 232
COUNTER 149
counters recovery 149
coupling facility 149, 404, 413, 418
Coupling Facility Control Code 406
CPU consumption 110, 113–116, 118, 160, 171, 175, 193, 217, 223, 228, 272, 283, 298, 307–308, 313–314, 374, 380, 388, 397
  significant difference 171
CPU contention 365
CPU cost 5, 8, 63, 111, 114, 117, 170–174, 291, 296, 298, 309, 313, 406
CPU time 113, 120, 192, 196, 212, 264, 268, 273, 292, 298, 310, 390, 405, 416–417, 421, 428
CPU usage 146, 266
CREATE TABLE 98, 139–140, 211, 361, 368
create, retrieve, update or delete 1
create, retrieve, update, and delete 75
created temporary tables 353
cross invalidation 415
cross-invalidation 408
CRUD 1, 75
CS 333
CS isolation 410
CSA 149
CTT 159, 211–212
current date 312
CURRENT PACKAGESET 261, 298, 376, 386
CURRENT TIMESTAMP 117, 141
CURRENTDATA 290, 301–302, 311, 327–328, 333, 336, 339–341, 346, 410
CURRENTDATA(NO) 339
CURRENTDATA(YES) 339
cursor repositionin 286
cursor stability 290, 339, 346
CURSOR WITH HOLD 209, 211, 286, 316, 351
CWA 149
CYCLE 81, 110, 120, 138–139, 142, 282, 315, 341, 375, 378

D

DAD file 180
data access 1, 61, 115, 118, 192, 211, 260–261, 276, 278–279, 350, 362, 369, 423, 425
  data rows 118
  improving performance 61
  unusual situations 276
Data capture 396
data clustering sequence 355
data consistency xxiv
data inconsistency 337
data integrity 24, 61, 63, 131–132, 168, 218, 245, 256, 269, 316, 329
data page 61–62, 68, 115, 118, 146, 193, 265, 337–338, 354, 357, 426, 428
Data Propagator 302, 396
data row 119–120, 136, 148, 426
data set xxiv, 265, 270, 274, 285, 397, 405
data set extend/delete/define 391
data sharing 58, 62–63, 93, 100, 114, 119, 139, 141, 143, 147, 149, 250, 280, 291–292, 319, 329, 334, 341, 359, 388, 391–393, 397, 401, 403–408, 413, 418
  architecture 359
data sharing group 404, 408–409, 413
data table 35, 135
  single insert 135
data type 4, 6, 57, 59, 139, 141, 162, 179, 186, 197–198, 255, 276, 293, 319, 321
data types
  bytes allocation 5
  Java 6
  programming language 6
  relative cost 5
database access thread 157, 390
database design xxv, 1–2, 110, 196, 359, 403
database engine 152, 179, 266
database modification 361
DATACAPTURE(CHANGES) 18
data-partitioned secondary indexes 100
DATE 15, 132, 198
date component 132
DB2 attachments 350
DB2 call 156, 158, 191, 297
DB2 Connect 318
DB2 engine 116, 137, 142, 185–186, 320
DB2 inactive connection support 386
DB2 index manager component 65–66
DB2 member 410, 415
DB2 repositioning 219–220, 222
DB2 subsystem 158, 192, 239, 315, 329, 341, 405, 408
  level 327
DB2 system wide parameter 366
DB2 table 60, 87, 111, 155, 162, 178–179, 264, 283, 353, 374
DB2 tools 311
DBAT 157, 212, 300, 318, 352, 386
DBCLOB 87, 178–179
DBD 89, 157, 308, 361
DBD01 361
DBRM 208, 229, 345, 363, 377, 387
DCLGEN 241
DD statement
  SYSDISC 274
  SYSREC00 274
DDF 157, 300, 318, 386, 390

DDL 169, 171, 212, 246, 292, 310, 361, 368, 375, 399, 411–412
deadlock 334
deadlock avoidance 342
deadlock information 398
deadlock situation 145, 258, 341
deadlocks 257, 340–341
DEADLOK 315, 341
DECLARE 235, 241
DECLARE CURSOR 154, 235, 241, 301, 350
DECLARE GLOBAL TEMPORARY TABLE 211–212
Declared temporary table 212
declared temporary table 157, 233
decompose 181
default value 15, 64, 131, 141, 290, 293–294, 297, 301, 304, 316
degree of parallelism 123
DELETE 372
  cost 119
delete 264
DELETE HOLE 238
delete hole 237
delete rule 130–131
delete trigger 273
DELETE using cursor 217
Denormalization 132, 136
denormalization 34
denormalization cost 41
dependent row 132–133
  foreign key 131
dependent table 22, 63, 107, 119, 130–135, 168, 344
  check constraint 134
  foreign key 22, 119, 130
  foreign key columns 134
  foreign key definition 133
  INSERT statements 130
  UPDATE statements 130
DEPT table 61
derived column 25
DESCRIBE 308
design option 114, 218, 221
detailed accounting report 388
DFSORT 65, 285
direct row access 139–140, 148, 276, 395, 423–424
dirty data 339
DISALLOW PARALLEL 167
disk space 5, 8–9, 25, 52, 82–83, 85, 110, 391, 439
DISTINCT 69
distributed environment
  COMMIT frequency 301
  thread reuse 300
Distribution statistics 273, 276, 314
document updates to catalog statistics 314
domain 4
domain verification 4
DPSI 93, 99, 101–104, 107, 123, 132
  disadvantages 101
drain 329, 333, 384, 396, 411, 416–417
DRAIN ALL 333
DRDA 153, 156, 290, 292–293, 295, 300–302, 318, 322
DRDA blocking 241
DRDA query block 302
DROP TABLE 157
DSC 305–306, 308, 396
DSMAX 45, 94, 391
DSN1COMP 51
DSN8810.EMP 416
DSNDB07 65
DSNI031I 383
DSNR035I 384
DSNTIAR 251–252, 259
DSNZPARM 24, 34, 46, 155, 294, 300, 303–304, 308, 310, 316, 335, 340–341, 346, 350, 383–384, 386, 396, 413–414
  CMTSTAT 386
  DEADLOK 315
  DSMAX 94
  EVALUNC 316
  IMMEDWRI 413–414
  IRLMRWT 315, 350
  NPGTHRSH 24
  NUMLKTS 315
  NUMLKUS 315, 335
  RELCURHL 316, 351
  RRULOCK 316
  SKIPUNCI 316
  URCHKTH 384
  URLGWTH 385
  UTIMOUT 315, 350
  XLKUPDLT 317
DSNZPARMs 327
DSSIZE 46, 50
DTT 154, 157, 211–212, 237
dummy value 15
duplicate output 284
duration lock 392
dynamic prefetch 344
dynamic scrollable cursor 234
Dynamic scrollable cursors 234–235, 239
dynamic SQL 202, 211, 227, 230–231, 269, 280, 291–294, 305–308, 312, 317, 321, 341, 361, 382, 395–396
  access path 230
dynamic SQL with parameter markers 306
Dynamic statement cache 305, 308, 396
dynamic statement cache 306, 308, 361

E

EDITPROC 55, 57, 143
EDM pool 229, 291–292, 297, 411
Elapsed time 388
Employee Table 416
employee table
  MQSEND function 164
  simple SELECT 416
ENABLE QUERY OPTIMIZATION 34
ENCODING 3, 7, 23, 26, 294, 298, 319
Encoding scheme 294, 319
end user response time 380
ENFORCED 18, 130–132, 134–135

EQUAL predicate 121, 200
Error handling 280
error handling control table 258
error SQLCODE 256
ESTAE 285
EVALUNC 316
EXEC SQL 191, 306, 369
EXECUTE privilege 304
EXISTS 202, 293, 295, 300, 395, 413, 422
EXPLAIN 67, 72, 157, 202, 226, 229, 287, 310, 313, 380–381
Explain 157, 287, 380
expression 163, 234, 275
extended address 111
extended format 94
external 140, 157, 163, 166, 180, 185, 271, 283, 406, 421, 429
    function 140, 162, 167
external sort 367
extract 65, 87, 112, 178–181, 183, 416–417

F

False contention 406
fast column processing 362
FCID
    22 395
FETCH 71, 111, 114, 116–117, 159, 165, 204, 213, 234–236, 253, 269, 277, 290, 302, 309, 316, 343, 357, 393, 416–417, 425
FETCH FIRST 1 ROW ONLY 204
FETCH FIRST n ROWS 204, 232
FETCH keywords 236
FETCH orientation 236
FETCH SENSITIVE 238
FETCH syntax 241
FIELDPROC 23
filter factor 69, 71, 229–230, 312
FINAL CALL 167
FIRST n ROWS ONLY 204
first subquery 202
FOR BIT DATA 139
FOR FETCH ONLY 195, 327, 341
FOR n ROWS 241, 269
FOR READ ONLY 235
FOR UPDATE 235
FOR UPDATE OF 195, 205, 208, 217, 235, 244, 273, 278, 286, 327, 339, 341–343, 357, 410, 424
forced write 352
foreign key 7, 14, 35, 38, 63, 74, 90, 107, 119, 130–132, 135, 361
    definition 130
    non-null values 130
    nullable column 14, 131
Forward key 220, 231
FREE PACKAGE 363
Free space 9, 64, 92–93, 107, 112, 265, 272, 334, 370, 399, 422
    initial percentage 64, 399
    table space partitions 106
free space 64, 121
FREEPAGE 399
FREQVAL 313

G

GBP-dependency 407, 413
GCLSN 410
GDG 285
GENERATE_UNIQUE 138–141, 148
GENERATED ALWAYS 140–143, 145
GENERATED BY DEFAULT 140, 142, 144
GET DIAGNOSTICS 241–242, 251, 253–256, 268
GET DIAGNOSTICS ALL 252
GETPAGE 388
getpage 8, 60, 68, 111, 115, 423, 426, 428
global highest committed log record sequence number 410
global locks 329, 408
global log record sequence number 410
global temporary table 211
GRANT 299, 304
Granularity 45, 58, 313, 327, 409
GRAPHIC 200, 319
group buffer pool dependent 416
GROUP BY 69, 75, 115, 162, 193, 234
Growth rate 1, 64, 400
growth rate 110
GSAM files 284

H

hash table 48
HEAP 160
High latency 375
historical data 20
hit ratio 58, 397
hole 146, 237, 358
horizontal table split 43
host variable 17, 197–199, 254–255, 268–269, 305, 307–308, 381
host variables 222, 241, 255, 270, 294, 305–306, 362, 381, 396
host-variable array 269
hot data page 354
hot pages 353
hot row 355
hot rows 353
hot spot 30, 81, 138, 168, 209, 250, 281, 334, 354, 357, 426
hot spots 353
Hybrid join 68, 79, 118, 394, 424
HyperSockets 153

I

I/O contention 365
I/O operation
    large number 283
I/O parallelism. 192
I/O time 120, 322, 392, 424
I/O wait time 388

Archived

identity column 138–139, 141–142, 144, 265, 415
GENERATED ALWAYS 142
IDENTITY_VAL_LOCAL 142–143
LOAD 144
major difference 138
MAXVALUE 138, 142
MINVALUE 138, 142
restrictions 142
when 138
identity columns 143
IFCID
125 69, 395
225 398
313 384–385
63 395
IFCID 0003 424
IFCID 0006 424
image copy 360
IMMEDWRI 413–414
IMMEDWRITE 413–414
IMS 75, 137, 148, 154, 209, 239, 247, 257, 282, 284, 297, 303, 342, 350, 389–390, 406
sequential numbers 148
virtual storage option 148
IMS Performance Analyzer 390
Inactive DBAT 157
index
duplicate entries 80
index access 60, 67, 73, 117–118, 189, 194, 197–198, 201, 251, 264, 276, 354, 358, 362, 380, 394–395, 422–424
index column 79, 81, 199
Index design 59, 62, 72–73, 75–76, 90, 110, 114
index design
join operations 79
index efficiency 72
index entry 81, 115, 117
index key 65, 91, 95, 194, 196, 265, 422, 426
index level 60, 68, 77, 93, 99, 428
index look-aside 349
index page 62, 64, 106, 141, 196, 411, 426
index scan 68, 147, 200, 224, 277–278, 381, 399, 422, 424
Index screening 62, 65, 67–68, 72, 77, 122, 196, 399, 422–423
index screening
predicate 422
index screening predicate 66
index selectivity 72
indexable 66, 189, 196–197, 199, 277
indexable predicate 65, 198, 228
index-based partitioning 61
index-only access 62, 70–71, 73, 75, 119, 193, 329, 331, 424–425
index-screening 286
indicator column 25
indicator variable 17
informational referential constraints 133
inline statistics 377
inner table 68, 74, 79, 201, 424
INSENSITIVE 233–236, 239
INSERT xxv, 45, 63–64, 71, 87, 92, 120, 130, 132–136, 139–143, 145, 148, 155, 167, 170, 172–173, 178–179, 181, 186, 207, 211, 234, 264–267, 269–271, 279, 293, 301–302, 316, 332, 336, 339, 347, 355, 360, 362–363, 368, 372–375, 380, 382, 393, 411, 413, 424–426, 429
cost 119
insert algorithm 119, 369
INSERT INTO SELECT 415
insert pattern 399
INSERT program 270
INSERT statement 143, 170, 266, 269
INSERT time 87, 141, 178, 181, 373
insert time
XML document 178
insert trigger 269
inter-DB2 R/W interest 407, 409, 413
page sets 408
Internal CF 406
International Committee for Information Technology Standards 185
intersection 68–69
IPROC 349, 429
IRLM 315–317, 327, 329, 335, 341, 348, 390, 406, 408–409, 416
IRLM time-out 366
IRLMRWT 315
IRLMRWT DSNZPARM
value 315, 350
IS NOT DISTINCT FROM 16
IS NULL 15–16, 130, 399
ISOLATION 62, 133, 193, 205, 209, 235, 237, 252, 260–261, 292, 311, 316, 327–328, 331, 336, 341, 346, 410–411, 413
isolation level 133, 235, 260, 262, 290, 293, 328, 334, 336, 346
recommended 339
UR 293
isolation repeatable read 413
ISOLATION UR 331
ISOLATION(CS) 290, 339
ISOLATION(RR) 218, 340
ISOLATION(RS) 339
ist prefetch 278

J

Java 6, 23, 154, 156, 158, 198, 317, 319, 322
join transformation 195

K

KEEP UPDATE LOCKS 207
KEEPDICTIONARY 53
KEEPDYNAMIC 300, 307, 309
key design 18
key entry 64, 111
key performance indicators 388
KEYCARD 313
KPI 388

L

large database 111
large tables 265
latch contention 265
latches 329
latency 178, 373
latency requirement 29
LE 432
LE storage 160
leaf page 81, 111, 117, 196, 423, 426–427
leaf pages 60
LEAFDIST 400
LEAST 314
LIKE 228, 381
LIST 71, 149, 198, 227, 273, 277–278, 290–292, 295–296, 299, 314, 381, 394, 423
List prefetch 118, 204, 394
list prefetch 67–69, 71, 90, 117–118, 204, 278–279, 391, 394, 424
  monitoring 69
  prime reason 118
  thresholds 69
list structure 149
literal 305
L-locks 408, 411, 416–417
LOAD 33, 63, 91, 93, 107, 130–131, 133–134, 136, 144, 155, 157, 161–162, 170, 259, 264, 270–271, 274, 279, 286, 313, 335, 360, 363, 370, 372, 377, 429
  SHRLEVEL CHANGE 270, 275
load module 159, 259, 363, 398
LOAD PART 100
LOAD utility 270, 272
LOB 9, 130, 139, 233, 300, 332, 336, 381, 383, 397, 405
LOB locks 332
local predicate 79
lock 27, 62–63, 68, 131, 133, 145, 147–148, 155, 192, 207–209, 211, 238, 260, 264–265, 269, 272, 290–292, 310, 315–316, 329, 331, 333, 336, 339, 341–343, 345, 352–355, 357–358, 361, 383, 391, 393, 405, 408–410, 413, 416
lock avoidance 133, 237, 290, 328, 334, 339–341, 348, 405, 409, 411
  data sharing 341
lock compatibility 331
lock compatibility matrix 331
Lock contention 47, 147, 291, 354–355, 411
lock contention 265, 335
lock duration 209, 339, 342, 345, 349, 355
  recommendation 347
Lock escalation 205, 315, 383, 411
lock escalation 205, 335, 349, 366, 383
lock granularity 346
lock mode 409
LOCK TABLE 34, 211, 292, 334, 336, 366, 393
LOCK TABLE IN SHARE MODE 330, 346
lock types 331
Locking 58, 62, 111, 114–115, 133, 147, 155, 192, 209, 232, 237, 260, 273, 276, 291–292, 312, 315–316, 327, 336, 344, 346, 366, 378, 393, 398, 404–405, 409, 411
locking 328, 363
  table space parameters effect 333
locking and DSNZPARMs 315
locking and latching 365
locking granularity 329
locking hierarchy 329
locking information application managed 347
locking levels 329
locking options 366
LOCKMAX 327, 335
LOCKMAX 0 328, 335
LOCKMAX SYSTEM 335
locks 329
locks with LOBs 332
LOCKSIZE 327, 333
LOCKSIZE ANY 334
LOCKSIZE PAGE 334
LOCKSIZE=ANY 366
log xxvi, 64, 132, 134, 141, 143, 148, 263–265, 268–271, 274, 285, 333, 352, 384, 391, 396–397, 405, 415, 417–418
log buffer
  forced write 349
log capture 374
log records 385
logical entity 1
logical model xxv, 1, 3–4, 18
logical unit 281, 350
  previous updates 347
logical unit of work 328, 347, 367
long columns 10
LONG VARCHAR 9
LONG VARGRAPHIC 9
long-running reader 384
look-up table 35, 135–136
LOOP 168, 242–243, 254, 277, 424, 429
low latency 373
LPAR 170, 257, 405–406
LRDRTHLD 384

M

mass delete 44, 331
mass delete lock 329, 411
Massive insert 144, 264
master copy 136
matching column 224, 351, 423
matching index access 312
Matching index scan 117, 122, 381, 422
matching index scan 66, 117–118, 226, 287
materialize 219–220, 232, 351
Materialized query table 133, 234
materialized query tables 27, 32
MAXASSIGNEDVAL 143
MAXDBAT 300
maximum length 56, 85, 164, 166
maximum number 67, 232, 241–242, 300, 315, 393–394, 423–424
maximum number of locks 315
maximum number of rows 268
maximum record size 57

MAXKEEPD 309
MAXROW 1 415
MAXROWS 58, 327
MAXROWS 1 147, 250, 334, 358, 411
  table space 147, 335
MEMBER CLUSTER 119, 411–412
memory 398
memory footprint 398
message metadata 165
METHOD 196
Migration 168
MINVALUE 138–139
module DSNTIAR 251
More partitions 422
MOST 314
MQ 155
MQ functions 164
MQ location 164–165
MQPUBLISHXML 166
MQREAD 164
MQREADALL 165
MQRECEIVE 164
MQRECEIVEALL 166
MQSEND 164
MQT 27, 32, 34, 125, 133, 234
MQT maintenance 33
MQTs 32
MRO 149
multi-column index 76
  column sequence 78
multiple batch job steps 347
multiple cursors 225
Multiple index
  access 67, 118, 423
multiple index 67, 77, 423
multiple index access 68
multiple index access path 77, 223
multiple index scan 118
multiple insert processes 334
multiple rows 241
multi-region operation 149
multi-row delete 244
Multi-row fetch 218, 240–242, 244, 290, 302
multi-row FETCH 218, 239
multi-row fetch 234
multi-row FETCH conditions 242
Multi-row INSERT 212, 254, 264, 266–267, 269–270, 272, 279, 286, 302
multi-row Insert 302
multi-row insert
  input variables 269
multi-valued column 38, 132

N

named counter servers 149
nested loop join 74, 79, 203, 424
New tables 83, 182
next value 111, 142–145, 149, 416–418
NO CACHE 415
NO CACHE report 435
NOFOR 207
non uniform distribution statistics 305
non-boolean predicat 223
Non-GBP-dependent 413
non-leaf page 423, 427
non-leaf pages 117
non-matching index scan 61, 117, 121, 223
non-null value 61, 130
non-uniform distribution statistics 312
non-unique indexes 362
NOT ATOMIC CONTINUE ON SQLEXCEPTION 253
  269
NOT IN 70, 265, 293, 386
NOT NULL WITH DEFAULT 13–14, 43, 130, 246
NOT PADDED 71
NPAGES 399
NPGTHRSH 24
NPI 92–94, 98–99, 102, 104–105, 111, 372
null value 13, 15, 64, 164
nullable columns
  indexing 16
number of partitions 50
NUMCOLS 313
NUMCOLS n 314
NUMLKTS 205, 315, 327, 335, 366
NUMLKUS 34, 315, 327, 336, 366
NUMLKUS threshold 349
NUMTCB 161

O

OA06729 406
ODBC 294, 301, 320, 395
ON COMMIT DELETE ROWS 157, 211–212, 300
ON COMMIT DROP TABLE 212
ON COMMIT PRESERVE ROWS 157, 212, 300
ON DELETE CASCADE 63
ON DELETE SET NULL 63
one fetch access 118
ONLINE LOAD 415
ONLINE REORG 110
Online REORG 92, 99, 110, 265, 274, 277, 360, 372, 385
online schema 361
OPEN C1 357
OPEN CURSOR 212–213, 221, 232–234, 241, 349, 351, 393, 410
open cursor 213, 349, 351
Open Transaction Environment 309
OPEN/CLOSE 391
optimistic locking 237, 248
optimize 71, 204, 301, 309, 386, 409, 421, 429
OPTIMIZE FOR 203–204
OPTIMIZE FOR 1 ROW 231
OPTIMIZE for n ROWS 302
optimizer hints 281
OR 223
OR 0 =
  HV 228
ORDER 415
ORDER BY 2, 69–70, 75, 115, 118, 193–194, 203–204, 221, 223, 226, 230, 235, 239, 259, 343, 351, 410

sort avoidance 194
order of locks 367
ORDER table 355
ORDER_SUMMARY table 355–356
OTE 309
Other Read 391
Other Write 392
outer subselect 234

P

package 116, 153, 158, 170–171, 173, 205, 213, 293–295, 297, 300, 304, 306, 308, 310, 312, 318, 336, 363–364, 376–377, 380, 382–383, 387, 391–392, 395, 410
package authorization 303
package level accounting 396
package list 304
package versioning 259, 363
PADDED 193
PADDED index 83
Page latch suspension 392
page level 68, 209, 329, 334, 344, 350, 366
page P-locks 62, 408, 411
page range screening 101
page set 111, 148, 407, 410, 412–413
page size 55
considerations 58
page splitting 64
PAGESAVE 54
parallel access volume 67
parallel batch job streams 365
parallel job 271
Parallel Sysplex
Availability 404
environment 149, 409
parallelism 46, 94, 125, 155, 167, 211, 271, 277, 280, 294, 373, 382, 391
degree 365
parameter markers 309, 396
PARAMETER TYPE 160
parent row 21, 130–133, 143
parent table 63, 119, 130–133, 135, 168, 273, 410
parent key 130
redundant checks 133
unique key 130
PARTITION 45, 47, 64, 74–75, 91, 98, 111, 121, 132, 143, 153, 192, 329, 331, 333, 360, 368, 372, 375, 383, 399, 409, 412, 422
Partition pruning 104, 106, 132
partition pruning 93
PARTITIONED 50, 104, 111–112, 122, 132, 192, 271, 280, 329, 333, 335, 372, 422
Partitioned index 121, 372, 407
partitioned tables 98
Partitioning 43, 45, 73–75, 85, 98, 104, 110, 112–113, 124–125, 132, 144, 192, 368, 372, 412
partitioning 43, 61, 89, 92–93, 104, 122, 125
benefits 44
table based 121
partitioning column 104
partitioning key 45
partitioning key size 85
PARTKEYU 45
PATH 118, 148, 158, 220–221, 296, 298–299, 309, 364, 388, 394, 409, 423, 429
PAV xxiv, 67
PCLOSEN 407
PCLOSET 407
PCTFREE 399
PCTFREE Value 64, 400
PCTROWCOMP 54
pessimistic locking 248
phase 2 commit 391
physical design 1–2, 22, 75, 109–110, 215, 355, 370
physical model 3
physical table design 75
PIECESIZE 98
PK01759 406
PK03878 406
PK05644 265
PK06185 362
PK10278 9
PK12389 7, 198
plan 113, 202, 278, 290–294, 297–299, 303, 306, 308, 311–312, 336, 345, 363, 372, 380–384, 387–388, 410–411, 424
plan and package efficienc 388
plan cache size 303
PLAN_TABLE 67, 277, 310, 380
P-lock 147, 334, 407–408, 411, 417
point-in-time 144, 362
point-in-time recovery 144
populate 65, 381
portability 235
POSITION 219, 230–231, 427
positioned UPDATE 233, 237
PQ45052 195
PQ70901 139
PQ79789 155
PQ82390 155, 316
PQ86037 106
PQ87381 106
PQ90263 9
PQ92833 155, 316
PQ94147 314
PQ96189 45
PQ98172 317
PQ99658 398
PRECISION 199, 299
Precompiler 157, 363
precompiler option NOFOR 207
predicate use 76
predictive governing 382
PREFETCH 115, 271, 280, 283, 355, 381, 392, 394, 423, 425–426
PREPARE 229, 309
PREPARE cost 308
primary key xxv, 3, 7, 18, 41–42, 61, 74, 76, 81–82, 88, 130, 132, 135, 143, 178, 356, 361, 363, 369
update 20

Archived

PRIQTY 361
privilege 303
program logic 355
program loop 279
programming language 6, 116–117, 156, 159, 166, 169, 191
propagation 22, 27, 357, 409
pseudo delete 63

Q

QBLOCKNO 196
QR TCB 309
QSAM file 285
QUALIFIER 211, 270, 274, 377
quasi-reentrant task control block 309
QUBE 120–121, 266
QUERY 38, 65, 82, 122, 125, 153, 185, 194–195, 211–212, 234, 238, 266, 277, 280, 290, 294, 301, 308, 313, 317, 337, 361–362, 380, 382, 391, 399, 411, 424
query environment 305
Query parallelism 294, 382, 396
QUERYNO 202
quick upper bound estimate 120

R

R/O 360, 397, 405, 407, 413
R/O switch 413
R/W 407
R/W interest 407, 409, 413
random I/O 96, 265, 279
RBDP 65, 362
reactive governing 382
read stability 339, 346
read-only cursor 234, 290, 301, 339–340
read-only transactions 384
Real contention 409
Real storage xxv, 315
Real-time statistics 398
real-time statistics 311
REBIND 194, 213, 280, 293, 307, 310, 313, 318, 362, 364, 377, 380, 382, 391, 429
REBIND of an active package 364
REBIND PACKAGE (*.*) 312
REBIND PLAN(*) 312
REBUILD INDEX 64–65, 377
Rebuild pending 65, 362
received SQLCODE
  full explanation 259
record ID 8
record size 55
RECOVER 63, 378, 415
recoverable VSAM files 284
recovery 210, 264, 272, 287, 347–348, 359, 362, 378, 384–385, 398, 404
recovery to a previous point in time 367
Redbooks Web site 439
referential constraint 130–131, 133
referential integrity 14, 21, 63, 116, 130–133, 135–136, 206–207, 217, 269, 423
REFRESH TABLE 32
REGION 149, 257, 297
relational database 135, 153, 186, 210, 311
relative number 232
RELEASE 311, 327
RELEASE DEALLOCATE 429
RELEASE(COMMIT) 347
RELEASE(DEALLOCATE) 347
REMARKS 144
REOPT(ALWAYS) 308
REOPT(VARS) 227, 231
re-optimization 305
REORG 63–64, 90–91, 93, 107, 265, 273–274, 276, 311, 313, 333, 335, 360, 362, 369, 372, 377, 385, 395
  DISCARD 274
  frequency 92
REORG DISCARD 92
REORG frequency 369, 400
REORG INDEX 64
REORG TABLESPACE 273
reorganization frequency 78, 400
repeatable read 62, 339, 346
replica 27
REPORT 160, 380, 386, 388, 393–394, 397, 416
repositioning 221
RESET 144, 291, 378, 386
Residual predicates 196
resource consumption 2, 113, 224–225, 232, 374, 381, 386–387, 390, 396, 398, 409
  upper boundary 232
Resource Limit Facility 232, 294, 381, 396–397
resource limit specification table 382
resource manager 209, 374
RESTART 142, 144, 207, 223–224, 268–269, 272, 280, 282–284, 287, 348–349, 351, 384–385
restart 144, 217, 224, 280, 282–283, 285, 349
restart coding 217
RESTORE 300, 359, 377–378
RESTRICT 130, 382, 412
result set 71, 115, 158, 192, 204, 212, 217–218, 222–224, 226, 229–232, 234, 278, 321
  determining 242
  Retrieving parts 217
result table 204, 233–234, 236
RI 14, 38, 95, 107, 116, 130–132, 134–136, 143, 206, 359, 410
  DB2 enforced 344
RID 50, 63, 67–69, 81, 111, 118, 121, 277–278, 310, 320, 392, 394, 423, 426
RID list 68–69, 277, 311, 394
RID Pool 68, 394
RIDs 64, 67–69, 71–72, 99, 111, 118, 223, 278–279, 394, 423–424
RLF 232, 294, 381, 397
RLFASUERR 382
RLFASUWARN 382
RMF 385
RO 384, 407
ROLLBACK 63, 145, 159, 163, 192, 207–208, 210–211, 213, 252, 256–258, 268, 272–273, 285, 337, 341,

345–347, 350, 352–353, 363, 378, 386, 413
ROLLBACK TO SAVEPOINT 353
ROTATE PARTITION 48
row length 8–9, 22, 40, 56, 121, 172
Row level
  concurrency benefits 334
row level 147, 209, 287, 317, 328–329, 334, 344, 348, 354, 411
row level locking 287, 331
row lock 62, 237, 331, 336, 409, 411
row locking 411
row trigge DDL 170
row trigger 169–170, 172–175
  logic 174
row trigger report 433
ROWID 130, 139–140, 148, 192, 212, 276, 395
  direct row access 276, 395
  UPDATE 148, 395
ROWID access 276
ROWID field 140
ROWID value 140, 276
row-level locking 334
Rowset 239, 242, 268, 302, 398
ROWSET positioning 241
RPTSTG 160
RR 62, 145, 316, 333, 339, 346, 410, 413
RRS 284
RRS attach 386
RRSAF 209, 282, 350
RS isolation 410
RTS 311
run time 304
RUNSTATS 74, 89, 213, 311–313, 362, 392, 400
  SAMPLE 313
RUNSTATS INDEX 314
RUNSTATS utility 74, 313
run-time option 160

S

safeguard 335
same data 192, 272, 281, 283
same table 130, 234, 269
  different pages 147, 358
same time xxiv, 239, 245, 259, 373, 391
SAMPLE keyword 313
SAVEPOINT 210
  restriction 353
savepoint 352
savepoint name 353
SAVEPOINT statement 352
savepoints 347
Scalability 179, 275, 404
SCHEMA 123, 140, 144, 163, 183–184, 186, 211, 364, 380
schema 183–184, 186, 364
Schema changes 183, 361
SCROLL 75, 218, 226, 232, 235–236, 239
scroll forward 219
scrollable cursor 233, 237, 239
  INSENSITIVE 233
  locking 237
  SENSITIVE 233, 235
second copy 232
SECQTY 265, 361
Security level 153
segmented table space 34, 36, 45, 51, 93, 147, 207, 273, 292, 329, 333, 358
SELECT 30, 63, 107, 115, 117, 133, 139–140, 148, 167, 170, 175, 183, 191, 193, 195–196, 201–204, 211, 213, 234, 259, 266, 273, 279, 282, 286–287, 306, 336, 346, 357, 373, 375–376, 380–382, 386, 393, 395, 399, 423–425, 428–429, 439
  cost 117
SELECT statement 193, 234, 273, 373, 424
select statement
  latency requirement 373
SENSITIVE DYNAMIC 234–235, 237
SENSITIVE DYNAMIC SCROLL 234
SENSITIVE STATIC 233, 235–236
sequence 142–143
sequence number 19–20, 142, 149, 265, 415
  first request 415
  next request 415
sequences
  accounting report 435
  performance 434
Sequential detection 118, 122, 132, 278–279, 291, 349, 391–392, 421, 423–426
sequential file 91, 162, 270–272, 286, 360
sequential number 81, 138, 146, 149, 357
Sequential prefetch 391
sequential prefetch 68, 117, 203, 278–279, 283, 391, 423–424
Service Task suspension 391
Service units 396
SESSION 157, 211
SET CURRENT PACKAGESET 261
SET NULL 131
shadow data set 385
SHARE 366, 409, 415
short prepare 308
shred 181
side table 178, 181
simplified view 120
single index access path 77
single row 90, 117, 141, 145–146, 174, 216, 218, 239, 241, 245, 264, 266, 290, 301
single table 38, 79, 130, 380, 383, 412
singleton SELECT 204
singleton select 165, 216, 341
SKCT 361
SKIPUNC 155
SKPT 361, 364
SLA 178, 373, 378, 404
S-lock 46, 308
SMF 385–386, 398
SMF 30 398
SMF type 30 report 398
snapshot of data 239
sort 115, 280

Archived

sort avoidance 69, 78, 124
sort pool 310
spacial data 162
Special register 141, 191, 261, 280, 294–295, 299, 386
special register 376
splitting tables 41
SQL 15, 111, 117, 120, 122, 139, 152, 155–156, 158–159, 183–184, 186–187, 189, 193, 195–198, 202, 211, 239, 251–252, 255, 261, 266–267, 271, 273, 275, 280–281, 283, 286, 291–295, 297–300, 312–313, 320, 327, 329, 331, 336, 345, 347, 350, 361, 363, 368, 370, 380–381, 387–388, 390, 393, 395–396, 407, 410–411, 414, 421–422, 424–426, 429
  statement execution order 328
SQL for sequence 416
SQL statement 116–117, 189, 193, 253, 267, 272, 295, 299, 301, 327, 336, 380–381, 388, 396, 421
  cost 116
  execution 306
  level 337
  locking 336
  processor 207, 382
SQL workload 399
SQL/XML 185
SQLCA 207, 242, 251–253, 257
SQLCODE 117, 138, 140–141, 204, 238, 242, 252, 256–257, 268, 293, 295, 315, 342, 345, 382, 393
SQLCODE 100 117, 204, 242
SQLERRD (3) 243
SQLJ 154, 320, 322
Stage 1 196–200, 228, 308, 316–317, 339–340, 410, 422
Stage 2 18, 67, 196–200, 228, 307, 339, 410
START WITH 139
state transition indicator 26
STATEMENT 255
statement 395
statement trigger 169–175
statement trigger report 434
STATIC 153, 159, 189, 211, 227, 229, 233, 236, 239, 269, 280, 292, 294, 312, 314, 317, 361, 363, 382
static scrollable cursor 212, 351
Static scrollable cursors 233, 237
static scrollable cursors
  updating 237
Static SQL 154, 294
static SQL 227, 306–308
statistics record 386
statistics trace
  classes 397
STDSQL(YES) 208
storage report 160, 432
STORE_ID 30
stored procedure 116, 153–159, 161–162, 166, 168–169, 174, 304, 364
stored procedures 304
Striping 44, 91, 94, 405
Subject 298, 425
Subquery 195, 234
subquery 228, 234
subquery to join transformation 195
subselect 234, 429
SUM 30, 75, 84, 120, 165, 174, 286, 365
surrogate key 19
Sync I/O 392
synchronous I/O
  operation 279
  processing 118, 424
synchronous read 68, 115, 391–392
SYSADM 384
SYSCOLDIST 314
SYSCOLUMNS 312
SYSCOPY 274
SYSIBM 252, 311, 314, 364, 399–400, 415
SYSIBM.INDEXSPACESTATS 311
SYSIBM.SYSINDEXES 89
SYSIBM.SYSINDEXSTATS 89
SYSIBM.SYSPACKAGE 259
SYSIBM.SYSPACKSTMT 202
SYSIBM.SYSROUTINES 167
SYSIBM.SYSSEQUENCES 143–144, 415–416
SYSIBM.SYSSTMT 202
SYSIBM.SYSTABLEPART 54
SYSLGRNG 391
SYSPACKSTMT 211
sysplex query parallelism 281

T

Table design 3, 20, 258, 275
table design 121
table function 153, 162, 166, 234
table space
  EXCLUSIVE lock 366
  lock hierarchy 335
  locksize parameter 333
  multiple tables 275, 333
  single S lock 330
table space level locking 340
table space parameters 327
table space scan 69, 380
table-based partitioning 61, 74, 105
TCP/IP 318, 391
TEMP database 157, 211
temporary table 172
TEXT 87, 180, 251, 293
third normal form 24
thread 118, 168, 212, 226, 251, 260, 262, 268, 291–292, 296–301, 309, 315, 329, 333, 341, 345, 347, 349, 384, 386, 390–392, 398, 405
thread pooling 352
thread reuse 157, 291–292, 296, 298–300, 390, 410
  implicit drop feature 157
threadsafe 309
time out 341
time slicing 406–407
timeouts 257
TIMESTAMP 15, 24, 105–106, 117, 139–140, 169, 246–247, 259–260
TIMESTAMP DESC 259
Tivoli® Decision Support for OS/390 390

touch 115
TRACKMOD NO 412
transaction monitor 303
transition table 166, 171, 174–175
Transition variables 116, 169–170, 172, 174
Trigger
  WHEN condition 175
trigger 25, 30, 82–83, 85, 116, 155, 166, 168–169, 172–174, 181, 273, 293, 374, 388, 424
  processing 269
  transition tables 116, 269
  WHEN 269
Trigger body 116, 169, 174–175
trigger transition table 269
triggers 269
  accounting detail report 433
TRIM function 320
TSO 282
Two-phase commit 163, 284
two-phase commit 284
Type 1 61
Type 2 61, 70, 318, 323, 386
Type 2 Driver 318
Type 2 index 70
Type 4 Driver 318

U

UK00296 314
uncommitted read 205, 238, 332, 337–338, 411
Unicode 7, 294, 319
UNION 68–69, 115, 234–235
UNION ALL 234–235
UNION IN VIEW 368
UNIQUE 16, 22, 61, 69, 74–76, 88, 90, 111, 119, 121, 130–131, 136, 138–141, 149, 179, 204, 225, 246, 286, 320, 415
unique clustering index 223
unique columns 222
unique constraint 130
unique index 19, 61, 131, 140–142, 362
UNIQUE WHERE NOT NULL 16
UNIQUE WHERE NOT NULL clause 83
uniqueness 61
  enforcing 76
unit of recovery 210, 217, 264, 347, 349, 353
unit of work 145–146, 152, 163, 207, 209–210, 249–250, 256–258, 260, 268–269, 272, 276–277, 279, 301, 316, 320, 339, 341, 344, 346, 349–350, 352, 355, 357–358, 378, 391
Universal Driver 301, 317–322
Universal Driver for SQLJ and JDBC 317
UNLOAD 110, 170, 360, 372
unordered updates 344
updatable views 369
UPDATE xxv, 2, 35, 62, 71, 83, 87, 119–120, 130, 133–135, 138, 141, 144, 146–148, 166–167, 181, 183, 191, 205, 211, 217, 245, 247, 260, 264, 270, 272, 275–279, 281, 312, 316, 332, 336, 339, 347, 350, 355, 357, 362–363, 372, 377, 380, 382, 391, 399, 405, 407, 410, 413–415, 417, 425–426, 428–429
  cost 120
update access 330
UPDATE COUNTER_TBL 357
update hole 238
update lock 217
UPDATE statement 192, 246, 248, 275, 277
update volatility 31
UPROC 349, 429
UQ50928 195
UR 193, 205, 235, 250, 252, 260, 331–332, 336, 339, 346, 384, 411
URCHKTH 384
URL 114
URLGWTH 385
USER 149, 167, 257, 274, 300, 303–304, 315, 318, 335, 382, 384, 390, 393, 411, 413
user defined function 167
user-defined 305, 380
UTF16 319
UTF8 319
utility contention 342
UTIMOUT 315
UTRO 65

V

VALIDATE 311
VALUES INTO 136
VARCHAR 7
VARCHAR column 83
VARCHAR variable 164, 166
VARGRAPHIC 200
VARIABLE 15, 56, 83, 85, 164, 166, 172, 202, 261, 280
variable character data types 5
vCF 406–407
Versioning 362, 364, 369
vertical table split 41
VIEW 193, 211, 368
view 234
Visual Explain 66–67, 157, 213, 313
VOLATILE 23, 62, 123, 147, 251, 354, 358
volatile 312, 360
Volatile table 312
VSO 148

W

W/W 408
wait time 114, 268, 339, 341, 381, 390–392
warning and error threshold 381
Web Site 439
WebSphere 88, 152, 155–156, 162–163, 168, 319, 374, 386, 389–390
WebSphere MQ 155, 163, 168, 374
  message 163
WebSphere MQ functions 163
WHERE clause 192–193
WHERE CURRENT OF 208, 217, 274, 286, 357, 381
WITH clause 336
WITH HOLD 157, 211–212, 286, 300, 327, 340, 349–352, 429

Archived

criteria to use 351
WITHOUT ROWSET POSITIONING 241
WLM xxvi, 116, 152, 155, 161–163, 380
work file 115, 174, 211, 219–220, 235, 351
work file creation 397
workfile 65, 94
Work-in-progress table 250–251
WRITE 333
WRITE CLAIM 333

X

XES 291–292, 409–410, 416
XES contention 410
XLKUPDLT 317
X-lock 364
XML 59, 87–88, 127, 153, 166, 177–178, 180, 182, 186, 405
Data Access Definition 180
publishing 183
shredding 181
XML Collection 178, 182–183
XML data 178–179, 183, 185–186
XML document 178–183, 186
simple search 180
XML Extender 88, 179–180, 182–184, 186
composition 186
functionality 186
publishing function 186
XML Publishing 166, 183
xternal data type. 185

Z

z900 116, 158, 169, 406
z990 xxvi, 170, 405–406
zAAP 156
zSeries 114, 153, 156, 405–406

Archived

Redbooks

DB2 UDB for z/OS: Application Design for High Performance and Availability

(1.0” spine)
0.875”<->1.498”
460 <-> 788 pages

Archived

Archived

Archived

DB2 UDB for z/OS: Application Design for High Performance and Availability

Develop applications for concurrency, performance, and availability

Learn useful programming techniques

Check your table and index design

Conducting business via the Web and remaining open for business 24 hours a day, seven days a week is now commonplace. Customers come in with unexpected workloads through the Web and operate in a self-service fashion with mostly context-sensitive metadata to guide them. The strong requirement is availability. However, even with new Web applications, most of the core business systems considerations still apply, and performance is critical.

Technology has been accelerating for mainframe systems. They had become adept at providing business resiliency accommodating strategic software that has been around for the last several decades, such as IMS, DB2, and CICS, and they have also become a host for developing and running Web applications built in Java accommodating the latest business requirements. Businesses need to leverage, extend, and integrate the strategic assets which represent multi-year investments to support leading edge technology.

DB2 for z/OS has come a long way and provides facilities to exploit the latest hardware and software technologies, accommodating a majority of user requirements. However, special considerations are required to build high performance applications. If you want to achieve high performance or high availability, you must use the design, programming, and operational techniques applicable to DB2.

In this IBM Redbook, we discuss many of these techniques and provide guidelines for database and application design. We expect the best practices described in this IBM Redbook will help DB2 professionals design high-performance and high-availability applications.

INTERNATIONAL TECHNICAL SUPPORT ORGANIZATION

BUILDING TECHNICAL INFORMATION BASED ON PRACTICAL EXPERIENCE

IBM Redbooks are developed by the IBM International Technical Support Organization. Experts from IBM, Customers and Partners from around the world create timely technical information based on realistic scenarios. Specific recommendations are provided to help you implement IT solutions more effectively in your environment.

For more information:
ibm.com/redbooks

SG24-7134-00 ISBN 0738494216