

Managing Ever-Increasing Amounts of Data with IBM DB2 for z/OS

Using Temporal Data Management, Archive Transparency, and the DB2 Analytics Accelerator

Mehmet Cuneyt Goksu

Claire McFeely

Craig McKellar

Xiao Hui Wang



Information Management



International Technical Support Organization

**Managing Ever-Increasing Amounts of Data with IBM
DB2 for z/OS**

September 2015

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

First Edition (September 2015)

This edition applies to DB2 11 for z/OS and DB2 10 for z/OS.

© Copyright International Business Machines Corporation 2015. 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

Notices	vii
Trademarks	viii
IBM Redbooks promotions	ix
Preface	xi
Authors	xii
Now you can become a published author, too!	xiv
Comments welcome	xiv
Stay connected to IBM Redbooks	xiv
Part 1. Temporal data management	1
Chapter 1. Temporal data management overview	3
1.1 Overview	4
1.1.1 Periods	5
1.1.2 System-period temporal tables	6
1.1.3 Application-period temporal tables	9
1.1.4 Bitemporal tables	11
1.1.5 Choosing the right type of temporal table	11
Chapter 2. System-period temporal tables	13
2.1 Overview	14
2.2 Relationship of a system-period temporal table to the associated history table	14
2.3 Moving from traditional to system-period temporal tables	16
2.3.1 Creating new tables	16
2.3.2 The history table and version enabling	17
2.3.3 Enabling versioning for an existing non-temporal table	18
2.3.4 Querying a system-period temporal table	20
2.3.5 Enabling versioning for an application managed temporal table	21
2.3.6 Other scenarios for migrating to system-period temporal tables	26
2.4 The who, what, and how of my row (audit columns)	26
2.4.1 Additional considerations for audit information and delete operations	27
2.4.2 The “who”, “what”, and “how” in action	27
2.5 Querying a system-period temporal table	29
2.5.1 System-period temporal table query examples	31
2.6 General restrictions with system-period data versioning	37
2.7 Utilities	37
2.7.1 LISTDEF	37
2.7.2 UNLOAD	38
2.7.3 LOAD	38
2.8 Recovery	39
Chapter 3. Application-period temporal tables	41
3.1 Overview and benefits	42
3.2 Using application-period temporal tables	42
3.2.1 New concept of a BUSINESS_TIME period	42
3.2.2 Defining a BUSINESS_TIME period	43
3.2.3 Extensions to primary keys, unique constraints, and unique indexes	44

3.2.4	Temporal UPDATE and DELETE rules	45
3.2.5	Queries for application-period temporal tables	49
3.2.6	BUSINESS_TIME period restrictions	50
3.2.7	Utilities	50
3.3	Case study: an application-period table in action	50
3.3.1	Create an application-period temporal table	50
3.3.2	Step-by-step usage scenarios	51
Chapter 4.	Bitemporal tables	57
4.1	Overview	58
4.2	Where application-period meets system-period temporal	58
4.2.1	Using bitemporal tables in data manipulation statements	59
4.3	Extending an application-period temporal table to become a bitemporal table	60
4.4	Bitemporal tables in action	61
4.4.1	Create a bitemporal table	61
4.4.2	Step-by-step usage scenarios	63
4.4.3	Bitemporal queries	71
Chapter 5.	Additional considerations	73
5.1	Temporal support for views	74
5.1.1	Queries	74
5.1.2	Data change operations on views	74
5.1.3	Query of a view using a period specification for SYSTEM_TIME	75
5.1.4	Query on a view using a period specification for BUSINESS_TIME	75
5.1.5	Data change operations on temporal views	76
5.2	Time travel: Temporal special registers	76
5.2.1	Two temporal special registers	77
5.2.2	Enabling or disabling the temporal special registers	77
5.2.3	How temporal registers affect queries	78
5.2.4	How temporal affects UPDATE and DELETE statements	79
5.2.5	How temporal special registers affect statements in user-defined functions	81
5.3	Auditing with temporal tables	82
5.3.1	Overview and benefit	82
5.3.2	Using non-deterministic generated expression columns	83
5.3.3	Detailed explanation of the preceding example	85
5.4	Performance	86
5.4.1	CURRENT TEMPORAL SYSTEM_TIME special register	86
5.4.2	Performance summary on temporal support	87
Part 2.	IBM DB2 archive transparency	89
Chapter 6.	DB2 archive transparency overview	91
6.1	Challenges with historical data	92
6.2	How DB2 archive transparency works	92
Chapter 7.	DB2 archive transparency concepts	95
7.1	Overview	96
7.2	What is transparent archiving?	98
7.3	Enabling transparent archiving	100
7.3.1	Create an archive table	101
7.3.2	Enable transparent archiving	101
7.4	Archive transparency controls	102
7.4.1	Archive transparency bind options	103
7.4.2	Archive transparency built-in global variables	104

7.4.3 Data change operations when the target is an archive-enabled table	104
7.4.4 Rules for deleting from an archive-enabled table	105
7.5 Query from an archive-enabled table	105
7.5.1 Querying the current data only	105
7.5.2 Transparently querying the current and archive data	105
7.6 Disable archiving for an archive-enabled table	106
7.7 Using an archive timestamp with archive transparency	106
7.8 Summary	107
Chapter 8. Case study: Using archive transparency	109
8.1 Sample application with step-by-step guidance	110
8.2 Define a table to enable transparent archiving	112
8.2.1 Creating the archive table	113
8.2.2 Moving data from the active table to the archive table	114
8.2.3 Loading data into the archive table	115
8.2.4 Deleting data from specific partitions of an archive-enabled table	118
8.3 Enabling archive transparency	119
8.4 Mechanics of moving data from an archive-enabled table to an archive table	120
8.5 Querying data from an archive-enabled table	122
Part 3. DB2 Analytics Accelerator	125
Chapter 9. DB2 Analytics Accelerator overview	127
9.1 Overview	128
9.2 Variations for using the DB2 Analytics Accelerator	129
9.2.1 Accelerator-shadow table	129
9.2.2 Accelerator-archived table	130
9.2.3 Accelerator-only table	130
9.2.4 Using temporal tables and DB2 Analytics Accelerator together	131
9.2.5 Using archive transparency and the DB2 Analytics Accelerator together	132
9.2.6 Using a row change timestamp column with archive transparency and the DB2 Analytics Accelerator	132
Chapter 10. DB2 Analytics Accelerator High Performance Storage Saver overview.	135
10.1 Overview	136
10.2 What is High Performance Storage Saver?	137
10.2.1 Restrictions	138
10.3 Online data archiving	139
10.3.1 Archive data process and operations	139
10.3.2 Accessing archived data	141
10.3.3 Restoring an archived partition from the accelerator	146
Chapter 11. Case study: Using the DB2 Analytics Accelerator	147
11.1 System setup of application with an accelerator	148
11.2 Moving and storing archived data with an accelerator	151
11.3 Querying archived data	153
Part 4. Creating an integrated solution.	155
Chapter 12. Case study: Combining DB2 temporal data management with the DB2 Analytics Accelerator	157
12.1 Using the DB2 Analytics Accelerator with application-period temporal tables	158
12.1.1 Step-by-step scenario	158
12.1.2 Define an application-period temporal table	159

12.1.3	Adding an application-period temporal table into DB2 Analytics Accelerator . .	160
12.1.4	How are queries processed when data exists on an accelerator?	162
12.1.5	How are data change statements processed?	162
12.1.6	Archiving parts of an application-period temporal table to the DB2 Analytics Accelerator	163
12.1.7	How are queries processed?	165
12.2	Using the DB2 Analytics Accelerator with system-period temporal tables	166
12.2.1	Step-by-step scenario	166
12.2.2	How are queries processed when data exists on an accelerator?	169
Chapter 13. Case study: Combining archive transparency with the DB2 Analytics Accelerator		171
13.1	Overview	172
13.2	Enabling an archive-enabled table and archive table for acceleration	172
13.3	Querying an accelerator-shadow table	175
13.4	Moving archived partitions to the accelerator as an accelerator-archived table	176
13.5	Querying both an accelerator-shadow table and an accelerator-archived table . . .	178
Glossary		181
Related publications		183
Online resources		183
Help from IBM		183

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 grant 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 product(s) and/or the program(s) described in this publication at any time without notice.

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

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

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 illustrate 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.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. These and other IBM trademarked terms are marked on their first occurrence in this information with the appropriate symbol (® or ™), indicating US registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at <http://www.ibm.com/legal/copytrade.shtml>

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

DB2®
IBM®
IMS™
MVS™

QMFTM
Redbooks®
Redbooks (logo) ®
System z®

z/OS®
zEnterprise®

The following terms are trademarks of other companies:

Netezza, and N logo are trademarks or registered trademarks of IBM International Group B.V., an IBM Company.

Java, and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

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

Find and read thousands of IBM Redbooks publications

- ▶ Search, bookmark, save and organize favorites
- ▶ Get up-to-the-minute Redbooks news and announcements
- ▶ Link to the latest Redbooks blogs and videos

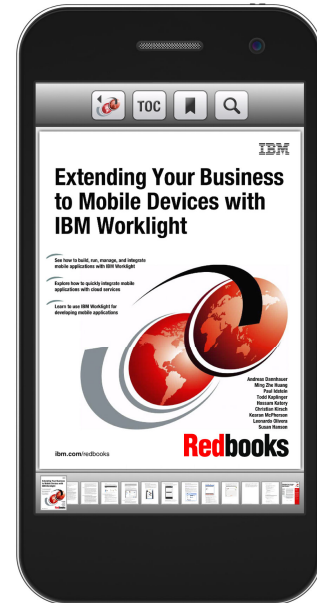
Get the latest version of the Redbooks Mobile App



iOS

Download
Now

Android



Promote your business in an IBM Redbooks publication

Place a Sponsorship Promotion in an IBM® Redbooks® publication, featuring your business or solution with a link to your web site.

Qualified IBM Business Partners may place a full page promotion in the most popular Redbooks publications. Imagine the power of being seen by users who download millions of Redbooks publications each year!



ibm.com/Redbooks

About Redbooks → Business Partner Programs

THIS PAGE INTENTIONALLY LEFT BLANK

Preface

IBM® DB2® Version 11.1 for z/OS® (*DB2 11 for z/OS* or just *DB2 11* throughout this book) is the fifteenth release of DB2 for IBM MVS™. The DB2 11 environment is available either for new installations of DB2 or for migrations from DB2 10 for z/OS subsystems only.

This IBM Redbooks® publication describes enhancements that are available with DB2 11 for z/OS. The contents help database administrators to understand the new extensions and performance enhancements, to plan for ways to use the key new capabilities, and to justify the investment in installing or migrating to DB2 11.

Businesses are faced with a global and increasingly competitive business environment, and they need to collect and analyze ever increasing amounts of data (Figure 1). Governments also need to collect and analyze large amounts of data. The main focus of this book is to introduce recent DB2 capability that can be used to address challenges facing organizations with storing and analyzing exploding amounts of business or organizational data, while managing risk and trying to meet new regulatory and compliance requirements.

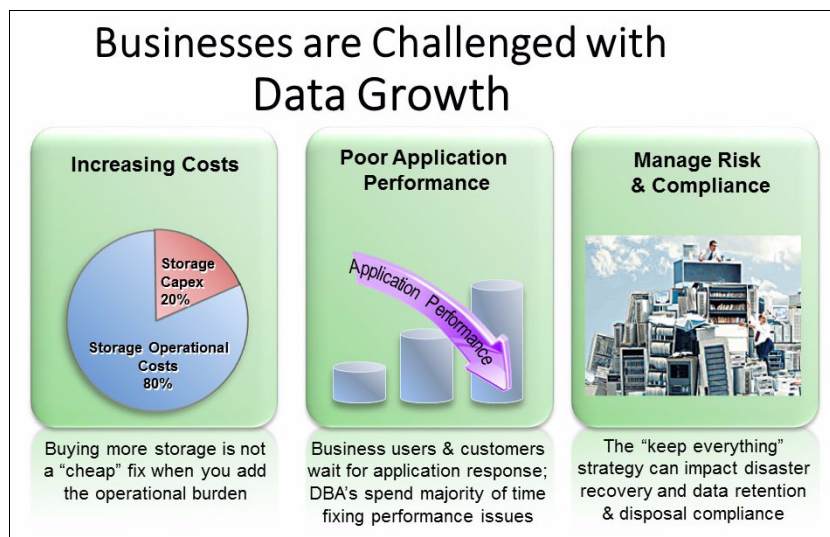


Figure 1 The challenge of data growth

This book describes recent extensions to DB2 for z/OS in V10 and V11 that can help organizations address these challenges.

Temporal tables

DB2 temporal support provides rich functionality for time-based data management. For example, you can choose to record the complete history of data changes for a database table so that you can "go back in time" and query any past state of your data. You can also indicate the business validity of data by assigning a pair of date or timestamp values to a row to indicate when the information is deemed valid in the real world. Using simple SQL syntax, you can easily insert, update, delete, and query data in the past, present, or future. The temporal support enables you to accurately track information and data changes over time and provide an efficient and cost-effective way to address auditing and compliance requirements.

Archive transparency

DB2 archive transparency is useful for managing historical data that is used infrequently. DB2 provides a row-based archiving solution that is implemented with a two-table approach. Using simple SQL syntax, you can define a table as an archive-enabled table that is associated with a separate archive table. As rows are deleted from the archive-enabled table, DB2 can automatically move them (or “copy them”) to the archive table. Built-in global variables and bind options control whether deleted data is automatically written to the archive table and whether queries of the archive-enabled table will consider data in the archive table as well. Existing queries do not need to be changed to reference only current data or current and archived data. By simply setting the built-in global variables and bind options, you can control whether the archive data is accessed or not.

DB2 Analytics Accelerator

The IBM DB2 Analytics Accelerator for IBM z/OS (simply called *DB2 Analytics Accelerator* in this book) is a union of the IBM System z® quality of service and IBM Netezza® technology to accelerate complex queries in a DB2 for z/OS highly secure and available environment. Together, DB2 Analytics Accelerator, DB2 for z/OS, and IBM zEnterprise® form a self-managing hybrid environment that can run online transaction processing (OLTP), online transactional analytical processing (OLTAP), and online analytical processing (OLAP) workloads concurrently and efficiently. A table can be defined as an “accelerator table” allowing data-intensive and complex queries to be run on the DB2 Analytics Accelerator, providing valuable analytics in a cost-effective way. With the hybrid system, short queries are processed by DB2 and the complex analytic queries are processed by the DB2 Analytics Accelerator, creating a solution that provides the best of both worlds.

These DB2 for z/OS capabilities provide significant functionality to help organizations address challenges with keeping and maintaining exploding amounts of data. Furthermore, these capabilities can be used in combination as building blocks to create an integrated solution.

Authors

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



Mehmet Cuneyt Goksu has been working as an IBM zAnalytics Technical Leader for MEA and is a member of zChampions Team specialized in Big Data and Analytics. He lives in Istanbul, Turkey. He served as DBA, DB2 Systems Programmer, and IBM DB2 Gold Consultant on IBM System z for over 25 years before joining IBM. He also served for International DB2 User Group (IDUG) about 10 years in Conference Planning Teams and as a BOD member. He provides technical presales support and consulting to clients who develop or extend the usage of core products, such as DB2, IBM IMS™, and all zAnalytics solutions including DB2 Analytics Accelerator. Cuneyt has co-authored many papers and articles. His most recent publication was about DB2 11 for z/OS developed by DB2 Gold Consultants on behalf of IDUG.



Claire McFeely is a Senior Software Engineer and a member of the DB2 for z/OS development team based in the Silicon Valley Lab. During her career with IBM in development, she worked primarily with IBM QMF™ and the DB2 database family. In her current role as lead SQL language architect for the DB2 for z/OS product, Claire oversees all SQL enhancements and extensions in the product and also focuses on SQL consistency across the DB2 family products. She has designed and implemented several key enhancements in DB2 for z/OS, and authored numerous internal specifications for enhancements to the product in the area of SQL extensions.



Craig McKellar has been a DBA on IBM System z for over 20 years, and over the last few years, he has used this experience in his role as an Accelerated Value Leader in Canberra, Australia. He provides hands-on support and consultation to large government clients for Information Management (IM) software. Craig has co-authored several IBM Redbooks publications on DB2 for z/OS tools.



Xiao Hui Wang is a Staff Software Engineer in IBM China Development Library. She has 7 years experience in the DB2 z/OS field. She holds a Master's degree in Computer Science from Bei Hang University in China.

This project was led by:

- Martin Keen

Special thanks to the sponsors of this project for their guidance:

- Steve Chen
- Maryela Weihrauch

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

- Patric Becker
- Chih-jieh Chang
- Xiao Hong Fu
- Robert Haimowitz
- Zhang Hao
- Terrie Jacopi
- Jeffrey Merritt
- Matthias Nicola
- Jim Reed
- David Zhang

Now you can become a published author, too!

Here's an opportunity to spotlight your skills, grow your career, and become a published author—all at the same time! Join an ITSO residency project and help write a book in your area of expertise, while honing your experience using leading-edge technologies. Your efforts will help to increase product acceptance and customer satisfaction, as you expand your network of technical contacts and relationships. Residencies run from two to six weeks in length, and you can participate either in person or as a remote resident working from your home base.

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 books to be as helpful as possible. Send us your comments about this book or other IBM Redbooks publications in one of the following ways:

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

ibm.com/redbooks

- Send your comments in an email to:

redbooks@us.ibm.com

- Mail your comments to:

IBM Corporation, International Technical Support Organization
Dept. HYTD Mail Station P099
2455 South Road
Poughkeepsie, NY 12601-5400

Stay connected to IBM Redbooks

- Find us on Facebook:

<http://www.facebook.com/IBMRedbooks>

- Follow us on Twitter:

<https://twitter.com/ibmredbooks>

- Look for us on LinkedIn:

<http://www.linkedin.com/groups?home=&gid=2130806>

- Explore new Redbooks publications, residencies, and workshops with the IBM Redbooks weekly newsletter:

<https://www.redbooks.ibm.com/Redbooks.nsf/subscribe?OpenForm>

- Stay current on recent Redbooks publications with RSS Feeds:

<http://www.redbooks.ibm.com/rss.html>



Part 1

Temporal data management

Part 1 contains the following chapters:

- ▶ Chapter 1, “Temporal data management overview” on page 3
- ▶ Chapter 2, “System-period temporal tables” on page 13
- ▶ Chapter 3, “Application-period temporal tables” on page 41
- ▶ Chapter 4, “Bitemporal tables” on page 57
- ▶ Chapter 5, “Additional considerations” on page 73



Temporal data management overview

This chapter outlines the temporal data management support in DB2 for z/OS.

1.1 Overview

Organizations are challenged to record and maintain ever increasing amounts of data. In many cases, organizations need to track and query different versions of application data over time to support queries of historical or future data. There are also requirements of regulatory and compliance laws to keep historical data. These are common challenges facing industries such as insurance, financial, retail, and human resources. DB2 for z/OS temporal tables provide time-based data management that can help businesses manage the increasing amounts of data and retention requirements.

Customers have requested support for having DB2 “remember” all past versions of rows in a table. An update operation on a table replaces the affected rows with new rows while a delete operation removes the affected rows from the table, leaving no historical information about the changes to the data. However, there are many application scenarios where it is important to preserve the history of data changes. A typical example of such applications is a banking application, where it is necessary to keep previous states of customer account information so that customers can be provided with a detailed history of their accounts. Current practice has been to maintain historical information through additional tables with triggers and stored procedures. This can be undesirable because it makes the application development more complex. It may also affect the runtime performance of such applications in a negative way.

Customers have also expressed a need to store future data in tables, that is, data that is not yet considered “valid” by a business or application.

DB2 for z/OS provides SQL extensions to define temporal tables that can be used to address these business challenges. For example, you can choose to record the complete history of data changes for a database table so that you can “go back in time” and query any past state of your data. You can also indicate the business validity of data by assigning a pair of date or timestamp values to a row to indicate when the information is deemed valid in the real world.

Using the DB2 for z/OS temporal extensions can:

- ▶ Simplify application development and maintenance (complexity and lines of code)
- ▶ Reduce application development time
- ▶ Simplify application management
- ▶ Enable movement of logic from applications to the database layer
- ▶ Provide automatic saving of historical data and changes
- ▶ Improve performance over home-grown temporal solutions
- ▶ Help address compliance requirements

DB2 provides support for defining several types of temporal tables:

- ▶ **System-period temporal table:** DB2 automatically keeps historical versions of each row.
- ▶ **Application-period temporal table:** A user or application maintains a “valid” period of time for each row.
- ▶ **Bitemporal table:** A table that is both a system-period temporal table and an application-period temporal table. A user or application maintains a “valid” period of time for each row while DB2 automatically keeps historical versions of each row.

Unlike regular base tables, every logical row in a temporal table can potentially have multiple stored versions. Every row in a temporal table is conceptually associated with two timestamp values, one for capturing the *begin time*, that is, the time when the row was inserted into the table and the other for capturing the *end time*, that is, the time when the row as either updated or deleted. A pair of these columns is used to define a “period”.

Temporal tables enable querying of rows that were current as of a specific point of time in the past. Temporal tables are mutually exclusive with archive transparency. A system-period temporal table or application-period temporal table cannot be defined as an archive-enabled table or archive table.

The temporal features in DB2 for z/OS product provide rich functionality for time-based data management. Using new and standardized SQL syntax, you can easily insert, update, delete, and query data in the past, present, or future.

The temporal features in the DB2 product enable you to accurately track information and data changes over time and provide an efficient and cost-effective way to address auditing and compliance requirements.

1.1.1 Periods

The definition of a temporal table includes a *period*, which is a pair of datetime columns that record the start and end time for each row of data. The column that records the start time for a row is the *row-begin column*, and the column that records the end time for a row is the *row-end column*. A period is a building block for a temporal table. DB2 supports the following periods:

- ▶ **SYSTEM_TIME:** The period maintained by DB2 to record when rows are active in a system-period temporal table.
- ▶ **BUSINESS_TIME:** The period maintained by an application to record the “valid time” for a row in an application-period temporal table, from a business perspective.

A bitemporal table contains both a SYSTEM_TIME period and a BUSINESS_TIME period.

A period indicates the starting and ending points of a time interval. With DB2, administrators identify two columns in a temporal table to indicate the start and end times of a period. Simple extensions to the syntax of the CREATE TABLE and ALTER TABLE statements accomplish this, enabling administrators to employ temporal data support for new or existing tables.

DB2 automatically enforces a constraint to ensure that the value of end column is always greater than the value of begin column. The semantic for the start and end points of a BUSINESS_TIME period is “inclusive-exclusive”. That is, the value of the *begin column* is included in the period (closed) and the value of the *end column* is not included in the period (open).

A period can be explicitly referenced in a *period specification* to indicate the period of time to apply to a table in a query. The following period specifications are supported:

- ▶ FOR *period-name* AS OF *value*
- ▶ FOR *period-name* FROM *value1* TO *value2*
- ▶ FOR *period-name* BETWEEN *value1* AND *value2*

These period specifications are used for both system-period temporal tables, and application-period temporal tables.

An implicit AS OF period specification can be specified by assigning a value to the appropriate temporal special register for the type of temporal table.

See Chapter 5, “Additional considerations” on page 73 for more information about using the temporal special registers.

1.1.2 System-period temporal tables

A *system-period temporal table* is a table defined so that DB2 automatically keeps a history of updated and deleted rows over time. Whenever a row is updated or deleted in a system-period temporal table, a copy of the old row is stored in the historical table by DB2. This is referred to as *system-period data versioning* (Figure 1-1). Using a period specification, queries can now be easily written to access rows from a system-period temporal table that has been deleted or updated. The use of DB2 temporal extensions for system-period temporal tables should be more efficient and easier to manage than current applications that are handling this type of historical versions of rows.

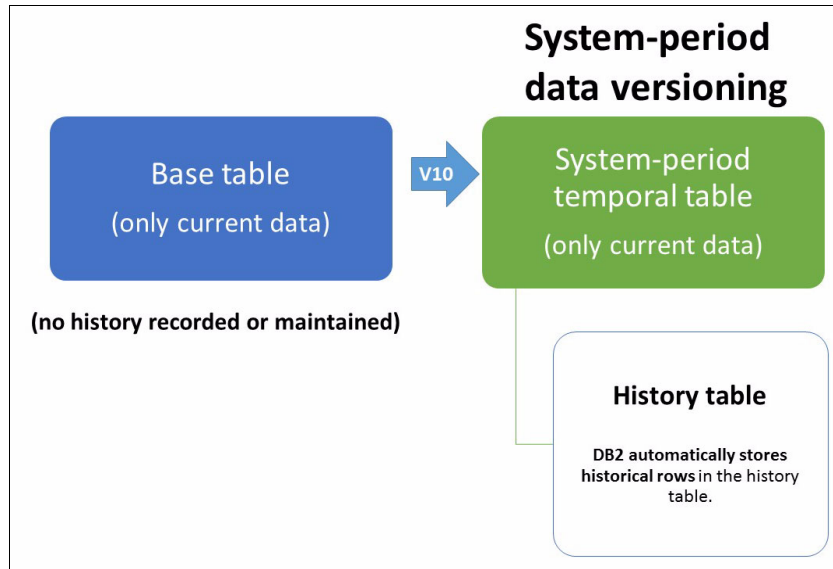


Figure 1-1 System-period data versioning

System-period temporal tables can be useful for clients that have requirements to query historical data, in addition to querying current data. For example:

- ▶ What was the price of a returned piece of merchandise on the day of purchase?
- ▶ What is the historical data for an account on 2015-02-20? The information is needed to resolve a client complaint about an old bill.
- ▶ What are the names of out-of-network specialists who billed for service to the insured client between 2006-05-01 and 2007-10-01?
- ▶ What is the equity position on 2009-02-20 for a client as reported on their 2009-02-28 statement?
- ▶ What is a customer's equity position on 2009-02-20 after all known corrections?
- ▶ What jobs has an employee held during his career?
- ▶ What is the salary history for an employee?

A candidate application for system-period data versioning is one where it is necessary to keep previous states of each row of data. Triggers and stored procedures are currently needed to maintain this historical data. This overhead may affect the runtime performance of such applications in a negative way. Using the DB2 temporal data support for data versioning moves the management of history to the DB2 database, providing consistent support across all applications.

DB2 for z/OS provides system-period data versioning capability using SQL through a two table approach that provides a single table image. The table that contains the current data is called the *system-period temporal table*, and the table that holds the historical data is called the *history table*. DB2 automatically stores historical data in the history table.

Defining a table as a system-period temporal table makes DB2 responsible for preserving the history of database changes, thus, moving the management of historical data from the application layer to the database layer. This results in simplification of application logic and the requirement for using triggers or stored procedures is reduced.

A system-period temporal table includes a system-maintained time period named `SYSTEM_TIME`. DB2 uses the `SYSTEM_TIME` period as rows in the table are modified to automatically preserve historical versions of the rows in the separate history table. A system-period temporal table differs from a “regular” base table in the following ways:

- ▶ The behavior of data manipulation operations on system-period temporal tables is quite different from the behavior on regular base tables. In contrast to the regular base tables, an update operation on a system-period temporal table does not replace affected rows with new rows. Instead, new versions of updated rows are inserted while preserving old versions of rows in the separate history table. Similarly, the delete operation on a system-period temporal table does not remove the affected rows. They too are preserved in the separate history table with an indication that they were deleted.
- ▶ System-period temporal tables include additional columns of a timestamp data type to capture the begin and end times of each row in the table. Values in these columns are set by DB2. In addition, users are not allowed to update the values in these columns. The presence of these additional columns makes it possible to identify the subset of rows that constitutes the content of the table, that is, the version, as of any point in time.
- ▶ A query on a system-period temporal table is allowed to specify a point in time or a period to evaluate the query for. This determines whether the query is evaluated only for current rows, or for current rows and historical rows. If a point in time or period is not specified, a query against a system-period temporal table is evaluated only on the current rows in the table.

With a system-period temporal table, DB2 maintains a history of changes to the content of the table. Conceptually, both the current rows and historical rows exist in a system-period temporal table. This is how the capability is defined in the SQL Standard. However, the DB2 for z/OS implementation involves the use of a separate *history table* to store the historical rows.

A system period named `SYSTEM_TIME` is used by DB2 to record when versions of rows are active or valid for a system-period temporal table. The `SYSTEM_TIME` period contains a pair of datetime columns, which are maintained by DB2 and record the period in which other values in the row are valid from the database or system point of view.

DB2 records in the period columns in the system-period temporal table when the data is “current”. The old rows of a system-period temporal table are kept as a set of historical rows. These historical rows are stored automatically by DB2 in a separate table, called the history table. For a row in the history table the columns, corresponding to the `SYSTEM_TIME` period in the system-period temporal table, record when that version of the row was valid or “active”.

The columns of the `SYSTEM_TIME` period are generated always by DB2. They must be NOT NULL, and they can be defined as either `TIMESTAMP(12)` or `TIMESTAMP(12) WITH TIME ZONE`.

For example:

```
begin-column-name    TIMESTAMP(12)    NOT NULL    GENERATED ALWAYS AS ROW BEGIN  
end-column-name      TIMESTAMP(12)    NOT NULL    GENERATED ALWAYS AS ROW END
```

In the system-period temporal table, the definition of the row-begin and row-end columns, SYSTEM_TIME period, and system-period data versioning attribute are all separate. This separation was designed to support the migration use case, where a table already exists and a user wants to reuse existing datetime columns. It also allows you to drop the system-period data versioning attribute to disable the special processing, while retaining the column and period definitions for future use.

After the tables have been defined, use the following ALTER TABLE statement to enable system-period data versioning and indicate the associated history table.

```
ALTER TABLE .... ADD VERSIONING .. USE HISTORY history-table
```

There is no requirement for separate authorization on the history table for a user that accesses data in the history table indirectly by explicitly referencing the system-period temporal table.

Non-deterministic generated expression columns can be used with a system-period temporal table to provide audit capabilities that are currently managed by applications, triggers, and stored procedures. See Chapter 5, “Additional considerations” on page 73 for more information.

System-period temporal tables can be used to track and query historical and current data in a straightforward and efficient manner. The functionality provided by system-period temporal tables ensures data integrity over time and addresses regulatory and compliance requirements. The use of system-period temporal tables is also simpler than implementing similar capability in applications, triggers, or stored procedures. Simple SQL extensions are used to define a system-period temporal table, and to specify a particular time period in a query.

Querying a system-period temporal table is simple. Optional period specifications can be used to indicate a specific period of time to evaluate a query against a system-period temporal table. A query with a period specification is a temporal query. A query without a period specification is a non-temporal query, and the syntax and semantics of such a query is unchanged when a system-period temporal table is accessed. A non-temporal query against a system-period temporal table is evaluated on the current data in the table, which means that existing applications, stored procedures, and database reports are not affected when an existing table is converted to a system-period temporal table. Use one of the following three period specification constructs in the FROM clause of a temporal query against a system-period temporal table to transparently access historical data, or a combination of current and historical data:

- ▶ FOR SYSTEM_TIME AS OF *value* enables you to query data as of the point in time specified by *value*.
- ▶ FOR SYSTEM_TIME FROM *value1* TO *value2* enables you to query data from a one point in time to another point in time. A row will satisfy this period specification if *value2* is greater than the starting value of the period in the row, and *value1* is less than the end value for the period in the row.
- ▶ FOR SYSTEM_TIME BETWEEN *value1* AND *value2* enables you to query data between a range of start and end times. A row will satisfy this period specification if *value2* is greater than or equal to the starting value of the period in the row, and *value1* is less than the end value for the period in the row.

Alternatively, an implicit time period to use for system-period temporal tables can be specified by assigning a value to the CURRENT TEMPORAL SYSTEM_TIME special register. See Chapter 5, “Additional considerations” on page 73 for more information about using the CURRENT TEMPORAL SYSTEM_TIME temporal special register.

See the SQL Reference for restrictions on what tables can be used as system-period temporal tables or history tables.

1.1.3 Application-period temporal tables

An application-period temporal table is a table for which users or applications specify a time period for each inserted row indicating when that row is “valid” from the perspective of the business. When a row is inserted, the user or application specifies the valid time for the row. To access rows from an application-period temporal table, a period specification can be used to retrieve data that is valid during a specified time period in the past, present, or future. Whenever a row is updated or deleted, the user can indicate a specific time period to apply the data change. DB2 applies the changes and adjusts the existing rows as necessary for any time period for which the row was not modified. If necessary, additional rows are inserted to retain the data for the row for the time period that was not affected by the data change. The use of DB2 temporal extensions for application-period temporal tables should be more efficient and easier to use than current applications that are managing business valid times.

Application-period temporal tables can be useful for application programmers and database administrators that for years have been facing the problem of managing different versions of application data. There is a large class of database applications that are interested in capturing the time periods during which the data is valid in the real world. A primary requirement of such applications is that it is the user, rather than the system, that is in charge of setting the start and end times of the validity period of rows. User-specified values may correspond to any past, current, or future time. The ability to specify future time is especially important in many application scenarios where the database is updated ahead of an event taking place. Another requirement of such applications is the ability to update or delete rows that are valid between any two points in time, which may cause some rows to be updated or deleted only for a portion of their validity period. If a row is updated or deleted only for a part of its validity period, one or more rows need to be inserted to preserve the information for the time period that the change did not apply to. The definition of an application-period temporal table also ensures that multiple rows with the same key value do not exist with overlapping valid periods.

Application-period temporal tables can be useful for customers that want to track and query when certain business conditions are, were, or will be valid. For example:

- ▶ Defining promotional or sale prices for items in a catalog in advance of when the special prices are available.
- ▶ Recording validity periods of insurance policies.
- ▶ Changing credit card interest rates over time.
- ▶ An online travel agency wants to detect inconsistencies in itineraries. If someone books a hotel in Rome for eight days and reserves a car in New York for three of those days, the agency would like to flag the situation for review.

Before DB2 for z/OS introduced support for application-period temporal tables, it was up to users and applications to manage application-level data versioning. This could be financially costly to an organization as a result of an exploding data model, additional application complexity, and the addition of error prone code to applications. The lack of data versioning in DB2 also prevented the protection and management of core business sensitive assets by DB2.

DB2 introduced initial support for application-period temporal tables in DB2 V10, with table-level specifications to control the management of application data based on time. Application programmers can now specify queries with search criteria based on the time frame that the data existed. This capability simplifies and reduces the cost of developing DB2 applications that require tracking differing data values over time, and it enables customers to meet new compliance laws faster and less expensively because DB2 automatically manages the different versions of data.

An application-period temporal table includes a user-specified period named `BUSINESS_TIME`. An application-period temporal table differs from a “regular” base table in the following ways:

- ▶ The behavior of data manipulation operations on application-period temporal tables is different from the behavior on regular base tables. In contrast to the regular base tables, a data change operation on an application-period temporal table may result in the insertion of additional rows into the table for “row splits” when the change applies only to a portion of the valid time for a row.
- ▶ Application-period temporal tables include additional columns of a datetime data type to capture the begin and end times of each row in the table. Values in these columns are initially provided by the user.
- ▶ A query on an application-period temporal table is allowed to specify a point in time or a period to apply to the query.

An application period named `BUSINESS_TIME` records when versions of rows are active or valid for an application-period temporal table. The `BUSINESS_TIME` period contains a pair of datetime columns that are initially specified by the user, and are adjusted by DB2 when data changes are specified for a portion of the period for which the row is valid from the business point of view.

The user records the beginning of the time period when the row is considered valid in the begin column, and when the row is no longer considered valid in the end column. The columns of the `BUSINESS_TIME` period must be defined as NOT NULL and DATE or TIMESTAMP(6).

For example:

```
begin-column-name    TIMESTAMP(6)    NOT NULL
end-column-name      TIMESTAMP(6)    NOT NULL
```

The definition of the begin column and end column, and the `BUSINESS_TIME` period are separate. This separation was designed to support the migration use case, where a table already exists and a user wants to reuse existing datetime columns.

Application-period temporal tables can be used to track and query valid times for data in a straightforward and efficient manner. The functionality provided by application-period temporal tables ensures data integrity over time. The use of application-period temporal tables is also simpler than implementing similar capability in applications, triggers, or stored procedures. Simple SQL extensions are used to define an application-period temporal table, and to specify a particular time period in a query.

Querying an application-period temporal table is simple. Optional period specifications can be used to indicate a specific period of time to evaluate a query against an application-period temporal table. A query with a period specification is a temporal query. A query without a period specification is a non-temporal query, and the syntax and semantics of such a query is unchanged when an application-period temporal table is accessed. Thus, existing applications, stored procedures, and database reports are not affected when an existing table is converted to an application-period temporal table.

Use one of the following three period specification constructs in the FROM clause of a temporal query against an application-period temporal table to retrieve data for past, current, and future business conditions:

- ▶ FOR BUSINESS_TIME AS OF *value* enables you to query data as of the point in time specified by *value*.
- ▶ FOR BUSINESS_TIME FROM *value1* TO *value2* enables you to query data from a one point in time to another point in time. A row will satisfy this period specification if *value2* is greater than the starting value of the period in the row, and *value1* is less than the end value for the period in the row.
- ▶ FOR BUSINESS_TIME BETWEEN *value1* AND *value2* enables you to query data between a range of start and end times. A row will satisfy this period specification if *value2* is greater than or equal to the starting value of the period in the row, and *value1* is less than the end value for the period in the row.

Alternatively, an implicit time period to apply to application-period temporal tables can be specified by assigning a value to the CURRENT TEMPORAL BUSINESS_TIME special register. See Chapter 5, “Additional considerations” on page 73 for more information about using the CURRENT TEMPORAL BUSINESS_TIME temporal special register.

See the SQL Reference for restrictions on what tables can be used as application-period temporal tables.

1.1.4 Bitemporal tables

A table that is both a system-period temporal table and an application-period temporal table is called a *bitemporal table*. A bitemporal table provides the benefits of both system-period temporal tables and application-period temporal tables. It records both system-provided historical data and application-based validity period information. Using a bitemporal table provides an application with an easy way to manage the business validity of their data, while DB2 maintains a full history of data changes.

A bitemporal table contains both a SYSTEM_TIME period and a BUSINESS_TIME period.

1.1.5 Choosing the right type of temporal table

You might be wondering what type of temporal table would best meet your business needs. The choice depends on the dimension of time that you want to track and the type of temporal queries that you need to support. The following table compares the key characteristics of system-period temporal tables and application-period temporal tables to help you make the correct choice (Table 1-1).

Table 1-1 Choosing what type of temporal table to use

Characteristics of a system-period temporal table	Characteristics of an application-period temporal table
Captures the time when changes happen to a row of data	Captures the time when changes happen to business objects in the real world
Automatically maintains a history of updated and deleted rows	Maintains application-driven changes to the time dimension of business objects
History recorded based on DB2 timestamps	Dates or timestamps provided by the user or application

Characteristics of a system-period temporal table	Characteristics of an application-period temporal table
DB2 view of physical time	Application logical view of time
Time spans from the past to the present	Time spans past, present, and future time
System validity (transaction time)	Business validity (valid time)

Choose the type of temporal table that best fits your needs:

- ▶ **System-period temporal table** to track and maintain a history of when information was physically inserted, updated, or deleted inside your DB2 database. A system-period temporal table is a good choice for applications that need to answer the following type of questions:
 “What was the policy in our database on June 30?”
- ▶ **Application-period temporal table** to describe when information is valid in the real world, outside DB2. An application-period temporal table is a good choice for applications that need to answer the following type of questions:
 “Which policies were valid on June 30?”
- ▶ **Bitemporal table** if you need to track both dimensions of time. With a bitemporal table, you can manage business valid time with full traceability of data changes and corrections. A bitemporal table is a good choice for applications that need to answer the following types of questions:
 “Which policies were stored in the database on June 30?”
 and:
 “Which policies were valid on June 30?”



System-period temporal tables

In this chapter, we look at where and how to implement system-period temporal tables.

This chapter contains the following sections:

- ▶ Overview
- ▶ Relationship of a system-period temporal table to the associated history table
- ▶ Moving from traditional to system-period temporal tables
- ▶ The who, what, and how of my row (audit columns)
- ▶ Querying a system-period temporal table
- ▶ General restrictions with system-period data versioning
- ▶ Utilities
- ▶ Recovery

2.1 Overview

Regulations require that historical data be kept for years. Every update and delete of data requires that old data be retained. The use of history tables is one of the ways used to hold this data. Application programs need to conform to the rules to maintain and access to time-varying data. The applications become more complex and difficult to maintain.

In many places, the recording of historical data is managed by the use of two tables and a set of AFTER triggers. This does help with reducing the complexity of the application program and support consistent handling of data changes.

DB2 for z/OS supports *system-period temporal tables* for system-period data versioning. The built-in temporal support removes the need for triggers and provides a consistent approach for the handling of data where the ability to see data at a given point in time is required. The DB2 support for system-period data versioning uses a two table approach that provides a single table image. The table that contains the current data is called the system-period temporal table, and the table that holds the historical data is called the history table. DB2 automatically stores historical data in the history table:

- ▶ When a row is INSERTED
The new row is placed into the base table and there is no change to the history table
- ▶ When a row is DELETED
The row is deleted from the base table and is inserted into the history table
- ▶ When a row is UPDATED
The row in the base table is updated and the old base table row image is inserted into the history table

A system-period temporal table includes a period named SYSTEM_TIME. DB2 uses the SYSTEM_TIME period as rows in the table are modified to automatically preserve historical versions of the rows in the separate history table. Queries against system-period temporal tables can make use of a “period specification” to provide easy access to rows that have been deleted or updated.

Note: System-period temporal is all about dates and timestamps. For simplicity in this chapter where only a date is provided, for example 9999-12-30, assume this to be equivalent to the timestamp 9999-12-30-00.00.00.000000000000. The date format used is yyyy-mm-dd.

2.2 Relationship of a system-period temporal table to the associated history table

A history table is a table associated with a system-period temporal table, and it is used to save the historical rows. The structure of the two tables must be very similar. The two tables must have the same number and order of columns. The following attributes for the corresponding columns of the two tables must be the same:

- ▶ Name
- ▶ Data type
- ▶ Length (excluding inline LOB lengths), precision, and scale
- ▶ FOR BIT, SBCS, or MIXED DATA attribute for character string columns
- ▶ Null attribute

- ▶ Hidden attribute
- ▶ CCSID
- ▶ Field procedure

The history table must not contain the following types of columns:

- ▶ Identity column
- ▶ Row change timestamp column
- ▶ Row-begin column
- ▶ Row-end column
- ▶ Transaction-start-ID column
- ▶ Generated expression column
- ▶ Security label column

The history table must not include a period, have a row permission defined, a column with a column mask defined, or have an incomplete status. Table 2-1 summarizes the relationship of the columns of the two tables:

Table 2-1 Relationship of columns

Type of column	Column definition in the system-period temporal table	Column definition in the history table Note: The column in the history table is defined with the same data type as the column in the system-period table, but without the GENERATED attributes.
row begin	TIMESTAMP (12) WITHOUT TIME ZONE NOT NULL GENERATED ALWAYS AS ROW BEGIN	TIMESTAMP (12) WITHOUT TIME ZONE NOT NULL
row end	TIMESTAMP (12) WITHOUT TIME ZONE NOT NULL GENERATED ALWAYS AS ROW END	TIMESTAMP (12) WITHOUT TIME ZONE NOT NULL
transaction start ID	TIMESTAMP (12) WITHOUT TIME ZONE GENERATED ALWAYS AS TRANSACTION START ID	TIMESTAMP (12) WITHOUT TIME ZONE
non-deterministic expression column	<i>data-type</i> GENERATED ALWAYS AS (<i>expression</i>)	<i>data-type</i>
rowid column	ROWID GENERATED ALWAYS	ROWID GENERATED ALWAYS
other columns	<i>data-type</i>	<i>data-type</i>

The corresponding columns of the system-period temporal table and associated history table must be defined as the same data type when *data-type* is used in the above table.

WITHOUT TIME ZONE does not need to be specified in the definition of a TIMESTAMP column as it is the default.

The row-end column in the system-period temporal table will always contain a maximum value for the data type of the column. The corresponding column in the history table is used to record when a row in the system-period temporal table is updated or deleted, and the historical row is moved into the history table. The system-period temporal table and associated history table are required to contain the same number of columns to provide a “single table” image to the user.

2.3 Moving from traditional to system-period temporal tables

This section describes moving to system-period temporal tables.

2.3.1 Creating new tables

If the new data model has a requirement to keep a history of changed rows, when you write your data definition statement to define for a system-period temporal table:

- ▶ Include the special system-period temporal table columns.
- ▶ Create a table that is like the base table to keep the history. This table is referred to as the *history table*.
- ▶ Enable versioning and identify the associated history table:

```
ALTER TABLE tb1 ADD VERSIONING USE HISTORY TABLE tb1_hist
```

The system-period temporal table and history table must reside in separate table spaces. Both the history and the base table space can be partitioned and they can be partitioned differently.

System period columns in bold are included in the base table (Example 2-1).

Example 2-1 System period columns

```
CREATE TABLE DB2AUTH.EMPS_S0
(EMPID    BIGINT NOT NULL,
 NAME     VARCHAR(20)  WITH DEFAULT NULL,

  DEPTID   INTEGER WITH DEFAULT NULL,
  SALARY   DECIMAL(7, 2) WITH DEFAULT NULL,
  SYSTEM_START      TIMESTAMP (12) WITHOUT TIME ZONE NOT NULL
    GENERATED ALWAYS AS ROW BEGIN,
  SYSTEM_END       TIMESTAMP (12) WITHOUT TIME ZONE NOT NULL
    GENERATED ALWAYS AS ROW END,
  TRANS_START      TIMESTAMP (12) WITHOUT TIME ZONE
    GENERATED ALWAYS AS TRANSACTION START ID,
  PERIOD SYSTEM_TIME (SYSTEM_START, SYSTEM_END),
  CONSTRAINT EMPID
  PRIMARY KEY (EMPID))
```

The TRANSACTION START ID column has minimal usage with DB2 10 and DB2 11. In general, it is defined as nullable and will not contain any values. If the column is defined as NOT NULL, the column will be the transaction start timestamp, which is the same as the value of the ROW BEGIN column. It is not used in system period queries. The usage may change with later versions of DB2.

While the system period columns are defined as timestamp (12), `TIMESTAMP(9)` is used with three trailing zeros for non-data sharing environments. For a data sharing environment, the trailing 3 digits identify the data sharing member ID used.

Example 2-2 shows an example being taken from a history table.

Example 2-2 History table

SYSTEM_START	SYSTEM_END
-----	-----
2015-06-26-12.09.46.598762794001	2015-06-26-12.16.51.665380751002

In this case, the original row was created on member 1 and the action that caused a write to the history table was on member 2. If the `TRANS_START` column was defined as `NOT NULL`, it would contain the same value as the `SYSTEM_START` column.

2.3.2 The history table and version enabling

Without the history table and versioning enabled, the base table with the temporal columns is usable, and any insert and updates on the base table will generate values for the `START_TS` and `END_TS`. This state is not recommended and does impact the integrity of your history table reflecting any data changes (Example 2-3).

Example 2-3 History table

SYSTEM_START	SYSTEM_END	DEPTID
-----	-----	-----
2015-06-17-17.00.35.609605073001	9999-12-30-00.00.00.000000000000	1

The history table must have the same number and order of columns as the table that is to be used as a system-period temporal table.

The history table can be created by using

```
CREATE TABLE emp_s0_hist LIKE emp_s0 in DATABASE apptmpd ;
```

An alternative to defining the history table with the `LIKE` clause (as shown above) is to create it from scratch with a standard `CREATE TABLE` statement.

The history table cannot be dropped if versioning is enabled. However, to prevent any unintended loss of history tables, it is recommended that the `RESTRICT ON DROP` option be specified as part of the table definition.

The base table (system-period temporal table) and the history table can be partitioned differently.

After versioning is enabled, any physical changes to history tables such as partitioning will require versioning to be disabled first. For example, if you start with the history table as a partition-by-growth (PBG), and you want to convert to a partition by range (PBR), you will need to disable versioning. Make your changes including reloading the history table if required, and then re-enable versioning. To maintain integrity, the base table should be placed in read-only (RO) during the change.

History table rows

Only changes to committed data are logged in the history table. If a single unit of work repeatedly updates the same row, this does not result in multiple rows in the history table.

The generation of history rows results in one history row per transaction (UOW) in a system-period temporal table as opposed to a trigger-based solution that records one row per SQL statement that affects the row. If the same row is updated multiple times in a single transaction, a trigger generates a history row for each of these updates. As a result, the history table contains intermediate versions of the row that were never committed in the database.

Version enablement

After the history table has been created, versioning can be enabled by using the following ALTER TABLE statement:

```
ALTER TABLE emps_s0
  ADD VERSIONING USE HISTORY TABLE emps_hist_s0;
```

The ADD VERSIONING clause enables system-period data versioning, and the USE HISTORY TABLE clause identifies the associated history table.

At this time, you now see the history table with a TYPE(column T below) value of H in the DB2 catalog (Example 2-4).

Example 2-4 System-period temporal table and history table catalog extract

Name	Schema	T	DB Name	TS Name	Cols
EMPS_S0	DB2AUTH	T	APPTMPD	EMPSRS0	7
EMPS_S0_HIST	DB2AUTH	H	APPTMPD	EMPSRSOR	7

When you disable versioning, the type value will be reset to T to indicate that the table is a normal table.

Implicitly hidden columns

If you did not want to have the system-period temporal table columns directly visible to SQL using SELECT *, and to load or unload utilities, you can define the columns with the IMPLICITLY HIDDEN attribute.

```
ADD COLUMN system_start TIMESTAMP(12) NOT NULL GENERATED AS ROW BEGIN IMPLICITLY
HIDDEN
```

The use of the hidden attribute for the special temporal columns will allow existing applications to see the same query results as before the migration. No changes to existing queries or insert, update, and delete statements are required.

2.3.3 Enabling versioning for an existing non-temporal table

Transitioning an existing table to a system-period temporal table where currently there is no collection of historical data can be reasonably straight forward. The process can be managed by using the following SQL statements (see Example 2-5 on page 19):

- ▶ ALTER TABLE statement to add the special columns and define the SYSTEM_TIME period
- ▶ CREATE TABLE to define the history table
- ▶ ALTER TABLE statement to enable system-period data versioning and identify the history table

Example 2-5 Define an existing table as a system-period temporal table

```
ALTER TABLE emps_s0
ADD COLUMN system_start TIMESTAMP(12) NOT NULL GENERATED AS ROW BEGIN
ADD COLUMN system_end TIMESTAMP(12) NOT NULL GENERATED AS ROW END
ADD COLUMN trans_start TIMESTAMP(12) GENERATED AS TRANSACTION START ID
ADD PERIOD SYSTEM_TIME (system_start, system_end);

CREATE TABLE emps_s0_hist LIKE emps_s0;

ALTER TABLE emps_s0
ADD VERSIONING USE HISTORY TABLE emps_s0_hist;
```

The SYSTABLES catalog information for the two tables will look like the following (see Example 2-6).

Example 2-6 SYSTABLES catalog information for the system-period temporal table and history table

EMPS_S0 <<< The system-period temporal table

Column Name	Col	No	Col Type	Length	Scale	Null	Def
*	*	*		*		*	*
-----	-----	-----	-----	-----	-----	-----	-----
EMPID	1		BIGINT	8	0	N	N
NAME	2		VARCHAR	20	0	Y	Y
DEPTID	3		INTEGER	4	0	Y	Y
SALARY	4		DECIMAL	7	2	Y	Y
SYSTEM_START	5		TIMESTMP	13	12	N	Q
SYSTEM_END	6		TIMESTMP	13	12	N	R
TRANS_START	7		TIMESTMP	13	12	Y	X

EMPS_S0_HIST <<< The history table

Column Name	Col	No	Col Type	Length	Scale	Null	Def
*	*	*		*		*	*
-----	-----	-----	-----	-----	-----	-----	-----
EMPID	1		BIGINT	8	0	N	N
NAME	2		VARCHAR	20	0	Y	Y
DEPTID	3		INTEGER	4	0	Y	Y
SALARY	4		DECIMAL	7	2	Y	Y
SYSTEM_START	5		TIMESTMP	13	12	N	N
SYSTEM_END	6		TIMESTMP	13	12	N	N
TRANS_START	7		TIMESTMP	13	12	Y	Y

You can see here from the catalog information that the default values (Def) for the columns of the base table (emps_s0) reflect their special use for the system-period temporal table:

- ▶ Q - ROW BEGIN
- ▶ R - ROW END
- ▶ X - TRANSACTION START ID

When you add the timestamp columns to an existing table, all the existing rows get the value 9999-12-30-00.00.00.000000000000 in the system_end column, indicating that all rows are current rows. However, DB2 does not know when these rows were originally inserted and what their system_start values should be. Hence, all existing rows initially get the system_start value 0001-01-01-00.00.00.000000000000, which is January 1 in the year 0001.

If you prefer to use the current time as the system start time for all existing rows, you have two options to achieve this:

- Add the `system_start` column with the wanted default value first, followed by `DROP DEFAULT` and then issue another `ALTER TABLE` statement to change the column to `GENERATED AS ROW BEGIN` (Example 2-7).

Example 2-7 Alter table

```
ALTER TABLE emps_s0 ADD COLUMN system_start TIMESTAMP(12) NOT NULL DEFAULT
CURRENT_TIMESTAMP

ALTER TABLE EMPS_S0 ALTER COLUMN system_start DROP DEFAULT ;

ALTER TABLE EMPS_S0 ALTER COLUMN system_start SET GENERATED ALWAYS AS ROW BEGIN
;
```

- You can unload and load all rows *before* you enable versioning.

2.3.4 Querying a system-period temporal table

A query can be written against a system-period temporal table (Example 2-8).

Example 2-8 Query

```
SELECT empid, name, deptid, salary, system_start, system_end
FROM emps_s0
WHERE empid = 544 ;
```

This query only returns results of rows from the base table as of the value of current time. Historical data will not be considered for this query (Table 2-2).

Table 2-2 Results

Empid	Name	Deptid	Salary	System_start	System_end
544	BRUCE	2	20000.00	0001-01-01-00.0 0.00.000000000 000	9999-12-30-00.0 0.00.000000000 000

Note that as this row existed before the table was converted to a system-period temporal table, the default values for the data type were used for the values of the `system_start` and `system_end` columns.

When a new row is added for Mary, the `system_start` column value is set by DB2 to the timestamp at the time of the insert (Table 2-3).

Table 2-3 Results

Empid	Name	Deptid	Salary	System_start	System_end
544	BRUCE	2	20000.00	0001-01-01-00.0 0.00.000000000 000	9999-12-30-00.0 0.00.000000000 000
800	MARY	2	20000.00	2015-06-19-19.4 2.55.400358574 001	9999-12-30-00.0 0.00.000000000 000

2.3.5 Enabling versioning for an application managed temporal table

As we have seen in the previous section, the transitioning to a system-period temporal table where currently there is no collection of historical data is reasonably straightforward. When you are already collecting history data within the one table or with multiple tables, the transition process becomes more complex and in many cases may not be considered viable.

The exact steps to migrate an existing application solution to use system-period temporal tables depends on the characteristics of your existing tables and data, such as:

- ▶ Whether you are already recording history rows, for example with triggers
- ▶ Whether you use a single table to hold current and historical data or two separate tables
- ▶ Whether you are using timestamps for versioning
- ▶ If you are using timestamps, are there one or two timestamps for each version of the row? Are you maintaining other details for each version, such as who and what changed the row?

Our scenario has separate tables recording history data using triggers. The general process to migrate follows along these lines:

- ▶ Continue using two tables.
- ▶ Dropping the triggers.
- ▶ Modify the existing tables to become a system-period temporal table and associated history table for built-in system-period data versioning.
- ▶ Modify application SQL and logic dependent on the time period to reduce the use of joins and complex SQL.
- ▶ Add audit support. This is covered in 2.4, “The who, what, and how of my row (audit columns)” on page 26.

Assume that the tables for an existing application managed temporal solution were defined with the following CREATE TABLE statements (Example 2-9).

Example 2-9 Data definition statements for traditional tables that keep track of history

```
CREATE TABLE EMPS_S1 (  
  EMPID          BIGINT NOT NULL ,  
  NAME           VARCHAR(20),  
  DEPTID         INTEGER,  
  SALARY         DECIMAL(7,2),  
  SYSTEM_START   TIMESTAMP(6) NOT NULL DEFAULT ,  
  SYSTEM_END     TIMESTAMP(6) NOT NULL DEFAULT '3000-01-01-00:00:00.000000' )  
IN DATABASE APPTMPD ;
```

```
CREATE TABLE EMPS_S1_HIST (  
  EMPID          BIGINT NOT NULL,  
  NAME           VARCHAR(20),  
  DEPTID         INTEGER,  
  SALARY         DECIMAL(7,2),  
  SYSTEM_START   TIMESTAMP(6) NOT NULL,  
  SYSTEM_END     TIMESTAMP(6) NOT NULL )  
IN DATABASE APPTMPD  
;
```

We assume that appropriate AFTER DELETE and AFTER UPDATE triggers are defined on the table emps_s1 to insert the before images of updated and deleted rows into the table emps_s1_hist.

Drop any triggers that create history rows

Because DB2 generates history rows for updated or deleted rows automatically, you should remove any triggers that generate history rows in your existing solution. Drop the triggers at the beginning of the migration process to avoid the unnecessary generation of (possibly) incorrect history rows, in case any rows need to be updated as part of the migration process itself.

Migrate the table definition and data

The existing table definitions that we assume in this scenario differ from a system-period temporal table in the following properties:

- ▶ The existing system_end date value of the timestamp for current rows is 3000-01-01, but should be 9999-12-30 in a system-period temporal table.
- ▶ The data type of the existing system_start and system_end columns is TIMESTAMP(6), but the data types of the columns must be TIMESTAMP(12) for a system-period temporal table.
- ▶ System-period temporal tables must have a transaction-start-ID column in the base table and a timestamp column in the history table.
- ▶ The definitions of the system_start and system_end columns in the existing table are different than what DB2 requires in a system-period temporal table. The columns in the existing table are defined as follows:
 - DEFAULT CURRENT TIMESTAMP
 - DEFAULT TIMESTAMP '...'

Whereas, in a system-period temporal table, the columns of the period must be defined as GENERATED ALWAYS AS ROW BEGIN or ROW END.

- ▶ A system-period temporal table requires a SYSTEM_TIME period.
- ▶ For a system-period temporal table, versioning must be enabled and the history table must be identified.

The following statements address these differences and convert the existing emps_s1 table into a system-period temporal table (Example 2-10).

Example 2-10 Converting a traditional table for an application managed temporal solution to a system-period temporal table

1. Change the system_end values to the same value that DB2 generates.

```
UPDATE emps_s1 SET system_end = '9999-12-30';
```

2. Change data types to TIMESTAMP(12) and add the transaction-start-ID column

```
ALTER TABLE EMPS_S1
ALTER COLUMN SYSTEM_START SET DATA TYPE TIMESTAMP(12)
ALTER COLUMN SYSTEM_END SET DATA TYPE TIMESTAMP(12)
ADD COLUMN TRANS_START TIMESTAMP(12)
GENERATED ALWAYS AS TRANSACTION START ID IMPLICITLY HIDDEN
;
ALTER TABLE EMPS_S1_HIST
ALTER COLUMN SYSTEM_START SET DATA TYPE TIMESTAMP(12)
ALTER COLUMN SYSTEM_END SET DATA TYPE TIMESTAMP(12)
ADD COLUMN TRANS_START TIMESTAMP(12) IMPLICITLY HIDDEN;
```

3. Set the auto generation of the columns `system_start` and `system_end`

```
ALTER TABLE EMPS_S1
  ALTER COLUMN SYSTEM_START DROP DEFAULT ;
ALTER TABLE EMPS_S1
  ALTER COLUMN SYSTEM_START
  SET GENERATED ALWAYS AS ROW BEGIN ;
```

```
ALTER TABLE EMPS_S1
  ALTER COLUMN SYSTEM_END DROP DEFAULT ;
ALTER TABLE EMPS_S1
  ALTER COLUMN SYSTEM_END
  SET GENERATED ALWAYS AS ROW END ;
```

4. Adding the `PERIOD SYSTEM_TIME` declaration.

```
ALTER TABLE EMPS_S1
  ADD PERIOD SYSTEM_TIME (SYSTEM_START, SYSTEM_END);
```

5. Reorg the tables

```
REORG TABLE EMPS_S1;
REORG TABLE EMPS_S1_HIST;
```

6. Enable versioning and identify the associated history table

```
ALTER TABLE EMPS_S1 ADD VERSIONING USE HISTORY TABLE EMPS_S1_HIST;
```

Detailed steps

Following are the steps to migrate to a system-period temporal table in more detail:

1. Adjust existing timestamp values

Updating the existing `system_end` values from 3000-01-01 to the same value that DB2 generates (9999-12-30) is highly recommended but not strictly necessary for the migration process. To be able to identify current rows by a single common `system_end` value after the migration, that value must be 9999-12-30.

Updating many rows in a single statement might require much log space. To avoid a log-full condition, ensure that your log is large enough or update the rows in a series of smaller batches with intermediate commits. A LOAD utility could be an option where you unload with the new value and then load replace the table.

Note that DB2 does not use the value 9999-12-31 as the `system_end` value for current rows. The reason is that the value 9999-12-31 might change to a value in the year 10000 if an application converts it to a different time zone. This is undesirable because a date with a five-digit year cannot be inserted or loaded in DB2 again. Additionally, to include rows with a date of 9999-12-31 in a query using a temporal range would not be possible as the end period is exclusive.

2. Adjust the attributes of existing timestamp columns

When you increase the precision of the `system_start` and `system_end` columns from `TIMESTAMP(6)` to `TIMESTAMP(12)`, existing values in these columns are automatically cast and padded with six additional zeros. For example, the value 2010-05-11 12:00:00.000000 becomes 2010-05-11 12:00:00.000000000000. When using DB2 data sharing, the `timestamp(12)` value is composed of a `timestamp(9)` value concatenated with 3 trailing bytes identifying the data sharing member.

The trans_start column is defined as IMPLICITLY HIDDEN so it will not show up in the result set of "SELECT *" queries. But the column can still be retrieved or compared if you use its column name explicitly in the SELECT or WHERE clause.

The changes need to apply to both the base and the history table. Note that this step could be done before step 1.

3. Alter existing timestamp columns to the required attributes

A system-period temporal table must include columns defined as follows:

- TIMESTAMP(12) NOT NULL GENERATED ALWAYS AS ROW BEGIN
- TIMESTAMP(12) NOT NULL GENERATED ALWAYS AS ROW END
- TIMESTAMP(12) GENERATED ALWAYS AS TRANSACTION START ID IMPLICITLY HIDDEN

Several ALTER TABLE statements may be needed to change the attributes of existing columns to make them acceptable for a system-period temporal table. IMPLICITLY HIDDEN is specified for the transaction start ID column to reduce need for application changes.

When the system_start and system_end columns have been changed to GENERATED ALWAYS AS ROW BEGIN or ROW END, users can no longer provide values for these columns in insert and update operations. Instead, DB2 always generates a timestamp value for these columns for the transaction in which the insert and update operations took place.

4. Adding the PERIOD SYSTEM_TIME

Adding the PERIOD SYSTEM_TIME declaration fails if the columns involved do not have the required attributes.

5. Consider running REORG and RUNSTATS

A REORG of both tables is required before any data change statements can be executed. You might also want to update statistics at this time by issuing the RUNSTATS command on the base and on the history table separately.

6. Enable system-period data versioning

Explicit activation of versioning is required for the table to be a system-period temporal table so that history is automatically recorded, and temporal queries are supported. It is important that the columns on the history table match the name and data type of the base table. See 2.2, "Relationship of a system-period temporal table to the associated history table" on page 14 for more details.

Application changes

Existing applications that read or write to your tables may or may not require minor changes, depending on how exactly they access the tables. For example, a Java application is not affected by the data type change from TIMESTAMP(6) to TIMESTAMP(12).

INSERT, UPDATE, DELETE, and MERGE operations

In this migration scenario, most if not all INSERT, UPDATE, DELETE, or MERGE statements continue to work unmodified. The reason is that in the existing application managed temporal solution the values for the system_start and system_end columns were automatically supplied by default values, so applications did not provide values for these columns explicitly. After the migration to a system-period temporal table, DB2 continues to generate values for these columns automatically. Also, in the existing solution, history rows were created by a database trigger, and the triggers are replaced by the automatic generation of history rows by DB2. Again, no changes on the application side are required.

If an application contains INSERT or UPDATE statements that write to the history table, or provide values for the system_start and system_end columns in the current table, those statements need to be changed. The reason is that DB2 performs these writes automatically for you.

Queries

Any application that explicitly tests for the previous system_end value, either in application code or in an SQL WHERE clause, such as WHERE system_end = TIMESTAMP ('3000-01-01'), should be changed to test for the DB2 generated system_end timestamp value 9999-12-30 instead. Many common temporal queries will still work unchanged, but can be greatly simplified.

Audit support

If you need to record the user ID or application ID for every change to a row, the following section shows you how to do this: 2.4, “The who, what, and how of my row (audit columns)” on page 26.

When the history table also contains existing rows

If your existing solution stores current rows redundantly in the base table and the history table, you must delete the current rows from the history table. Otherwise, temporal queries might return incorrect results.

When the history table contains rows with overlapping periods

For a given key value, such as an empID value in our employee example, the history table must not contain two or more rows with the same key and overlapping periods. Overlapping periods for the same key value means that at some point, there were two current rows for the same primary key in the current table, which would be inconsistent. As a result, temporal queries might return unexpected results.

If you are not sure, you should verify that your existing history data does not contain overlapping values for the SYSTEM_TIME period columns. The following query returns a row for each case of overlapping value, if any exist (Example 2-11).

Example 2-11 A query that detects temporal overlaps

```
SELECT empID,  
       previous_end AS overlap_start,  
       system_start AS overlap_end  
FROM  
    (SELECT empID, system_start, system_end, MIN(system_end)  
     OVER (PARTITION BY empID ORDER BY system_start  
           ROWS BETWEEN 1 PRECEDING AND 1 PRECEDING)  
     AS previous_end  
     FROM emps_s1_hist)  
WHERE system_start < previous_end;
```

2.3.6 Other scenarios for migrating to system-period temporal tables

Several scenarios can be covered for migrating to a system-period temporal table. We just covered a couple of scenarios with a few of the things that you need to deal with. These examples and more are covered in the following articles:

- *Adopting temporal tables in DB2, Part 1: Basic migration scenarios for system-period tables*

http://www.ibm.com/developerworks/data/library/techarticle/dm-1210temporaltable_sdb2

- *Adopting temporal tables in DB2, Part 2: Advanced migration scenarios for system-period temporal tables*

http://www.ibm.com/developerworks/data/library/techarticle/dm-1210temporaltable_sdb2pt2

2.4 The who, what, and how of my row (audit columns)

The implementation of system-period data versioning provided the history of when the row was changed but did not provide *who*, *what*, and *how* it was changed. At the end of 2014, an APAR was delivered that provided additional function to provide this information: *Temporal Auditing Support: APAR PM99683*. This support is only for DB2 V11 on and is not available on DB2 v10.

Without this change, there would still be a need to maintain triggers to capture this information. More details about auditing features and DELETE support can be found in Chapter 5, “Additional considerations” on page 73.

With this functionality being available, we can now add or modify columns with the additional syntax to auto generate information as below:

- The *who* modified the data:

Last_user VARCHAR(128) GENERATED ALWAYS AS (SESSION_USER) ,

- The *how*:

Last_operation CHAR (1) GENERATED ALWAYS AS (DATA CHANGE OPERATION) ,

- The *what* can be defining one or more of the many additional available expressions for auditing purposes (Example 2-12).

Example 2-12 Expressions

```
CHGACCT VARCHAR(255) GENERATED ALWAYS AS (CURRENT_CLIENT_ACCTNG),
CHGAPPL VARCHAR(255) GENERATED ALWAYS AS (CURRENT_CLIENT_APPLNAME),
CHGCORR VARCHAR(255) GENERATED ALWAYS AS (CURRENT_CORR_TOKEN),
CHGUID VARCHAR(255) GENERATED ALWAYS AS (CURRENT_CLIENT_USERID),
CHGWKST VARCHAR(255) GENERATED ALWAYS AS (CURRENT_CLIENT_WRKSTNNAME),
CHGSVRV CHAR (16) GENERATED ALWAYS AS (CURRENT_SERVER),
CHGSQLID VARCHAR(128) GENERATED ALWAYS AS (CURRENT_SQLID),
```

2.4.1 Additional considerations for audit information and delete operations

In order to capture in the history table that a delete operation triggered an entry in the history table, we need an extra row to reflect the value for the column defined as DATA CHANGE OPERATION. This requires the following to be added to the ADD VERSIONING clause. The ON DELETE ADD EXTRA ROW clause is specified in addition to defining the extra column. For example:

```
ALTER TABLE EMPS_S2 ADD VERSIONING
USE HISTORY TABLE EMPS_S2_HIST ON DELETE ADD EXTRA ROW ;
```

One row records the history resulting from the delete operation issued by the user; a second row records additional information about the delete itself, storing the ID of the user that initiated the delete operation. The additional row is recorded in the history table because the ON DELETE ADD EXTRA ROW clause was specified in the definition of the system-period temporal table. The values of the columns in the history table that correspond to the row-begin and row-end columns, have the same value as the first row, reflecting the time of the deletion and the values of the initial row.

2.4.2 The “who”, “what”, and “how” in action

Consider the following example (Example 2-13), which modifies the existing emp_s0 table so that it is defined as a system-period temporal table. We add additional columns to generate the answer to our questions. We create the history table with the additional columns before versioning is enabled. After the columns have been added, versioning can be enabled with another ALTER TABLE statement that includes the ADD VERSIONING clause, along with the additional ON DELETE ADD EXTRA ROW clause.

Remember that any column can use the attribute IMPLICITLY HIDDEN to hide the column from SELECT *.

Example 2-13 Add the who, what, and how to an existing table

```
ALTER TABLE EMPS_S2
ADD COLUMN SYSTEM_START TIMESTAMP(12) NOT NULL GENERATED AS ROW BEGIN
ADD COLUMN SYSTEM_END TIMESTAMP(12) NOT NULL GENERATED AS ROW END
ADD COLUMN TRANS_START TIMESTAMP(12) GENERATED AS TRANSACTION START ID
ADD COLUMN CHGAPPL VARCHAR(255) GENERATED ALWAYS AS (CURRENT CLIENT_APPLNAME)
ADD COLUMN LAST_USER VARCHAR(8) GENERATED ALWAYS AS (CURRENT SQLID)
ADD COLUMN LAST_OPERATION CHAR(1) GENERATED ALWAYS AS (DATA CHANGE OPERATION)
ADD PERIOD SYSTEM_TIME (SYSTEM_START, SYSTEM_END);
```

```
CREATE TABLE EMPS_S2_HIST LIKE EMPS_S2 IN DATABASE APPTMPD;
```

```
ALTER TABLE EMPS_S2
ADD VERSIONING USE HISTORY TABLE EMPS_S2_HIST ON DELETE ADD EXTRA ROW ;
```

Our system-period temporal table information from the catalog looks like this (Example 2-14).

Example 2-14 System-period temporal table information

Select	Column Name	Col	No	Col Type	Length	Scale	Null	Def
*		*	*		*		*	*
-----	-----	-----	-----	-----	-----	-----	-----	-----
	EMPID	1		BIGINT	8		0 N	N
	NAME	2		VARCHAR	20		0 Y	Y

DEPTID	3	INTEGER	4	0	Y	Y
SALARY	4	DECIMAL	7	2	Y	Y
SYSTEM_START	5	TIMESTAMP	13	12	N	Q
SYSTEM_END	6	TIMESTAMP	13	12	N	R
TRANS_START	7	TIMESTAMP	13	12	Y	X
CHGAPPL	8	VARCHAR	255	0	Y	a
LAST_USER	9	VARCHAR	8	0	Y	a
LAST_OPERATION	10	CHAR	1	0	Y	d

Note the attributes of the default (Def) reflecting the auto generated attribute.

We run a couple of inserts and updates such that the emps_s2 table has rows that look like Table 2-4.

Table 2-4 Results

emp_id	name	salary	dept_id	system_start	system_end	chgappl	Last_user	Last_op
77	Claire	30000	2	2015-06-21 -23.06.43.5 5429679	9999-12-30 -00.00.00.0 000000000 00	MKRES6	MKRES6	U
54	Cameron	35000	2	2015-06-21 -23.15.22.6 920364040 01	9999-12-30 -00.00.00.0 000000000 00	db2jcc_app lication	MKRES6	U
49	Tom	41000	5	2015-06-22 -09.18.31.6 210887580 01	9999-12-30 -00.00.00.0 000000000 00	MKRES6	DB2AUT H	I

We delete Tom and the history table now has two rows for Tom, with the same values for the system_start and system_end columns (Table 2-5).

Table 2-5 Results

emp_id	name	salary	dept_id	system_start	system_end	chgappl	Last_user	Last_op
77	Claire	10000	2	2015-06-21 -00.05.36.3 069353140 01	2015-06-21 -23.06.43.5 542967990 01	MKRES6	MKRES6	U
54	Cameron	35000	2	2015-06-21 -23.15.22.6 920364040 01	9999-12-30 -00.00.00.0 000000000 00	db2jcc_app lication	MKRES6	U
49	Tom	41000	5	2015-06-22 -09.18.31.6 210887580 01	2015-06-22 -09.49.15.2 960833750 01	MKRES6	DB2AUT H	I
49	Tom	41000	5	9999-12-30 -00.00.00.0 000000000 00	2015-06-22 -09.49.15.2 960833750 01	MKRES6	MKRES6	D

A select on the base table at a point where Tom existed returns only the one row. If an application has a requirement to retrieve the rows that have been deleted, this can be done by explicitly querying the history table for rows with the last operation of D (Table 2-6).

```
SELECT * FROM EMPS_S2
FOR SYSTEM_TIME AS OF '2015-06-22-09.19.00.000000' WHERE EMPID=49;
```

Table 2-6 Results

emp_id	name	salary	dept_id	system_start	system_end	chgappl	Last_user	Last_op
49	Tom	41000	5	2015-06-22-09.18.31.621088758001	2015-06-22-09.49.15.296083375001	MKRES6	DB2AUTH	I

2.5 Querying a system-period temporal table

Querying a system-period temporal table can return results for a specified point or period in time. Those results can include current values and previous historic values.

It is worth noting that the unit of recovery or unit of work begin timestamp is used as the foundation for system-period data versioning activity. Hence, all rows impacted by a unit of work will have the same row-begin values.

You can run regular SQL queries on a system-period temporal table as on any other table.

DB2 managed temporal data came with supporting additions to SQL. The select statement now includes a period specification FOR SYSTEM_TIME.

The rows returned from an SQL statement can be the current row and historical rows. DB2 transparently examines the current table and the history table, and returns the correct result based on the content of the query.

DB2 presents a system-period temporal table along with the associated history table as a “single table” to the user. DB2 pulls data from both tables as needed to return a result that corresponds to the user’s query. DB2 processes these queries as a UNION ALL. You can write an SQL statement to reference both the system-period temporal table and the associated history table without using the temporal syntax. However, this requires an explicit UNION ALL, which results in a more complex query. This is not recommended.

The value specified in the FOR SYSTEM_TIME AS OF clause can be a date, timestamp, expression, parameter marker, or a host variable (Figure 2-1).

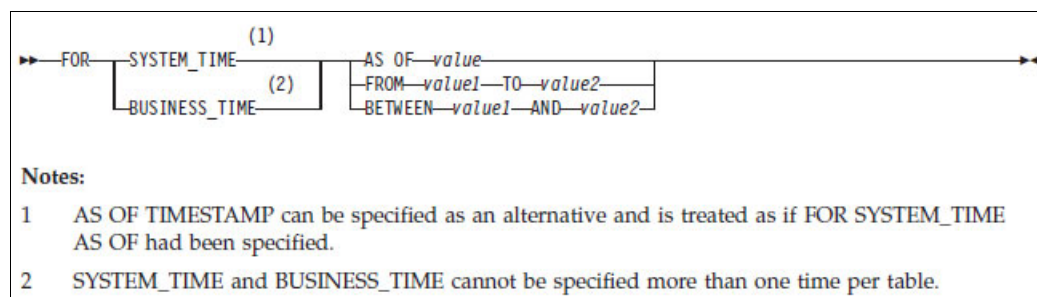


Figure 2-1 FOR SYSTEM_TIME

In a system-period temporal table, the columns of the SYSTEM_TIME period are always TIMESTAMP(12), whereas with an application-period temporal table the columns of the BUSINESS_TIME period can be either DATE or TIMESTAMP(6).

Application programmers can request data from a temporal table based on a time criteria using simple SQL queries. The FROM clause of a query is extended with an optional “period specification” that can be specified for the SYSTEM_TIME period following the name of a system-period temporal table. A period specification for the SYSTEM_TIME period indicates time criteria for the system-period temporal table:

- ▶ FOR SYSTEM_TIME AS OF *timestamp-expression*
- ▶ FOR SYSTEM_TIME FROM *timestamp-expression1* TO *timestamp-expression2*
- ▶ FOR SYSTEM_TIME BETWEEN *timestamp-expression1* AND *timestamp-expression2*

Whenever the data is requested from a system-period temporal table with the period specification syntax, DB2 rewrites the user’s query in the following way (Table 2-7).

Table 2-7 Transformation of a temporal query

Temporal query – period specification	Logical equivalent predicate
FOR SYSTEM_TIME AS OF <i>value1</i>	Where the SYSTEM_TIME period is defined with sys_start as the first column and sys_end as the second column of the period. WHERE sys_start <= <i>value1</i> AND sys_end > <i>value1</i>
FOR SYSTEM_TIME FROM <i>value1</i> TO <i>value2</i>	WHERE sys_start < <i>value2</i> AND sys_end > <i>value1</i> Note: A period is defined such that the time range for the period is inclusive of the starting value and exclusive of the end value.
FOR SYSTEM_TIME BETWEEN <i>value1</i> AND <i>value2</i>	WHERE sys_start <= <i>value2</i> AND sys_end > <i>value1</i> Note: A period is defined such that the time range for the period is exclusive of the end value. This means that the semantic for a BETWEEN period specification differs from the normal BETWEEN predicate. In a BETWEEN period specification, <i>value2</i> must be greater or equal to the starting point of the SYSTEM_TIME period.

If required, a date can be cast to a timestamp, for example TIMESTAMP(‘2015-06-15’).

To access columns defined with the implicitly hidden attribute, explicitly specify the column names in the select list of a query.

Note: It is important to note the difference for providing the literal value of a date in period specifications expressions. DB2 for Linux, UNIX, and Windows supports casting a date to the timestamp where this is not supported on System z. The date value must be written as a timestamp, for example:

‘2015-06-15’ becomes TIMESTAMP(‘2015-06-15’) or ‘2015-06-15-00.00.00’

2.5.1 System-period temporal table query examples

Using our emps_s2 table, we can provide examples of temporal queries using system-period temporal tables with SQL.

The SQL statement is shown in Example 2-15.

Example 2-15 SQL query

Base system-period temporal table.

```
select empid,name,deptid,salary,system_start,system_end,last_operation
from "db2auth"."emps_s2"
```

Table 2-8 shows the contents of the base table.

Note: In DB2 for z/OS, the literal value 2011-02-01 must be written as `TIMESTAMP '2011-02-01'` so that the value is correctly cast to the target data type.

Table 2-8 Base system-period temporal table

Empid	Name	Deptid	Salary	System_start	System_end	Last_operation
77	CLAIRE	2	30000	2015-06-23-10.4 1.33.32817	9999-12-30-00. 00.00.00000	U
54	CAMERON	2	70000	2015-06-23-10.4 2.52.15378	9999-12-30-00. 00.00.00000	U
13	PETER	4	25000	2015-06-23-10.4 2.52.15378	9999-12-30-00. 00.00.00000	U
87	ROGER	5	0	2015-06-22-11.5 7.04.93788	9999-12-30-00. 00.00.00000	U

By using the following query (Example 2-16), we can see the content of the emps_s2 history table (Table 2-9).

Example 2-16 SQL query

```
select empid,name,deptid,salary,system_start,system_end,last_operation
from "db2auth"."emps_s2_hist"
order by empid asc,system_start desc ;
```

Table 2-9 Emps_s2_hist table contents

Empid	Name	Deptid	Salary	System_start	System_end	Last_operation
12	ROGER	2	50000	2015-06-22-00.1 3.36.36044	2015-06-22-00. 13.36.36044	D
12	ROGER	2	50000	2015-06-20-23.5 9.24.98997	2015-06-22-00. 13.36.36044	I
13	PETER	4	25000	2015-06-22-11.5 7.04.93788	2015-06-23-10. 42.52.15378	U
13	PETER	4	70000	2015-06-21-23.5 9.36.41058	2015-06-22-11. 57.04.93788	I

Empid	Name	Deptid	Salary	System_start	System_end	Last_operation
13	DAVID	2	50000	2015-06-20-23.5 9.24.98997	2015-06-21-00. 08.26.61891	I
49	TOM	5	41000	2015-06-22-09.4 9.15.29608	2015-06-22-09. 49.15.29608	D
49	TOM	5	41000	2015-06-22-09.1 8.31.62108	2015-06-22-09. 49.15.29608	I
54	CAMERON	2	0	2015-06-22-11.5 7.04.93788	2015-06-23-10. 42.52.15378	U
54	CAMERON	2	35000	2015-06-21-23.1 5.22.69203	2015-06-22-11. 57.04.93788	U
54	CAMERON	2	70000	2015-06-20-23.5 9.24.98997	2015-06-21-23. 15.22.69203	I
77	CLAIRE	2	0	2015-06-22-11.5 7.04.93788	2015-06-23-10. 41.33.32817	U
77	CLAIRE	2	30000	2015-06-21-23.0 6.43.55429	2015-06-22-11. 57.04.93788	U
77	CLAIRE	2	10000	2015-06-21-00.0 5.36.30693	2015-06-21-23. 06.43.55429	U
77	CLAIRE	2	50000	2015-06-20-23.5 8.27.79271	2015-06-21-00. 05.36.30693	I
87	ROGER	5	23000	2015-06-22-09.1 8.31.62108	2015-06-22-11. 57.04.93788	I

We use the following SQL to retrieve the current row for empid 77 (Example 2-17).

Example 2-17 SQL query

```
select empid, name, deptid, salary, system_start, system_end, last_operation
from db2auth.emps_s2
where empid = 77 ;
```

Even though a period specification was not included in the query, it is implied that FOR SYSTEM_TIME AS OF CURRENT_TIMESTAMP was used (Table 2-10).

Table 2-10 Results

Empid	Name	Deptid	Salary	System_start	System_end	Last_operation
77	CLAIRE	2	30000	2015-06-23-10.4 1.33.32817	9999-12-30-00. 00.00.00000	U

A temporal query using a FROM and TO period specification

The use of the FROM and TO period specification indicates that rows changed from '2015-06-21-23.06.43.554296799001' to the current time should be returned for empid 77 (Example 2-18 on page 33).

Example 2-18 SQL query

```
select empid, name, deptid, salary, system_start, system_end, last_operation  
  
  from db2auth.emps_s2  
    for system_time from '2015-06-21-23.06.43.554296799001'  
                      to current timestamp  
  
 where empid = 77  
 order by empid, system_start desc;
```

A row changed at the time '2015-06-21-23.06.43.554296799001' will be returned (Table 2-11).

Table 2-11 Results

Empid	Name	Deptid	Salary	System_start	System_end	Last_operation
77	CLAIRE	2	30000	2015-06-23-10.4 1.33.32817	9999-12-30-00. 00.00.00000	U
77	CLAIRE	2	0	2015-06-22-11.5 7.04.93788	2015-06-23-10. 41.33.32817	U
77	CLAIRE	2	30000	2015-06-21-23.0 6.43.55429	2015-06-22-11. 57.04.93788	U

A temporal query using a BETWEEN period specification

This is the same as the previous query, but this time using the BETWEEN period specification (Example 2-19).

Example 2-19 SQL query

```
select empid, name, deptid, salary, system_start, system_end, last_operation  
  
  from db2auth.emps_s2  
    for system_time between '2015-06-21-23.06.43.554296799001'  
                      and current timestamp  
 where empid = 77  
 order by empid, system_start desc;
```

The same results are returned as the FROM and TO syntax (Table 2-12).

Table 2-12 Results

Empid	Name	Deptid	Salary	System_start	System_end	Last_operation
77	CLAIRE	2	30000	2015-06-23-10.4 1.33.32817	9999-12-30-00. 00.00.00000	U
77	CLAIRE	2		2015-06-22-11.5 7.04.93788	2015-06-23-10. 41.33.32817	U
77	CLAIRE	2	30000	2015-06-21-23.0 6.43.55429	2015-06-22-11. 57.04.93788	U

What were the details for empid 77 two days ago?

The AS OF clause can be used to request the rows that were current at a particular point. In this case, the query is requesting the rows that were current two days ago for employee 77 (Example 2-20).

Example 2-20 SQL query

```
select empid, name, deptid, salary, system_start, system_end, last_operation
from db2auth.emps_s2
  for system_time as of current timestamp - 2 days
where empid = 77
order by empid, system_start desc;
```

The results from the query shown in Table 2-13 have come from the history table due to an update made at 2015-06-21-23.06.43.55429 (SYSTEM_END). You can verify by looking at the current row with the SYSTEM_START equal to the SYSTEM_END value in the history table.

Table 2-13 Results

Empid	Name	Deptid	Salary	System_start	System_end	Last_operation
77	CLAIRE	2	10000	2015-06-21-00.05.36.30693	2015-06-21-23.06.43.55429	U

The following use of the FROM and TO period specification shows that the time specified after the TO keyword is not included in the time period to satisfy the query. This query returns rows for from 3 days ago up until, but not including, time '2015-06-21-23.06.43.55429'. See Example 2-21.

Example 2-21 SQL query

```
select empid, name, deptid, salary, system_start, system_end, last_operation
from db2auth.emps_s2
  for system_time from current timestamp - 3 days
    to '2015-06-21-23.06.43.554296799001'
where empid = 77
order by empid, system_start desc;
```

The result shows the history row from the history table for a change that occurred at '2015-06-21-23.06.43.554296799001' but it does not show the row as at '2015-06-21-23.06.43.554296799001' due to the semantic for the timestamp specified after the TO keyword is exclusive of that value (Table 2-14).

Table 2-14 Results

Empid	Name	Deptid	Salary	System_start	System_end	Last_operation
77	CLAIRE	2	10000	2015-06-21-00.05.36.30693	2015-06-21-23.06.43.55429	U
77	CLAIRE	2	50000	2015-06-20-23.58.27.79271	2015-06-21-00.05.36.30693	I

The following is the same query as above but not using an explicit period specification. It is provided to highlight the difference in behavior with a datetime range. In this case, explicit BETWEEN predicates are specified for the traditional behavior of a range where it is inclusive on both ends (Example 2-22).

Example 2-22 SQL query

```

select empid, name, deptid, salary, system_start, system_end,
       last_operation
  from db2auth.emps_s2
   where system_start between current timestamp - 4 days
                        and '2015-06-21-23.06.43.554296799001'
   and empid = 77
union all
(select empid, name, deptid, salary, system_start, system_end,
       last_operation
  from db2auth.emps_s2_hist
   where system_start between current timestamp - 4 days
                        and '2015-06-21-23.06.43.554296799001'
   and empid = 77 ) ;

```

The results are shown in Table 2-15.

Table 2-15 Results

Empid	Name	Deptid	Salary	System_start	System_end	Last_operation
77	CLAIRE	2	10000	2015-06-21-00.05.36.30693	2015-06-21-23.06.43.55429	U
77	CLAIRE	2	50000	2015-06-20-23.58.27.79271	2015-06-21-00.05.36.30693	I
77	CLAIRE	2	30000	2015-06-21-23.06.43.55429	2015-06-22-11.57.04.93788	U

Finding deleted rows

The following query against the system-period temporal table, without a period specification, does not return any rows for empid 49 (Example 2-23).

Example 2-23 Finding deleted rows

```

SELECT EMPID, NAME, DEPTID, SALARY, SYSTEM_START, SYSTEM_END,
       LAST_OPERATION
  FROM DB2AUTH.EMPS_S2
 WHERE EMPID=49 ;

```

As there are no current rows found, the row may have been deleted. We need to modify our query to include the history table. In this case, we know a row existed as it was listed in the contents of the history table earlier.

In our example, we know from the contents of the history table there was a row for empid 49 on 2015-06-22 but later in the day followed by a delete. The AS OF clause can specify a particular point in time.

This query returns data for empid 49 that was current at the point in time of '2015-06-22-09.45.00' (Example 2-24).

Example 2-24 SQL query

```
select empid, name, deptid, salary, system_start, system_end, last_operation
  from db2auth.emps_s2
    for system_time as of '2015-06-22-09.45.00'
 where empid=49      ;
```

This could also be written without the “FOR SYSTEM_TIME” clause, where FOR SYSTEM_TIME is implied by the use of the AS OF clause (Example 2-25).

Example 2-25 SQL query

```
select empid, name, deptid, salary, system_start, system_end, last_operation
  from db2auth.emps_s2
    as of timestamp( '2015-06-22-09.45.00')
 where empid=49      ;
```

We know from information provided earlier about DELETE that there will be two rows in the history for a delete statement when the ON DELETE ADD EXTRA ROW clause is used. A temporal query will not return a row where the LAST_OPERATION = D (Table 2-16).

Table 2-16 Results

Empid	Name	Deptid	Salary	System_start	System_end	Last_operation
49	TOM	5	41000	2015-06-22-09.18.31.62108	2015-06-22-09.49.15.29608	I

To find all the deleted rows, you need to run an explicit query on the history table (Example 2-26).

Example 2-26 SQL query

```
select empid, name, deptid, salary, system_start, system_end
  from db2auth.emps_s2_hist
 where last_operation='D'
 order by empid, system_start desc;
```

There might be a requirement to know just the previous information of an employee. You do not want all the history, just what were the details before the latest change.

The following query returns the latest row before the current row for empid 77 (Example 2-27). The query joins the emps_s2 table to itself and returns a row where the system_end value matches the current value for the system_start column.

Example 2-27 SQL query

```
select prev.*
  from emps_s2 curr,
    emps_s2 for system_time between timestamp('0001-01-01') and
    current_timestamp prev
 where curr.empid = 77
    and prev.empid = 77
```

```
and curr.system_start = prev.system_end;
```

The results are shown in Table 2-17.

Table 2-17 Results

Empid	Name	Deptid	Salary	System_start	System_end	Last_operation
77	CLAIRE	2	0	2015-06-22-11.5 7.04.93788	2015-06-23-10. 41.33.32817	U

2.6 General restrictions with system-period data versioning

While system-period data versioning is defined for a table, the following restrictions apply:

- ▶ The history table must have the same number and order of columns as the table that is to be used as a system-period temporal table.
- ▶ No alter of schema (data type, add column, and so on) is allowed on system-period temporal table and history table.
- ▶ Cannot drop the history table, history table space, or history database.
- ▶ Cannot define a CLONE table to the system-period temporal table and history table.
- ▶ Cannot create another table in either the table space of either the system-period temporal table or history table.
- ▶ Cannot use the UPDATE and DELETE syntax that specifies the BUSINESS_TIME period on the history table.
- ▶ Cannot RENAME a column for the system-period temporal table or history table.
- ▶ For point-in-time recovery to keep the system-maintained temporal table and history data in sync, the history table space and system-period temporal table table space must be recovered as a set and cannot be done individually unless the keyword VERIFYSET NO is used.
- ▶ No utility operation is allowed that will delete data from the system-period temporal table. This includes LOAD REPLACE, REORG DISCARD, and CHECK DELETE YES.

See the SQL Reference for all of the restrictions on system-period temporal tables and history tables.

2.7 Utilities

Using system-period temporal tables with versioning, DB2 Utilities have been enhanced to support history table spaces. With a couple of exceptions where the system-period temporal table and history table space should be processed as a set, most utilities can operate on the history table space independently from its associated system-period temporal table space, just like any other user table space.

2.7.1 LISTDEF

LISTDEF is enhanced to include the history table spaces and index spaces as part of the list of objects to be processed. The keyword HISTORY has been added to the LISTDEF syntax.

Load option details:

- **OVERRIDE**

Allows unloaded data to be reloaded into the specified types of GENERATED ALWAYS columns.

- **SYSTEMPERIOD**

Allows unloaded data to be reloaded into a GENERATED ALWAYS row-begin or row-end column. Row-begin and row-end columns are intended to be used in the definition of a system period, but the period does not need to exist when the SYSTEMPERIOD keyword is specified.

If you specify OVERRIDE(SYSTEMPERIOD) and include input field specifications in the LOAD statement, both the row-begin and row-end columns that can be used to define a system period must be specified. In the specification for these columns, the NULLIF and DEFAULTIF options are not allowed.

- **TRANSID**

Allows unloaded data to be reloaded into a GENERATED ALWAYS column that is defined as a transaction-start-ID column.

2.8 Recovery

When you recover the table space for a system-period temporal table to a point in time, you must also recover the table space for the associated history table to the same point in the same RECOVER utility job. If there is a need to recover the system-period temporal table quickly to open the critical application, the VERIFYSET NO option can be used on the RECOVER utility. Care must be taken to recover the history table to the same point before updating and deleting from the system-period temporal table.



Application-period temporal tables

This chapter introduces application-period temporal tables whereby application programmers can specify queries with a search criteria based on time the data existed.

This chapter contains the following sections:

- ▶ Overview and benefits
- ▶ Using application-period temporal tables
- ▶ Case study: an application-period table in action

3.1 Overview and benefits

For years, application programmers and database administrators have been facing the problem of managing different versions of application data. There is a large class of database applications that are interested in capturing the time periods during which the data is valid in the real world. A primary requirement of such applications is that it is the user rather than the system that is in charge of setting the start and end times of the validity period of rows.

User-specified values may correspond to any past, current, or future time. The ability to specify future time is especially important in many application scenarios where the database is updated ahead of an event taking place. Another requirement of such applications is the ability to update or delete rows that are valid between any two points in time, which may cause some rows to be updated or deleted only for a portion of their validity period. If a row is updated or deleted only for a part of its validity period, the system will have to insert one or more rows to preserve the information for the period during which the update or delete did not apply.

In addition, users also want to ensure that constraints are enforced by considering the validity periods of rows. For example, additional syntax might need to be provided to allow for the existence of multiple rows with the same key value when their validity periods do not overlap.

Before DB2 V10, it was up to the user to organize the application level versioning. It not only can explode the table design but adds complexity and is potentially error prone in the application code. The lack of data versioning in DB2 prevents the protection and management of core business sensitive assets by DB2.

In DB2 V10, we provide a capability to specify table-level specifications to control the management of application data based on time. Application programmers are able to specify queries that specify a search criteria based on time the data existed. This function simplifies and reduces the cost of developing DB2 applications requiring data versioning and allows customers to meet new compliance laws faster and cheaper because DB2 will automatically manage the different versions of data.

3.2 Using application-period temporal tables

Application-period temporal tables provide methods to enable the definition of a user-specified `BUSINESS_TIME` period, which could be processed with extensions to the `UPDATE` and `DELETE` statements to allow updating and deleting based on a period of time.

For `BUSINESS_TIME` periods, the user controls the values that would be put into the period columns. DB2 supports a temporal `UPDATE` and a temporal `DELETE` to handle these operations based on the `BUSINESS_TIME` period. A user can also decide whether to create a `UNIQUE` index with `BUSINESS TIME WITHOUT OVERLAPS` to ensure that no overlaps of periods occur.

3.2.1 New concept of a `BUSINESS_TIME` period

In order to use versioning to keep rows, there is a new concept called a *period*. A period is defined by two columns of a table in which the first column marks the start of a period and the second column marks the end of a period. It is important to know that a period in DB2 temporal has the start value being inclusive and the end value being exclusive. For example, if the start column has a value of '01/01/1995' and an end column has a value of '03/21/1995',

the date of '01/01/1995' belongs in the row, while the date of '03/21/1995' is not part of the row period.

A BUSINESS_TIME period is built with a start column defined as either TIMESTAMP(6) WITHOUT TIMEZONE NOT NULL or DATE NOT NULL and the end column matching the data type definition of the start column.

Whenever a BUSINESS_TIME period is specified for a table, DB2 generates a check constraint in which the end column must be greater than the start column value.

To create a temporal table with BUSINESS_TIME period, the table needs to be defined with:

- ▶ Two same type columns defined as TIMESTAMP(6) WITHOUT TIME ZONE NOT NULL or DATE NOT NULL. This data type cannot be a distinct type.
- ▶ A period BUSINESS_TIME specified on the two above columns.

Example of period specification: PERIOD BUSINESS_TIME (col1, col2).

3.2.2 Defining a BUSINESS_TIME period

An application-period temporal table is a table that includes a BUSINESS_TIME period. Here are examples about how to enable the BUSINESS_TIME period.

Simple creation of an application-period temporal table

Example 3-1 shows the policy_info table.

Example 3-1 Create table

```
CREATE TABLE policy_info
(policy_id CHAR(4) NOT NULL,
coverage INT NOT NULL,
bus_start DATE NOT NULL,
bus_end DATE NOT NULL,
PERIOD BUSINESS_TIME (bus_start, bus_end));
```

Define a unique index with the BUSINESS_TIME WITHOUT OVERLAPS clause (Example 3-2). This index ensures uniqueness for the values for columns, other than the columns of the BUSINESS_TIME period, regarding the time period represented by the BUSINESS_TIME period.

Example 3-2 Create unique index

```
CREATE UNIQUE INDEX ix_policy
ON policy_info (policy_id, BUSINESS_TIME WITHOUT OVERLAPS);
```

Simple migration to an application-period temporal table

Example 3-3 shows the policy_info table.

Example 3-3 Create table

```
CREATE TABLE policy_info
(policy_id CHAR(4) NOT NULL,
coverage INT NOT NULL,
bus_start DATE NOT NULL,
bus_end DATE NOT NULL);
```

If the user is keeping track of start and end columns, the following statements can be used to migrate from the existing table to use the DB2 support for application-period temporal tables:

```
ALTER TABLE policy_info ADD PERIOD BUSINESS_TIME (bus_start, bus_end);
```

Define a unique index with the BUSINESS_TIME WITHOUT OVERLAPS clause (Example 3-4). This index ensures uniqueness for the values for columns, other than the columns of the BUSINESS_TIME period, regarding the time period represented by the BUSINESS_TIME period.

Example 3-4 Create unique index

```
CREATE UNIQUE INDEX ix_policy_info
ON policy_info (policy_id, BUSINESS_TIME WITHOUT OVERLAPS);
```

Now, users can use the DB2 provided support for application-period temporal tables for the policy_info table.

3.2.3 Extensions to primary keys, unique constraints, and unique indexes

Primary key and unique constraints using the current syntax with exactly the same behavior can be defined on the application-period temporal table. However, the presence of a BUSINESS_TIME period provides an opportunity to enhance the notion of primary key and unique constraints on that table.

For example, assume that we are interested in creating a primary key for the emp table that corresponds to the combination of the emp_id, bus_start, and bus_end columns, such that for a given emp_id value and a given point in time T, there is exactly one row whose BUSINESS_TIME period (that is, set of values from bus_start value through to but not including bus_end value) contains T. This means that for any selection based on a specified emp_id value and a specified date, only one row is retrieved.

Assume that the emp table contains the following rows (Table 3-1).

Table 3-1 The emp table

emp_id	name	salary	dept_id	bus_start	bus_end
100	Tom	3500	10	2008-01-01	2013-01-01
100	Tom	4000	20	2013-01-01	2015-01-01
100	Tom	4500	20	2015-01-01	2017-01-01

From the preceding example, it is clear that we need a capability to specify that the emp table contains no two rows with the same emp_id value and overlapping application-time periods. We provide an additional syntax for primary keys, unique constraints, and unique index declarations to provide such a capability. Example 3-5 illustrates this new syntax.

Example 3-5 Create table

```
CREATE TABLE emp
(emp_id INTEGER NOT NULL,
name VARCHAR(30),
salary DECIMAL(5,2),
dept_id INTEGER,
bus_start DATE NOT NULL,
bus_end DATE NOT NULL,
```

```
PERIOD BUSINESS_TIME (bus_start, bus_end),  
PRIMARY KEY (emp_id, BUSINESS_TIME WITHOUT OVERLAPS)  
);
```

```
CREATE UNIQUE INDEX ix_emp  
ON emp (emp_id, BUSINESS_TIME WITHOUT OVERLAPS);
```

The PRIMARY KEY constraint in the above definition ensures that the table has no two rows with the same emp_id value having overlapping application-time periods. The reference to the BUSINESS_TIME in the PRIMARY KEY declaration effectively makes the bus_start and bus_end columns as part of the primary key with check for overlapping periods rather than for equality of periods. Using our sample emp table, an attempt to insert a new row with an emp_id value of 100 and application-time period from 2017-01-01 to 2018-06-01 will succeed while an attempt to insert a new row with an emp_id value of 100 and application-time period from 2015-01-01 to 2015-06-01 will fail.

BUSINESS_TIME WITHOUT OVERLAPS clause can be on CREATE TABLE, ALTER TABLE, or CREATE UNIQUE INDEX statements. BUSINESS_TIME WITHOUT OVERLAPS must be the last expression specified. It will add, in ascending order, the end column and begin column of the period BUSINESS_TIME to the key and have special support to enforce that there are no overlaps in time given the rest of the key expressions.

Referential integrity considerations: Referential constraints can be specified on temporal tables with BUSINESS_TIME period, but there is no support for having a referential constraint relationship across periods of time. That is, a foreign key specification cannot include the BUSINESS_TIME WITHOUT OVERLAPS clause.

Specifying BUSINESS_TIME WITHOUT OVERLAPS for a key: The enforcement of uniqueness over a period of time is an important functionality delivered for application-period temporal tables. A primary or unique key can be defined with the optional BUSINESS_TIME WITHOUT OVERLAPS clause to have DB2 ensure that primary or unique key values are unique for any point in the BUSINESS_TIME period. If the clause is not specified for the key definition, overlaps in time can occur and DB2 does not do any overlap checking.

3.2.4 Temporal UPDATE and DELETE rules

For application-period temporal table, the UPDATE and DELETE statements have been enhanced to allow updating and deleting based on a period of time.

The new clause is specified as follows:

```
FOR PORTION OF BUSINESS_TIME FROM expression1 TO expression2
```

The new FOR PORTION OF clause for UPDATE or DELETE statements, updates, or deletes rows as usual if the period for the row is contained in the period specified in the FROM and TO clause.

When a row is not contained within the FROM/TO values, only the part of the row that is contained between the FROM and TO values is updated or deleted. This may cause additional inserts of rows into the table as the original row needs to be split into multiple rows to reflect the old values for the period that are not affected by the update or delete.

All UPDATE and DELETE triggers defined on the table will get activated in the usual way for all rows that are updated or deleted. In addition, all INSERT triggers will get activated for all rows whenever a split of an original row causes a row to be inserted into the table.

Temporal update logic

Here is what happens for a temporal UPDATE.

Given temporal UPDATE (Example 3-6).

Example 3-6 Update

```
UPDATE emp
  FOR PORTION OF BUSINESS_TIME FROM value1 TO value2
  SET salary = 8000
  WHERE emp_id = 100;
```

This is logically equivalent to the following UPDATE statement (Example 3-7).

Example 3-7 Update

```
UPDATE emp
  SET salary = 8000,
      bus_start = MAX (bus_start, value1),
      bus_end = MIN (bus_end, value2)
  WHERE emp_id = 100 AND
      bus_start < value2 AND bus_end > value1;
```

Additionally, zero to two additional rows can be inserted (internal trigger type logic, **green** bar – UPDATE period, **red** bar – INSERT period). See Figure 3-1.

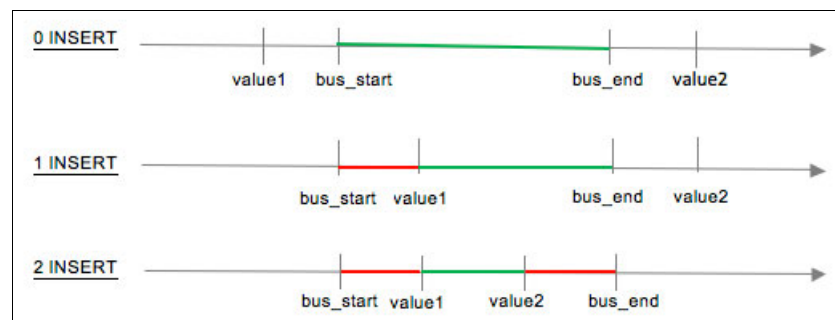


Figure 3-1 Inserting additional rows

More precisely, the effect of UPDATE statements that contain the FOR PORTION OF BUSINESS_TIME FROM value1 to value2 clause are as follows:

- ▶ For each row R in the table that qualifies for update and whose application-time period overlaps with the period formed by value1 and value2, let BPS be its application-time period start value, and let BPE be its application-time period end value.
- ▶ If $BPS < value1$ and $BPE > value1$, a copy of R with its application-time period end value set to value1 is inserted.
- ▶ If $BPS < value2$ and $BPE > value2$, a copy of R with its application-time period start value set to value2 is inserted.
- ▶ R is updated with its application-period start value set to the maximum of BPS and value1 and the application-time end value set to the minimum of BPE and value2.

Temporal delete logic

Here is what happens for a temporal DELETE.

Given temporal DELETE (Example 3-8).

Example 3-8 Delete

```
DELETE emp
  FOR PORTION OF BUSINESS_TIME FROM value1 TO value2
  WHERE emp_id = 100;
```

This is logically equivalent to the following DELETE statement ().

Example 3-9 Delete

```
DELETE emp
  WHERE emp_id = 100 AND
        bus_start < value2 AND bus_end > value1;
```

Additionally, zero to two additional rows can be inserted (internal trigger type logic, **green** bar – DELETE period, **red** bar – INSERT period). See Figure 3-2.

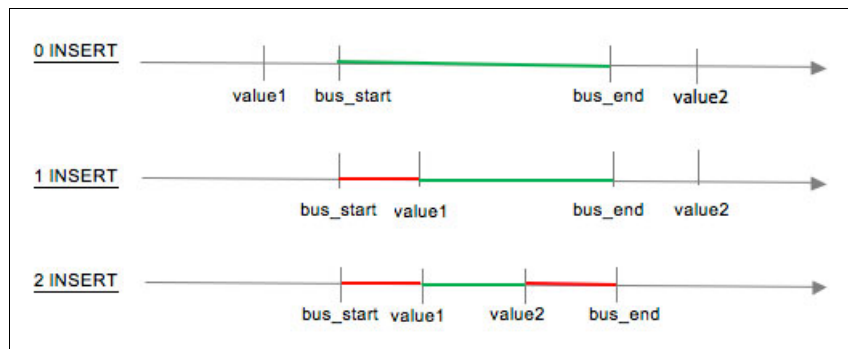


Figure 3-2 Inserting additional rows

More precisely, the effect of DELETE statements that contain the FOR PORTION OF BUSINESS_TIME FROM value1 to value2 clause are as follows:

- ▶ For each row R in the table that qualifies for delete and whose application-time period overlaps with the period formed by value1 and value2, let BPS be its application-time period start value, and let BPE be its application-time period end value.
- ▶ If $BPS < value1$ and $BPE > value1$, a copy of R with its application-time period end value set to value1 is inserted.
- ▶ If $BPS < value2$ and $BPE > value2$, a copy of R with its application-time period start value set to value2 is inserted.
- ▶ R is deleted with its application-period start value set to the maximum of BPS and value1 and the application-time end value set to the minimum of BPE and value2.

Example

The following example illustrates this new syntax for temporal update and delete. Table 3-2 shows the emp table contains the following rows.

Table 3-2 The emp sample table

emp_id	name	salary	dept_id	bus_begin	bus_end
100	Tom	3500	10	2008-01-01	2013-01-01
100	Tom	4000	20	2013-01-01	2015-01-01
100	Tom	4500	20	2015-01-01	2017-01-01

Now, execute the following statement (Example 3-10).

Example 3-10 Update

```
UPDATE emp FOR PORTION OF BUSINESS_TIME FROM  
DATE '2015-06-01' TO DATE '2016-06-01'  
SET dept_id = 15  
WHERE emp_id = 100;
```

The row in **bold** is selected for update. After successful execution of the UPDATE statement, the content of emp table is changed while two new rows in *italics* are inserted (Table 3-3).

Table 3-3 The emp table after the UPDATE

emp_id	name	salary	dept_id	bus_begin	bus_end
100	Tom	3500	10	2008-01-01	2013-01-01
100	Tom	4000	20	2013-01-01	2015-01-01
<i>100</i>	<i>Tom</i>	<i>4500</i>	<i>20</i>	<i>2015-01-01</i>	<i>2015-06-01</i>
100	Tom	4500	15	2015-06-01	2016-06-01
<i>100</i>	<i>Tom</i>	<i>4500</i>	<i>20</i>	<i>2016-06-01</i>	<i>2017-01-01</i>

Then, we execute the following statement (Example 3-11).

Example 3-11 Delete

```
DELETE emp FOR PORTION OF BUSINESS_TIME FROM  
DATE '2008-01-01' TO DATE '2010-01-01'  
WHERE emp_id = 100;
```

The row in example font will be selected to delete. After successful execution of the preceding DELETE statement, the content of emp table is changed as shown below while one new row in *italics* is inserted (Table 3-4 on page 49).

Table 3-4 The emp table after the DELETE

emp_id	name	salary	dept_id	bus_begin	bus_end
100	Tom	4000	20	2013-01-01	2015-01-01
100	Tom	4500	20	2015-01-01	2015-06-01
100	Tom	4500	15	2015-06-01	2016-06-01
100	Tom	4500	20	2016-06-01	2017-01-01
100	Tom	3500	10	2010-01-01	2013-01-01

3.2.5 Queries for application-period temporal tables

Application programmers can request data from a temporal table based on a time criteria using simple SQL queries. The FROM clause of a query is extended with an optional “period specification” that can be specified for the BUSINESS_TIME period following the name of an application-period temporal table. A period specification for the BUSINESS_TIME period indicates time criteria for the application-period temporal table.

- ▶ FOR BUSINESS_TIME AS OF *datetime-expression*
- ▶ FOR BUSINESS_TIME FROM *datetime-expression1* TO *datetime-expression2*
- ▶ FOR BUSINESS_TIME BETWEEN *datetime-expression1* AND *datetime-expression2*

Whenever the data is requested from an application-period temporal table with the period specification syntax, DB2 rewrites the user's query as shown in Table 3-5.

Table 3-5 Transformation of a temporal query

Temporal query – period specification	Logical equivalent predicate
FOR BUSINESS_TIME AS OF <i>value1</i>	WHERE bus_start <= <i>value1</i> AND bus_end > <i>value1</i>
FOR BUSINESS_TIME FROM <i>value1</i> TO <i>value2</i>	WHERE bus_start < <i>value2</i> AND bus_end > <i>value1</i> Note: A period is defined such that the time range for the period is inclusive of the starting value and exclusive of the end value.
FOR BUSINESS_TIME BETWEEN <i>value1</i> AND <i>value2</i>	WHERE bus_start <= <i>value2</i> AND bus_end > <i>value1</i> Note: A period is defined such that the time range for the period is inclusive of the starting value and exclusive of the end value. This means that the semantic for a BETWEEN period specification differs from the normal BETWEEN predicate. In a BETWEEN period specification, <i>value2</i> must be greater or equal to the starting point of the BUSINESS_TIME period.

3.2.6 BUSINESS_TIME period restrictions

When a BUSINESS_TIME period is defined, some restrictions apply to the table.

These include:

- ▶ No ALTER INDEX ADD BUSINESS_TIME WITHOUT OVERLAPS.
- ▶ No SELECT FROM DELETE or SELECT FROM UPDATE when UPDATE or DELETE with FOR PORTION OF specified.
- ▶ If the new FOR PORTION OF BUSINESS_TIME clause is specified, the BUSINESS_TIME period columns are not allowed to be specified in the SET clause of the UPDATE statement.

3.2.7 Utilities

For tables with BUSINESS_TIME periods, unique indexes defined with the BUSINESS_TIME WITHOUT OVERLAPS clause require a new type of uniqueness checking for overlapping time ranges.

For data manipulative utilities that insert into this new type of index, DB2 catches a unique key violation at key insert time, so there is no impact to utilities like REORG, LOAD, and REBUILD.

CHECK INDEX is enhanced to validate there are no overlapping periods in these unique indexes when the BUSINESS_TIME WITHOUT OVERLAPS clause is specified in the definition of the index.

3.3 Case study: an application-period table in action

Define an application-period temporal table to have data versioned from insert, update, and delete activity. Assume there is a health insurance company who wants to keep the customer insurance information changes based on the basic flow of events shown in Table 3-6.

Table 3-6 Basic flow of events

Step	Date	Activity
1	1/1/2004 (Future)	Issue PPO policy P138 to customer C882 with copay amount \$10 starting from 2/1/2004
2	9/1/2004 (Present)	Customer called and changed to HMO as of today
3	3/1/2006 (Future)	Copay increase to \$15 starting 1/1/2007
4	6/1/2008 (Present)	Cancel policy as of today
5	9/1/2008 (Past)	Correct error by retroactively updating policy to POS from 5/01/06 to 10/01/07

3.3.1 Create an application-period temporal table

First of all, we need to create an application-period temporal table as shown in Example 3-12.

Example 3-12 Create table

```
CREATE TABLE Policy
(Bk VARCHAR(4) NOT NULL,
```

```

Eff_Beg DATE NOT NULL,
Eff_End DATE NOT NULL,
Client CHAR(4) NOT NULL,
Type CHAR(3),
Copoly CHAR(3),
PERIOD BUSINESS_TIME (Eff_Beg, Eff_End),
PRIMARY KEY (Bk, Client, BUSINESS_TIME WITHOUT OVERLAPS)
);

CREATE UNIQUE INDEX ix_policy
ON Policy (Bk, Client, BUSINESS_TIME WITHOUT
OVERLAPS);

```

This creates the table shown in Table 3-7.

Table 3-7 New table

Bk	Eff-Beg	Eff-End	Client	Type	Copoly
----	---------	---------	--------	------	--------

3.3.2 Step-by-step usage scenarios

In this section, we proceed through each step in the flow of events in our scenario.

Step 1

This step is shown in Table 3-8.

Table 3-8 Basic flow of events

Step	Date	Activity
1	1/1/2004 (Future)	Issue PPO policy P138 to customer C882 with copay amount \$10 starting from 2/1/2004

We insert one row into table POLICY as shown in Example 3-13 and Figure 3-3.

Example 3-13 Step 1

```

INSERT INTO POLICY
VALUES ('P138', '2/1/2004', '12/31/9999', 'C882', 'PPO', '$10');

```

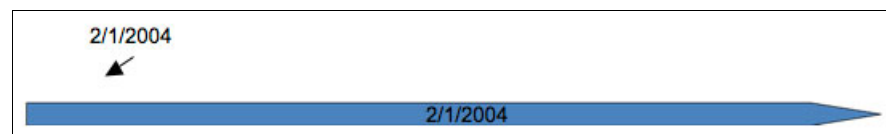


Figure 3-3 Step 1

This adds the row in the table as shown in Figure 3-4.

Bk	Eff-Beg	Eff-End	Client	Type	Copoly
P138	2/1/2004	12/31/9999	C882	PPO	\$10

Figure 3-4 First row in the table

Step 2

The next step is executed (Table 3-9).

Table 3-9 Basic flow of events

Step	Date	Activity
1	1/1/2004 (Future)	Issue PPO policy P138 to customer C882 with copay amount \$10 starting from 2/1/2004
2	9/1/2004 (Present)	Customer called and changed to HMO as of today

We execute the following temporal update as shown in Example 3-14 and Figure 3-5.

Example 3-14 Step 2

```
UPDATE POLICY FOR PORTION OF BUSINESS_TIME
FROM '9/1/2004' TO '12/31/9999'
SET Type='HMO'
WHERE Bk='P138'AND Client='C882';
```

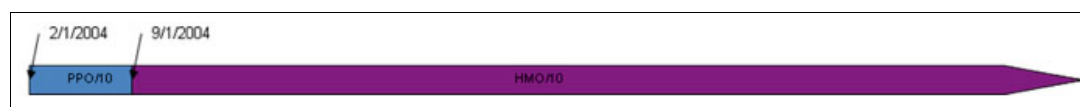


Figure 3-5 Step 2

And the data in the table is as shown in Figure 3-6.

Bk	Eff-Beg	Eff-End	Client	Type	Copay
P138	2/1/2004	9/1/2004	C882	PPO	\$10
P138	9/1/2004	12/31/9999	C882	HMO	\$10

Figure 3-6 Rows in the table

Step 3

The next step is executed (Table 3-10).

Table 3-10 Basic flow of events

Step	Date	Activity
1	1/1/2004 (Future)	Issue PPO policy P138 to customer C882 with copay amount \$10 starting from 2/1/2004
2	9/1/2004 (Present)	Customer called and changed to HMO as of today
3	3/1/2006 (Future)	Copay increase to \$15 starting 1/1/2007

Again, we execute a temporal update (Example 3-15).

Example 3-15 Step 3

```
UPDATE POLICY FOR PORTION OF BUSINESS_TIME
FROM DATE '1/1/2007' TO '12/31/9999'
SET Copay='$15'
WHERE Bk='P138' AND Client='C882';
```

After the update, the rows in the table are as shown in Figure 3-7 and Figure 3-8.

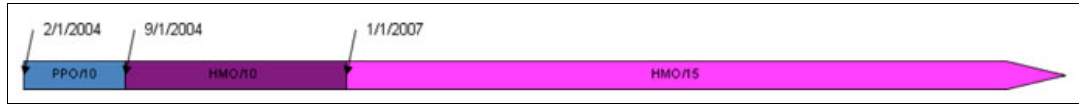



Figure 3-7 Step 3



Bk	Eff-Beg	Eff-End	Client	Type	Copay
P138	2/1/2004	9/1/2004	C882	PPO	\$10
P138	9/1/2004	1/1/2007	C882	HMO	\$10
P138	1/1/2007	12/31/9999	C882	HMO	\$15

Figure 3-8 Rows in the table

Step 4

The next step is executed (Table 3-11).

Table 3-11 Basic flow of events

Step	Date	Activity
1	1/1/2004 (Future)	Issue PPO policy P138 to customer C882 with copay amount \$10 starting from 2/1/2004
2	9/1/2004 (Present)	Customer called and changed to HMO as of today
3	3/1/2006 (Future)	Copay increase to \$15 starting 1/1/2007
4	6/1/2008 (Present)	Cancel policy as of today

Now, we execute a temporal delete on the table (Example 3-16).

Example 3-16 Step 4

```
DELETE FROM POLICY FOR PORTION OF BUSINESS_TIME
FROM '6/1/2008' TO '12/31/9999'
WHERE Bk='P138' AND Client='C882';
```

The values are changed as shown in Figure 3-9 and Figure 3-10.

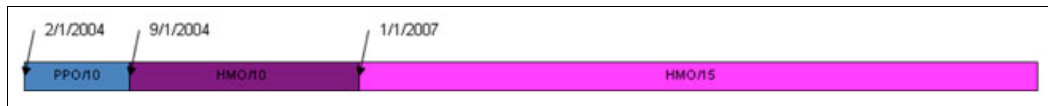



Figure 3-9 Step 4



Bk	Eff-Beg	Eff-End	Client	Type	Copay
P138	2/1/2004	9/1/2004	C882	PPO	\$10
P138	9/1/2004	1/1/2007	C882	HMO	\$10
P138	1/1/2007	6/1/2008	C882	HMO	\$15

Figure 3-10 Rows in the table

Step 5

The next step is executed (Table 3-12).

Table 3-12 Basic flow of events

Step	Date	Activity
1	1/1/2004 (Future)	Issue PPO policy P138 to customer C882 with copay amount \$10 starting from 2/1/2004
2	9/1/2004 (Present)	Customer called and changed to HMO as of today
3	3/1/2006 (Future)	Copay increase to \$15 starting 1/1/2007
4	6/1/2008 (Present)	Cancel policy as of today
5	9/1/2008 (Past)	Correct error by retroactively updating policy to POS from 5/01/06 to 10/01/07

At last, we execute the temporal update to change the policy to POS from 5/01/06 to 10/01/07 (Example 3-17).

Example 3-17 Step 5

```
UPDATE POLICY FOR PORTION OF BUSINESS_TIME
FROM DATE '5/1/2006' TO DATE '10/1/2007'
SET Type='POS'
WHERE Bk='P138' AND Client='C882' ;
```

Here's the data in the table after the temporal update as shown in Figure 3-11 and Figure 3-12.

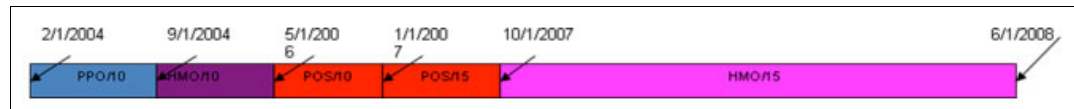


Figure 3-11 Step 5

Bk	Eff-Beg	Eff-End	Client	Type	Copay
P138	2/1/2004	9/1/2004	C882	PPO	\$10
P138	9/1/2004	5/1/2006	C882	HMO	\$10
P138	5/1/2006	1/1/2007	C882	POS	\$10
P138	1/1/2007	10/1/2007	C882	POS	\$15
P138	10/1/2007	6/1/2008	C882	HMO	\$15


Figure 3-12 Rows in the table

We can run the temporal query on the table as shown in Example 3-18.

Example 3-18 Run temporal query

```
SELECT * FROM POLICY FOR BUSINESS_TIME AS OF '7/1/2007'  
WHERE Bk='P138' AND CLIENT='C882';
```

The results of the query are shown in Figure 3-13.



Bk	Eff-Beg	Eff-End	Client	Type	Copay
P138	1/1/2007	10/1/2007	C882	POS	\$15

Figure 3-13 Results



Bitemporal tables

This chapter introduces bitemporal tables, which are both a system-period temporal table and an application-period temporal table.

This chapter contains the following sections:

- ▶ Overview
- ▶ Where application-period meets system-period temporal
- ▶ Extending an application-period temporal table to become a bitemporal table
- ▶ Bitemporal tables in action

4.1 Overview

A *bitemporal table* is defined with both an application period and a system period. It is both a system-period temporal table and an application-period temporal table. The application period allows the recording of business history and data currency. The system period provides the audit history.

Consider an insurance company that can be constantly tweaking or introducing new policies. While the policy information can exist in the DB2 database, it might not be effective or in fact is expired. An application period would keep track of policy effective dates while the system period tracks when the information was added, changed, or deleted in a database (such as used for auditing). Managing both sets of information is called bitemporal (Figure 4-1).

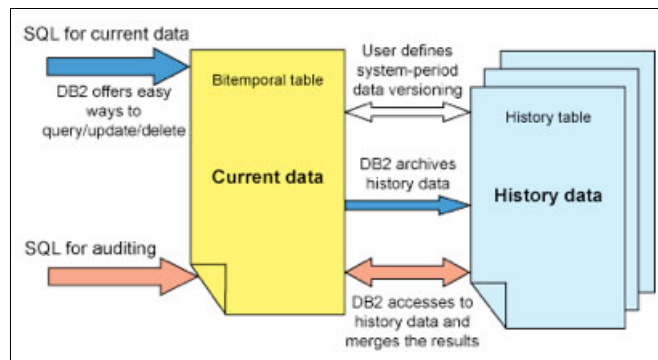


Figure 4-1 Basic principle of a bitemporal table

4.2 Where application-period meets system-period temporal

We covered how to implement and migrate to a system-period temporal table in Chapter 2, “System-period temporal tables” on page 13 and Chapter 3, “Application-period temporal tables” on page 41.

In Chapter 3, “Application-period temporal tables” on page 41, we covered the use of application-period temporal tables. Here, we discuss a case study that demonstrates adding functionality for a system-period temporal table to an existing application-period temporal, thus creating a bitemporal table.

The following list summarizes temporal data management:

- ▶ Use a system period to track when data was changed inside the DB2 system.
- ▶ Use an application period to track when data was, is, or will be valid in the real-world.
- ▶ Use bitemporal tables to combine system and business time as needed.
- ▶ DB2 uses inclusive-exclusive periods rather than inclusive-inclusive periods that you may have in your application logic.

Creating a new bitemporal table will look like Example 4-1 where **bold** is for the application-period temporal table and *italic* is for the system-period temporal table.

Example 4-1 Bitemporal example

```
CREATE TABLE policy  
(policy_id CHAR(4) NOT NULL,  
coverage INT NOT NULL,
```

```

bus_start DATE NOT NULL,
bus_end DATE NOT NULL,
sys_start TIMESTAMP(12) NOT NULL GENERATED ALWAYS AS ROW BEGIN,
sys_end TIMESTAMP(12) NOT NULL GENERATED ALWAYS AS ROW END,
PERIOD BUSINESS_TIME(bus_start, bus_end),
PERIOD SYSTEM_TIME(sys_start, sys_end));

CREATE TABLE policy_hist
(policy_id CHAR(4) NOT NULL,
coverage INT NOT NULL,
bus_start DATE NOT NULL,
bus_end DATE NOT NULL,
sys_start TIMESTAMP(12) NOT NULL,
sys_end TIMESTAMP(12) NOT NULL);

ALTER TABLE policy_info
ADD VERSIONING USE HISTORY TABLE hist_policy_info;

CREATE UNIQUE INDEX ix_policy
ON policy_info (policy_id, BUSINESS_TIME WITHOUT OVERLAPS);

```

4.2.1 Using bitemporal tables in data manipulation statements

So now with a bitemporal table, the SQL statements can combine the use of application period and system period syntax.

For an application-period temporal table, we specify a period specification for the BUSINESS_TIME period (Example 4-2).

Example 4-2 Period specification

```

SELECT ... FROM BTT FOR BUSINESS_TIME
    AS OF exp ...
SELECT ... FROM BTT FOR BUSINESS_TIME
    FROM exp1 TO exp2 ... ;
SELECT ... FROM BTT FOR BUSINESS_TIME
    BETWEEN exp1 AND exp2 ...;

```

And for a system-period temporal table, we specify a period specification for the SYSTEM_TIME period (Example 4-3).

Example 4-3 Period specification

```

SELECT ... FROM BTT FOR SYSTEM_TIME
    AS OF exp ...
SELECT ... FROM BTT FOR SYSTEM_TIME
    FROM exp1 TO exp2 ... ;
SELECT ... FROM BTT FOR SYSTEM_TIME
    BETWEEN exp1 AND exp2 ...;

```

A standard SELECT for a bitemporal table (without any period specifications) implies current date or timestamp for the system period. If the business period specification is excluded, all rows meeting the provided predicates are returned.

Example 4-4 shows how to return the current row for the business period of current date/time and current system date/time.

Example 4-4 Period specification

```
SELECT ... FROM table
      FOR BUSINESS_TIME AS OF CURRENT DATE
      -- or CURRENT_TIMESTAMP depending on column data type
      FOR SYSTEM_TIME AS OF CURRENT_TIMESTAMP
```

A FOR PORTION OF BUSINESS_TIME clause can be specified to limit the effect of updates and deletes of an application-period temporal table (Example 4-5).

Example 4-5 FOR PORTION OF BUSINESS_TIME clause

```
UPDATE table
FOR PORTION OF BUSINESS_TIME FROM datea TO dateb

DELETE FROM table
FOR PORTION OF BUSINESS_TIME FROM datea TO dateb
```

It is worth remembering here again that DB2 temporal queries are inclusive from the start of the period and exclusive to the end of the period, which is different to the non-temporal BETWEEN predicate. This should also serve as a reminder to never set end dates to 31/12/9999 as this date will be excluded in a query. The default end date with DB2 temporal data is 30/12/9999.

4.3 Extending an application-period temporal table to become a bitemporal table

It is most likely that DB2 maintained application period temporal support would not be added to an existing data model without application changes. This could involve some significant development work. DB2 maintained system period support could be defined on existing data models and in some cases without application changes.

A model that has already implemented an application-period temporal table can extend that table to be a bitemporal table by adding the system period and defining the table for data versioning. This would be just as was covered in Chapter 2, “System-period temporal tables” on page 13.

The data model will now have a history table that is maintained by DB2. Example 4-6 describes how an existing application-period temporal table can be extended with system-period data versioning to become a bitemporal table.

Example 4-6 Adding a system period and data versioning to an existing table

```
ALTER TABLE emps_s0
ADD COLUMN system_begin TIMESTAMP(12) NOT NULL GENERATED AS ROW BEGIN
ADD COLUMN system_end TIMESTAMP(12) NOT NULL GENERATED AS ROW END
ADD COLUMN trans_start TIMESTAMP(12) GENERATED AS TRANSACTION START ID
ADD PERIOD SYSTEM_TIME (system_begin, system_end);
```

```
- Create the associated history table
CREATE TABLE emps_s0_hist LIKE emps_s0;
```

```
- Alter the table for system-period data versioning
- and identify the history table
ALTER TABLE emps_s0
ADD VERSIONING USE HISTORY TABLE emps_s0_hist;
```

4.4 Bitemporal tables in action

To best demonstrate bitemporal support, let us expand on the example in 3.3, “Case study: an application-period table in action” on page 50. This time, we add system-period data versioning to make the table a bitemporal table and we observe the activity on the history table.

In this scenario, we define a bitemporal table to have data versioned from insert, update, and delete activity. Assume that there is a health insurance company who is required to keep customer insurance information changes based on time.

It is now a requirement to have an audit of when these changes have been made, and who initiated the changes. This now makes the business requirement for the table to be bitemporal.

The basic flow of events is shown in Table 4-1.

Table 4-1 Basic flow of events

Step	Date	Activity
1	1/1/2004 (Future)	Issue PPO policy P138 to customer C882 with copay amount \$10 starting from 2/1/2004
2	9/1/2004 (Present)	Customer called and changed to HMO as of today
3	3/1/2006 (Future)	Copay increase to \$15 starting 1/1/2007
4	6/1/2008 (Present)	Cancel policy as of today
5	9/1/2008 (Past)	Correct error by retroactively updating policy to POS from 5/01/06 to 10/01/07
6	Current	Executive requires to know when and who canceled the policy

4.4.1 Create a bitemporal table

We use the policy table and add the additional attributes for a system-period temporal table to create a bitemporal table (Example 4-7).

To comply with audit requirements, we include non-deterministic generated expression columns to record the user and the last operation. These audit columns can be defined with the IMPLICITLY HIDDEN column attribute, which may reduce changes required for applications. Refer to Chapter 5, “Additional considerations” on page 73 for more details about these columns.

Example 4-7 Create table

```
CREATE TABLE Policy
(Bk          VARCHAR(4) NOT NULL,
Eff_Beg     DATE NOT NULL,
```

```

Eff_End    DATE NOT NULL,
Client     CHAR(4) NOT NULL,
Type       CHAR(3),
Copoly     CHAR(3),
Last_user  VARCHAR(128)
           GENERATED ALWAYS AS (SESSION_USER) IMPLICITLY HIDDEN,
Last_op    CHAR (1)
           GENERATED ALWAYS AS (DATA CHANGE OPERATION) IMPLICITLY HIDDEN,
sys_beg    TIMESTAMP (12) NOT NULL
           GENERATED ALWAYS AS ROW BEGIN IMPLICITLY HIDDEN,
Sys_end    TIMESTAMP (12) NOT NULL
           GENERATED ALWAYS AS ROW END IMPLICITLY HIDDEN,
trans_start TIMESTAMP (12)
           GENERATED ALWAYS AS TRANSACTION START ID IMPLICITLY HIDDEN,
PERIOD BUSINESS_TIME (Eff_Beg, Eff_End),
PERIOD SYSTEM_TIME(sys_beg, sys_end),
PRIMARY KEY (Bk, Client, BUSINESS_TIME WITHOUT OVERLAPS)
) IN DATABASE apptmpd;

CREATE TABLE policy_hist LIKE policy in DATABASE apptmpd ;

ALTER TABLE policy
ADD VERSIONING USE HISTORY TABLE policy_hist ON DELETE ADD EXTRA ROW;

CREATE UNIQUE INDEX ix_policy
ON Policy (Bk, Client, BUSINESS_TIME WITHOUT OVERLAPS);

```

Next, we issue:

```
SELECT * FROM policy
```

This shows the following table (Table 4-2).

Table 4-2 New table

Bk	Eff-Beg	Eff-End	Client	Type	Copoly
----	---------	---------	--------	------	--------

As in our case, we defined the non-deterministic generated columns (Last_user and Last_op) with the IMPLICITLY HIDDEN attribute. A query that wants to retrieve those columns needs to explicitly specify the names of the columns in the select list to retrieve the values (Example 4-8 and Table 4-3).

Example 4-8 Select

```

SELECT bk,eff_beg,eff_end,client,type,copoly,
substr(last_user,1,8) as last_user,last_op,
sys_beg,sys_end
FROM policy ;

```

Table 4-3 Policy table

Bk	Eff-Beg	Eff-End	Client	Type	Copoly
Last_user		Last_op	Sys_beg		Sys_end

4.4.2 Step-by-step usage scenarios

In this section, we proceed through each step in the flow of events in our scenario.

Step 1

This step is shown in Table 4-4.

Table 4-4 Basic flow of events

Step	Date	Activity
1	1/1/2004 (Future)	Issue PPO policy P138 to customer C882 with copay amount \$10 starting from 2/1/2004

We insert one row into table POLICY as shown in Example 4-9 and Figure 4-2.

Example 4-9 Step 1

```
INSERT INTO POLICY
VALUES ('P138','2/1/2004','12/31/9999','C882','PPO','$10');
```

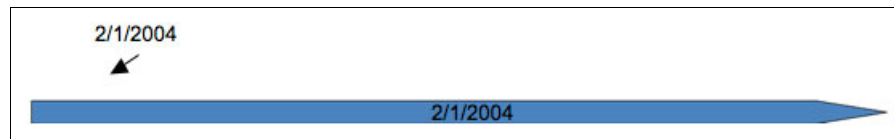


Figure 4-2 Step 1

This adds the row in the table as shown in Table 4-5.

Table 4-5 Policy table

Bk	Eff-Beg	Eff-End	Client	Type	Copay
P138	2/1/2004	12/31/9999	C882	PPO	\$10
Last_user	Last_op	Sys_beg	Sys_end		
MKRES6	I	2004-01-01-17.28.53.6941 26188001	9999-12-30-00.00.00.00000000 00000		

There are no rows in policy_hist.

Step 2

The next step is executed (Table 4-6).

Table 4-6 Basic flow of events

Step	Date	Activity
1	1/1/2004 (Future)	Issue PPO policy P138 to customer C882 with copay amount \$10 starting from 2/1/2004
2	9/1/2004 (Present)	Customer called and changed to HMO as of today

We execute the following temporal update as shown in Example 4-10 and Figure 4-3.

Example 4-10 Step 2

```
UPDATE POLICY FOR PORTION OF BUSINESS_TIME
FROM '9/1/2004' TO '12/31/9999'
SET Type='HMO'
WHERE Bk='P138'AND Client='C882';
```

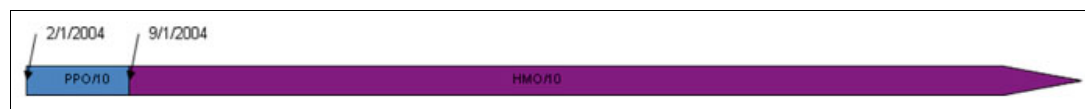


Figure 4-3 Step 2

And the data in the table is as shown in Figure 4-4.

Bk	Eff-Beg	Eff-End	Client	Type	Copay
P138	2/1/2004	9/1/2004	C882	PPO	\$10
P138	9/1/2004	12/31/9999	C882	HMO	\$10

Figure 4-4 Rows in the table

The Policy table is updated as shown in Table 4-7.

Table 4-7 Policy table

Bk	Eff-Beg	Eff-End	Client	Type	Copay
P138	2/1/2004	9/1/2004	C882	PPO	\$10
Last_user	Last_op	Sys_beg	Sys_end		
MKRES6	I	2004-09-01-17.40.10.5234 41733001	9999-12-30-00.00.00.00000000 00000		
Bk	Eff-Beg	Eff-End	Client	Type	Copay
P138	9/1/2004	12/31/9999	C882	HMO	\$10
Last_user	Last_op	Sys_beg	Sys_end		
MKRES6	U	2004-09-01-17.40.10.5234 41733001	9999-12-30-00.00.00.00000000 00000		

The Policy_hist table is shown in Table 4-8.

Table 4-8 Policy_hist table

Bk	Eff-Beg	Eff-End	Client	Type	Copay
P138	2/1/2004	12/31/9999	C882	PPO	\$10
Last_user	Last_op	Sys_beg	Sys_end		
MKRES6	I	2004-01-01-17.28.53.6941 26188001	2004-09-01-17.40.10.5234417 33001		

Note here that the new row was inserted into the base table to cover the application period before the change. The original row was written to the history table.

Step 3

The next step is executed (Table 4-9).

Table 4-9 Basic flow of events

Step	Date	Activity
1	1/1/2004 (Future)	Issue PPO policy P138 to customer C882 with copay amount \$10 starting from 2/1/2004
2	9/1/2004 (Present)	Customer called and changed to HMO as of today
3	3/1/2006 (Future)	Copay increase to \$15 starting 1/1/2007

Again, we execute a temporal update (Example 4-11).

Example 4-11 Step 3

```
UPDATE POLICY FOR PORTION OF BUSINESS_TIME
FROM DATE '1/1/2007' TO '12/31/9999'
SET Copay='$15'
WHERE Bk='P138' AND Client='C882';
```

After the update, the rows in the table are as shown in Figure 4-5 and Figure 4-6.

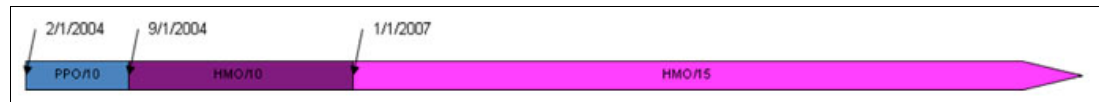


Figure 4-5 Step 3

Bk	Eff-Beg	Eff-End	Client	Type	Copay
P138	2/1/2004	9/1/2004	C882	PPO	\$10
P138	9/1/2004	1/1/2007	C882	HMO	\$10
P138	1/1/2007	12/31/9999	C882	HMO	\$15

Figure 4-6 Rows in the table

The Policy table is updated as shown in Table 4-10.

Table 4-10 Policy table

Bk	Eff-Beg	Eff-End	Client	Type	Copay
P138	2/1/2004	9/1/2004	C882	PPO	\$10
Last_user	Last_op	Sys_beg	Sys_end		
MKRES6	I	2004-09-01-17.40.10.5234 41733001	9999-12-30-00.00.00.00000000 00000		
Bk	Eff-Beg	Eff-End	Client	Type	Copay
P138	9/1/2004	1/1/2007	C882	HMO	\$10
Last_user	Last_op	Sys_beg	Sys_end		
MKRES6	I	2006-03-01-09.14.04.2278 40961001	9999-12-30-00.00.00.00000000 00000		

Bk	Eff-Beg	Eff-End	Client	Type	Copay
P138	1/1/2007	12/31/9999	C882	HMO	\$15
Last_user	Last_op	Sys_beg	Sys_end		
MKRES6	U	2006-03-01-09.14.04.2278 40961001	9999-12-30-00.00.00.0000000 00000		

The Policy_hist table is shown in Table 4-11.

Table 4-11 Policy_hist table

Bk	Eff-Beg	Eff-End	Client	Type	Copay
P138	2/1/2004	12/31/9999	C882	PPO	\$10
Last_user	Last_op	Sys_beg	Sys_end		
MKRES6	I	2004-01-01-17.28.53.6941 26188001	2004-09-01-17.40.10.5234417 33001		
Bk	Eff-Beg	Eff-End	Client	Type	Copay
P138	9/1/2004	12/31/9999	C882	HMO	\$10
Last_user	Last_op	Sys_beg	Sys_end		
MKRES6	U	2004-09-01-17.40.10.5234 4173300	2006-03-01-09.14.04.2278409 610		

Here, the row affected by the update operation is updated with the new eff-beg value. The original row is written to the history table and a new row is inserted into the base table for the application period to 1/1/2007. The current period row has an eff-beg value of 1/1/2007 and the updated Copay value.

Step 4

The next step is executed (Table 4-12).

Table 4-12 Basic flow of events

Step	Date	Activity
1	1/1/2004 (Future)	Issue PPO policy P138 to customer C882 with copay amount \$10 starting from 2/1/2004
2	9/1/2004 (Present)	Customer called and changed to HMO as of today
3	3/1/2006 (Future)	Copay increase to \$15 starting 1/1/2007
4	6/1/2008 (Present)	Cancel policy as of today

Now, we execute a temporal delete on the table (Example 4-12).

Example 4-12 Step 4

```
DELETE FROM POLICY FOR PORTION OF BUSINESS_TIME
FROM '6/1/2008' TO '12/31/9999'
WHERE Bk='P138' AND Client='C882';
```

The values are changed as shown in Figure 4-7 and Figure 4-8.

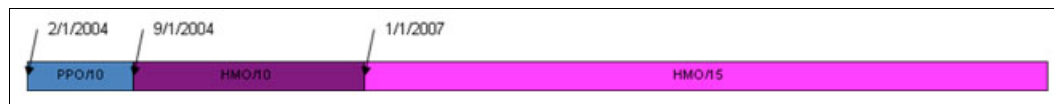


Figure 4-7 Step 4

Bk	Eff-Beg	Eff-End	Client	Type	Copay
P138	2/1/2004	9/1/2004	C882	PPO	\$10
P138	9/1/2004	1/1/2007	C882	HMO	\$10
P138	1/1/2007	6/1/2008	C882	HMO	\$15

Figure 4-8 Rows in the table

The Policy table is updated as shown in Table 4-13.

Table 4-13 Policy table

Bk	Eff-Beg	Eff-End	Client	Type	Copay
P138	2/1/2004	9/1/2004	C882	PPO	\$10
Last_user	Last_op	Sys_beg	Sys_end		
MKRES6	I	2004-09-01-17.40.10.5234 41733001	9999-12-30-00.00.00.0000000 00000		
Bk	Eff-Beg	Eff-End	Client	Type	Copay
P138	9/1/2004	1/1/2007	C882	HMO	\$10
Last_user	Last_op	Sys_beg	Sys_end		
MKRES6	I	2006-03-01-09.14.04.2278 40961001	9999-12-30-00.00.00.0000000 00000		
Bk	Eff-Beg	Eff-End	Client	Type	Copay
P138	1/1/2007	6/1/2008	C882	HMO	\$15
Last_user	Last_op	Sys_beg	Sys_end		
MKRES6	I	2008-06-01-10.39.53.4783 2052200	9999-12-30-00.00.00.0000000 00000		

The Policy_hist table is shown in Table 4-14.

Table 4-14 Policy_hist table

Bk	Eff-Beg	Eff-End	Client	Type	Copay
P138	2/1/2004	12/31/9999	C882	PPO	\$10
Last_user	Last_op	Sys_beg	Sys_end		
MKRES6	I	2004-01-01-17.28.53.6941 26188001	2004-09-01-17.40.10.5234417 33001		
Bk	Eff-Beg	Eff-End	Client	Type	Copay
P138	9/1/2004	12/31/9999	C882	HMO	\$10

Last_user	Last_op	Sys_beg	Sys_end			
MKRES6	U	2004-09-01-17.40.10.5234 4173300	2006-03-01-09.14.04.2278409 610			
Bk	Eff-Beg		Eff-End	Client	Type	Copay
P138	7/1/2007		12/31/9999	C882	HMO	\$15
Last_user	Last_op	Sys_beg	Sys_end			
MKRES6	U	2006-03-01-09.14.04.2278 40961001	2008-06-01-10.39.53.4783205 22001			
Bk	Eff-Beg		Eff-End	Client	Type	Copay
P138	1/1/2007		12/31/9999	C882	HMO	\$15
Last_user	Last_op	Sys_beg	Sys_end			
MKRES6	D	2008-06-01-10.39.53.4783 20522001	2008-06-01-10.39.53.4783205 22001			

The delete statement wrote the current effective row to the history table and inserted a new row into the base table with the eff-end date of the policy cancellation. As we defined system-period data versioning with the ADD ROW ON DELETE clause, there is the extra row in the history table with the last_op value of D.

Step 5

The next step is executed (Table 4-15).

Table 4-15 Basic flow of events

Step	Date	Activity
1	1/1/2004 (Future)	Issue PPO policy P138 to customer C882 with copay amount \$10 starting from 2/1/2004
2	9/1/2004 (Present)	Customer called and changed to HMO as of today
3	3/1/2006 (Future)	Copay increase to \$15 starting 1/1/2007
4	6/1/2008 (Present)	Cancel policy as of today
5	9/1/2008 (Past)	Correct error by retroactively updating policy to POS from 5/01/06 to 10/01/07

At last, we execute the temporal update to change the policy to POS from 5/01/06 to 10/01/07 (Example 4-13).

Example 4-13 Step 5

```
UPDATE POLICY FOR PORTION OF BUSINESS_TIME
FROM DATE '5/1/2006' TO DATE '10/1/2007'
SET Type='POS'
WHERE Bk='P138' AND Client='C882' ;
```

Here is the data in the table after the temporal update as shown in Figure 4-9 and Figure 4-10.

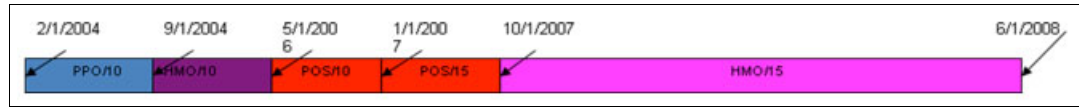


Figure 4-9 Step 5

Bk	Eff-Beg	Eff-End	Client	Type	Copay
P138	2/1/2004	9/1/2004	C882	PPO	\$10
P138	9/1/2004	5/1/2006	C882	HMO	\$10
P138	5/1/2006	1/1/2007	C882	POS	\$10
P138	1/1/2007	10/1/2007	C882	POS	\$15
P138	10/1/2007	6/1/2008	C882	HMO	\$15

Figure 4-10 Rows in the table

The Policy table is updated as shown in Table 4-16.

Table 4-16 Policy table

Bk	Eff-Beg	Eff-End	Client	Type	Copay
P138	2/1/2004	9/1/2004	C882	PPO	\$10
Last_user	Last_op	Sys_beg	Sys_end		
MKRES6	I	2004-09-01-17.40.10.5234 41733001	9999-12-30-00.00.00.0000000 00000		
Bk	Eff-Beg	Eff-End	Client	Type	Copay
P138	9/1/2004	5/1/2006	C882	HMO	\$10
Last_user	Last_op	Sys_beg	Sys_end		
MKRES6	I	2008-09-01-10.55.49.0666 06574001	9999-12-30-00.00.00.0000000 00000		
Bk	Eff-Beg	Eff-End	Client	Type	Copay
P138	5/1/2006	1/1/2007	C882	POS	\$10
Last_user	Last_op	Sys_beg	Sys_end		
MKRES6	U	2008-09-01-10.55.49.0666 06574001	9999-12-30-00.00.00.0000000 00000		
Bk	Eff-Beg	Eff-End	Client	Type	Copay
P138	1/1/2007	10/1/2007	C882	POS	\$15
Last_user	Last_op	Sys_beg	Sys_end		
MKRES6	U	2008-09-01-10.55.49.0666 06574001	9999-12-30-00.00.00.0000000 00000		
Bk	Eff-Beg	Eff-End	Client	Type	Copay
P138	10/1/2007	6/1/2008	C882	HMO	\$15

Last_user	Last_op	Sys_beg	Sys_end	
MKRES6	I	2008-09-01-10.55.49.0666 06574001	9999-12-30-00.00.00.0000000 00000	

The Policy_hist table is shown in Table 4-17.

Table 4-17 Policy_hist table

Bk	Eff-Beg	Eff-End	Client	Type	Copay
P138	2/1/2004	12/31/9999	C882	PPO	\$10
Last_user	Last_op	Sys_beg	Sys_end		
MKRES6	I	2004-01-01-17.28.53.6941 26188001	2004-09-01-17.40.10.5234417 33001		
Bk	Eff-Beg	Eff-End	Client	Type	Copay
P138	9/1/2004	12/31/9999	C882	HMO	\$10
Last_user	Last_op	Sys_beg	Sys_end		
MKRES6	U	2004-09-01-17.40.10.5234 4173300	2006-03-01-09.14.04.2278409 610		
Bk	Eff-Beg	Eff-End	Client	Type	Copay
P138	9/1/2004	1/1/2007	C882	HMO	\$10
Last_user	Last_op	Sys_beg	Sys_end		
MKRES6	I	2006-03-01-09.14.04.2278 40961001	2008-09-01-10.55.49.0666065 74001		
Bk	Eff-Beg	Eff-End	Client	Type	Copay
P138	1/1/2007	12/31/9999	C882	HMO	\$15
Last_user	Last_op	Sys_beg	Sys_end		
MKRES6	U	2006-03-01-09.14.04.2278 40961001	2008-06-01-10.39.53.4783205 22001		
Bk	Eff-Beg	Eff-End	Client	Type	Copay
P138	1/1/2007	12/31/9999	C882	HMO	\$15
Last_user	Last_op	Sys_beg	Sys_end		
MKRES6	D	2008-06-01-10.39.53.4783 20522001	2008-06-01-10.39.53.4783205 22001		
Bk	Eff-Beg	Eff-End	Client	Type	Copay
P138	1/1/2007	6/1/2008	C882	HMO	\$15
Last_user	Last_op	Sys_beg	Sys_end		
MKRES6	I	2008-06-01-10.39.53.4783 20522001	2008-09-01-10.55.49.0666065 74001		

The changes that crossed application periods here involved inserting two new rows for application periods that were unaffected and updating two rows. The two updates resulted in two rows written to the history table.

4.4.3 Bitemporal queries

This section follows up our case study with queries that can be run with bitemporal tables.

The business time query shown in Example 4-14 reflects current system time. That is, since a period specification is not specified for SYSTEM_TIME, the query is applied to data that is considered “current”.

Example 4-14 Select statement

```
SELECT * FROM POLICY FOR BUSINESS_TIME AS OF '7/1/2007'
WHERE BK='P138' AND CLIENT='C882';
```

This returns the results shown in Table 4-18.

Table 4-18 Policy table

Bk	Eff-Beg	Eff-End	Client	Type	Copay
P138	1/1/2007	10/1/2007	C882	POS	\$15

If we wanted to know the client policy detail that was used at a particular point, we would add a FOR SYSTEM_TIME period specification (Example 4-15).

Example 4-15 Select statement

```
SELECT * FROM POLICY FOR BUSINESS_TIME AS OF '7/1/2007'
      FOR SYSTEM_TIME AS OF '2015-06-24-17.40.10'
WHERE BK='P138' AND CLIENT='C882';
```

This returns the results shown in Table 4-19.

Table 4-19 Policy table

Bk	Eff-Beg	Eff-End	Client	Type	Copay
P138	2/1/2004	12/31/9999	C882	POS	\$10

The business may request the following information: “Show me the history of changes to policy for client C882 when the policy was active on the 1st July 2007”. The BUSINESS_TIME AS OF clause specifies that data should be returned for the policy that was active on 1st July 2007.

The query might look like Example 4-16.

Example 4-16 Select statement

```
SELECT BK,EFF_BEG,EFF_END,CLIENT,TYPE,COPAY,
CHAR(LAST_USER,8) AS LAST_USER, LAST_OP, SYS_BEG,SYS_END

FROM POLICY FOR BUSINESS_TIME AS OF '7/1/2007'
      FOR SYSTEM_TIME FROM TIMESTAMP('0001-01-01')
                        TO CURRENT_TIMESTAMP
WHERE BK='P138' AND CLIENT='C882'
ORDER BY SYS_BEG DESC,BK      ;
```

In this type of query, we want to retrieve the system period columns that we defined as IMPLICITLY HIDDEN, so the columns need to be explicitly specified in the select list of a query (Table 4-20).

Table 4-20 Policy table

Bk	Eff-Beg	Eff-End	Client	Type	Copay
P138	1/1/2007	10/1/2007	C882	PPO	\$15
Last_user	Last_op	Sys_beg	Sys_end		
MKRES6		2015-06-25-10.55.49.0666 06574001	9999-12-30-00.00.00.00000000 00000		
Bk	Eff-Beg	Eff-End	Client	Type	Copay
P138	1/1/2007	6/1/2007	C882	HMO	\$15
Last_user	Last_op	Sys_beg	Sys_end		
MKRES6		2015-06-25-10.39.53.4783 20522001	2015-06-25-10.55.49.0666065 74001		
Bk	Eff-Beg	Eff-End	Client	Type	Copay
P138	1/1/2007	12/31/9999	C882	HMO	\$15
Last_user	Last_op	Sys_beg	Sys_end		
MKRES6		2015-06-25-09.14.04.2278 40961001	2015-06-25-10.39.53.4783205 22001		
Bk	Eff-Beg	Eff-End	Client	Type	Copay
P138	9/1/2004	12/31/9999	C882	HMO	\$10
Last_user	Last_op	Sys_beg	Sys_end		
MKRES6		2015-06-24-17.40.10.5234 41733001	2015-06-25-09.14.04.2278409 61001		
Bk	Eff-Beg	Eff-End	Client	Type	Copay
P138	9/1/2004	12/31/9999	C882	HMO	\$10
Last_user	Last_op	Sys_beg	Sys_end		
MKRES6		2015-06-24-17.28.53.6941 26188001	2015-06-24-17.40.10.5234417 33001		



Additional considerations

This chapter covers the temporal enhancements of IBM DB2 11 for z/OS, which includes temporal support for views, temporal special registers, and auditing used with temporal tables. Additionally, information is provided about how temporal tables can improve performance and considerations for designing a history table.

5.1 Temporal support for views

Views have been used by DB2 clients intensively to get more data access control. DB2 V11 removed the V10 restriction that a period specification or period clause could only be specified with references to base tables, which are defined as the appropriate type of temporal table.

Beginning with V11, for temporal applications with references to views, a period specification or period clause can be specified for a view of a relevant type of temporal table, without modifying the existing view definition. If a period specification or period clause is specified for a view that does not reference a temporal table, the clause is ignored. This is a major enhancement to the original temporal support, which performs better and is easier to manage compared to home-grown solutions.

5.1.1 Queries

In DB2 V11, any of the supported period specifications may be specified following the name of a view. A period specification after the name of a view in a table reference applies to all of the temporal table references in the fullselect of the definition of that view:

- ▶ A period specification referencing the `SYSTEM_TIME` period is applied to all system-period temporal tables referenced in the definition of the view.
- ▶ A period specification referencing the `BUSINESS_TIME` period is applied to all application-period temporal tables referenced in the definition of the view.
- ▶ A query can reference a combination of regular tables, system-period temporal tables, application-period temporal tables, and bitemporal tables and include period specifications.
- ▶ A period specification is ignored if the view does not reference a temporal table containing the period referenced in the period specification.
- ▶ Multiple period specifications cannot be applied to a table reference.
- ▶ A period specification must not be specified for a view that is defined with a reference to a user-defined function.

5.1.2 Data change operations on views

In DB2 V11, an update or delete statement for a view could include a period clause for the `BUSINESS_TIME` period subject to the following conditions:

- ▶ The `FROM` clause of the outer fullselect of the view definition must reference, directly or indirectly, an application-period temporal table.
- ▶ The result table of the outer fullselect of the view definition explicitly, or implicitly, includes the start and end columns of the `BUSINESS_TIME` period. This restriction only applies to an update statement.
- ▶ The view definition must not contain a period specification in the `FROM` clause of the outer fullselect of the view definition.
- ▶ An `INSTEAD OF` trigger must not exist for the view.

A period clause starts with the following syntax:

```
FOR PORTION OF BUSINESS_TIME
```

5.1.3 Query of a view using a period specification for SYSTEM_TIME

As a user, you want to query a view that references a system-period temporal table or a bitemporal table, and specify a point in time or time range (period specification) for a SYSTEM_TIME period. You want to get the same behavior whether querying a base table or a view. Assume that the definition of your view does not include a specification for a point in time or time range for a SYSTEM_TIME period.

Assume that there is a system-period temporal table STT, a bitemporal table BTT, and a regular table RTT. For a query that references all three of these tables, a period specification for SYSTEM_TIME applies to all of the underlying system-period temporal tables. In this example, the period specification applies to STT and BTT, both of which have a SYSTEM_PERIOD period and data versioning (Example 5-1).

Example 5-1 Case 1 create view

```
CREATE VIEW v0 (col1, col2, col3) AS SELECT * FROM RT, STT, BTT;  
SELECT * FROM v0  
FOR SYSTEM_TIME AS OF TIMESTAMP '2010-01-10 10:00:00';
```

This is transformed to the following query against the base tables (Example 5-2).

Example 5-2 Case 1 select

```
SELECT * FROM  
  (SELECT * FROM RT,  
    STT FOR SYSTEM_TIME AS OF TIMESTAMP '2010-01-10 10:00:00',  
    BTT FOR SYSTEM_TIME AS OF TIMESTAMP '2010-01-10 10:00:00'  
  );
```

5.1.4 Query on a view using a period specification for BUSINESS_TIME

As a user, you want to query a view that references an application-period temporal table or a bitemporal table, and specify a point in time or time range (period specification) for a BUSINESS_TIME period. You want to get the same behavior whether querying a base table or a view. Assume that the definition of your view does not include a specification for a point in time or time range for a BUSINESS_TIME period.

Assume that there is an application-period temporal table ATT, a bitemporal table BTT, and a regular table RTT. The period specification for BUSINESS_TIME applies to all of the underlying application-period temporal tables. In this example, the period specification applies to ATT and BTT, both of which have a BUSINESS_PERIOD period (Example 5-3).

Example 5-3 Case 2 create view

```
CREATE VIEW v0 (col1, col2, col3) AS SELECT * FROM RT, ATT, BTT;  
SELECT * FROM v0  
FOR BUSINESS_TIME AS OF TIMESTAMP '2010-01-10 10:00:00';
```

This is transformed to the following query against the base tables (Example 5-4).

Example 5-4 Case 2 select

```
SELECT * FROM
  (SELECT * FROM RT,
    ATT FOR BUSINESS_TIME AS OF TIMESTAMP '2010-01-10 10:00:00',
    BTT FOR BUSINESS_TIME AS OF TIMESTAMP '2010-01-10 10:00:00'
  );
```

5.1.5 Data change operations on temporal views

As a user, you would like to specify a period of time for a BUSINESS_TIME period to apply an update or delete operation on a view that references an application-period temporal table or bitemporal table. You want to get the same behavior for a data change operation whether the target is a base table or a view. Assume that a trigger is not defined for your view.

Table ATT is an application-period temporal table (Example 5-5).

Example 5-5 Case 3 create view

```
CREATE VIEW v8 (col1, col2, col3) AS SELECT * FROM ATT;
UPDATE v8
FOR PORTION OF BUSINESS_TIME FROM '2009-01-01' TO '2009-06-01'
SET c2 = c2 + 1.10;
```

This is transformed to the following query against the base table (Example 5-6).

Example 5-6 Case 3 update

```
UPDATE ATT
FOR PORTION OF BUSINESS_TIME FROM '2009-01-01' TO '2009-06-01'
SET c2 = c2 + 1.10
```

5.2 Time travel: Temporal special registers

Clients would like to get data at a certain point of time from temporal tables without changing SQL query and data change statements. Consider a scenario where the client has a packaged application that they would like to run against the state of the business AS OF today, the same application against the state of the business AS OF the end of last quarter, the same application against the state of the business AS OF last year, and so on, changing the application and adding AS OF period specifications to each SQL statement. This is cumbersome, and might not even be possible in case of packaged applications.

Two new special registers (CURRENT TEMPORAL SYSTEM_TIME and CURRENT TEMPORAL BUSINESS_TIME) and two new bind options (SYSTIMESENSITIVE and BUSTIMESENSITIVE) for BIND and REBIND commands are introduced. The two temporal registers have an effect on SQL statements that reference temporal tables directly, or indirectly through a view in static and dynamic data manipulation statements that are bound with the new bind options set to YES. In addition, two new routine options (SYSTEM TIME SENSITIVE and BUSINESS TIME SENSITIVE) are introduced to control the effects of the temporal registers for CREATE and ALTER statements for SQL routines. These new routine options correspond to the new bind options for external routines and normal packages.

When a temporal register is in effect, DB2 generates an implicit period specification for the associated period in SQL statements that reference that period. By setting a temporal register to a specific point in time, you can retrieve data from temporal tables without the need to add explicit period specifications and predicates to their original SQL statements. In this way, DB2 provides a time machine that requires fewer application changes for temporal support.

5.2.1 Two temporal special registers

Two new special registers `CURRENT TEMPORAL SYSTEM_TIME` and `CURRENT TEMPORAL BUSINESS_TIME` are introduced. As with other special registers, the temporal registers are storage areas that are defined for an application process by DB2 and they are used to store information that can be referenced in SQL statements. A reference to the temporal special register is a reference to the value at the current server:

- ▶ **Scope:** The temporal special registers are defined for a session.
- ▶ For temporal queries, when a temporal special register is in effect, the appropriate period specification is implicitly added to SQL statements that reference the corresponding temporal period. That is, a “`FOR SYSTEM_TIME AS OF CURRENT TEMPORAL SYSTEM_TIME`” clause is added to a statement that references a system-period temporal table, and a “`FOR BUSINESS_TIME AS OF CURRENT TEMPORAL BUSINESS_TIME`” clause is implicitly added to a statement that references an application-period temporal table.
- ▶ Set the temporal special registers to a specific point in time to affect subsequent SQL statements, including those from invoked functions, stored procedures, and triggers. This allows an application to see data for a specific point in time without modifying the SQL statements in multiple packages or routines.
- ▶ The data type of the special registers is `TIMESTAMP(12)`, and the default value is `NULL`.

5.2.2 Enabling or disabling the temporal special registers

The following options are added to control sensitivity to the settings of the `CURRENT TEMPORAL SYSTEM_TIME` and `CURRENT TEMPORAL BUSINESS_TIME` special registers.

SYSTIMESENSITIVE and BUSTIMESENSITIVE bind options

Two new bind options allow an application to choose whether to be sensitive to the temporal special registers:

- ▶ `CURRENT TEMPORAL SYSTEM_TIME`
 - `YES` -- References to system-period temporal tables are affected by the value of the `CURRENT TEMPORAL SYSTEM_TIME` special register.
 - `NO` -- References to system-period temporal tables are not affected by the value of the `CURRENT TEMPORAL SYSTEM_TIME` special register.
- ▶ `CURRENT TEMPORAL BUSINESS_TIME`
 - `YES` -- References to application-period temporal tables are affected by the value of the `CURRENT TEMPORAL BUSINESS_TIME` special register.
 - `NO` -- References to application-period temporal tables are not affected by the value of the `CURRENT TEMPORAL BUSINESS_TIME` special register.

Both bind options default to `YES`.

SYSTEM_TIME SENSITIVE and BUSINESS_TIME SENSITIVE routine options

Similar routine options are provided for SQL routines on the following SQL statements:

- ▶ CREATE FUNCTION (compiled SQL scalar)
- ▶ ALTER FUNCTION (compiled SQL scalar)
- ▶ CREATE PROCEDURE (SQL native)
- ▶ ALTER PROCEDURE (SQL native)

The default value for the routine options is YES.

SYSTEM_TIME SENSITIVE and BUSINESS_TIME SENSITIVE options on DB2I panels

The bind options are specified on the following DB2I panels:

- ▶ DB2I Panel DSNEBP10
- ▶ DB2I Panel DSNEBP11
- ▶ DB2I Panel DSNEBP19

5.2.3 How temporal registers affect queries

When an application-period temporal table is referenced and the value in effect for the CURRENT TEMPORAL BUSINESS_TIME special register is not the null value, the following period specification is implicitly added:

FOR BUSINESS_TIME AS OF CURRENT TEMPORAL BUSINESS_TIME.

When a system-period temporal table is referenced and the value in effect for the CURRENT TEMPORAL SYSTEM_TIME special register is not the null value, the following period specification is implicitly added:

FOR SYSTEM_TIME AS OF CURRENT TEMPORAL SYSTEM_TIME.

Assume that there is an application-period temporal table ATT, a system-period temporal table STT, a bitemporal table BTT, a regular table RT, and a view is created as shown in Example 5-7.

Example 5-7 Create view

```
CREATE VIEW VW
AS
SELECT ... FROM RT, ATT, STT, BTT;
```

Given the following configuration:

- ▶ SYSTIMESENSITIVE = YES and BUSTIMESENSITIVE = YES
- ▶ CURRENT TEMPORAL SYSTEM_TIME or CURRENT TEMPORAL BUSINESS_TIME is NOT NULL

The following query against a view without any explicit period specifications in the query (Example 5-8).

Example 5-8 User-specified SELECT statement

```
SELECT ... FROM VW;
```

Is transformed to the following query against the view (Example 5-9).

Example 5-9 Transformed SELECT statement: Intermediate version

```
SELECT ... FROM VW
  FOR SYSTEM_TIME AS OF CURRENT TEMPORAL SYSTEM_TIME
  FOR BUSINESS_TIME AS OF CURRENT TEMPORAL BUSINESS_TIME;
```

Which is in turn transformed to the following query against the base tables (Example 5-10).

Example 5-10 Transformed SELECT statement: Final version

```
SELECT ... FROM RT,
  ATT FOR BUSINESS_TIME AS OF CURRENT TEMPORAL BUSINESS_TIME,
  STT FOR SYSTEM_TIME AS OF CURRENT TEMPORAL SYSTEM_TIME,
  BTT FOR BUSINESS_TIME AS OF CURRENT TEMPORAL BUSINESS_TIME
  FOR SYSTEM_TIME AS OF CURRENT TEMPORAL SYSTEM_TIME ;
```

5.2.4 How temporal affects UPDATE and DELETE statements

When an application-period temporal table is the target of a searched UPDATE or searched DELETE statement, the following additional search condition is implicitly added (Example 5-11).

Example 5-11 Additional search condition

```
(CURRENT TEMPORAL BUSINESS_TIME IS NULL OR
 (BUS_START <= CURRENT TEMPORAL BUSINESS_TIME AND
  BUS_END > CURRENT TEMPORAL BUSINESS_TIME));
```

Where BUS_START and BUS_END are the begin and end columns of the BUSINESS_TIME period of the target table of the UPDATE or DELETE statement.

If the CURRENT TEMPORAL SYSTEM_TIME special register is set to a non-null value, the underlying target (direct or indirect) of INSERT, UPDATE, or DELETE statement cannot be a system-period temporal table or bitemporal table. The restriction also applies to insert and update operations of MERGE statement.

Assume that there is an application-period temporal table ATT and a view is created as follows, and BUSTIMESENSITIVE = YES (Example 5-12).

Example 5-12 Create view

```
CREATE VIEW VW (VW_COL1, ...)
  AS SELECT C01, ... FROM ATT
  WHERE COL2 = 'ABC';
```

The following UPDATE statement for an application-period temporal table (Example 5-13).

Example 5-13 User-specified UPDATE statement

```
UPDATE ATT
SET COL1 = 123
WHERE COL1 = 234;
```

Is transformed into the following statement with references to the BUSINESS_TIME temporal special register (Example 5-14).

Example 5-14 Transformed UPDATE statement

```
UPDATE ATT
SET COL1 = 123
WHERE COL1 = 234 AND
(CURRENT TEMPORAL BUSINESS_TIME IS NULL OR
(BUS_START <= CURRENT TEMPORAL BUSINESS_TIME AND
BUS_END > CURRENT TEMPORAL BUSINESS_TIME));
```

The following UPDATE statement for an application-period temporal table (Example 5-15).

Example 5-15 User-specified UPDATE statement

```
UPDATE VW
SET COL1 = 123
WHERE COL1 = 234 AND COL2 = 'ABC';
```

Is transformed into the following statement with references to the BUSINESS_TIME temporal special register (Example 5-16).

Example 5-16 Transformed UPDATE statement

```
UPDATE VW
SET COL1 = 123
WHERE COL1 = 234 AND COL2 = 'ABC' AND
(CURRENT TEMPORAL BUSINESS_TIME IS NULL OR
(BUS_START <= CURRENT TEMPORAL BUSINESS_TIME AND
BUS_END > CURRENT TEMPORAL BUSINESS_TIME));
```

The following UPDATE statement with a FOR PORTION OF clause against a view defined on an application-period temporal table (Example 5-17).

Example 5-17 User-specified UPDATE statement with FOR PORTION clause

```
UPDATE VW
  FOR PORTION OF BUSINESS_TIME FROM value1 TO value2
  SET VW_COL1 = 123
WHERE VW_COL1 = 234;
```

Is transformed into the following statement with references to the BUSINESS_TIME temporal special register (Example 5-18).

Example 5-18 Transformed UPDATE statement

```
UPDATE ATT
  FOR PORTION OF BUSINESS_TIME FROM value1 TO value2
  SET COL1 = 123
WHERE COL1 = 234 AND COL2 = 'ABC' AND
(CURRENT TEMPORAL BUSINESS_TIME IS NULL OR
BUS_END > CURRENT TEMPORAL BUSINESS_TIME));
```

5.2.5 How temporal special registers affect statements in user-defined functions

Assume the following:

- ▶ There are three system-period temporal tables STT1, STT2, and STT3
- ▶ In an application, there is an SQL statement:

```
SELECT UDF_1 (STT1.C1) FROM STT1, STT2;
```
- ▶ In a function UDF_1, there is an SQL statement:

```
SELECT * FROM STT3;
```
- ▶ Today is 2012-03-26-17.44.49.000000, and the application wants to find the result of the function from last week.

Approach with DB2 10 temporal support

Using DB2 V10, a user had to:

- ▶ Modify the application, and add an explicit period specification to the SQL statement (Example 5-19).

Example 5-19 Select

```
SELECT UDF_1 (STT1.C1)
      FROM STT1 FOR SYSTEM_TIME AS OF '2012-03-19-17.44.49',
      STT2 FOR SYSTEM_TIME AS OF '2012-03-19-17.44.49' ;
```

- ▶ Modify the function UDF_1, and add an explicit period specification to the SQL statement (Example 5-20).

Example 5-20 Select

```
SELECT * FROM STT3
      FOR SYSTEM_TIME AS OF '2012-03-19-17.44.49' ;
```

- ▶ Precompile, compile, and bind both the application and the function UDF_1.

Approach with DB2 11 temporal support using the temporal special registers

Using DB2 V11, the process is greatly simplified:

- ▶ Assign a value to the CURRENT TEMPORAL SYSTEM_TIME special register. This eliminates the need to change the SQL statements in the application or function UDF_1, before calling the application:

```
SET CURRENT TEMPORAL SYSTEM_TIME = '2012-03-19-17.44.49' ;
```

- ▶ Assign a null value to the special register if the application just wants to access the active data:

```
SET CURRENT TEMPORAL SYSTEM_TIME = NULL ;
```

When the temporal special registers are used, there is no need to change SQL statements in an existing application to get current data only or to get both current and historical data.

The same scenario applies to an application-period temporal table using the CURRENT TEMPORAL BUSINESS_TIME special register.

5.3 Auditing with temporal tables

This section discusses auditing considerations with temporal tables.

5.3.1 Overview and benefit

DB2 V10 system-period data versioning capability provides auditing information about WHEN data is modified. However, it is not enough for many customers just to track WHEN. To meet regulatory compliance requirements, it is common and mandatory to audit and track WHO modified the data, and WHAT action (which SQL statement) caused the data modification.

Without built-in DB2 support, complex application logic (for example, usage of complex triggers) is usually needed, which can result in significant maintenance overhead and poor performance.

DB2 V11 delivers integrated auditing support (with good performance and easy management characteristics as compared to the home-grown auditing solutions) to allow for automatic tracking of the following types of audit information:

- ▶ WHO modified the data in the table
- ▶ WHAT SQL operation modified the data in the table

Non-deterministic generated expression columns

Non-deterministic generated expression columns can be used with a system-period temporal table to provide audit capabilities. New syntax is introduced to define “non-deterministic generated expression columns”. Aside from the use cases with temporal data, these special columns are also useful for non-temporal applications that want to record auditing data.

You can define a non-deterministic generated expression column using the following syntax:

```
GENERATED ALWAYS AS ( expression )
```

Where *expression* is one of the following:

- ▶ DATA CHANGE OPERATION
- ▶ One of the following special registers:
 - CURRENT CLIENT_ACCTNG
 - CURRENT CLIENT_APPLNAME
 - CURRENT CLIENT_CORR_TOKEN
 - CURRENT CLIENT_USERID
 - CURRENT CLIENT_WKSSTNNAME
 - CURRENT SERVER
 - CURRENT SQLID
 - SESSION_USER
- ▶ One of the following session variables:
 - SYSIBM.PACKAGE_NAME
 - SYSIBM.PACKAGE_SCHEMA
 - SYSIBM.PACKAGE_VERSION

ON DELETE ADD EXTRA ROW clause

When defining a table for system-period data versioning, the new ON DELETE ADD EXTRA ROW clause can be specified. The clause is specified on the ALTER TABLE statement (Example 5-21 on page 83).

Example 5-21 Alter table

```
ALTER TABLE stt ADD VERSIONING USE HISTORY TABLE hist_table  
ON DELETE ADD EXTRA ROW;
```

This clause specifies that an extra row is inserted into the associated history table when a row is deleted from a system-period temporal table. The content of the columns of the additional row in the history table are determined as follows:

- ▶ New values are generated for each column that corresponds to a non-deterministic generated expression column.
- ▶ The column that corresponds to the row-begin column is set to the same value as the column that corresponds to the row-end column.
- ▶ The other columns are set to the same value as in the row inserted into the history table for the delete.
- ▶ The ON DELETE ADD EXTRA ROW clause is intended to be used when the system-period temporal table contains a non-deterministic generated expression column.

5.3.2 Using non-deterministic generated expression columns

The following example uses a system-period temporal table and associated history table to track the lifecycle of a bank account (Example 5-22). The example was derived from discussions with customers. A more detailed explanation of the example follows at 5.3.3, “Detailed explanation of the preceding example” on page 85.

The generated expression columns in the system-period temporal table are defined with the GENERATED ALWAYS AS clause. The corresponding columns in the history table are defined as normal columns with a data type that is appropriate for the generated column in the system-period temporal table. In other words, the columns in the history table are not defined as generated expression columns.

Simple integer values are used in place of timestamps for the period columns in this example for simplicity.

Example 5-22 Defining the BANK_ACCOUNT_STT and BANK_ACCOUNT_HIST tables

```
CREATE Table bank_account_stt  
(account_no INT NOT NULL,  
  balance INT,  
  user_id VARCHAR(128) GENERATED ALWAYS AS (SESSION_USER),  
  op_code CHAR(1) GENERATED ALWAYS AS (DATA CHANGE OPERATION),  
  sys_start TIMESTAMP(12) NOT NULL GENERATED ALWAYS AS ROW BEGIN,  
  sys_end TIMESTAMP(12) NOT NULL GENERATED ALWAYS AS ROW END,  
  trans_id TIMESTAMP(12) GENERATED ALWAYS AS TRANSACTION START ID,  
  PERIOD SYSTEM_TIME (sys_start, sys_end));
```

```
CREATE TABLE bank_account_hist  
(account_no INT NOT NULL,  
  balance INT,  
  user_id VARCHAR(128),  
  op_code CHAR(1),  
  sys_start TIMESTAMP(12) NOT NULL,  
  sys_end TIMESTAMP(12) NOT NULL,  
  trans_id TIMESTAMP(12));
```

```
ALTER TABLE bank_account_stt ADD VERSIONING USE HISTORY TABLE bank_account_hist
ON DELETE ADD EXTRA ROW;
```

At time 2, Claire inserts a row into the bank_account_stt table (account_no 13452, balance \$2000):

```
INSERT INTO bank_account_stt (account_no, balance) VALUES (13452, 2000);
```

The current table BANK_ACCOUNT_STT is shown in Figure 5-1.

account_no	balance	user_id	op_code	sys_start	sys_end	...
13452	\$2000	Claire	'I'	2	max	...

Figure 5-1 BANK_ACCOUNT_STT

The history table BANK_ACCOUNT_HIST is empty (Figure 5-2).

account_no	balance	user_id	op_code	sys_start	sys_end	...

Figure 5-2 BANK_ACCOUNT_HIST

At time 6, Steve updates the row increasing the balance by \$500 (account_no 13452, balance \$2500):

```
UPDATE bank_account_stt
SET balance = balance + 500
WHERE account_no = 13452;
```

The current contents of the BANK_ACCOUNT_STT table are shown in Figure 5-3.

account_no	balance	user_id	op_code	sys_start	sys_end	...
13452	\$2500	Steve	'U'	6	max	...

Figure 5-3 BANK_ACCOUNT_STT

The current contents of the history table BANK_ACCOUNT_HIST are shown in Figure 5-4.

account_no	balance	user_id	op_code	sys_start	sys_end	...
13452	\$2000	Claire	'I'	2	6	...

Figure 5-4 BANK_ACCOUNT_HIST

At time 15, Rick deletes all the rows in the BANK_ACCOUNT_STT table. Given that the table currently contains a single row, only that row is deleted.

```
DELETE FROM bank_account_stt;
```

The BANK_ACCOUNT_STT table is now empty as shown in Figure 5-5.

account_no	balance	user_id	op_code	sys_start	sys_end	...
------------	---------	---------	---------	-----------	---------	-----

Figure 5-5 BANK_ACCOUNT_STT

The contents of the history table BANK_ACCOUNT_HIST is shown in Figure 5-6.

account_no	balance	user_id	op_code	sys_start	sys_end	...
13452	\$2000	Claire	'I'	2	6	...
13452	\$2500	Steve	'U'	6	15	
13452	\$2500	Rick	'D'	15	15	

Figure 5-6 BANK_ACCOUNT_HIST

5.3.3 Detailed explanation of the preceding example

A more detailed explanation of this example follows:

Row 1 records the history resulting from the *update statement issued by Steve*. The update caused a row to be inserted into the history table at time 2 with balance=2000, and the row was valid until time 6. Row 1 in the history table records the values of the row in the system-period temporal table before the update. The values in the new history row reflect the values of the row after the original insert by Claire. Assume that a commit was issued.

Row 2 records the history resulting from the *delete statement issued by Rick*. The delete caused a row to be inserted into the history table at time 6 with balance=2500, and the row was valid until time 15. Row 2 in the history table records the values of the row in the system-period temporal table before the delete. The values in the new history row reflect the values of the row after the update by Steve.

Row 3 records additional information about *the delete statement itself*, storing the ID of the user that initiated the delete operation. This row is recorded in the history table because the new ON DELETE ADD EXTRA ROW clause was specified in the definition of the system-period temporal table.

The values (2500, 'Rick', 'D', 15, 15), record that Rick deleted the row at time 15, and the balance was 10 at the time of the delete. Notice that the values of the history table columns that correspond to the row-begin and row-end columns both have the value 15, reflecting the time of the deletion.

Row 2 and row 3 are identical for user data (balance). The difference is in the auditing columns: the new generated expression columns that record who initiated the action, and which data change operation the row represents. A select from the system-period temporal table will not return row 3 from the history table.

5.4 Performance

This section discusses performance considerations.

5.4.1 CURRENT TEMPORAL SYSTEM_TIME special register

For a query of a system-period temporal table or bitemporal table, when the CURRENT TEMPORAL SYSTEM_TIME temporal register is in effect, DB2 implicitly adds the “FOR SYSTEM_TIME AS OF CURRENT TEMPORAL SYSTEM_TIME” clause to the query. The clause applies to all system-period temporal tables or bitemporal tables referenced in the SQL statement. The resulting query is transformed to a 'union all' from both the system-period temporal tables and the associated history tables.

To avoid runtime performance degradation regarding the impacts of the CURRENT TEMPORAL SYSTEM_TIME special register, DB2 generates two sections per static query of a system-period temporal table (STT) or bitemporal table (BTT) reference.

For a static SQL `SELECT * FROM STT1`, if the package is bound with `SYSTIMESENSITIVE='YES'`, DB2 binds this statement twice:

- ▶ Original section: The original `SELECT * FROM STT1` is bound to access current data only. The majority of applications using system-period temporal tables request current data only. There is no performance degradation caused by the union all query transformation.
- ▶ Extended section: The statement is bound again as an extended section by expanding STT1 to UNION ALL of STT1 and its associated history table. This accesses both current and historical data.

At execution time, DB2 determines which section to execute based on the value of the CURRENT TEMPORAL SYSTEM_TIME special register:

- ▶ If the CURRENT TEMPORAL SYSTEM_TIME special register is NULL, the first and regular section is executed.
- ▶ If the CURRENT TEMPORAL SYSTEM_TIME special register contains a timestamp, the extended section that contains the implicit union all query transformation is executed.

Monitoring the impacts at bind time

An SQL statement that references a system-period temporal table may be transformed to union All of itself with the associated history table, and an SQL statement that references an application-period temporal table can be expanded with extra predicates. How would this transformation be monitored?

The EXPLAIN tables provide information about the use of these packages. At bind time, the EXPANSION_REASON column in catalog tables and EXPLAIN tables can be used to monitor the implicit query transformations (SYSIBM.SYSPACKSTMT, SYSIBM.SYSQUERYPLAN, and all EXPLAIN tables).

Given a static data manipulation statement that references a system-period temporal table, two separate sections and plans for two access paths may be found in the SYSIBM.SYSPACKSTMT and PLAN_TABLE tables. The different sections and access paths will have:

- ▶ Same STMTNO
- ▶ Same STATEMENT
- ▶ Different and unique SECTNO; the SECTNO of the extended section is assigned by DB2

- ▶ Different and unique STMT_ID
- ▶ Different EXPANSION_REASON; the extended section with a runtime error may have the same value or a blank

EXPANSION_REASON indicates the impact of the CURRENT TEMPORAL BUSINESS_TIME special register, the CURRENT TEMPORAL SYSTEM_TIME special register, and the SYSIBMADM.GET_ARCHIVE built-in global variable that are controlled by the BUSTIMESENSITIVE, SYSTIMESENSITIVE, and ARCHIVESENSITIVE bind options:

- ▶ **A:** The query was implicitly transformed according to the setting of the GET_ARCHIVE global variable.
- ▶ **B:** The query was implicitly transformed according to the setting of the BUSINESS_TIME special register.
- ▶ **S:** The query was implicitly transformed according to the setting of the SYSTEM_TIME special register.
- ▶ **SB:** The query was implicitly transformed according to the setting of the temporal special registers.
- ▶ **Blank:** The query was not implicitly transformed by the settings of the GET_ARCHIVE global variable, or the CURRENT TEMPORAL BUSINESS_TIME, or CURRENT TEMPORAL SYSTEM_TIME special registers.

5.4.2 Performance summary on temporal support

DB2 provides a capability to specify table-level specifications to control the management of application data based on time.

Application programmers can specify search criteria based on the time the data existed or was valid. This built-in capability simplifies DB2 application development requiring data versioning.

Customers can satisfy compliance laws faster and cheaper because DB2 automatically manages the different versions of data.

System-period temporal tables

The performance expectations are as follows:

- ▶ DB2 provided system-period data versioning support for UPDATE out-performs user-defined triggers by 30 - 39%, and by 10 - 19% for DELETE respectively in DB2 CPU time.
- ▶ UPDATE and DELETE operations against a base table: The performance is affected by the history insert performance, which in turn is controlled by the number of indexes, number of columns in the index, clustering order, space map search, and other information about the history table.
- ▶ INSERT and UPDATE of current data (that is, from the base table) might perform slower than tables not performing data versioning.
- ▶ SELECT against base tables (that is, current data) will perform similar to tables not performing data versioning.
- ▶ SELECT of historical data using the new approach might be slower. The performance overhead of temporal query statement is determined by efficiency of indexing on both base and history table, type of join, and number of table joining.

Considerations for the associated history table

Usually, there are two operations on a history table, which include:

- ▶ For updates and deletes from the system-period temporal table (the base table), a copy of the updated or deleted rows is inserted into the history table.
- ▶ When a query refers to historical data, the query is transformed to access the system-period temporal table with a union all to the history table.

When designing a history table, consider both insert performance and select performance. When the performance for insert and select conflict, there are several general suggestions about how to create a history table:

- ▶ Generally, sequential insert performance is better on a table in a partition by range table space than a table in a partition-by-growth table space.
- ▶ If the history data is to be archived to the Accelerator, define the history table in a partition by range table space, and specify the column of the history table (that corresponds to the row-end column) for the partition key.
- ▶ Considering that most queries do not reference historical data, it is recommended that you do the following:
 - Define as few indexes as possible to minimize the performance impact on inserts.
 - Define the history table as APPEND YES. This improves insert performance.
 - Monitor the table as a candidate for REORG due to low cluster ratio.

Application-period temporal tables

The performance expectations are as follows:

- ▶ 1 - 3% overhead is expected for ensuring that there are no overlapping BUSINESS_TIME periods.
- ▶ The DB2 provided support for application-period temporal tables for automatic row splitting out performs a user-defined stored procedure doing the same processing by 57% to 68% in DB2 CPU time.

For detailed performance data of system-period temporal tables and application-period temporal tables, see the IBM Redbooks publication: *DB2 10 for z/OS Performance Topics*, SG24-7942, chapter 7.1: Temporal Support.



Part 2

IBM DB2 archive transparency

Part 2 contains the following chapters:

- ▶ Chapter 6, “DB2 archive transparency overview” on page 91
- ▶ Chapter 7, “DB2 archive transparency concepts” on page 95
- ▶ Chapter 8, “Case study: Using archive transparency” on page 109



DB2 archive transparency overview

This chapter outlines the archive transparency support in DB2 for z/OS.

It contains the following sections:

- ▶ Challenges with historical data
- ▶ How DB2 archive transparency works

6.1 Challenges with historical data

Organizations have long had a need to “save” old data, and this is becoming more of an urgent “must do” requirement as a result of recent regulatory and compliance changes. With ever-increasing business and government requirements to save historical data, querying, and managing tables that contain a large amount of data is a common problem that is becoming increasingly challenging. Many organizations are required to retain data for a long time, even after it is considered current. Old data tends to be accessed less and less over time. One way to address the requirements for saving historical data is to move or “archive” inactive data (cold or static data) to a separate archive, which can be on less expensive storage media. Using DB2 archive transparency support, which is delivered in V11, you can separate the active data from the inactive data in order to provide faster access to current data, while still providing easy access to both current data and archived data within a single query without changing the existing application. Another consideration is minimizing the impact on existing applications while providing archiving of the data and transparent access to the data.

Organizations today are either not saving as much data as they should be or they are doing it with their own home-grown applications making use of application logic, partitioning, triggers, stored procedures, or partitioning schemes. However, the methods used today in home-grown solutions are complex, require additional maintenance, and can result in unnecessary costs. In home-grown archive solutions, the user must manage the archive and adjust application code to include or exclude archived data as needed.

6.2 How DB2 archive transparency works

DB2 for z/OS provides a basic row-based archiving solution using SQL, referred to as *archive transparency*. DB2 archive transparency support is implemented with a two table approach, with a single table image to users (Figure 6-1 on page 93). The table that contains the current data is called an *archive-enabled table*, and the table that holds the pre-existing rows is called an *archive table*. Archive tables are useful for managing historical data. An application can design its own way to archive data, or DB2 can automatically insert rows that are deleted from an archive-enabled table into a separate table that is called the archive table, depending on the value of a built-in global variable. Additionally, the LOAD utility with resume behavior can be used to archive data. The retrieval of data from an archive-enabled table or both the archive-enabled table and the associated archive table is controlled by the setting of a built-in global variable and there is no need to change the original application.

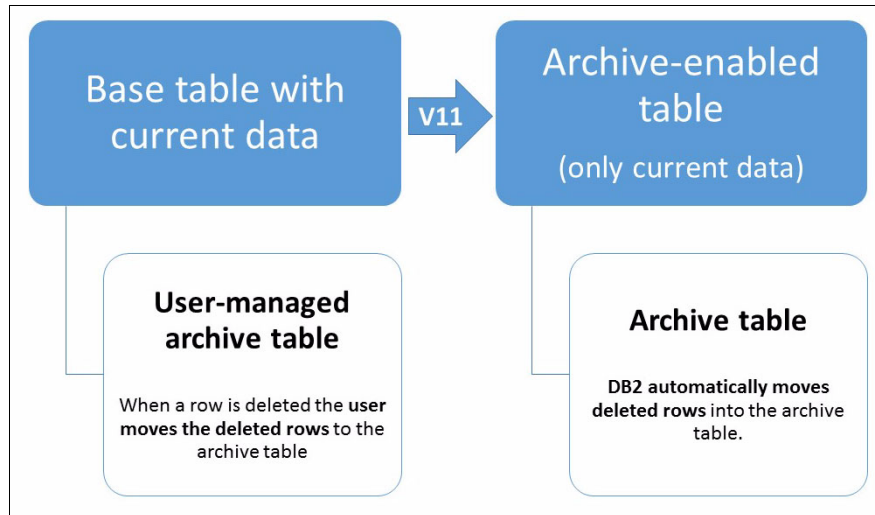


Figure 6-1 Comparison of user-managed archiving to DB2 archive transparency

DB2 archive transparency support addresses the issues of tables that contain large amounts of historical data that are accessed infrequently. The benefits of DB2 archive transparency include:

- ▶ Faster access for retrieval of current data.
- ▶ SQL data access for both current and archived data.
- ▶ Makes data archiving and access “transparent” with minimum application changes.
- ▶ Provides easy access to both current and archived data in a single query.
- ▶ No changes to existing applications.

You can modify queries to include or exclude archive table data without having to change the SQL statement and prepare the application again. Instead, you can control the scope of the query with a global variable. Similarly, you can control whether deleted rows are moved to the archive table with a global variable rather than modifying the original data change statement.

- ▶ Automatic movement of deleted data to the archive.

DB2 can manage historical data for you. You do not have to manually move data to a separate table.

- ▶ Simplified management of tables with a large amount of data that becomes static over time.
- ▶ Improved performance of online transaction processing (OLTP) and data maintenance for large tables by separating the older static data from the more active current data.

When rows that are infrequently accessed are stored in a separate table, you can potentially improve the performance of queries against the archive-enabled table.

- ▶ Reduced cost of storage for older static data.

Archive tables can be stored on a low-cost device to reduce operating costs.

The implementation of archive transparency on DB2 uses two built-in global variables and a bind option as “switches” so that existing applications can continue to be used without modification. Use the appropriate switches to control whether rows deleted from an archive-enabled table are transparently written to the associated archive table, and whether references to an archive-enabled table should transparently consider the rows in the associated archive table.

The archive transparency switches allow transparent access to the archive table in that original queries or data change statements are not modified when an archive-enabled table is defined. The original data change statements and queries can continue to be used, and whether or not the archive table is affected is determined by the switches that are provided.

Migration consideration: A user can move data to the archive table using REORG with the DISCARD option to extract data from the archive-enabled table and then use LOAD RESUME to move the data into the archive table. When the data is removed from an archive-enabled table, the user is responsible for loading the data from the discard file to the archive table. This user-controlled movement of data from an archive-enabled table to an archive table can be useful when migrating to DB2 archive transparency support.

By using the DB2 for z/OS support for archive transparency, you can archive data from the OLTP table to DB2 archive table. Specifying an update operation on an archive-enabled table only affects the current data that is stored in the archive-enabled table.

There is no requirement for separate authorization on the archive table for a user that accesses data in the archive by referencing the archive-enabled table (retrieval or deletions).

Archive transparency is mutually exclusive with temporal tables. A system-period temporal table or application-period temporal table cannot be defined as an archive-enabled table or archive table.

To define an existing table as an archive-enabled table, use an ALTER TABLE statement with the following additional syntax:

```
ENABLE ARCHIVE USE archive-table-table
```

The table that is the target of the ALTER TABLE statement becomes an archive-enabled table and the table specified after the USE keyword becomes the associated archive table. See the SQL Reference for restrictions on what tables can be used as archive-enabled tables or archive tables.

Storing historical, read-only data on the DB2 Analytics Accelerator can save z/OS storage. The data on the Accelerator can be retrieved with SQL, and more data can be made available on the Accelerator for operational analytics in a cost-efficient way.

The DB2 Analytics Accelerator can be used with DB2 archive transparency capability to store the archived data on the Accelerator. See “Chapter 13, “Case study: Combining archive transparency with the DB2 Analytics Accelerator ” on page 171.

The DB2 Analytics Accelerator High Performance Storage Saver can also be used with DB2 archive transparency capability to store the archived data on the Accelerator.



DB2 archive transparency concepts

This section contains more details about the concepts and implementation of DB2 archive transparency. It contains the following sections:

- ▶ Overview
- ▶ What is transparent archiving?
- ▶ Enabling transparent archiving
- ▶ Archive transparency controls
- ▶ Query from an archive-enabled table
- ▶ Disable archiving for an archive-enabled table
- ▶ Using an archive timestamp with archive transparency
- ▶ Summary

7.1 Overview

Over the years, DB2 for z/OS development has continuously improved the product to keep and maintain large data. Large data means that TBs of data are stored in a single table and accessed by hundreds of users concurrently. Traditionally, application designers and DBAs agreed over the years to separate active and archive data. Archive data is historical data that is no longer subject to change and is accessed rarely.

The requirement for archiving data has always existed. The DBA's primary focus is performance and optimization of SQL access paths. The size of the data is related with these factors. Size does matter and less data is better.

Following are the benefits of archiving and separating active data from historical data:

- ▶ Performance
 - Tables are smaller
 - Indexes have fewer levels
 - Number of duplicate index entries are eliminated in NPIs
 - Fewer getpages and index entries
 - Fewer application and database interactions
- ▶ Improved utility operations
 - Less data to process
 - Reduced sorting and index rebuild costs
- ▶ Reduced storage costs
 - Store archived data on lower-cost media
 - Peak performance for archived data usually not necessary

We have seen different usage patterns of historic data. The most common one is a legacy application where access to the active data is changed to access both set of tables (active and archive). Sometimes data maintenance is completed by the mainstream application, and sometimes by operational procedures or DB2 Utilities set by DBAs. Although the set of tables are separate and maintained, the access pattern was the same, which is OLTP type access.

Another usage was when archive data is extracted with another application or DB2 Utility such as UNLOAD or REORG DISCARD and then loaded to another platform such as a central Warehouse, DataMart, or a corporate archive platform. In such a case, the separation of application was necessary like the separation of the data. The main obstacle was that there was no simple way to access the two different data sets from a single application. Basically there were two different applications for two different sets of data in two different platforms. An automated process was moving data from active data to archive data. Archiving has been implemented in the following ways:

- ▶ Application Logic
 - Insert into archive table before deleting data
- ▶ Triggers
 - If the row gets deleted, it is stored in an archive table
- ▶ Partitioning
 - Use range keys and avoid reading older data
- ▶ Log based technology
 - Scan log for deletes and insert deleted data into an archive table

- System-period temporal tables (delivered with DB2 10 for z/OS)
 - Archiving was not the primary design point for temporal data
- Third-party tools

Unfortunately none of these methods provide a good solution. They each require application change, usually limited access to archived data, reduced flexibility for the live application, NPI costs for a partition-based solution, and inflexibility in making schema changes, and more.

Today's world requires mixed workloads such as OLTP, batch, and analytics, which have to be executed on top of corporate data no matter where the data is, and accessing the data has to be transparent to applications. A single view of data has to be accessed by every type of application and the result has to be consistent and integrated.

Having said that, in either way, the separation of operational (or active) data and the archive data in different tables provides increased efficiency for applications. The operational table and indexes for the application process is to a much smaller extent burdened by non-operational data. This also applies to housekeeping processes. In addition, housekeeping processes can run at a much lower frequency for non-operational data, creating another opportunity to reduce data management cost and elapsed time.

Before DB2 11, it was up to the user to organize not only the archiving process, but also ensure proper coding of SQL to include or exclude the archive data. Splitting an existing table into an operational table and an archive table thus meant reviewing and recoding of all SQL statements against the original table.

In many cases, the requirement to optimize access to the operational data is recognized well after the initial creation of the applications. The user community may have developed a wealth of SQL queries against the table. As such, turning a table into a combination of an operational table and an archive table is associated with much effort. In such a development cycle or business case, design teams are challenged to recode the application and separate the data. Those two tasks are not so easy for an already running application in production.

Let's have an example from a real world business case. There is a user-defined archive process, which reads data from the active or operational table (Table A) and moves rows into the archive table (Table A archive). From the DB2 perspective, they are both separate, mutually exclusive tables. It is the responsibility of the archiving process to keep the data in an organized way. The user is in charge of having separate access layers for both tables in case different SQL statements need to access the operational table, the archive table or a combination of both the operational and archive tables.

Further, the user is also responsible for ensuring the layout of the operational table and the archive are kept in sync. Changes must be applied to both tables as well as the archiving process. If the identification of data to be archived is simple (that is, can be expressed with stage 1 predicates), the easiest way to implement the user-defined archive process is usually using a combination of REORG DISCARD of the operational table space and a subsequent LOAD of the discarded rows into the archive table.

An additional advantage of a REORG/LOAD based process is that the base table space and its statistics are in pristine condition afterward (presuming, inline image copy and inline stats are included in the REORG). See Figure 7-1.

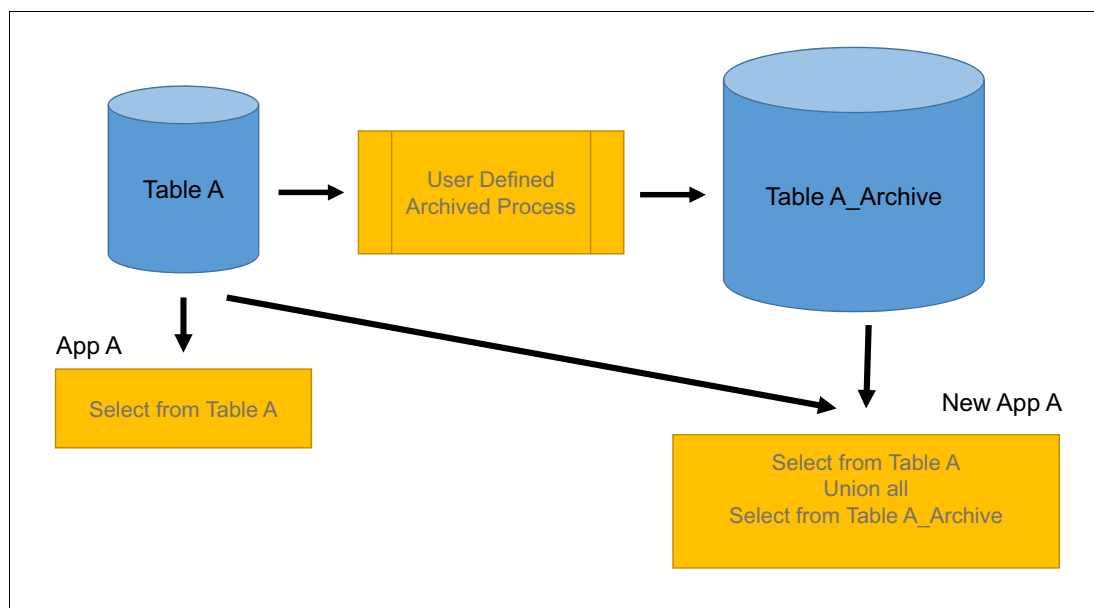


Figure 7-1 Manual or user-controlled archiving example

7.2 What is transparent archiving?

DB2 11 transparent archiving is built on the infrastructure that was introduced in DB2 V10 for bitemporal support (Figure 7-2 on page 99). It is a row-based archiving solution, and it is similar in many ways to system-period temporal tables. However, with transparent archiving there is no need for time period columns. Transparent archiving is included in the base product, and is available beginning with DB2 11 in new function mode. Transparent archiving allows you to define a table, the archive table, and associate it with an existing table, which becomes an archive-enabled table.

Because DB2 is now aware of the association between the operational table and the archive table, DB2 takes care of most of the activities associated with archiving. Access to archive data is independent from applications. It is controlled by a new built-in global variable, and there is no need to make code changes to existing applications to retrieve archive data. A new bind option allows additional control at a package level, and similar bind options are also provided for triggers and SQL routines.

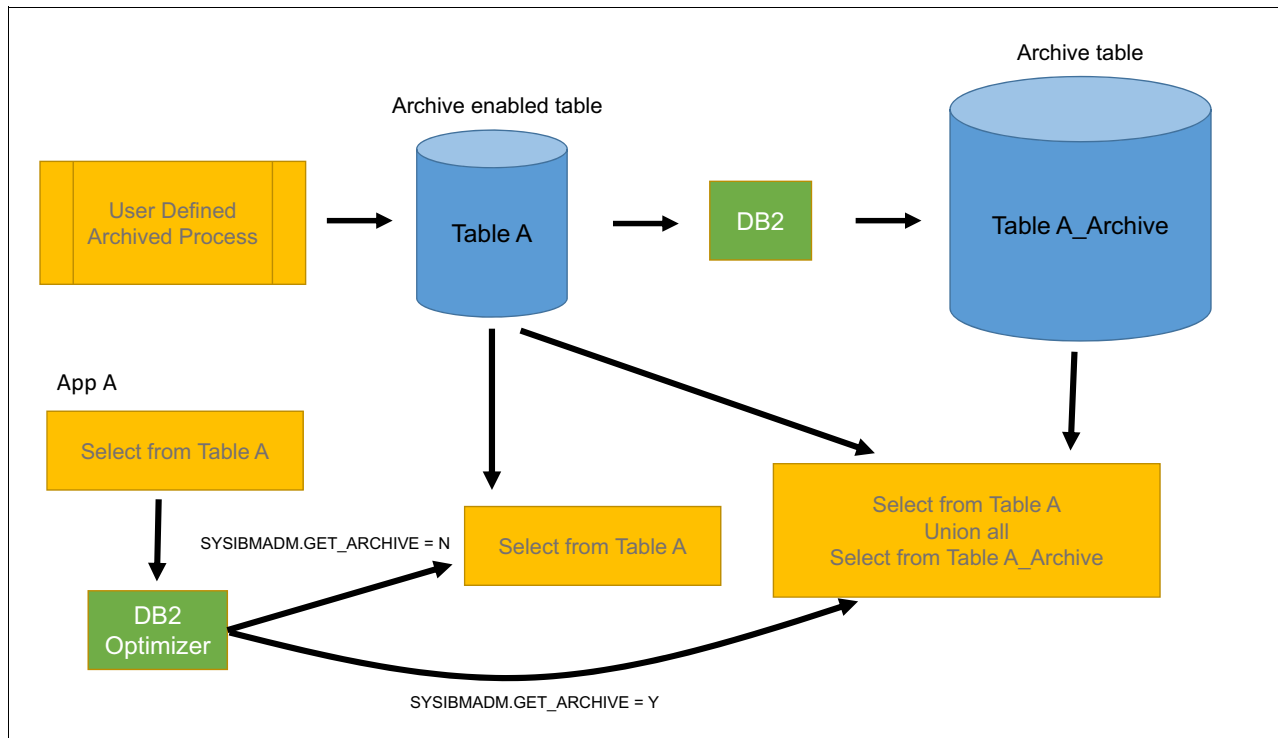


Figure 7-2 Transparent archiving example

The user is still responsible for defining the archiving process. However, the process now only needs to determine what data to archive and simply issue a DELETE statement for those rows of data. The movement of the rows from the archive-enabled table (the operational table) to the archive table is handled automatically by DB2.

When an SQL DELETE statement results in removing a row from the archive-enabled table, DB2 automatically inserts it into the corresponding archive table. There is a new system-defined built-in global variable `SYSIBMADM.MOVE_TO_ARCHIVE` (with default = 'N') that allows the user to control this behavior so that truly unwanted rows can still be deleted from the archive-enabled table without automatically appearing in the archive table.

The process above also works if the data is deleted from the archive-enabled table as the result of a cascading delete from a parent table, where DB2 enforced referential constraint is defined.

The archiving method can also use REORG DISCARD and LOAD to store data in the archive table. The rows that are deleted from the base table by REORG DISCARD are moved to the archive table.

The same SQL statement can be used to retrieve data from the archive-enabled table, or from a combination of the archive-enabled table and the archive table. The latter option ensures that the result set of SQL statements can remain the same before and after the enabling of an archive solution. The impact on the user is thus massively reduced.

When selecting from the archive-enabled table, the user can control whether or not the data contained in the archive table is automatically included. If the built-in global variable `SYSIBMADM.GET_ARCHIVE` (with default = 'N') has been set to 'N', only the table explicitly referenced in the SQL statement will be searched. If this variable is set to 'Y', DB2 will automatically transform the SQL statement to use a union of the archive-enabled table and the archive table.

There is no performance impact caused by union all processing if the application requires only the operational data in the archive-enabled table. This is achieved by optimizing the same static SQL statement twice during bind, resulting in one regular section without the union all expansion, and an extended section with the union all expansion. At execution time, if `SYSIBMADM.GET_ARCHIVE = 'N'`, the regular section is executed, and if `SYSIBMADM.GET_ARCHIVE = 'Y'`, the extended section is executed. In other words, the added flexibility of archive transparency will not impact the SQL performance when accessing operational data in the archive-enabled table only. For dynamic SQL, the SQL statement is optimized based on the value of `SYSIBMADM.GET_ARCHIVE`, which determines whether to perform a union all expansion.

The package must have been bound with `ARCHIVESENSITIVE(YES)` in order for DB2 to take the value of the `SYSIBMADM.GET_ARCHIVE` global variable into consideration. For SQL scalar functions or SQL native stored procedures, the SQL routine option `ARCHIVE SENSITIVE YES` must be used.

With the use of the bind options and global variables, DB2 is able to shield the user from having to recode SQL statements to distinguish between queries that access the archive-enabled table or a combination of the archive-enabled table and the archived table.

DB2 supports adding new columns to the archive-enabled table, and corresponding columns are automatically added to the associated archive table. However, more complicated changes to the archive-enabled table are not supported while the archiving relationship exists. In order to implement such changes, archiving must be disabled first.

7.3 Enabling transparent archiving

To use archive transparency, you need to have two tables properly defined, and issue an `ALTER TABLE` statement to define the relationship between the two tables. One table becomes the archive-enabled table containing the current data, and the other table (the archive table) stores the historical data that has been archived.

You can change an existing table into an archive-enabled table with an associated archive table for historical rows, using an `ALTER TABLE` statement with an `ENABLE ARCHIVE` clause. The table to use as the archive table is specified in the `USE` clause. The archive table must exist before the `ALTER TABLE` statement that references it. Defining a table as an archive-enabled table results in package invalidation of existing applications that reference the table.

Steps for enabling an existing table for archiving:

1. Create the archive table, which is the same structure as the base table.
Identical column structure and attributes
2. Use the `ALTER TABLE` statement with an `ENABLE ARCHIVE` clause:
 - The original table is now an archive-enabled table
 - Only supported by `ALTER TABLE` statement (not `CREATE TABLE`)
 - Identify the archive table after the `USE` keyword

3. Archive the data:
 - Manually: Typically using REORG DISCARD and LOAD
 - System managed: Issue a DELETE statement and DB2 transparently moves the deleted rows to the archive table
4. Applications can access archived data transparently:
 - Built-in global variable controls whether the result set includes archived data from the archive table
 - Packages can be made (in)sensitive to this variable
 - No application changes are required to the SQL statements that reference the original table, which has become an archive-enabled table

7.3.1 Create an archive table

Let's assume you have a base table named T1. The first step to take is to define an archive table that is associated with (what becomes) the archive-enabled table. It is important that the column definitions of the two tables match. If you use a CREATE TABLE statement with the LIKE clause to create this, be aware that the identified archive table must not contain an identity column, row-begin column, row-end column, transaction-start-ID column, or a generated expression column.

But a row change timestamp column is allowed in the archive table, which could track WHEN the event occurs. The row change timestamp column in the archive-enabled table can be defined as either GENERATED BY DEFAULT or GENERATED ALWAYS, but the corresponding row change timestamp column in the archive table must be defined as GENERATED ALWAYS:

```
CREATE TABLE T1_ARC LIKE T1;
```

7.3.2 Enable transparent archiving

The next step is to associate the archive table with the existing table. This is done by executing:

```
ALTER TABLE T1 ENABLE ARCHIVE USE T1_ARC;
```

At this point, we have an archive-enabled table and an associated archive table that DB2 is aware of.

After a table is defined as an archive-enabled table:

- ▶ ALTER TABLE with the ADD COLUMN clause also implicitly adds the new column to the associated archive table.
- ▶ If the SYSIBMADM.GET_ARCHIVE global variable is set to Y, data is retrieved from the archive table when an archive-enabled table is referenced in a table-reference.
- ▶ The access of historical data in the archive table is "transparent" to the application. All subsequent SQL statements including those from an invoked function, stored procedure, or trigger. This allows the application to see both active and archive data without modifying the SQL statements in multiple packages. DB2 internally rewrites the query with the union all operator.

- ▶ If the SYSIBMADM.MOVE_TO_ARCHIVE built-in global variable is set to Y, historical data is stored in the associated archive table when a row is deleted in an archive-enabled table. The storing of a row of historical data in the archive table is “transparent” to the application. When the built-in global variable is set to Y, an update operation returns an error.
- ▶ Any reference to an archive-enabled table for existing values in an INSERT, UPDATE, DELETE, or MERGE statement will not include rows of the associated archive table.

A system-period temporal table or application-period temporal table cannot be referenced in a data manipulation statement when the archive-enabled table is also referenced, when both tables are considered for transparent archive transformations. An archive-enabled table cannot be defined as a materialized query table, with a column mask or row permission.

Once the table is enabled for archiving:

- ▶ All the packages dependent on the archive-enabled table are invalidated
- ▶ ALTER TABLE ADD COLUMN is allowed and the new column is automatically added to the archive table
- ▶ No other column alterations are allowed
- ▶ Other table changes such as rotating partitions or defining the table as a clone table are not allowed
- ▶ LOAD REPLACE is not allowed on an archive-enabled table although you can use it on the archive table

7.4 Archive transparency controls

The implementation of archive transparency on DB2 uses a new bind option and two built-in global variables in order to provide the capability “transparently” to existing applications. That is, you do not need to modify existing delete statements to get deleted rows stored in the archive table, and you do not need to modify existing retrieval statements to have rows in the archive table considered. The new bind option and built-in global variables serve as “switches” to control whether archive transparency is active or not.

The bind option determines whether SQL statements are impacted by the value of the SYSIBMADM.GET_ARCHIVE built-in global variable for transparent access to the historical data in the archive table.

The built-in global variables control whether rows deleted from an archive-enabled table are written to the associated archive table, and whether references to an archive-enabled table should consider the rows in the associated archive table (Figure 7-3 on page 103). The variables allow transparent access to the archive table in that original queries or data change statements do not need to be modified when an archive-enabled table is defined. The original data change statements and queries can continue to be used and whether or not the archive table is affected is determined by a built-in global variable.

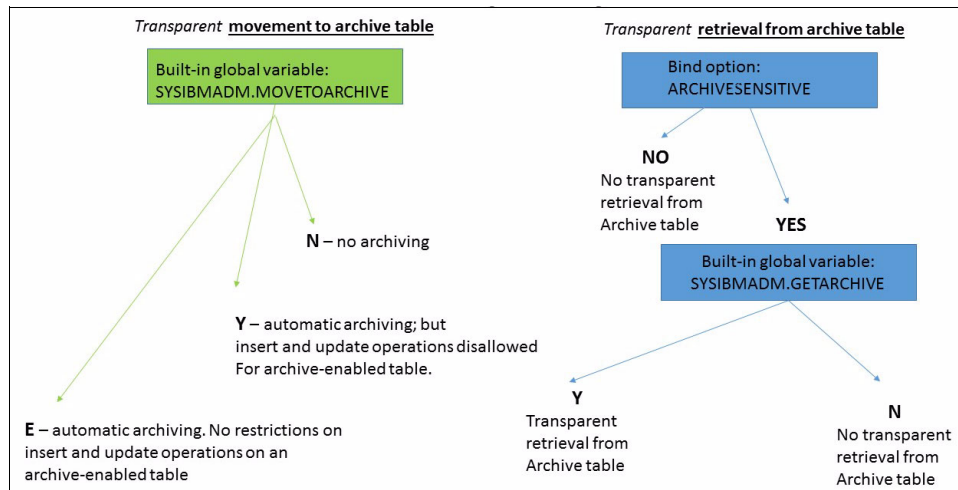


Figure 7-3 Archive transparency controls

7.4.1 Archive transparency bind options

This section addresses two archive transparency bind options.

ARCHIVESENSITIVE bind option

The ARCHIVESENSITIVE bind option determines whether references to an archive-enabled table in both static and dynamic SQL statements are affected by the value of the SYSIBMADM.GET_ARCHIVE built-in global variable:

- ▶ BIND/REBIND PACKAGE
- ▶ REBIND TRIGGER PACKAGE
- ▶ CREATE TRIGGER

The bind option can be specified as:

- ▶ ARCHIVESENSITIVE (YES)

References to archive-enabled tables are affected by the value of the SYSIBMADM.GET_ARCHIVE global variable.

- ▶ ARCHIVESENSITIVE (NO)

References to archive-enabled tables are not affected by the value of the SYSIBMADM.GET_ARCHIVE global variable.

SQL statements in the body of a trigger will fail if an archive-enabled table is referenced in the WHEN clause and the ARCHIVESENSITIVE (YES) bind option is specified.

ARCHIVESENSITIVE SQL routine option

The ARCHIVESENSITIVE routine option determines whether references to archive-enabled tables in user-defined SQL functions or native SQL procedures are affected by the value of the SYSIBMADM.GET_ARCHIVE built-in global variable:

- ▶ CREATE/ALTER FUNCTION (SQL scalar)
- ▶ CREATE/ALTER PROCEDURE (SQL native)

The bind option can be specified as:

- **ARCHIVE SENSITIVE YES**

References to archive-enabled tables are affected by the value of the SYSIBMADM.GET_ARCHIVE global variable.

- **ARCHIVE SENSITIVE NO**

References to archive-enabled tables are not affected by the value of the SYSIBMADM.GET_ARCHIVE global variable.

If the bind option is set to NO, the package ignores GET_ARCHIVE setting and only active data in the archive-enabled table is retrieved.

7.4.2 Archive transparency built-in global variables

This section addresses two built-in global variables.

SYSIBMADM.MOVE_TO_ARCHIVE

The SYSIBMADM.MOVE_TO_ARCHIVE built-in global variable controls whether rows that are deleted from an archive-enabled table are automatically moved to the associated archive table. The default value is N, which disables automatic archiving of deleted rows. Use the value Y to enable automatic archiving of deleted rows and disallow insert or update operations on the archive-enabled table. Use the value E to enable automatic archiving of deleted rows and allow insert or update operations on the archive-enabled table.

SYSIBMADM.GET_ARCHIVE

The SYSIBMADM.GET_ARCHIVE built-in global variable controls whether transparent retrieval of historical rows of data from an archive table occurs when the corresponding archive-enabled table is referenced. Using this built-in global variable, an existing application that references the archive-enabled table does not need to be changed to return archived data. The default value is N, which disables transparent retrieval access to rows in the archive table. When the built-in global variable is set to Y, and the ARCHIVESENSITIVE bind option is set to Y, references to an archive-enabled table will automatically consider rows in the associated archive table.

Performance note: To provide optimal performance of data manipulation SQL statements, DB2 prepares two paths for an SQL statement that accesses an archive-enabled table. DB2 decides at run time which path to use depending on the setting of the SYSIBMADM.GET_ARCHIVE built-in global variable.

7.4.3 Data change operations when the target is an archive-enabled table

When transparent archiving is active, INSERT, UPDATE, and MERGE SQL statements may be disabled depending on the value of the SYSIBMADM.MOVE_TO_ARCHIVE built-in global variable as follows:

- If SYSIBMADM.MOVE_TO_ARCHIVE = 'Y', the INSERT, UPDATE, and MERGE statements fail with new SQLCODE -20555 with reason code 2, based on the assumption that in archive mode, you can only delete data.

- ▶ If SYSIBMADM.MOVE_TO_ARCHIVE = 'N', no archive mode failure occurs. All data manipulation SQL statements are allowed against archive-enabled table including a DELETE statement, and archiving does not occur.
- ▶ If SYSIBMADM.MOVE_TO_ARCHIVE = 'E', the behavior is similar to Y, but adds flexibility. The E setting does not restrict the use of the data change statements. Users that favor the restriction on data change operations can set the built-in global variable to Y, and users that do not want the restriction can set the built-in global variable to E.

7.4.4 Rules for deleting from an archive-enabled table

A single DELETE statement triggers transparent archive behavior when MOVE_TO_ARCHIVE = Y or E. No additional privileges are required on an archive table. Only the privileges to delete from the archive-enabled table are needed, and there is no requirement for insert privileges on the archive table.

Given a DELETE from an archive-enabled table, regardless of whether it is dynamic or static SQL and whether the package is bound with the ARCHIVESENSITIVE option YES or NO:

- ▶ If SYSIBMADM.MOVE_TO_ARCHIVE built-in global variable is set to Y or E, for each row deleted from the archive-enabled table, DB2 inserts it into the corresponding archive table.
- ▶ If SYSIBMADM.MOVE_TO_ARCHIVE built-in global variable is set to N (the default), DB2 does no data propagation to the archive table. It basically works similar to a regular delete statement.

Transparent archiving is basically an SQL performance improvement to help archiving data with a single DELETE statement. For the application, there is no need to change the syntax of the data change SQL statements (that is, the archiving is transparent to the application).

7.5 Query from an archive-enabled table

There is no need to change SQL statements in an existing application to switch between retrieving current data only or to access both the current and archive data. The setting of the SYSIBMADM.GET_ARCHIVE built-in global variable offers transparent access to archive data for queries.

7.5.1 Querying the current data only

Set the SYSIBMADM.GET_ARCHIVE built-in global variable to N to have a query of an archive-enabled table ignore the rows in the associated archive table. For example:

```
SET SYSIBMADM.GET_ARCHIVE = 'N'
```

There is no need to change the original query to tell DB2 to ignore the archive table. DB2 transparently ignores the archive table based on the setting of the SYSIBMADM.GET_ARCHIVE built-in global variable.

7.5.2 Transparently querying the current and archive data

Set the SYSIBMADM.GET_ARCHIVE built-in global variable to Y to have a query of an archive-enabled table consider the rows in the associated archive table. For example:

```
SET SYSIBMADM.GET_ARCHIVE = 'Y'
```

There is no need to change the original query to reference the archive table. DB2 transparently accesses the archive table based on the setting of the SYSIBMADM.GET_ARCHIVE built-in global variable.

7.6 Disable archiving for an archive-enabled table

To disable archiving and remove the relationship between the archive-enabled table and the associated archive table, specify the DISABLE ARCHIVE clause of the ALTER TABLE statement for the archive-enabled table. When the DISABLE ARCHIVE is specified, the statement must identify an archive-enabled table. After the ALTER statement is successfully processed, both tables continue to exist and are considered ordinary tables. The column definitions for both tables, and the data in both tables are unaffected by the ALTER statement.

Subsequent queries that reference the table (that was previously defined as the archive-enabled table) will not consider rows in the archive table regardless of the setting of the SYSIBMADM.GET_ARCHIVE built-in global variable.

Rows that are subsequently deleted (from the table that was the archive-enabled table) will not be moved to the archive table regardless of the setting of the SYSIBMADM.MOVE_TO_ARCHIVE built-in global variable.

Disabling archiving for an archive-enabled table results in invalidation of packages that reference the table that was the archive-enabled table:

```
ALTER TABLE TABLE_A DISABLE ARCHIVE;
```

7.7 Using an archive timestamp with archive transparency

An “archive timestamp” can be recorded in an archive table. A row change timestamp column is a good choice for this. As rows are moved to the archive table, DB2 generates values for the row change timestamp column in the archive table, which reflect when the rows were inserted into the archive table, which is the timestamp of archiving.

An archive-enabled table and the associated archive table do not need to be partitioned in the same way. It can be useful to partition the archive-enabled table and the archive table differently. Partitioning the archive table on the row change timestamp column allows for further archiving (“offloading”) to the DB2 Analytics Accelerator on a partition level from the archive table (Figure 7-4 on page 107).

Given the rules for transparent archiving that the number of columns and column attributes must be the same for the archive-enabled table and corresponding archive table, you need to add a row change timestamp column to both tables. Define the column in both tables as a row change timestamp using the following syntax for the column definition:

```
GENERATED ALWAYS FOR EACH ROW ON UPDATE  
AS ROW CHANGE TIMESTAMP
```

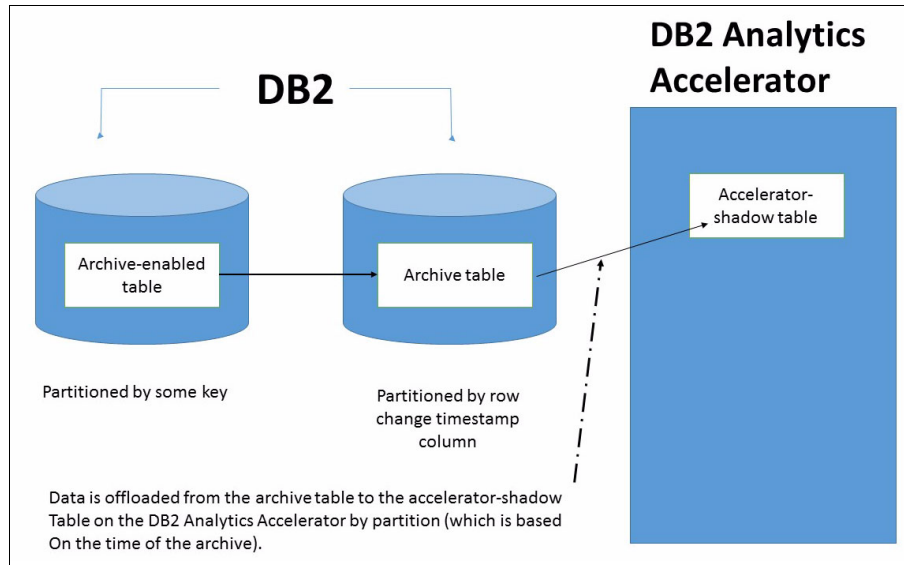


Figure 7-4 Offloading archived data to the DB2 Analytics Accelerator by partition

7.8 Summary

DB2 has delivered many improvements in the area of application enablement and online schema development. Situations may still remain where a DROP is required.

Important: If an archive-enabled table is dropped, the archive table is automatically dropped as well. If this is not your intent, ensure to disable the archiving relationship before dropping an archive-enabled table.

The implementation requires an application to both bind the package with ARCHIVESENSITIVE (YES) and set the built-in global variable SYSIBMADM.GET_ARCHIVE to 'Y'. One could argue that one could be enough. Let's assume the bind parameter would not exist. The effect would be that if a module somewhere in the calling path would set the global variable SYSIBMADM.GET_ARCHIVE to 'Y', then all of a sudden all packages in the thread would be affected. It is after all a global variable. So, without being touched at all, the behavior of the module would have changed. With the ARCHIVESENSITIVE bind parameter, we get additional control to avoid that situation.

Similarly, one could question why setting another built-in global variable (SYSIBMADM.MOVE_TO_ARCHIVE) is necessary in applications that delete from an archive-enabled table. To be fair, this opens the opportunity that things go wrong and the archive is no longer complete. Then again, if moving data would not be under control of a built-in global variable, deleting truly unwanted data could become difficult. In order to remove the unwanted data from the archive enabled table (that should not end up in the archive), one would have to disable the archiving and perform the delete. But what if you have more applications that delete and expect the data to be moved to the archive? It must now be ensured that none of such applications run while the relation between the archive-enabled table and the archive is disabled.

One thing that certainly must be considered is the fact that if a row is now moved to an archive table, the primary key of that row becomes available again in the archive-enabled table. It could occur that the same primary key (or any other unique key for that matter) is reinserted into the archive-enabled table. This might or might not lead to problems.

Tip: Do not define unique indexes on an archive table.

A bind with ARCHIVESENSITIVE (YES) uses 10 - 80% more CPU. It does not result in more execution time and the cost is low compared to the cost of changing the application.

Consider the following tips:

- ▶ Avoid union all costs when not accessing the archive data:
 - DB2 binds the statement twice
 - First (original) section contains just the current data
 - Extended section includes archive data
 - Visible in various PLAN_TABLEs - EXPANSION_REASON = 'A'
- ▶ Beware of ORDER BY when accessing archive data

Access of archive data requires the use of an implicit union all operator, which likely results in a sort
- ▶ New catalog columns and values support archive transparency:
 - ARCHIVESENSITIVE in SYSPACKAGE, SYSPACKCOPY
 - ARCHIVING_SCHEMA and ARCHIVING_TABLE in SYSTABLES
 - TYPE = 'R' in SYSTABLES for the archive table only
- ▶ Plan management features check for archiving changes

Bind or rebind fails if you change ARCHIVESENSITIVE and use APCOMPARE or APREUSE with PLANMGMT BASIC or EXTENDED
- ▶ Utilities
 - Cannot run CHECK DATA SHRLEVEL REFERENCE with DELETE YES or any of LOBERROR, AUXERROR, or XMLERROR INVALIDATE
 - LISTDEF enhanced to include ARCHIVE option. Specifies that only archive objects are to be included in the resulting list clause.



Case study: Using archive transparency

In this case study, we practice starting from a stand-alone DB2 table that includes both active and history data and move forward with transparent archiving capability.

This chapter contains the following sections:

- ▶ Sample application with step-by-step guidance
- ▶ Define a table to enable transparent archiving
- ▶ Enabling archive transparency
- ▶ Mechanics of moving data from an archive-enabled table to an archive table
- ▶ Querying data from an archive-enabled table

8.1 Sample application with step-by-step guidance

The application has a table called *Table_A*. It is partitioned by date and contains both active and history data in the same table. In the initial situation, data in historical partitions is technically read/write, but read-only from a logical perspective only.

Table_A (Figure 8-1) is keeping total call durations for a phone number per day for each customer. The structure of the table looks like this:

- ▶ PhoneNo: Phone number that made the call
- ▶ CustNo: Customer number of the phone
- ▶ CallDur: Total call duration
- ▶ Date: Date

The table is partitioned by date.

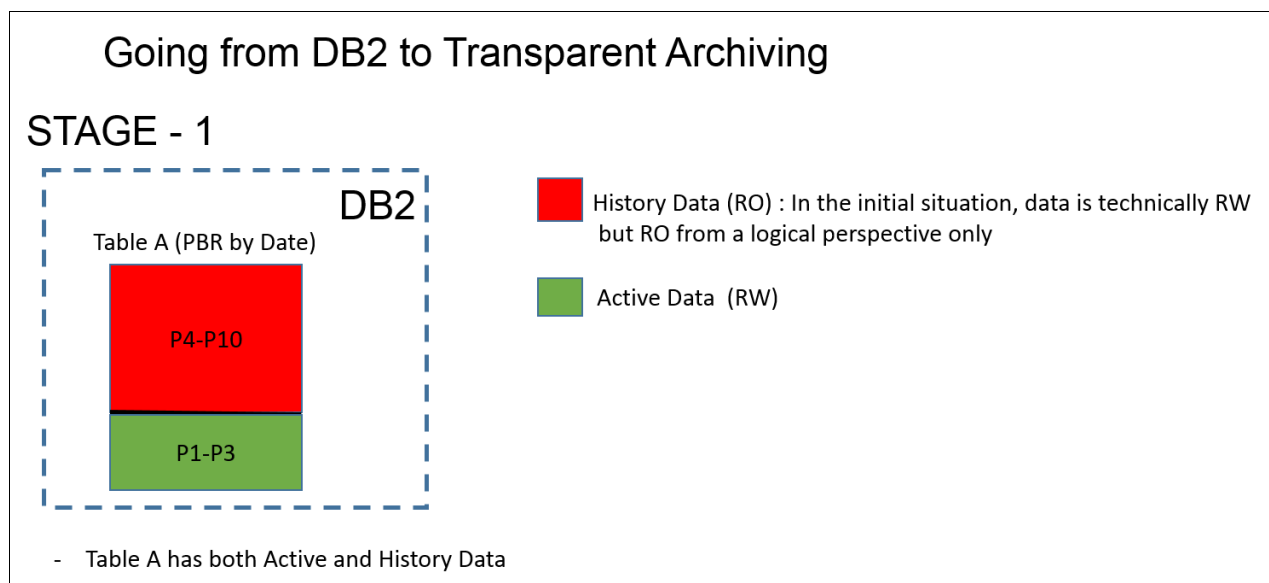


Figure 8-1 Initial view of Table_A

Example 8-1 shows the complete Data Definition Language (DDL) for this sample table.

Example 8-1 DDL for the sample application: Initial table

```
SET CURRENT SQLID='MKRES7';
CREATE DATABASE PHONE
  BUFFERPOOL BPO
  INDEXBP    BPO
  CCSID      EBCDIC
  STOGROUP   SYSDEFLT;
COMMIT;
CREATE TABLESPACE PHONE
  IN PHONE
  USING STOGROUP SYSDEFLT
  PRIQTY -1 SECQTY -1
  FREEPAGE 0 PCTFREE 5
  GBPCACHE CHANGED
  TRACKMOD YES
  LOGGED
```

```

DSSIZE 4 G
NUMPARTS 3
SEGSIZE 32
BUFFERPOOL BP0
LOCKSIZE ANY
LOCKMAX SYSTEM
CLOSE YES
COMPRESS NO
CCSID      EBCDIC
DEFINE YES
MAXROWS 255;
COMMIT;
CREATE TABLE MKRES7.TABLE_A
  (PHONENO          INTEGER NOT NULL WITH DEFAULT,
   CUSTONO          INTEGER NOT NULL WITH DEFAULT,
   CALLDUR          SMALLINT NOT NULL WITH DEFAULT,
   DATE             DATE NOT NULL WITH DEFAULT)
  IN PHONE.PHONE
  PARTITION BY (DATE ASC)
  (PARTITION 1 ENDING AT ('2015-04-30'),
   PARTITION 2 ENDING AT ('2015-05-31'),
   PARTITION 3 ENDING AT ('2015-06-30'))
  AUDIT NONE
  DATA CAPTURE NONE
  CCSID      EBCDIC
  NOT VOLATILE
  APPEND NO ;
COMMIT;
CREATE UNIQUE INDEX MKRES7.XTABLE_A
  ON MKRES7.TABLE_A
  (PHONENO          ASC,
   CUSTONO          ASC,
   DATE             ASC)
  USING STOGROUP SYSDEFLT
  PRIQTY -1 SECQTY -1
  ERASE NO
  FREEPAGE 0 PCTFREE 10
  GBPCACHE CHANGED
  NOT CLUSTER
  COMPRESS NO
  INCLUDE NULL KEYS
  BUFFERPOOL BP0
  CLOSE YES
  COPY NO
  DEFER NO
  DEFINE YES
  PIECESIZE 4 G;
COMMIT;

```

For simulation purposes, we created three partitions. Partitions 1 and 2 include history data from April and May 2015. Partition 3 includes active data of June 2015. Partitions 1 and 2 will include read-only data. Partition 3 allows read/write activity.

Here is the sample data in Example 8-2.

Example 8-2 Sample data for application

```
-- ** HISTORIC DATA ****
INSERT INTO "MKRES7"."TABLE_A" VALUES(3171427, 1, 10, '2015-04-13') ;
INSERT INTO "MKRES7"."TABLE_A" VALUES(3171427, 1, 15, '2015-04-15') ;
INSERT INTO "MKRES7"."TABLE_A" VALUES(3171427, 1, 15, '2015-05-15') ;
INSERT INTO "MKRES7"."TABLE_A" VALUES(3171201, 2, 12, '2015-05-15') ;
INSERT INTO "MKRES7"."TABLE_A" VALUES(3171201, 2, 11, '2015-04-15') ;
INSERT INTO "MKRES7"."TABLE_A" VALUES(3171201, 2, 11, '2015-04-16') ;
-- ** ACTIVE DATA ****
INSERT INTO "MKRES7"."TABLE_A" VALUES(3171201, 2, 12, '2015-06-14') ;
INSERT INTO "MKRES7"."TABLE_A" VALUES(3171427, 1, 25, '2015-06-15') ;
SELECT * FROM TABLE_A;
```

PHONENO	CUSTONO	CALLDUR	DATE
3171427	1	10	2015-04-13
3171427	1	15	2015-04-15
3171201	2	11	2015-04-15
3171201	2	11	2015-04-16
3171427	1	15	2015-05-15
3171201	2	12	2015-05-15
3171201	2	12	2015-06-14
3171427	1	25	2015-06-15

8.2 Define a table to enable transparent archiving

For enabling transparent archiving, we need a second table for the archived data (Figure 8-2 on page 113). To review the terminology that we're using, archived data is stored in the "archive table" and active data is stored in the "archive-enabled table". So, we're going to create a table exactly like Table_A for the archive, read-only data. The table space and table structures will be the same for the two tables. Then, we'll move the historic data that is in partition 1 and partition 2 of Table_A to the same partitions of the archive table Table_A_ARC, and DELETE data from Table_A.

We'll use an UNLOAD/LOAD method to move data between the partitions and we'll use the LOAD REPLACE DD DUMMY method to clean up Partition 1 and 3 of Table_A.

Going from DB2 to Transparent Archiving

STAGE - 2

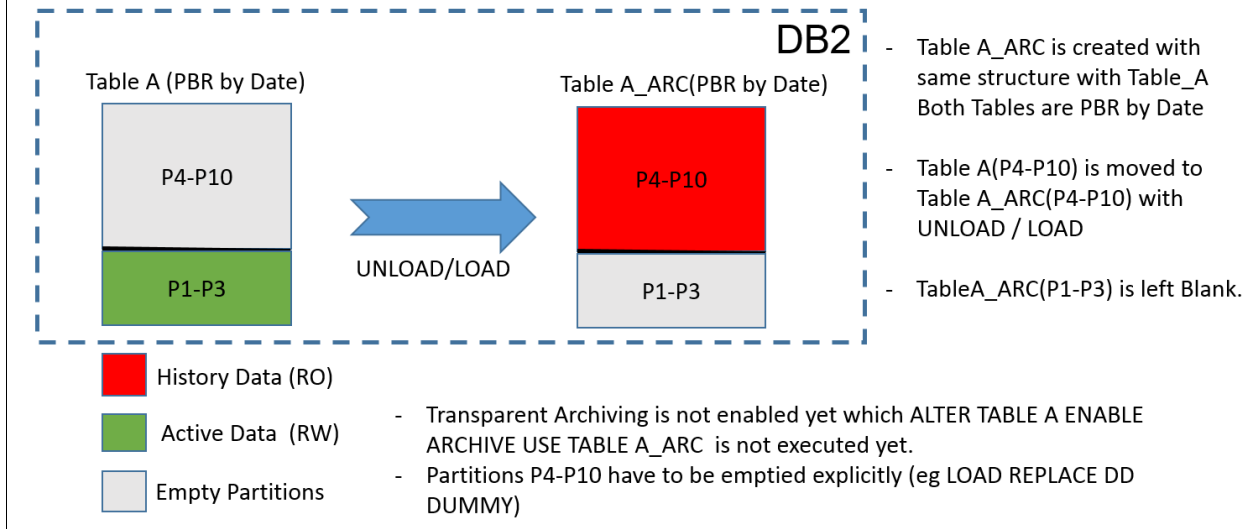


Figure 8-2 Creating the archive table and loading it with historic data

8.2.1 Creating the archive table

In Example 8-3, the archive table space named **PHONEH**, and the archive table named **TABLE_A_ARC**, are created.

Example 8-3 DDL for the sample application: Archive table and table space

```
CREATE TABLESPACE PHONEH
  IN PHONE
  USING STOGROUP SYSDEFLT
  PRIQTY -1 SECQTY -1
  FREEPAGE 0 PCTFREE 5
  GBPCACHE CHANGED
  TRACKMOD YES
  LOGGED
  DSSIZE 4 G
  Numparts 3
  SEGSIZE 32
  BUFFERPOOL BP0
  LOCKSIZE ANY
  LOCKMAX SYSTEM
  CLOSE YES
  COMPRESS NO
  CCSID      EBCDIC
  DEFINE YES
  MAXROWS 255;
COMMIT;
CREATE TABLE MKRES7.TABLE_A_ARC
  (PHONENO      INTEGER NOT NULL WITH DEFAULT,
   CUSTONO      INTEGER NOT NULL WITH DEFAULT,
   CALLDUR      SMALLINT NOT NULL WITH DEFAULT,
```

```

DATE                DATE NOT NULL WITH DEFAULT)
IN PHONE.PHONEH
PARTITION BY (DATE ASC)
(PARTITION 1 ENDING AT ('2015-04-30'),
 PARTITION 2 ENDING AT ('2015-05-31'),
 PARTITION 3 ENDING AT ('2015-06-30'))
AUDIT NONE
DATA CAPTURE NONE
CCSID      EBCDIC
NOT VOLATILE
APPEND NO  ;
COMMIT;

```

8.2.2 Moving data from the active table to the archive table

Once the archive table is created, it's ready to be loaded with “cold” data from the active table (the archive-enabled table). Cold data is UNLOADED and LOADED with DB2 utilities. As seen from Example 8-4, partition 1 and partition 2 are UNLOADED from Table_A.

Example 8-4 UNLOAD process from Table_A

UNLOAD for first partition

```

//DSNUPROC.SYSUT1 DD DSN=MKRES7.SYSUT1,
//                DISP=(MOD,DELETE,CATLG),
//                SPACE=(TRK,(8,5),RLSE),
//                UNIT=SYSDA
//DSNUPROC.SORTOUT DD DSN=MKRES7.SORTOUT,
//                DISP=(MOD,DELETE,CATLG),
//                SPACE=(TRK,(8,5),RLSE),
//                UNIT=SYSDA
//SYSPUNCH DD DSN=MKRES7.D1DG.CNTL.PHONE.PHONE.P1,
//                DISP=(,CATLG,DELETE),
//                DCB=(LRECL=80,BLKSIZE=0,RECFM=FB,DSORG=PS),
//                SPACE=(TRK,(5,5),RLSE),
//                UNIT=SYSDA
//SYSREC DD DSN=MKRES7.D1DG.UNLD.PHONE.PHONE.P1,
//                DISP=(,CATLG,DELETE),
//                SPACE=(TRK,(45,5),RLSE),
//                UNIT=SYSDA
//DSNUPROC.SYSIN DD *
UNLOAD TABLESPACE PHONE.PHONE PART 1
FROM TABLE
"MKRES7"."TABLE_A"
/*

```

The Output UNLOAD for Partition 1

```

OUTPUT START FOR UTILITY, UTILID = MKRES7.UNLOAD1
PROCESSING SYSIN AS EBCDIC
UNLOAD TABLESPACE PHONE.PHONE PART 1
S - FROM TABLE "MKRES7"."TABLE_A"
UNLOAD PHASE STATISTICS - NUMBER OF RECORDS UNLOADED=4 FOR TABLE MKRES7.TABLE_A
UNLOAD PHASE STATISTICS - NUMBER OF RECORDS UNLOADED=4 FOR TABLESPACE

UNLOAD PHASE COMPLETE, ELAPSED TIME=00:00:00
UTILITY EXECUTION COMPLETE, HIGHEST RETURN CODE=0

```

```

UNLOAD for second partition
//DSNUPROC.SYSUT1 DD DSN=MKRES7.SYSUT1,
//          DISP=(MOD,DELETE,CATLG),
//          SPACE=(TRK,(8,5),RLSE),
//          UNIT=SYSDA
//DSNUPROC.SORTOUT DD DSN=MKRES7.SORTOUT,
//          DISP=(MOD,DELETE,CATLG),
//          SPACE=(TRK,(8,5),RLSE),
//          UNIT=SYSDA
//SYSPUNCH DD DSN=MKRES7.D1DG.CNTL.PHONE.PHONE.P2,
//          DISP=(,CATLG,DELETE),
//          DCB=(LRECL=80,BLKSIZE=0,RECFM=FB,DSORG=PS),
//          SPACE=(TRK,(5,5),RLSE),
//          UNIT=SYSDA
//SYSREC DD DSN=MKRES7.D1DG.UNLD.PHONE.PHONE.P2,
//          DISP=(,CATLG,DELETE),
//          SPACE=(TRK,(45,5),RLSE),
//          UNIT=SYSDA
//DSNUPROC.SYSIN DD *
UNLOAD TABLESPACE PHONE.PHONE PART 2
FROM TABLE
"MKRES7"."TABLE_A"

```

The Output UNLOAD for Partition 2

```

OUTPUT START FOR UTILITY, UTILID = MKRES7.UNLOAD1
PROCESSING SYSIN AS EBCDIC
UNLOAD TABLESPACE PHONE.PHONE PART 2
S - FROM TABLE "MKRES7"."TABLE_A"
UNLOAD PHASE STATISTICS - NUMBER OF RECORDS UNLOADED=2 FOR TABLE MKRES7.TABLE_A
UNLOAD PHASE STATISTICS - NUMBER OF RECORDS UNLOADED=2 FOR TABLESPACE
UNLOAD PHASE COMPLETE,
ELAPSED TIME=00:00:00
UTILITY EXECUTION COMPLETE, HIGHEST RETURN CODE=0

```

8.2.3 Loading data into the archive table

UNLOADed and extracted data is LOAded to the same partitions of the archive table, which is TABLE_A_ARC. In Example 8-5 on page 116, you'll find LOAD JCLs and related outputs for LOAD process to archive table TABLE_A_ARC.

Note: We edited following two data sets and performed the following changes:

- ▶ MKRES7.D1DG.CNTL.PHONE.PHONE.P1
- ▶ MKRES7.D1DG.CNTL.PHONE.PHONE.P2
- ▶ LOG NO to LOG YES
- ▶ RESUME YES to REPLACE
- ▶ TABLE_A to TABLE_A_ARC

Example 8-5 LOAD process to TABLE_A_ARC

LOAD first partition

```
//LOAD EXEC DSNUPROC,SYSTEM=D1DG,
//          LIB='DB1DT.SDSNLOAD',
//          UID=' '
//DSNUPROC.SYSREC DD DISP=SHR,
//          DSN=MKRES7.D1DG.UNLD.PHONE.PHONE.P1
//DSNUPROC.SYSUT1 DD DSN=MKRES7.TEST.SYSUT1,
//          DISP=(MOD,DELETE,CATLG),
//          SPACE=(TRK,(10,5),RLSE),
//          UNIT=SYSDA
//DSNUPROC.SORTOUT DD DSN=MKRES7.TEST.SORTOU,
//          DISP=(MOD,DELETE,CATLG),
//          SPACE=(TRK,(10,5),RLSE),
//          UNIT=SYSDA
//DSNUPROC.SORTWK01 DD DSN=MKRES7.TEST.SORTWK01,
//          DISP=(MOD,DELETE,CATLG),
//          SPACE=(TRK,(10,5),RLSE),
//          UNIT=SYSDA
//DSNUPROC.SYSMAP DD DSN=MKRES7.TEST.SYSMAP,
//          DISP=(MOD,DELETE,CATLG),
//          SPACE=(TRK,(10,5),RLSE),
//          UNIT=SYSDA
//DSNUPROC.SYSERR DD DSN=MKRES7.TEST.SYSERR,
//          DISP=(MOD,CATLG,CATLG),
//          SPACE=(TRK,(10,5),RLSE),
//          UNIT=SYSDA
//DSNUPROC.SYSDISC DD DSN=MKRES7.TEST.SYSDISC,
//          DISP=(MOD,CATLG,CATLG),
//          SPACE=(TRK,(10,5),RLSE),
//          UNIT=SYSDA
//DSNUPROC.SYSIN DD *
//  DD DSN=MKRES7.D1DG.CNTL.PHONE.PHONE.P1,
//          DISP=SHR
//  DD DATA,DLM='$$'
$$
```

The Output LOAD for Partition 1

```
OUTPUT START FOR UTILITY, UTILID = MKRES7.LOAD1
PROCESSING SYSIN AS EBCDIC
  LOAD DATA INDDN SYSREC LOG YES EBCDIC CCSID(37, 0, 0)
  - INTO TABLE "MKRES7". "TABLE_A_ARC" PART 1 REPLACE WHEN(1:2)=X'0003'

  - ("PHONENO" POSITION(3:6) INTEGER,
  -   "CUSTONO" POSITION(7:10) INTEGER,
  -   "CALLDUR" POSITION(11:12) SMALLINT,
  -   "DATE" POSITION(13:22) DATE EXTERNAL)
T - EXISTING RECORDS DELETED FROM TABLESPACE PARTITION 1
  - (RE)LOAD PHASE STATISTICS - NUMBER OF RECORDS=4 FOR TABLE MKRES7.TABLE_A_ARC

  - (RE)LOAD PHASE STATISTICS - TOTAL NUMBER OF RECORDS LOADED=4 FOR TABLESPACE

(RE)LOAD PHASE STATISTICS - NUMBER OF INPUT RECORDS PROCESSED=4
(RE)LOAD PHASE COMPLETE, ELAPSED TIME=00:00:00
BUILD PHASE STATISTICS - NUMBER OF INDEXES=0
```

BUILD PHASE COMPLETE, ELAPSED TIME=00:00:00
 UTILITY EXECUTION COMPLETE, HIGHEST RETURN CODE=0

LOAD second partition

```
//LOAD EXEC DSNUPROC,SYSTEM=D1DG,
//          LIB='DB1DT.SDSNLOAD',
//          UID=' '
//DSNUPROC.SYSREC DD DISP=SHR,
//          DSN=MKRES7.D1DG.UNLD.PHONE.PHONE.P2
//DSNUPROC.SYSUT1 DD DSN=MKRES7.TEST.SYSUT1,
//          DISP=(MOD,DELETE,CATLG),
//          SPACE=(TRK,(10,5),RLSE),
//          UNIT=SYSDA
//DSNUPROC.SORTOUT DD DSN=MKRES7.TEST.SORTOU,
//          DISP=(MOD,DELETE,CATLG),
//          SPACE=(TRK,(10,5),RLSE),
//          UNIT=SYSDA
//DSNUPROC.SORTWK01 DD DSN=MKRES7.TEST.SORTWK01,
//          DISP=(MOD,DELETE,CATLG),
//          SPACE=(TRK,(10,5),RLSE),
//          UNIT=SYSDA
//DSNUPROC.SYSMAP DD DSN=MKRES7.TEST.SYSMAP,
//          DISP=(MOD,DELETE,CATLG),
//          SPACE=(TRK,(10,5),RLSE),
//          UNIT=SYSDA
//DSNUPROC.SYSERR DD DSN=MKRES7.TEST.SYSERR,
//          DISP=(MOD,CATLG,CATLG),
//          SPACE=(TRK,(10,5),RLSE),
//          UNIT=SYSDA
//DSNUPROC.SYSDISC DD DSN=MKRES7.TEST.SYSDISC,
//          DISP=(MOD,CATLG,CATLG),
//          SPACE=(TRK,(10,5),RLSE),
//          UNIT=SYSDA
//DSNUPROC.SYSIN DD *
//  DD DSN=MKRES7.D1DG.CNTL.PHONE.PHONE.P2,
//          DISP=SHR
//  DD DATA,DLM='$$'
$$
```

The Output LOAD for Partition 2

```
OUTPUT START FOR UTILITY, UTILID = MKRES7.LOAD1
PROCESSING SYSIN AS EBCDIC
  LOAD DATA INDDN SYSREC LOG YES EBCDIC CCSID(37, 0, 0)
  - INTO TABLE "MKRES7". "TABLE_A_ARC" PART 2 REPLACE WHEN(1:2)=X'0003'

  - ("PHONENO" POSITION(3:6) INTEGER,
  - "CUSTONO" POSITION(7:10) INTEGER,
  - "CALLDUR" POSITION(11:12) SMALLINT,
  - "DATE" POSITION(13:22) DATE EXTERNAL)
T - EXISTING RECORDS DELETED FROM TABLESPACE PARTITION 2
  - (RE)LOAD PHASE STATISTICS - NUMBER OF RECORDS=2 FOR TABLE MKRES7.TABLE_A_ARC

  - (RE)LOAD PHASE STATISTICS - TOTAL NUMBER OF RECORDS LOADED=2 FOR TABLESPACE
```

```
(RE)LOAD PHASE STATISTICS - NUMBER OF INPUT RECORDS PROCESSED=2
(RE)LOAD PHASE COMPLETE, ELAPSED TIME=00:00:00
BUILD PHASE STATISTICS - NUMBER OF INDEXES=0
BUILD PHASE COMPLETE, ELAPSED TIME=00:00:00
UTILITY EXECUTION COMPLETE, HIGHEST RETURN CODE=0
```

The target archive table (TABLE_A_ARC) is loaded successfully.

8.2.4 Deleting data from specific partitions of an archive-enabled table

Now it's time to erase the records in partitions 1 and 2 of Table_A with LOAD REPLACE DD DUMMY. In Example 8-6, we used the LOAD utility to DELETE rows from partitions 1 and 2 of TABLE_A.

Example 8-6 LOAD REPLACE DD DUMMY process for Table_A

```
//LOAD EXEC DSNUPROC,SYSTEM=D1DG,
//          LIB='DB1DT.SDSNLOAD',
//          UID=''
//DSNUPROC.SYSUT1 DD DSN=MKRES7.TEST.SYSUT1,
//          DISP=(MOD,DELETE,CATLG),
//          SPACE=(TRK,(10,5),RLSE),
//          UNIT=SYSDA
//DSNUPROC.SORTOUT DD DSN=MKRES7.TEST.SORTOU,
//          DISP=(MOD,DELETE,CATLG),
//          SPACE=(TRK,(10,5),RLSE),
//          UNIT=SYSDA
//DSNUPROC.SYSREC DD DUMMY
//DSNUPROC.SYSIN DD *
LOAD DATA INDDN SYSREC LOG YES
EBCDIC CCSID(00037,00000,00000)
INTO TABLE
"MKRES7".
"TABLE_A"
PART 00001 REPLACE
LOAD DATA INDDN SYSREC LOG YES
EBCDIC CCSID(00037,00000,00000)
INTO TABLE
"MKRES7".
"TABLE_A"
PART 00002 REPLACE
```

The final view of both tables is as shown in Example 8-7.

Example 8-7 Final view of the tables

```
DB2 Admin -- D1DG BROWSE MKRES7.TABLE_A
Command ==>

***** Top of Dat
      PHONENO      CUSTONO  CALLDUR  DATE
-----
      3171201          2       12 2015-06-14
      3171427          1       25 2015-06-15
DB2 Admin -- D1DG BROWSE MKRES7.TABLE_A_ARC
```

Command ==>

```
***** Top of Data
```

PHONENO	CUSTONO	CALLDUR	DATE
3171427	1	10	2015-04-13
3171427	1	15	2015-04-15
3171201	2	11	2015-04-15
3171201	2	11	2015-04-16
3171427	1	15	2015-05-15
3171201	2	12	2015-05-15

Table_A table has only rows from June and TABLE_A_ARC has rows from April and May.

8.3 Enabling archive transparency

Now we have both active and archive tables. Data is also distributed between them from usage perspective. It's time to activate archive transparency. According to Figure 8-3, when we enable archive transparency, Table_A will be the archive-enabled table and TABLE_A_ARC will be the archive table.

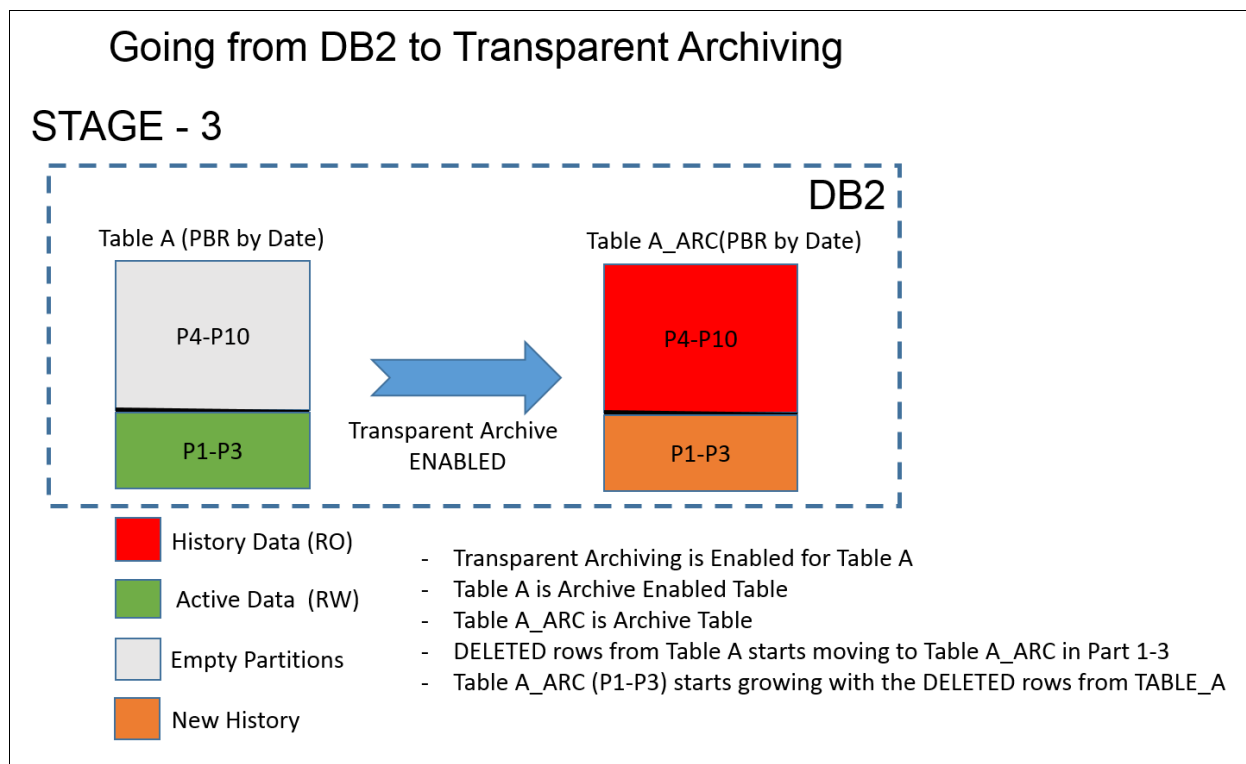


Figure 8-3 Enabling archive transparency

Example 8-8 shows how to enable archiving for Table_A.

Example 8-8 Enabling transparent archive

```
ALTER TABLE TABLE_A ENABLE ARCHIVE USE TABLE_A_ARC;
-----+-----+-----+-----+-----+-----
DSNE616I STATEMENT EXECUTION WAS SUCCESSFUL, SQLCODE IS 0
-----+-----+-----+-----+-----+-----
```

When we enable transparent archiving, it is reflected in SYSIBM.SYSTABLES as seen in Example 8-9.

- ▶ Archive-enabled table:
TYPE = 'T', ARCHIVING_TABLE refers to the archive-enabled table
- ▶ Archive table:
TYPE = 'R', ARCHIVING_TABLE refers to the archive table

Example 8-9 Catalog view of archive transparency

Seq	Name	Schema	T DB Name	TS Name	Cols
	*	*	* *	*	*
-----+-----+-----+-----+-----+-----					
*	TABLE_A	MKRES7	T PHONE	PHONE	4
*	TABLE_A_ARC	MKRES7	R PHONE	PHONEH	4

When we enable transparent archiving, it is reflected in SYSIBM.SYSTABLES, as seen in Example 8-9.

If the column names, data types, lengths, null attributes, and so on, do not match between the archive-enabled table and the archive table, you might get the following error message (Example 8-10). Details of the reason code can be found in the DB2 Codes manual.

Example 8-10 Error message

```
TABLE ADMF001.TABLE_A_ARC WAS SPECIFIED AS AN ARCHIVE TABLE, BUT THE TABLE
DEFINITION IS NOT VALID FOR AN ARCHIVE TABLE. REASON CODE 10. SQLCODE=-20554,
SQLSTATE=428HX,
DRIVER=4.18.60
```

8.4 Mechanics of moving data from an archive-enabled table to an archive table

Two methods are available to move data from the active table into the archive table:

- ▶ DELETE from archive-enabled table
- ▶ REORG DISCARD on archive-enabled table

DELETE behavior is controlled by the built-in global variable SYSIBMADM.MOVE_TO_ARCHIVE, which has three modes:

- ▶ Archiving mode, all rows deleted will be moved, no insert/update allowed:
SETSYSIBMADM.MOVE_TO_ARCHIVE='Y';--no insert/update allowed

- Enabled mode, all rows deleted will be moved, insert/update allowed:
SETSYSIBMADM.MOVE_TO_ARCHIVE='E';--insert/update allowed
- Disabled mode, all rows deleted will be gone:
SETSYSIBMADM.MOVE_TO_ARCHIVE='N';--'N' is the default

First, we do a quick test to update a row in archive mode in Example 8-11.

Example 8-11 Update

```

SET SYSIBMADM.MOVE_TO_ARCHIVE='Y';
-----+-----+-----+-----+-----+-----+-----+
DSNE616I STATEMENT EXECUTION WAS SUCCESSFUL, SQLCODE IS 0
-----+-----+-----+-----+-----+-----+-----+
      UPDATE TABLE_A SET CALLDUR = 15 WHERE PHONENO=3171201;
-----+-----+-----+-----+-----+-----+-----+
DSNT408I SQLCODE = -20555, ERROR:  AN ARCHIVE-ENABLED TABLE IS NOT ALLOWED IN THE
SPECIFIED CONTEXT. REASON CODE  2

```

Now, let's update and move a row to the archive table after updating it as seen in Example 8-12.

Example 8-12 Update a row in the archive-enabled table and move it to the archive table

```

SET SYSIBMADM.MOVE_TO_ARCHIVE='E';
-----+-----+-----+-----+-----+-----+-----+
DSNE616I STATEMENT EXECUTION WAS SUCCESSFUL, SQLCODE IS 0
-----+-----+-----+-----+-----+-----+-----+
      UPDATE TABLE_A SET CALLDUR = 15 WHERE PHONENO=3171201;
-----+-----+-----+-----+-----+-----+-----+
DSNE615I NUMBER OF ROWS AFFECTED IS 1
DSNE616I STATEMENT EXECUTION WAS SUCCESSFUL, SQLCODE IS 0
-----+-----+-----+-----+-----+-----+-----+
      DELETE FROM TABLE_A WHERE PHONENO=3171201;
-----+-----+-----+-----+-----+-----+-----+
DSNE615I NUMBER OF ROWS AFFECTED IS 1
DSNE616I STATEMENT EXECUTION WAS SUCCESSFUL, SQLCODE IS 0
-----+-----+-----+-----+-----+-----+-----+

```

As you'll see from Example 8-13, the updated row is also moved to TABLE_A_ARC after the DELETE statement is executed against Table_A. The row that is bold in TABLE_A_ARC, is updated and moved.

Example 8-13 Final view of the tables after UPDATE and DELETE

```

SELECT * FROM TABLE_A;
-----+-----+-----+-----+-----+-----+-----+
      PHONENO      CUSTONO  CALLDUR  DATE
-----+-----+-----+-----+-----+-----+-----+
      3171427          1       25  2015-06-15
DSNE610I NUMBER OF ROWS DISPLAYED IS 1
DSNE616I STATEMENT EXECUTION WAS SUCCESSFUL, SQLCODE IS 100
-----+-----+-----+-----+-----+-----+-----+
      SELECT * FROM TABLE_A_ARC;
-----+-----+-----+-----+-----+-----+-----+
      PHONENO      CUSTONO  CALLDUR  DATE
-----+-----+-----+-----+-----+-----+-----+

```

8.5 Querying data from an archive-enabled table

Note the following access levels:

- ▶ Access to current data only or current data plus archive data is controlled by the built-in global variable `SYSIBMADM.GET_ARCHIVE`.
- ▶ Access to current data only:
`SET SYSIBMADM.GET_ARCHIVE='N';` --'N' is the default.
- ▶ Access to both current and archive data:
`SET SYSIBMADM.GET_ARCHIVE='Y';`

DB2 transforms any query to implicitly union all the archive-enabled table with the archive table for transparent access.

You can query active and archive data separately. As you'll see when `SYSIBMADM.GET_ARCHIVE` global variable is set to `NO`, each table can be accessed individually (Example 8-14).

Example 8-14 Query active and archive data separately

```

SET SYSIBMADM.GET_ARCHIVE='N';
-----+-----+-----+-----+-----
DSNE616I STATEMENT EXECUTION WAS SUCCESSFUL, SQ
-----+-----+-----+-----+-----
      SELECT * FROM TABLE_A;  -- the archive-enabled table
-----+-----+-----+-----+-----
      PHONENO          CUSTONO    CALLDUR    DATE
-----+-----+-----+-----+-----
      3171427          1          25    2015-06-15
DSNE610I NUMBER OF ROWS DISPLAYED IS 1
DSNE616I STATEMENT EXECUTION WAS SUCCESSFUL, SQ
-----+-----+-----+-----+-----
      SELECT * FROM TABLE_A_ARC;  -- the archive table
-----+-----+-----+-----+-----
      PHONENO          CUSTONO    CALLDUR    DATE
-----+-----+-----+-----+-----
      3171427          1          10    2015-04-13
      3171427          1          15    2015-04-15
      3171201          2          11    2015-04-15
      3171201          2          11    2015-04-16
      3171427          1          15    2015-05-15
      3171201          2          12    2015-05-15

```

```

3171201          2      15  2015-06-14
DSNE610I NUMBER OF ROWS DISPLAYED IS 7

```

And now we run the same query as in Example 8-14 on page 122, but this time it will transparently access the archived data. By setting the SYSIBMADM.GET_ARCHIVE global variable to YES, the query against the archive-enabled table now transparently retrieves data from both tables (Example 8-15).

Example 8-15 Query from the archive-enabled table

```

SET SYSIBMADM.GET_ARCHIVE='Y';
-----+-----+-----+-----+-----+
DSNE616I STATEMENT EXECUTION WAS SUCCESSFUL, SQLCODE IS 0
-----+-----+-----+-----+-----+
      SELECT * FROM TABLE_A; -- the archive-enabled table
-----+-----+-----+-----+-----+
      PHONENO      CUSTONO  CALLDUR  DATE
-----+-----+-----+-----+-----+
      3171427          1      25  2015-06-15
      3171427          1      10  2015-04-13
      3171427          1      15  2015-04-15
      3171201          2      11  2015-04-15
      3171201          2      11  2015-04-16
      3171427          1      15  2015-05-15
      3171201          2      12  2015-05-15
      3171201          2      15  2015-06-14
DSNE610I NUMBER OF ROWS DISPLAYED IS 8
DSNE616I STATEMENT EXECUTION WAS SUCCESSFUL, SQLCODE IS 100

```

Note that the SELECT in Example 8-15 returned data from both the archive-enabled table as well as the archive table. This was the same query that was run in Example 8-14 on page 122 where it only returned rows from the archive-enabled table (the active rows). The difference is in the setting of the SYSIBMADM.GET_ARCHIVE global variable. For Example 8-15, the global variable was set to 'Y' so the query transparently accesses the archive table as well as the archive-enabled table.



Part 3

DB2 Analytics Accelerator

Part 3 contains the following chapters:

- ▶ Chapter 9, “DB2 Analytics Accelerator overview” on page 127
- ▶ Chapter 10, “DB2 Analytics Accelerator High Performance Storage Saver overview” on page 135
- ▶ Chapter 11, “Case study: Using the DB2 Analytics Accelerator” on page 147



DB2 Analytics Accelerator overview

The DB2 Analytics Accelerator can help in creating the insight that your organization needs by delivering the right answers for your business, with optimal performance and cost effectiveness.

This chapter contains the following sections:

- ▶ Overview
- ▶ Variations for using the DB2 Analytics Accelerator

9.1 Overview


Driven by the global economy, businesses are faced with an increasingly competitive environment and the amount of data that they need to capture and analyze for business decisions is exploding. Companies constantly seek new ways to leverage the information that is available to them to identify opportunities, enabling them to respond quickly to changing business situations and improve their business advantage. The use of analytics has become a differentiating factor to explore insight from information that is available, and provide analytics that are critical for making business decisions. In order to maintain a competitive edge, businesses must use all the information at their disposal to create actionable insights that drive decision making and smarter transactions. DB2 for z/OS and the IBM DB2 Analytics Accelerator form a self-managing hybrid environment to address the needs of business intelligence and analytic processing workloads, while continuing to run mission-critical transaction processing and analytical workload concurrently and efficiently. The DB2 Analytics Accelerator leverages the power of zEnterprise, DB2 for z/OS, and Netezza technology, which enables the integration of analytic insights into operational processes to drive business critical analytics resulting in exceptional business value.

The DB2 Analytics Accelerator (Figure 9-1) keeps a copy of the DB2 for z/OS tables. This combination leverages DB2 for z/OS performance for transactional queries with industry-leading performance for analytical queries. With this integration, you can accelerate data-intensive and complex queries in a DB2 for z/OS highly secure and available environment. When a query takes a long time to provide the information that is necessary to make a key decision, the business opportunity may be lost. With the DB2 Analytics Accelerator, complex queries can be run in significantly less time, providing necessary information for business decisions. The combination of DB2 for z/OS and the DB2 Analytics Accelerator implements the vision of a universal relational DBMS that processes OLTP and analytical workloads in a single system. By using heuristics, DB2 for z/OS determines if a query should be executed in DB2 or offloaded to the attached DB2 Analytics Accelerator.

DB2 Analytics Accelerator

Accelerating decisions to the speed of business

Blending System z and Netezza technologies to deliver unparalleled, mixed workload performance for complex analytic business needs.



- **Get more insight from your data timely**
- Fast, predictable response times for “right-time” analysis
- Accelerate analytic query response times
- Improve price/performance for analytic workloads
- Minimize the need to create data marts for performance
- Highly secure environment for sensitive data analysis
- Transparent to the application

Figure 9-1 DB2 Analytics Accelerator

With DB2 for z/OS combined with the DB2 Analytics Accelerator, high-performance analytics are available to the DB2 SQL user. This combination:

- ▶ Allows mixing transactional and analytics workloads
- ▶ Transparent access to the data on the DB2 Analytics Accelerator
- ▶ Supports running complex queries on very large volumes of data
- ▶ Accelerates analytic query response times
- ▶ Lowers the cost of storing, managing, and processing historical data
- ▶ Improves price and performance for analytic workloads
- ▶ Minimizes the need to have data marts for performance
- ▶ Reduces capacity requirements on IBM z Systems
- ▶ Reduces operational costs and risk
- ▶ Complements existing DB2 for z/OS investments

9.2 Variations for using the DB2 Analytics Accelerator

Several different types of “accelerator tables” are supported by DB2 for z/OS and the Analytics Accelerator. Choose the type of accelerator table that best fits your application and needs for using the DB2 Analytics Accelerator.

9.2.1 Accelerator-shadow table

An accelerator-shadow table exists both in DB2 and in the DB2 Analytics Accelerator (Figure 9-2). The table in the accelerator contains all or a subset of the columns in the DB2 table. After the table is defined on the accelerator, you can load data into the table on the accelerator by copying data from the original DB2 table to the corresponding table on the accelerator. After the data is loaded on the accelerator, you can enable query acceleration for this table by using one of the following methods: QUERY_ACCELERATION subsystem parameter, CURRENT QUERY ACCELERATION special register, QUERYACCELERATION bind option, or connection properties (JDBC and ODBC) and profile tables.

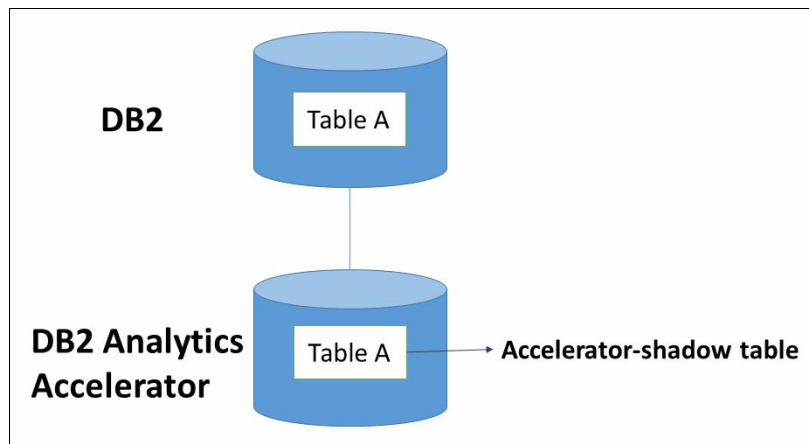


Figure 9-2 Accelerator shadow table

9.2.2 Accelerator-archived table

An accelerator-archived table is a table on the accelerator for which partitions of a DB2 table, or the entire table, are archived on the accelerator (Figure 9-3). When the data is archived on the accelerator, the original DB2 data is deleted. The table space for a DB2 table that is associated with an archived accelerator table is set to a permanent read-only state so that the original DB2 table can no longer be changed. An image copy of the data is also created as part of the archive, to enable recovery of the data in an emergency situation. The recovery function in the DB2 Analytics Accelerator uses the image copies to restore the data in the original DB2 table.

Accelerator-archived tables are used primarily for historical data that is no longer actively used or maintained. The archive saves storage space on z Systems. Normally, the archived data is not included in an accelerated query. However, you can specify that archived data is to be included in an accelerated query by using one of the following methods:

QUERY_ACCELERATION subsystem parameter, CURRENT QUERY ACCELERATION special register, QUERYACCELERATION bind option, and connection properties (JDBC and ODBC) and profile tables.

An accelerator-archived table can be a High Performance Storage Saver table. See Chapter 10, “DB2 Analytics Accelerator High Performance Storage Saver overview” on page 135.

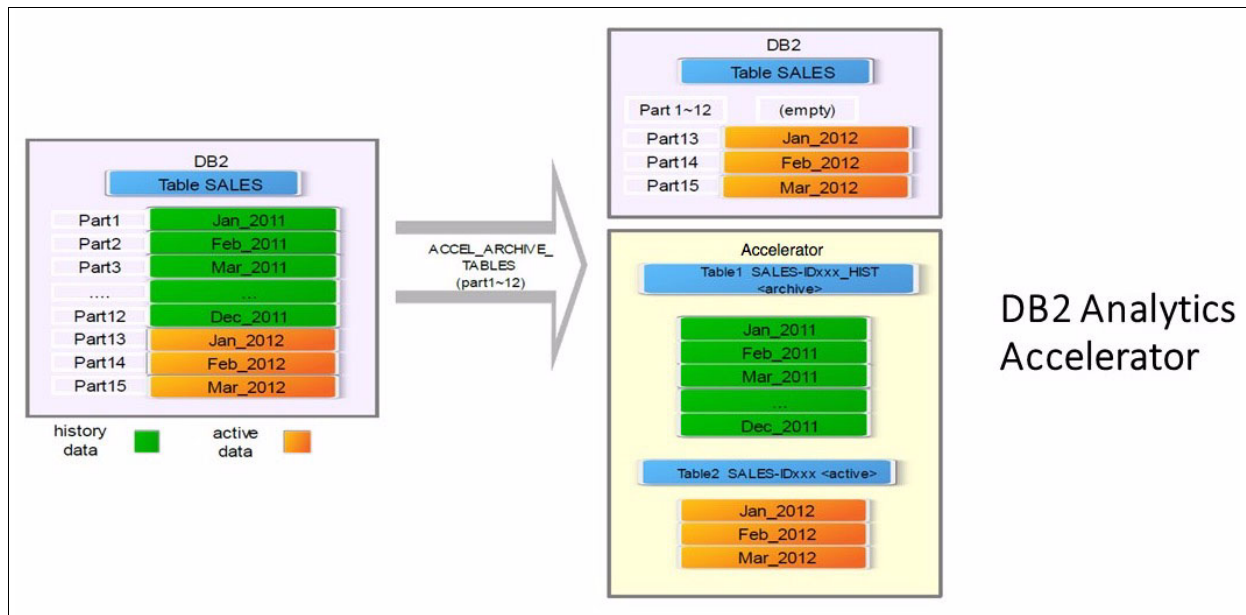


Figure 9-3 Accelerator-archived table

9.2.3 Accelerator-only table

An *accelerator-only table* is a table for which the data exists only on the DB2 Analytics Accelerator (Figure 9-4 on page 131). Accelerator-only tables are the newest type of accelerator tables that are supported by DB2 for z/OS.

An accelerator-only table is automatically created in the DB2 Analytics Accelerator when the CREATE TABLE statement is issued on DB2 with the IN ACCELERATOR clause. The table and column definitions of the accelerator-only table are reflected in the DB2 catalog. Any queries that reference an accelerator-only table must be executed in the accelerator and will be accelerated. A data change statement for an accelerator-only table must be executed in the accelerator. If the data change statement type is not supported by the accelerator, or if the statement contains any expression that is not supported by the accelerator, an error is returned.

When validate run behavior is in effect, a static query or data change statement that references an accelerator-only table is incrementally bound at run time. A static query or data change statement that references an accelerator-only table is eligible for acceleration during an incremental bind only if the QUERYACCELERATION bind option that is in effect is ENABLE or ELIGIBLE. When a static query or data change statement that references an accelerator-only table is issued and the QUERYACCELERATION bind option that is in effect is ALL, an error is returned.

Running queries and data change statements on the accelerator can significantly speed up SQL statements, such as INSERT from SELECT statements. This is not possible with the other two types of accelerator tables. Accelerator-only tables are used by statistics and analytics tools, which can quickly gather all the data that is required for reports. Because the data in these tables can be modified so quickly, they are also ideal for data-preparation tasks that must be completed before the data can be used for predictive modeling.

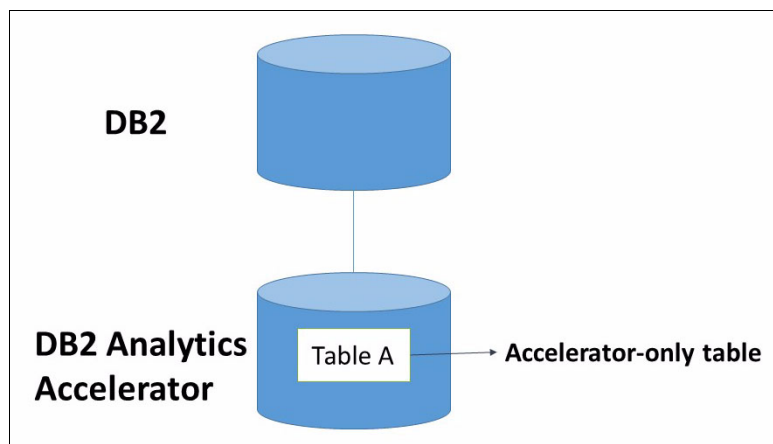


Figure 9-4 Accelerator-only table

9.2.4 Using temporal tables and DB2 Analytics Accelerator together

It is possible to use application-period temporal tables with the DB2 Analytics Accelerator (Figure 9-5 on page 132). In this case, the application-period temporal table is defined as an accelerator-shadow table. See Chapter 12, “Case study: Combining DB2 temporal data management with the DB2 Analytics Accelerator” on page 157.

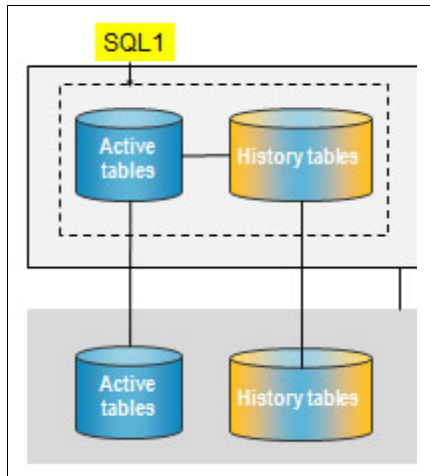


Figure 9-5 Using temporal tables with the DB2 Analytics Accelerator

9.2.5 Using archive transparency and the DB2 Analytics Accelerator together

It is possible to use archive transparency in combination with the DB2 Analytics Accelerator (Figure 9-6). In this case, both the archive-enabled table and the archive table can be enabled for query acceleration. Additionally, the archive table can be moved to the DB2 Analytics Accelerator with the High Performance Storage Saver feature. See Chapter 13, “Case study: Combining archive transparency with the DB2 Analytics Accelerator ” on page 171.

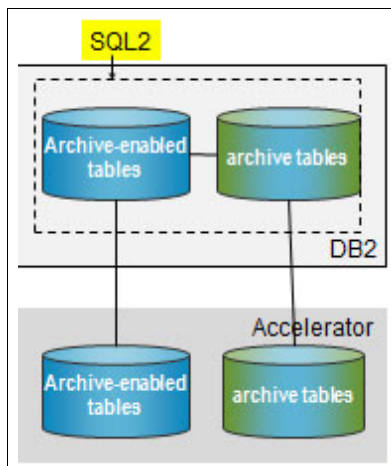


Figure 9-6 Using archive transparency with the DB2 Analytics Accelerator

9.2.6 Using a row change timestamp column with archive transparency and the DB2 Analytics Accelerator

A row change timestamp column can be useful with archive transparency in offloading older data in the archive table on DB2 to the DB2 Analytics Accelerator.

Define a row change timestamp column in both the archive-enabled table and archive tables. In the archive-enabled table, the column records the time of data change operations. The archive table is partitioned on the row change timestamp column. The archive-enabled table can be partitioned on the same column or a different key.

When a row of an archive-enabled table is moved to the associated archive table, DB2 automatically records a value in the row change timestamp column that indicates when the row was archived to the archive table. Later, when moving some of the archived data on DB2 to the DB2 Analytics Accelerator, whole partitions of the archive table can be offloaded based on the time the rows were moved to the archive table.

To define a row change timestamp column in the archive-enabled table and archive table, use the following syntax for the column definition:

```
GENERATED ALWAYS FOR EACH ROW ON UPDATE  
    AS ROW CHANGE TIMESTAMP
```




DB2 Analytics Accelerator High Performance Storage Saver overview

This chapter introduces the IBM DB2 Analytics Accelerator for z/OS (DB2 Analytics Accelerator) High Performance Storage Saver feature and its functional capabilities.

This chapter contains the following sections:

- ▶ Overview
- ▶ What is High Performance Storage Saver?
- ▶ Online data archiving

10.1 Overview

High Performance Storage Saver enables users to archive partitions of a table, or the entire table, from the DB2 environment to the DB2 Analytics Accelerator. The data belonging to older partitions is moved to the DB2 Analytics Accelerator and purged from DB2. From this perspective, the feature can be considered an archiving option, and the partitions that are only in the DB2 Analytics Accelerator are often referred to as *archived partitions*. However, note that there are no drawbacks regarding reduced availability of data, which is usually a by-product of archiving. The partitions that are in both DB2 and the DB2 Analytics Accelerator are referred to as *non-archived partitions*.

High Performance Storage Saver (first introduced in V3.1 and extended in V4.1) provides enhanced capabilities such as better access control to High Performance Storage Saver-archived partitions, improved flexibility, and protection of image copies created by High Performance Storage Saver, built-in restore for High Performance Storage Saver, and High Performance Storage Saver archiving for multiple accelerators.

High Performance Storage Saver provides a partition or table-based archiving solution rather than row-based archiving. It supports both DB2 10 for z/OS and DB2 11 for z/OS. High Performance Storage Saver can be used to store data in partitions or tables that exist only in DB2. Unlike normal partitions, the data does not exist in DB2. Data in archived partitions is available using read-only SQL. Archived data can be restored to DB2 in case needed.

High Performance Storage Saver is more restrictive than transparent archiving as the DB2 Analytics Accelerator does not support all:

- ▶ Data types
- ▶ SQL clauses

Today, typical business analytic query systems are based on data where more than 95% of host disk space is occupied by static historical data. The history data generally is not subject to change or regular updates, if at all.

High Performance Storage Saver offers several benefits:

- ▶ Reduced usage of physical disk storage in DB2.
- ▶ With the data being removed from the DB2 table, the size of corresponding indexes shrinks, which further reduces disk storage requirements. More importantly, removing the data improves performance when accessing the remaining data.
- ▶ Administrative operations on the data remaining in DB2, such as backup and recovery times, become faster because of the reduced data volume.
- ▶ The data moved to the DB2 Analytics Accelerator is still available, and it can be accessed by using SQL. Access to the archive data in the DB2 Analytics Accelerator delivers the usual high performance.
- ▶ High Performance Storage Saver is one of the most advanced multi-temperature data solutions, and the next step toward an integrated online transaction processing (OLTP) and online analytical processing (OLAP) system.
- ▶ High Performance Storage Saver tables can exist on multiple accelerators, which provide high availability.

High Performance Storage Saver is a fully integrated solution. Its use has the positive side effect that it also makes the database system more responsive. This is because fewer objects need to be maintained in the catalog, smaller or fewer indexes need to be searched, reorganizations become quicker, and report data and statistics can be gathered faster.

10.2 What is High Performance Storage Saver?

The DB2 Analytics Accelerator can function as a High-Performance Storage Saver for online archive. High Performance Storage Saver moves data of table partitions or an entire table in DB2 for z/OS to an accelerator. This capability stores static, historical, time-partitioned partitions, or an entire table solely on the accelerator. This effectively reduces the host storage requirement and data volume in DB2 and it improves query performance. Subsequently, it removes the requirement for the data to be replicated on both DB2 for z/OS and the accelerator storage.

Tables or partitions can now be divided between traditional database resident partitions on DB2 for z/OS and the DB2 Analytics Accelerator:

- ▶ Users continue high-speed query access while reducing the storing and maintaining of the data in DB2.
- ▶ With the data being removed from the DB2 table, the size of corresponding indexes shrinks, which further reduces disk storage requirements. More importantly, removing the data improves performance when accessing the remaining data.
- ▶ Administrative operations, such as backup and recovery times, become faster because of the reduced data volume.
- ▶ High Performance Storage Saver tables can exist on multiple accelerators, which provide high availability.
- ▶ Users do not need to create a separate data warehouse or extract, transform, and load (ETL) to another platform.

The more recent data partitions are typically used with frequent data changes in a transactional context and with short running queries. The entire table is typically used for analytics that are data intensive or to aggregate complex queries.

With High Performance Storage Saver, DB2 for z/OS still has ownership of all data, and all the queries that target the data archived in the accelerator are now only directed to the accelerator. This not only provides storage savings on IBM z Systems, it also allows the organization to substantially increase the amount of history data to be continuously archived and stored in the accelerator over time. The accelerator now hosts both the recent “active” operational partition as well as the newly “archived” partitions (Figure 10-1).

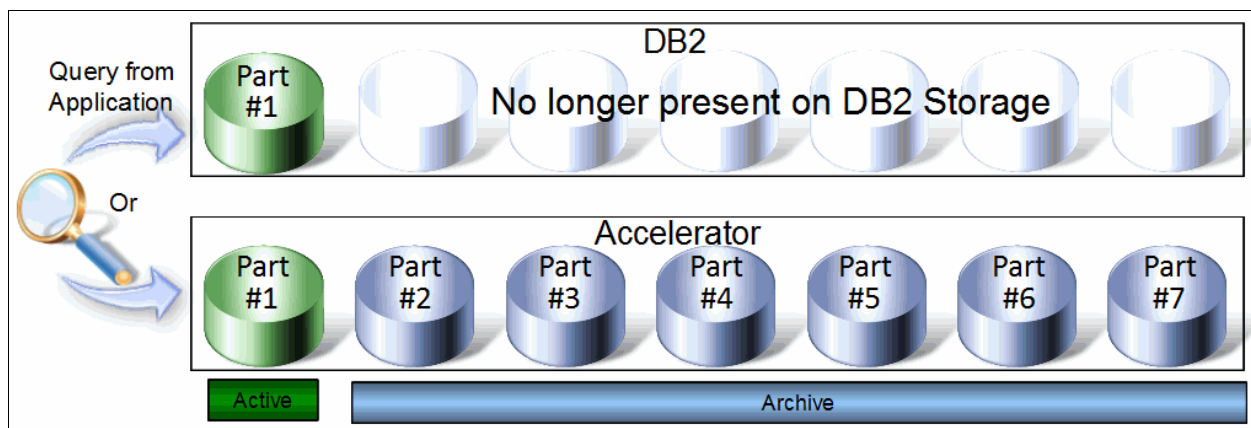


Figure 10-1 Archived partitions

High Performance Storage Saver is designed to integrate into DB2 at the SQL level, which provides transparency for any user application accessing the online archive in the accelerator.

You can determine whether moved data is to be included in queries. By default, this type of data is excluded, and the IBM DB2 Analytics Accelerator for z/OS automatically skips moved records when it processes a query. This often leads to different results for queries that are run repeatedly because the query is run just against the remaining DB2 data. If this behavior is not wanted, you can set the CURRENT GET_ACCEL_ARCHIVE special register or the GET_ACCEL_ARCHIVE configuration parameter to YES.

High Performance Storage Saver is flexible. It is also possible to move the data of one or more partitions to an accelerator, while the data of other partitions remains in DB2 for z/OS and on the same or other accelerators. Before any data is moved to an accelerator and deleted from DB2, full image copies with SHRLEVEL REFERENCE are created, which allow you to restore the data if needed. The data sets of the moved partitions are set to a persistent read-only state (PRO), which prevents future insert, update, and delete operations on these partitions. Furthermore, the High Performance Storage Saver ensures that the data in the image copies is consistent with the data that is going to be moved.

The table space of a moved partition continues to exist after the move. It will be empty, but it will still allocate as much disk space as defined by the primary quantity. To meet your space-saving goal, check the (PRIQTY) for the table spaces of the partitions that you want to move and decrease the value if needed by submitting an appropriate ALTER TABLESPACE ... PRIQTY statement.

10.2.1 Restrictions

The following restrictions exist with the High Performance Storage Saver:

- ▶ The High Performance Storage Saver works on range-partitioned tables only (PBR). If the partitioning is controlled by an index, the index must already exist.
- ▶ The smallest unit that you can move is a partition as opposed to transparent archiving where the smallest unit is table. That is, all the table rows in specified partitions are copied to the accelerator and are finally removed from DB2.
- ▶ You can only move partitions of tables, which are not enabled for incremental update.
- ▶ A table on High Performance Storage Saver cannot be a parent table in a referential integrity relationship. The reason for this is that the original data is deleted at the end of the move process, which removes the values of the foreign key.
- ▶ Columns that cannot be loaded into accelerator tables because their data types are not supported also cannot be moved. The following data types exist:
 - DECFLOAT
 - TIMESTAMP(n) where n has a value other than 6
 - TIMESTAMP WITH TIME ZONE
 - ROWID
 - BLOB
 - CLOB
 - DBCLOB
 - XML
- ▶ For a move operation to succeed, an S-Lock is required on the involved tables or partitions. Long-running queries might prevent the lock from being obtained. Ensure that those long-running queries are finished before you start a data move.
- ▶ If the GET_ACCEL_ARCHIVE special register (or system parameter) has been set to the value YES, and a query references tables whose data has been moved to different accelerators, the query fails.

10.3 Online data archiving

This section focuses on online data archiving, accessing archived data, and the restore processes. High Performance Storage Saver has been available in the DB2 Analytics Accelerator since V3.1 and provides enhanced features, such as better access control to High Performance Storage Saver-archived partitions, improved flexibility and protection of image copies created by High Performance Storage Saver, built-in restore for High Performance Storage Saver, and High Performance Storage Saver archiving for multiple accelerators.

The following two stored procedures are required to automate the High Performance Storage Saver archive and restore processes. They are `SYSPROC.ACCEL_ARCHIVE_TABLES` and `SYSPROC.ACCEL_RESTORE_ARCHIVE_TABLES`:

- ▶ `SYSPROC.ACCEL_ARCHIVE_TABLES` moves table partitions from DB2 for z/OS to a storage area on a DB2 Analytics Accelerator.
- ▶ `SYSPROC.ACCEL_RESTORE_ARCHIVE_TABLES` restores the data of moved partitions to their original locations. It works on partitions that were moved with High Performance Storage Saver.

10.3.1 Archive data process and operations

The archive partition process uses the `SYSPROC.ACCEL_ARCHIVE_TABLES` stored procedure to move data from one or more table partitions in DB2 to a DB2 Analytics Accelerator. The stored procedure triggers or completes the following tasks against the partition or table to be moved:

- ▶ Locks the partition or table
- ▶ Unloads data from the partition or table
- ▶ Creates the image copies
- ▶ Creates (adds) a table in the DB2 Analytics Accelerator
- ▶ Transfers data into the DB2 Analytics Accelerator
- ▶ Sets the partition to the UT state
- ▶ Removes data from DB2 table (pruning)
- ▶ Invalidates the DB2 dynamic statement cache and packages for the tables
- ▶ Updates `SYSACCEL.SYSACCELERATEDTABLES`

Objects must be defined to the accelerator and must be in either the `InitialLoadPending` or the `Loaded State`. Incremental update must not be active. It is possible to reinstate this once the archiving process is completed.

Data in a base partition is pruned by using `LOAD REPLACE`. Partitions remain in DB2 using the minimum primary allocation to save DASD storage. It might be better to `ALTER TABLESPACE PRIQTY` to a small value *before* archiving. This allows `LOAD REPLACE` to shrink the table space size to the minimum required.

If there are non-partitioning index (NPI) rebuilds, it can slow down the archiving process. It is better to drop NPIs before the archive process.

During the archive process, DB2 image copies are created automatically. These image copies are used to restore the original data to DB2.

The stored procedure enables up to four new image copies to be created for each moved partition. The stored procedure uses the DB2 `COPY` utility to create up to two local images and two recovery images.

The naming convention for the image copy data sets is controlled by the values specified in the new environment variables in the AQTENV data set. Template support was added with PTF2 (now up to PTF4). Data set names need to be unique: use &PART or &UNIT:

- ▶ AQT_ARCHIVE_COPY1
- ▶ AQT_ARCHIVE_COPY2
- ▶ AQT_ARCHIVE_RECOVERYCOPY1
- ▶ AQT_ARCHIVE_RECOVERYCOPY2

When a partition is moved to the DB2 Analytics Accelerator, accelerated queries can only be run against the data in that partition in the accelerator if the GET_ACCEL_ARCHIVE=YES special register or subsystem parameter is used. The table data can no longer be updated, not in the original DB2 partition, and not on the DB2 Analytics Accelerator because DB2 partitions are set to the PRO state.

Tables are processed serially but partitions in a table can be moved in parallel. By default, the data is transferred four partitions at a time. This can be changed by adjusting the AQT_MAX_UNLOAD_IN_PARALLEL environment variable in the AQTENV data set.

The data removal process is called *pruning*, and it requires exclusive access to the tables in a partition. The stored procedure issues the START DATABASE ACCESS(UT) command to make the partition available only for DB2 online utilities (and the SQL DROP statement). Data is deleted with a LOAD REPLACE operation.

If the data removal process fails for any partition, the operation is not rolled back. The object remains in the UT state. It is better to fix the problem and invoke the stored procedure. More serious issues may require manual recovery.

The UT lock remains active when the stored procedure fails in the pruning process, but enables the process to resume where it left off. The number of retries, and the time interval between retries, are controlled by the AQT_MAX_RETRIES_DSNUTILU and AQT_SECONDS_BEFORE_RETRY_DSNUTILU environment variables.

The connection to the client must remain active while the stored procedure runs. It could take some time (hours) so pay attention if the IBM Data Studio is being used. It might be better to run the stored procedure from a batch job.

Once the archiving process completes, the columns of the SYSACCEL.SYSACCELERATEDTABLES table are updated as follows:

- ▶ ARCHIVE
 - **Blank** Table is not archived: Active requests only
 - **A** Archived: Handles active and archive requests
 - **C** Archived elsewhere: Active requests only
 - **B** Similar to C but archiving is partially complete: Active only
 - **X** Table has been restored on another accelerator: Disabled
- ▶ ENABLE
 - **Y** Enabled and queries can be offloaded
 - **N** Disabled
 - **T** Transition

10.3.2 Accessing archived data

Querying data in accelerator is easy using regular SQL. To ask for archived data, specify range predicates. With the introduction of High Performance Storage Saver online archive, for each query, DB2 looks at query processing against recent, non-archived partitions, or archived partitions. For every query queued for processing, DB2 checks to see whether the following conditions are met:

- Query acceleration is enabled
- All tables referenced by the query have been loaded on the accelerator
- The query is qualified and meets the routing criteria for query acceleration
- Heuristics decisions, optimal query access plan, and cost-based optimization for query routing apply

Accessing archived data requires two settings before the request works:

- ▶ Acceleration must be enabled:
 - DSNZPARM QUERY_ACCELERATION sets the default.
 - Use the QUERY ACCELERATION special register to provide an override for dynamic SQL statements.
 - Use the QUERYACCELERATION bind option to provide an override for static SQL statements.
- ▶ Access to archived data must be permitted:
 - DSNZPARM GET_ACCEL_ARCHIVE sets the default.
 - Use the GET_ACCEL_ARCHIVE special register to provide an override for dynamic SQL statements.
 - Use the GETACCELARCHIVE bind option to provide an override for static SQL statements.

It is your choice whether to include archive data for query processing. This is accomplished by setting the appropriate special register or bind option:

- ▶ With archive data, the query can only run in the accelerator
- ▶ The DB2 heuristics check and query routing rule do not apply in this case

With High Performance Storage Saver, the query can be routed and needs to access the online archived data. DB2 rewrites the query to combine the active partitions (in a table on the DB2 side) with the archived partitions (which reside in the corresponding table on the accelerator) via the UNION ALL SQL operator.

We explain this with an example. In Figure 10-2 on page 142, assume that we have a table that is partitioned by date. Between part n and part n-1, there is active data and the remaining partitions contain “cold” or read-only data. We want to keep the active data both in DB2 and in the accelerator, and the cold data will be moved to the accelerator to be available for queries.

The ACCEL_ARCHIVE_TABLES stored procedure encapsulates the archiving process. In this example, we assume that the table is already added and loaded to accelerator.

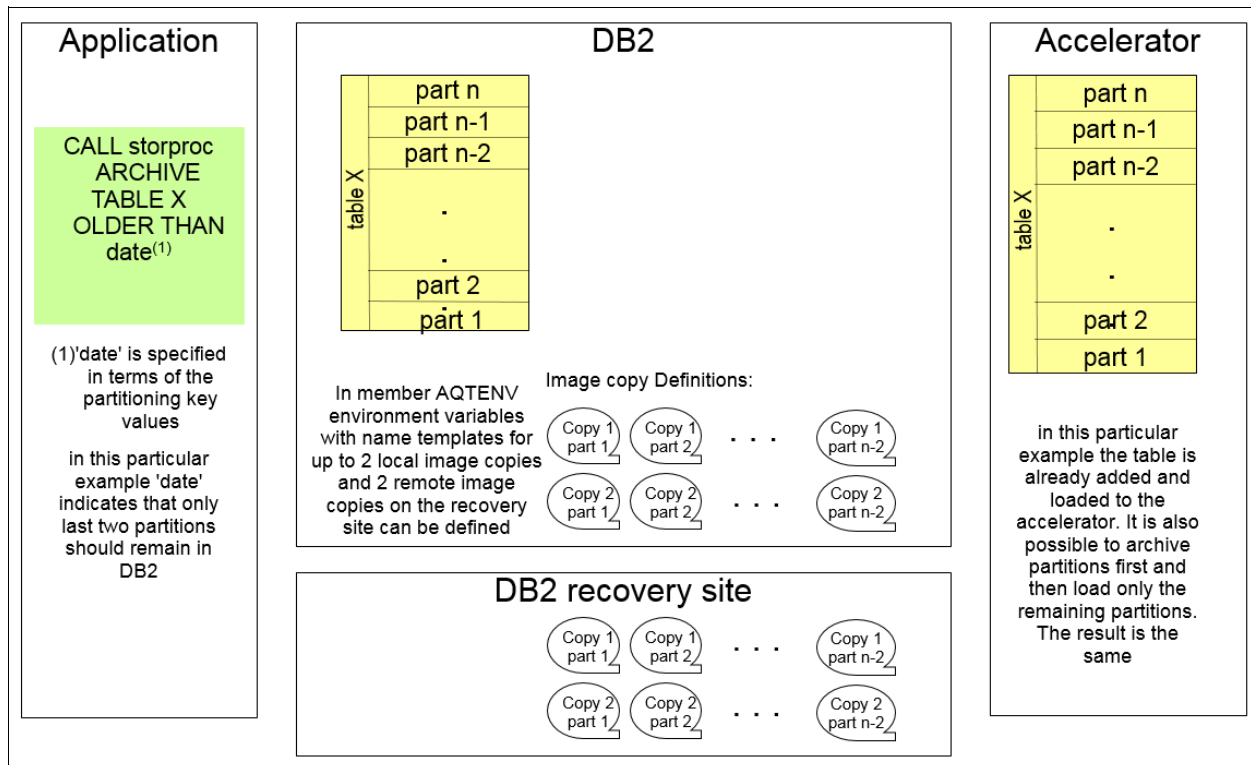


Figure 10-2 Initial view

As seen in Figure 10-2, the ACCEL_ARCHIVE_TABLES stored procedure creates up to four image copies for both the local and recovery sites. The requested number of copies are defined in the AQTENV member in the installation library:

- ▶ New AQTENV environment variables to allow up to four image copies:
AQT_ARCHIVE_COPY1, AQT_ARCHIVE_COPY2,
AQT_ARCHIVE_RECOVERYCOPY1, AQT_ARCHIVE_RECOVERYCOPY2
- ▶ Each value is a template specification as in the DB2 TEMPLATE utility, for example,
AQT_ARCHIVE_COPY1=&USERID..&DB..&TS..P&PART..&UNIQ.

The following rules apply to the values:

- ▶ Variables can be used as documented for the DB2 COPY utility. Variables &SEQ, &LIST, &DSNUM are not valid for the IBM DB2 Analytics Accelerator context.
- ▶ The values must evaluate to qualifiers that are mapped with DFSMS to a suitable data class.
- ▶ The values must ensure uniqueness of names among all archived partitions.
Recommendation: use variables &PART and &UNIQ.
- ▶ The values must evaluate to valid z/OS data set names

Persistent read-only (PRO) restricted status is set for archived partitions as seen in Figure 10-3 on page 143:

- ▶ PRO status:
 - Read access is allowed; updates are prohibited
 - Only the archived table space partitions are set
 - UTS is required
 - STOP DB and START DB commands do not affect the PRO status

- ▶ DB2 issues SQLCODE -904 with a new resource unavailable reason code, RC00C90635, when a SQL INSERT is attempted on a table space partition in PRO restricted status.
- ▶ SQL update or delete operations on a table space partition in PRO restricted status has the same behavior as though the partition was started for RO (read-only) access.

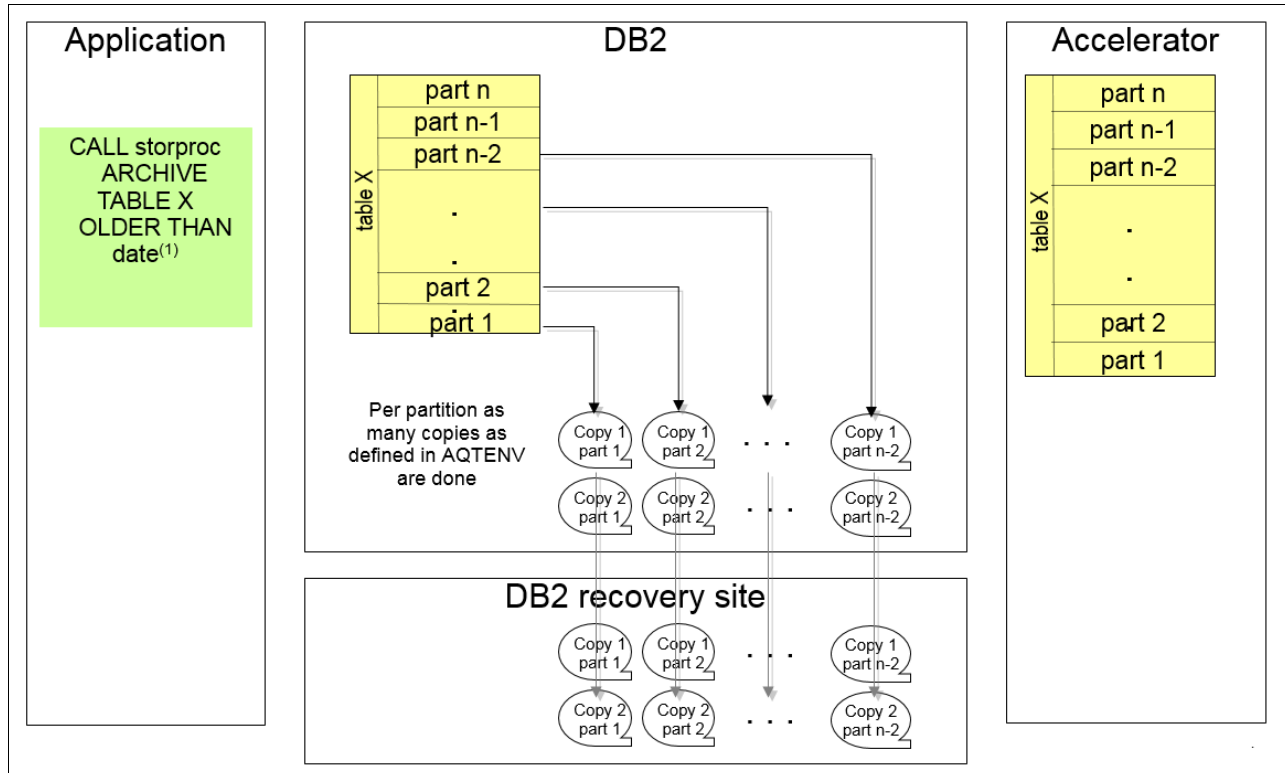


Figure 10-3 Image copy create step

The persistent read-only state is turned on and off by the REPAIR utility (DB2 10 and 11 New Function Mode):

```
REPAIR SET TABLESPACE ... PART n PRO
REPAIR SET TABLESPACE ... PART n NOPRO
```

The DIS DB command shows PRO status:

- ▶ DIS DB
- ▶ DIS DB(*) SP(*)
- ▶ DIS DB(*) SP(*) RESTRICT
- ▶ DIS DB(*) SP(*) RESTRICT(PRO)

Figure 10-4 on page 144 shows that the old partitions are deleted and the table is split.

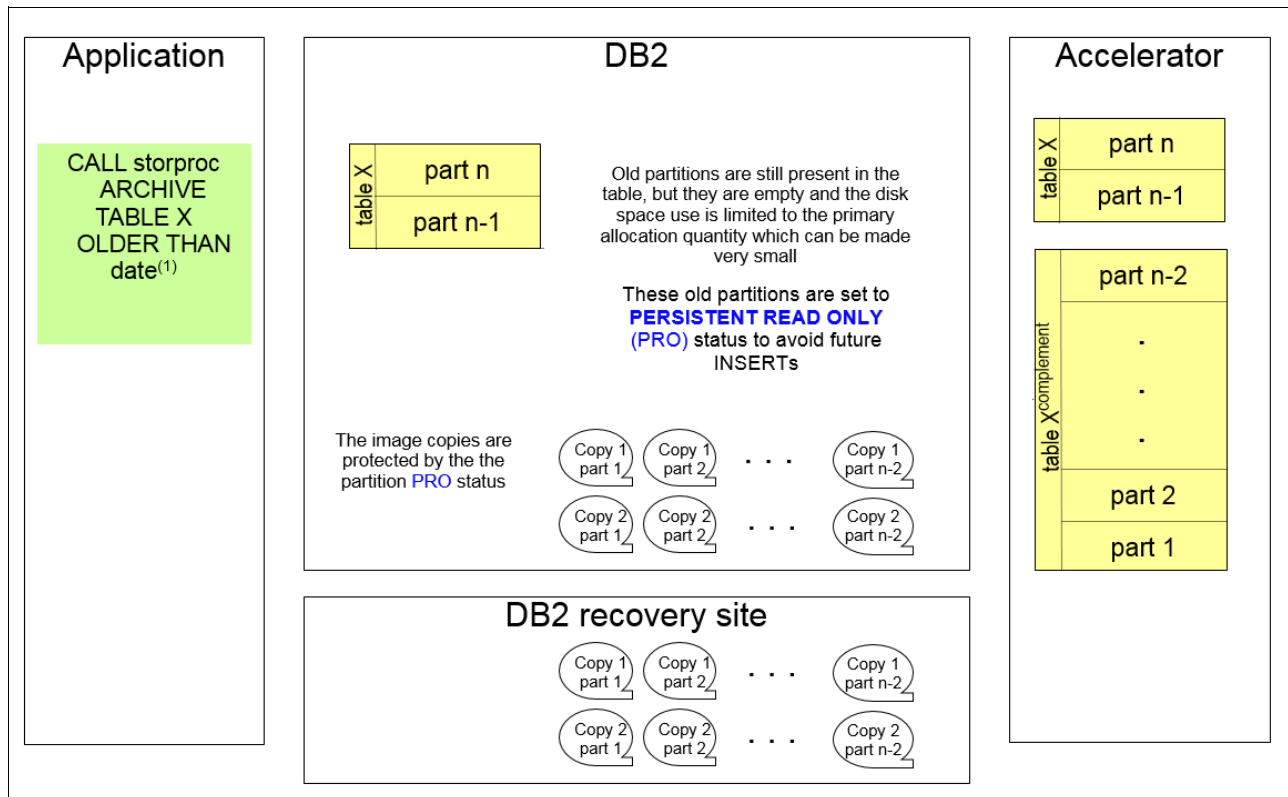


Figure 10-4 Old partitions deleted; table is split

Access to archived data is controlled by the system parameter GET_ACCEL_ARCHIVE special register:

- NO (default)
 - Specifies when a table is archived in the accelerator
 - A table reference does not include the archived data
- YES
 - Specifies when a table is archived in the accelerator
 - A table reference includes the archived data
 - Query can run *only* in the accelerator

Figure 10-5 on page 145 shows applications have transparent access (no SQL statement changes needed) to the table.

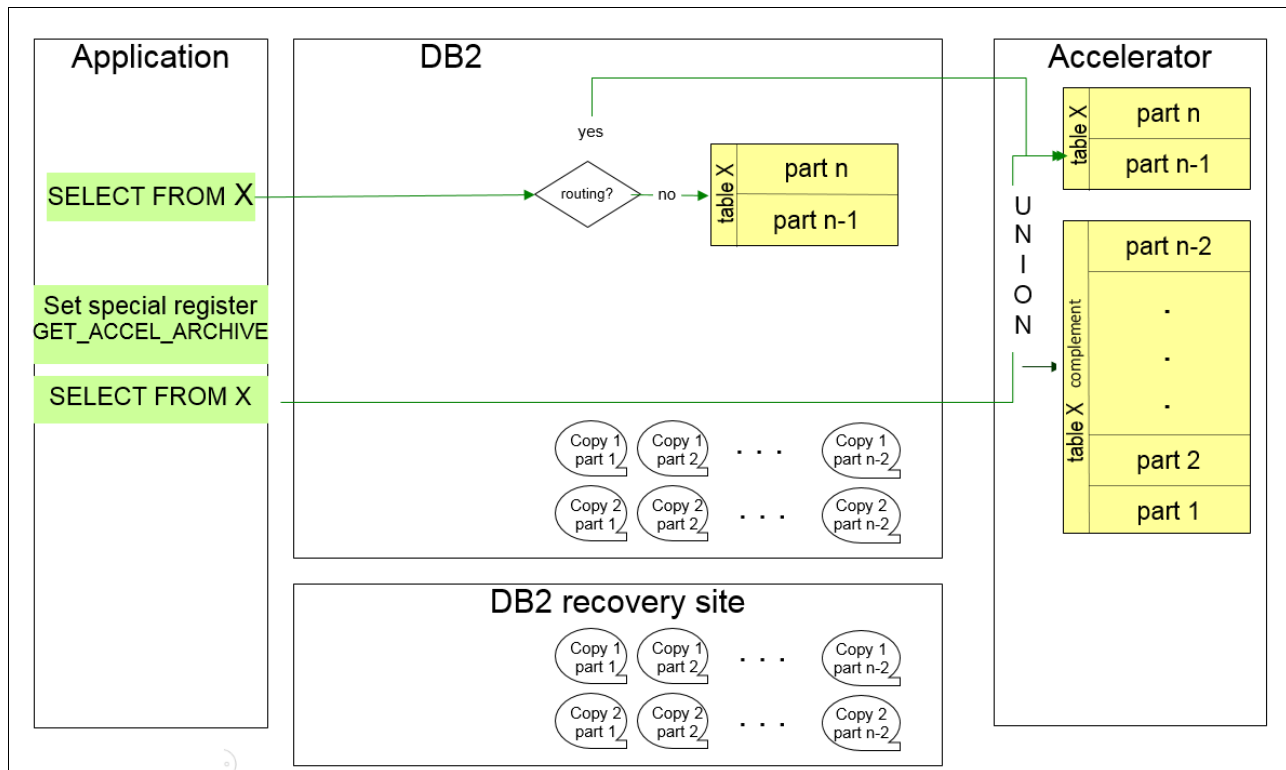


Figure 10-5 Applications have transparent access (no SQL statement changes needed) to the table

As previously stated, a dynamic query cannot be accelerated without setting the `SET CURRENT QUERY ACCELERATION` special register to a value other than `NONE`, and the `GET_ACCEL_ARCHIVE` special register determines whether data that was moved with High Performance Storage Saver is included in a query.

If a query can neither be executed in DB2 for z/OS or on an accelerator, the following error is issued:

```
4742 THE STATEMENT CANNOT BE EXECUTED BY DB2 OR IN THE ACCELERATOR (reason-code
<code>)
```

A query that would be executed in DB2 for z/OS according to the special register or parameter settings fails with `SQLCODE -4742` if the query references data that can only be found on the accelerator because it has been moved.

The statement cannot be executed by DB2 and receives `SQLCODE -4742` if any of the following conditions are true:

- ▶ When the special register `CURRENT GET_ACCEL_ARCHIVE` is set to `YES`, the statement needs to access data that is stored only in the accelerator server.
- ▶ When the special register `CURRENT QUERY ACCELERATION` is set to `ALL`, the statement must be executed in the accelerator server.
- ▶ A function that was referenced can be run only on an accelerator server. For example, the `MEDIAN` function can be run only on an accelerator server.

The statement cannot be executed by an accelerator server if any condition that is indicated by a *reason-code* is true. A reason code is a numeric value that provides additional information about why the statement cannot be executed in the accelerator server.

10.3.3 Restoring an archived partition from the accelerator

The restore operations are a set of administrative manual steps that are necessary in restoring data, rebuilding an index, and support structures in DB2. It restores one or more archived partitions from the accelerator back into DB2. Historical and archived data typically will not be updated and most likely will not be targeted for query processing. Therefore, they are being backed up and stored separately from the more recent, active operational data.

Restoring a moved partition data might become necessary if you must add or update data in the original DB2 partition or if an error in IBM DB2 Analytics Accelerator for z/OS has led to the loss of moved data. On the accelerator, the archived data is directly moved back into the shadow table for the non-archived data and the corresponding catalog information is updated.

The DB2 Analytics Accelerator provides a full-scale, built-in restore function for High Performance Storage Saver data. The restore operation uses the `SYSPROC.ACCEL_RESTORE_ARCHIVE_TABLES` stored procedure.

`SYSPROC.ACCEL_RESTORE_ARCHIVE_TABLES` does the following actions:

- ▶ Invokes `RECOVER TO LASTCOPY` to recover data from image copies created in the High Performance Storage Saver archive process. If that image copy is damaged, the procedure automatically uses the next image copy in backwards chronological order. The utility works on a partition level. Several threads can run in parallel if multiple partitions are specified. A sample job can be found in `SAQTSAMP(AQTSJI04)`.
- ▶ Invokes `REBUILD INDEX (Partitioned and NPIs)`. The utility is run once per table.
- ▶ Invokes `CHECK DATA` for a consistency check. This is started automatically for partitions that are in a check-pending state after a restore operation.
- ▶ A moved partition is not deleted from the accelerator, but is kept as a regular partition for query acceleration and the table can be used for regular accelerated queries.
- ▶ All image copies and entries in the DB2 catalog table `SYSIBM.SYSCOPY` are kept.

One or more explicitly specified partitions or an entire table can be restored. These partitions need to be defined on a DB2 Analytics Accelerator, and the DB2 Analytics Accelerator has to be active (in online status). If the DB2 Analytics Accelerator is unavailable, or if the table had been removed from the DB2 Analytics Accelerator, you can only recover the DB2 data manually from the DB2 copies. If an error occurs during the restore process, indexes are not rebuilt. They are put into a rebuild pending state. This might cause performance degradation on queries using these indexes and cause the static SQL queries to fail due to unavailable resources. Tables are recovered serially, partitions in parallel just like the archiving process.

If an operation fails on a single partition, only the failing partition is affected. All changes to previously processed partitions are kept and not rolled back. For those partitions that failed during the restore process, you must call the `SYSPROC.ACCEL_RESTORE_ARCHIVE_TABLES` procedure again, after fixing detected problems, to resume processing.



Case study: Using the DB2 Analytics Accelerator

In this case study, we demonstrate how an existing application can be enabled for acceleration and then how we can move (archive) some of the partitions to the DB2 Analytics Accelerator by using the High Performance Storage Saver function.

This chapter contains the following sections:

- ▶ System setup of application with an accelerator
- ▶ Moving and storing archived data with an accelerator
- ▶ Querying archived data

11.1 System setup of application with an accelerator

There is a table called Table_A with three partitions, partitioned by Date as shown in Figure 11-1. Partitions 1 and 2 contain read-only accessed data and partition 3 contains read/write accessed data. This table is added to the accelerator to obtain performance benefits.

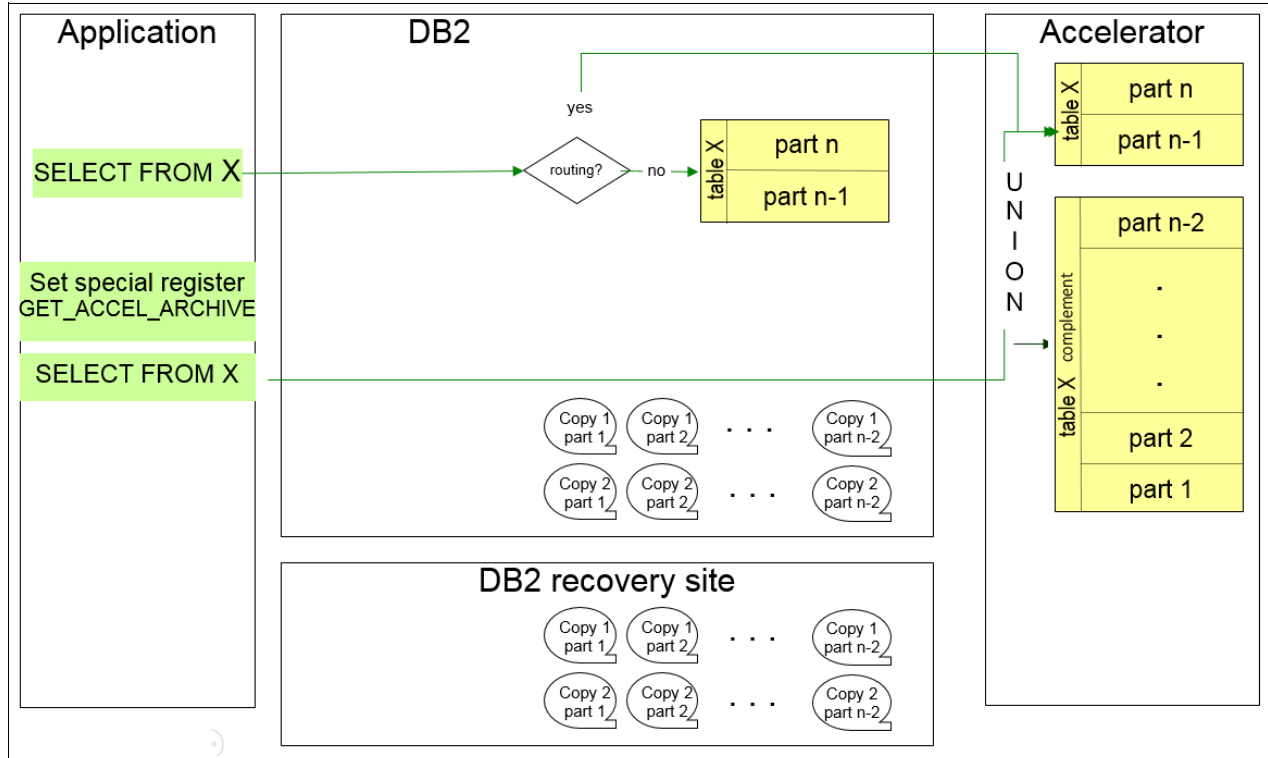


Figure 11-1 Final view of the accelerator-archived table used in this case study

Here is the Data Definition Language (DDL) and sample data of Table_A as seen in Example 11-1 and Figure 11-2 on page 149.

Example 11-1 Sample DDL and test data

```
CREATE TABLESPACE "PHONE"
  IN PHONE
  USING STOGROUP CDCSG
    PRIQTY -1
    SECQTY -1
  DSSIZE 4 G
  NUMPARTS 3
  SEGSIZE 32
  BUFFERPOOL BPO
  CCSID EBCDIC
  LOCKMAX SYSTEM
  LOCKSIZE ANY
  MAXROWS 255;

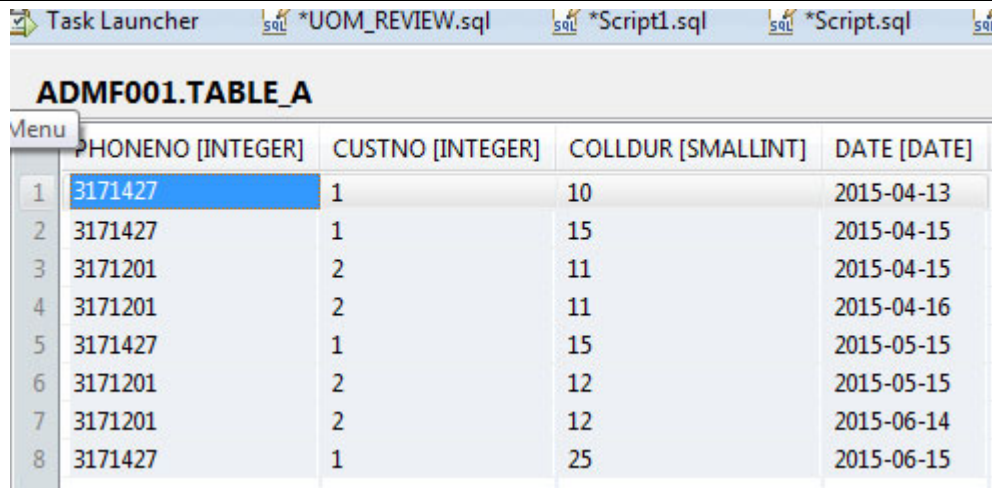
CREATE TABLE "ADMFO01"."TABLE_A" (
  "PHONENO" INTEGER NOT NULL WITH DEFAULT,
  "CUSTNO" INTEGER NOT NULL WITH DEFAULT,
```

```

"COLLDUR" SMALLINT NOT NULL WITH DEFAULT,
"DATE" DATE NOT NULL WITH DEFAULT
)
IN "PHONE"."PHONE"
PARTITION BY (DATE)
(PARTITION 1 ENDING AT ('2015-04-30'),
PARTITION 2 ENDING AT ('2015-05-31'),
PARTITION 3 ENDING AT ('2015-06-30'))
AUDIT NONE
DATA CAPTURE CHANGES
CCSID EBCDIC;

INSERT INTO ADMF001.TABLE_A VALUES(3171427, 1, 10, '2015-04-13') ;
INSERT INTO ADMF001.TABLE_A VALUES(3171427, 1, 15, '2015-04-15') ;
INSERT INTO ADMF001.TABLE_A VALUES(3171427, 1, 15, '2015-05-15') ;
INSERT INTO ADMF001.TABLE_A VALUES(3171201, 2, 12, '2015-05-15') ;
INSERT INTO ADMF001.TABLE_A VALUES(3171201, 2, 11, '2015-04-15') ;
INSERT INTO ADMF001.TABLE_A VALUES(3171201, 2, 11, '2015-04-16') ;
INSERT INTO ADMF001.TABLE_A VALUES(3171201, 2, 12, '2015-06-14') ;
INSERT INTO ADMF001.TABLE_A VALUES(3171427, 1, 25, '2015-06-15') ;

```



	PHONENO [INTEGER]	CUSTNO [INTEGER]	COLLDUR [SMALLINT]	DATE [DATE]
1	3171427	1	10	2015-04-13
2	3171427	1	15	2015-04-15
3	3171201	2	11	2015-04-15
4	3171201	2	11	2015-04-16
5	3171427	1	15	2015-05-15
6	3171201	2	12	2015-05-15
7	3171201	2	12	2015-06-14
8	3171427	1	25	2015-06-15

Figure 11-2 Sample Data in Table_A

Table_A is added to the accelerator by using the Data Studio Accelerator Management Interface as seen in Figure 11-3.

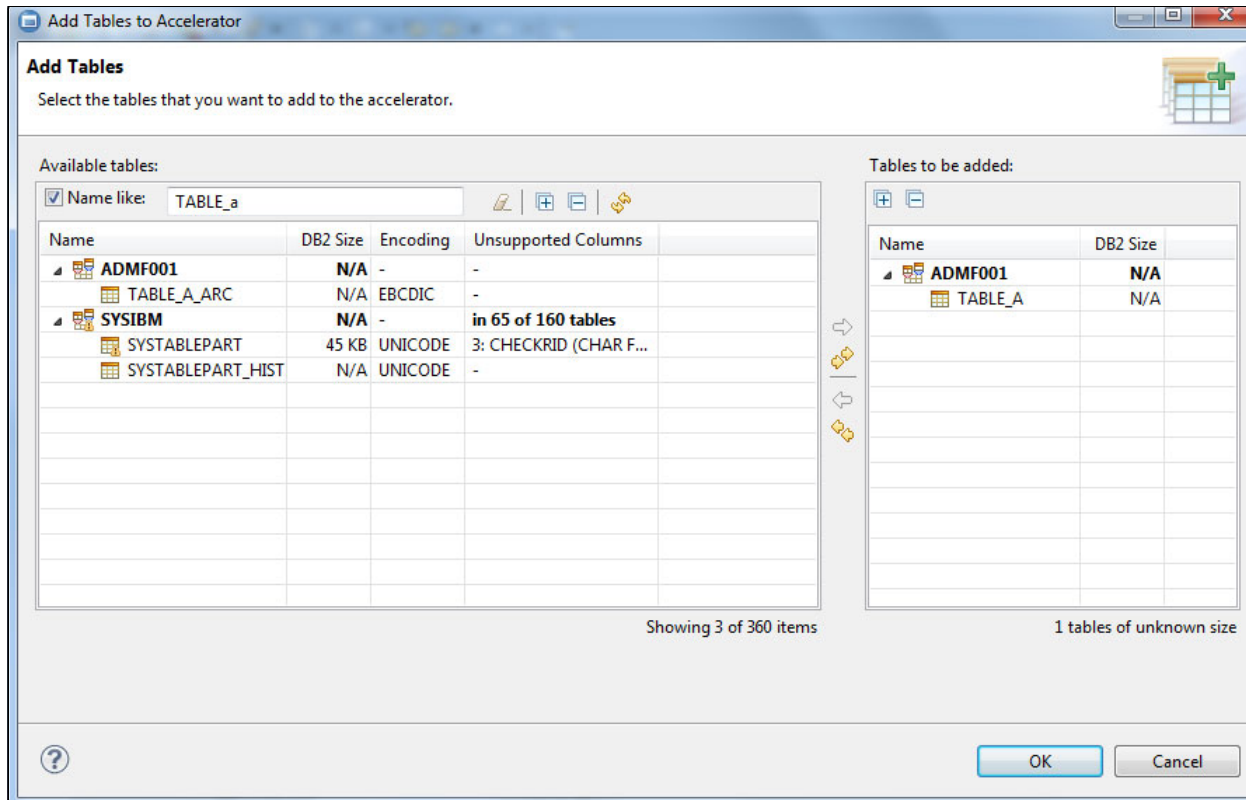


Figure 11-3 Adding Table_A to the accelerator

After Table_A is added to the accelerator, its recent status is set to “Initial Load Pending”. Then, the table is loaded with the data from DB2 for z/OS. This results in both DB2 and the accelerator having the same contents for the data, and the table is considered an accelerator-shadow table. All the data in all of the partitions has been copied to the accelerator as seen in Figure 11-4 on page 151. For this study, we have all partitions of Table_A to the accelerator.

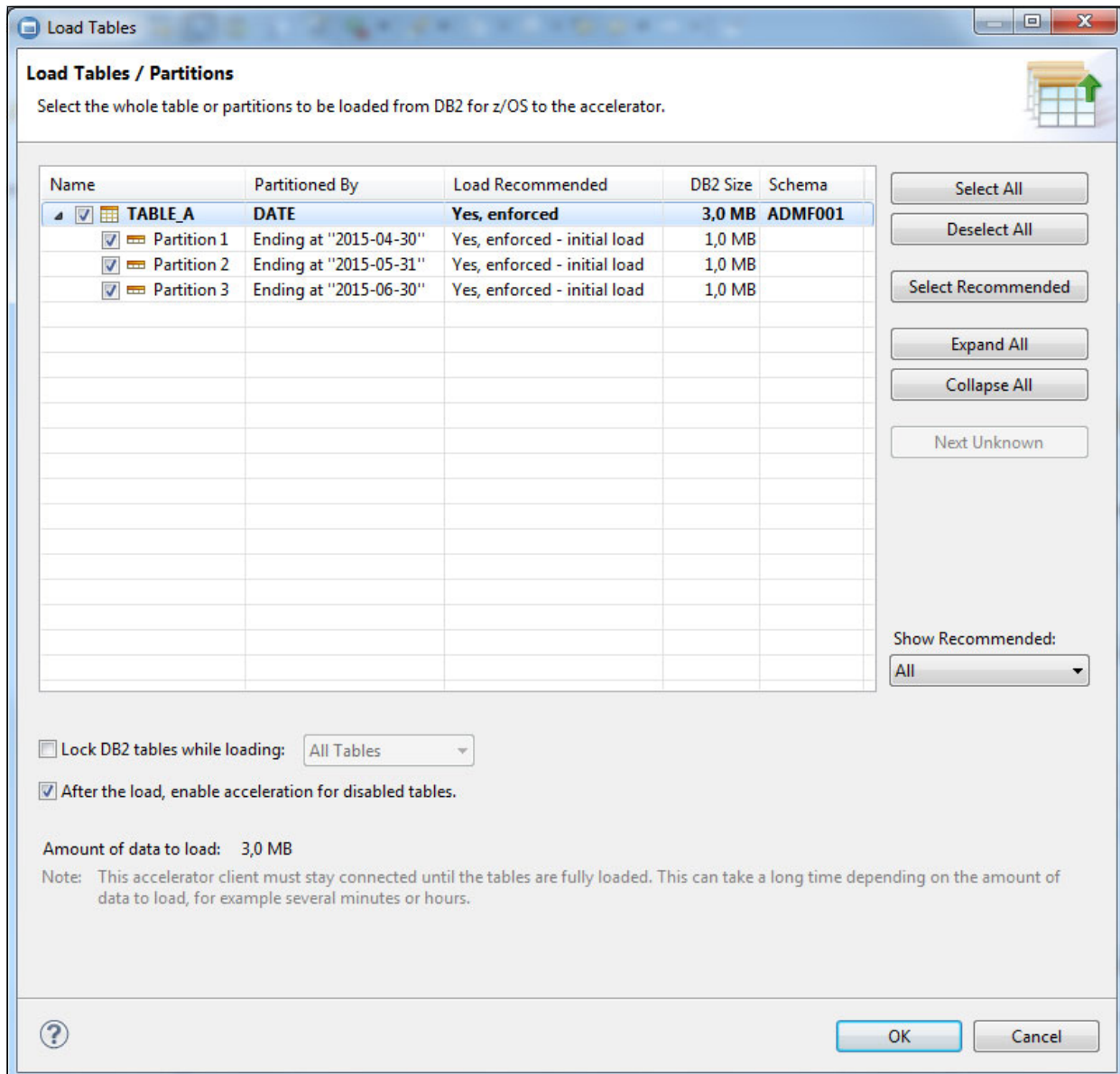


Figure 11-4 Loading partitions

From now on, the data for this table is in both DB2 and the accelerator. Depending on the complexity of a SELECT statement, DB2 determines whether to execute the query in DB2 or route it to accelerator.

11.2 Moving and storing archived data with an accelerator

Let's say, you wanted to archive the data in partitions 1 and 2 to the accelerator only, and erase that data from DB2 so that you'll end up with only one copy of the archived data from partitions 1 and 2 in the accelerator. This saves DASD storage and all queries that require the archived data must execute in the accelerator, and will get the benefit of acceleration. At this point, the table is considered an accelerator-archived table.

The archive process, which is a stored procedure, copies data to the accelerator, backs up the moved partitions, deletes the data in DB2, and puts those partitions in PRO state. So that nobody can access the table in DB2, the image copies are removed from SYSCOPY. As you see in Figure 11-5, Table_A is selected for Storage Saver Partitions to the accelerator, which uses the High Performance Storage Saver function.

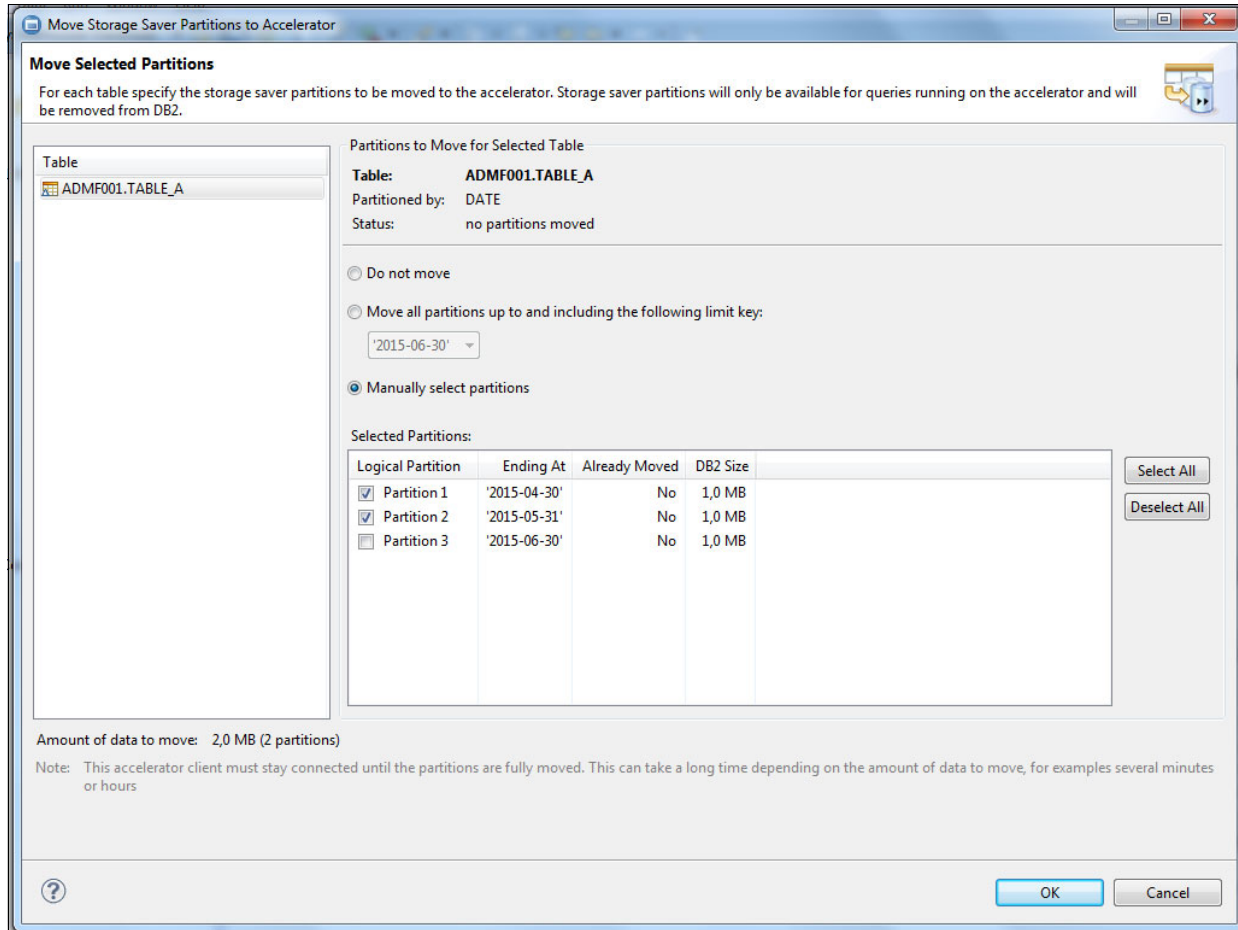


Figure 11-5 Moving partitions to the accelerator

When the archive process is complete, the status of partitions is changed to PRO status (Example 11-2).

Example 11-2 Database status after the archive

NAME	TYPE	PART	STATUS
PHONE	TS	0001	RW
-THRU		0003	
PHONE	TS		
PHONEH	TS	0001	RW, PRO
-THRU		0002	
PHONEH	TS	0003	RW
PHONEH	TS		

11.3 Querying archived data

Now we have partition 3 of Table_A both in DB2 and in the accelerator, which is already enabled for acceleration. Partitions 1 and 2 of Table_A have been archived to the accelerator and removed from DB2. So the data from partitions 1 and 2 exists only in the accelerator.

Based on these facts, we tested some access combinations. In Example 11-3, the CURRENT QUERY ACCELERATION special register is set to ALL to execute queries in the accelerator if they are eligible for acceleration. The CURRENT GET_ACCEL_ARCHIVE special register is set to YES to include data archived on an accelerator server when a query references an accelerator table. In addition:

- Query is routed to the accelerator.
- Archived data from the accelerator-archived table is accessed.

The result contains the full set of data from the accelerator-archived table (both current and archived data), which is all retrieved from the accelerator.

Example 11-3 Access an accelerator-archived table

```
SET CURRENT QUERY ACCELERATION = ALL ;
SET CURRENT GET_ACCEL_ARCHIVE = YES ;
SELECT * FROM ADMF001.TABLE_A FOR READ ONLY ;
```

PHONENO	CUSTNO	COLLDUR	DATE
3171427	1	15	2015-04-15
3171201	2	11	2015-04-16
3171427	1	15	2015-05-15
3171201	2	12	2015-06-14
3171427	1	25	2015-06-15
3171427	1	10	2015-04-13
3171201	2	11	2015-04-15
3171201	2	12	2015-05-15

```
DSNE610I NUMBER OF ROWS DISPLAYED IS 8
DSNE616I STATEMENT EXECUTION WAS SUCCESSFUL, SQLCODE IS 100
```

In Example 11-4, the CURRENT QUERY ACCELERATION special register is set to ALL to execute queries in the accelerator if they are eligible for acceleration. The CURRENT GET_ACCEL_ARCHIVE special register is set to NO so that data archived on an accelerator server will not be accessed when a query references an accelerator table. In addition:

- ▶ Query is routed to the accelerator.
- ▶ Archived data from the accelerator-archived table is not accessed.

The query is executed in the accelerator. Because the special register GET_ACCEL_ARCHIVE is set to NO, only active rows from Table_A are retrieved. The result contains only current rows from the accelerator-archived table.

Example 11-4 Access an accelerator-archived table

```

SET CURRENT QUERY ACCELERATION = ALL ;
SET CURRENT GET_ACCEL_ARCHIVE = NO ;
SELECT * FROM ADMF001.TABLE_A FOR READ ONLY ;
-----+-----+-----+-----+-----+-----+-----+
      PHONENO      CUSTNO  COLLDUR  DATE
-----+-----+-----+-----+-----+-----+-----+
      3171201           2       12  2015-06-14
      3171427           1       25  2015-06-15
DSNE610I NUMBER OF ROWS DISPLAYED IS 2
DSNE616I STATEMENT EXECUTION WAS SUCCESSFUL, SQLCODE IS 100

```



Part 4

Creating an integrated solution

Part 4 contains the following chapters:

- ▶ Chapter 12, “Case study: Combining DB2 temporal data management with the DB2 Analytics Accelerator” on page 157
- ▶ Chapter 13, “Case study: Combining archive transparency with the DB2 Analytics Accelerator ” on page 171



Case study: Combining DB2 temporal data management with the DB2 Analytics Accelerator

This chapter addresses combining application-period temporal data management with IBM DB2 Analytics Accelerator.

This chapter contains the following sections:

- ▶ Using the DB2 Analytics Accelerator with application-period temporal tables
- ▶ Using the DB2 Analytics Accelerator with system-period temporal tables

12.1 Using the DB2 Analytics Accelerator with application-period temporal tables

You can add an application-period temporal to the DB2 Analytics Accelerator as an accelerator-shadow table. A temporal query against the application-period temporal table can then be offloaded to the DB2 Analytics Accelerator.

The benefits of the DB2 Analytics Accelerator can be leveraged by application-period temporal tables to reduce:

- ▶ CPU time
- ▶ Elapsed time
- ▶ Storage

12.1.1 Step-by-step scenario

This scenario consists of the following steps:

1. Assume that customer A has an application-period temporal table ATT. The table is partitioned by the bus_end column, and contains three partitions.
2. Create an accelerator-shadow table in the DB2 Analytics Accelerator for ATT.
3. Load data into the accelerator-shadow table on DB2 Analytics Accelerator.
4. A temporal query for table ATT can be run on the accelerator. Based on the value of the bind option QUERYACCELERATION, or the special register CURRENT QUERY ACCELERATION, there are several behaviors as follows:
 - **NONE:** The query is executed on DB2.
 - **ENABLE:** The query is accelerated only if DB2 determines that it is advantageous to do so. If an accelerator failure occurs while the query is running or if the accelerator returns an error, DB2 returns an error to the application.
 - **ENABLE WITH FAILBACK:** The query is accelerated only if DB2 determines that it is advantageous to do so. If the accelerator returns an error during the PREPARE or first OPEN for the query, DB2 executes the query without the accelerator. If the accelerator returns an error during a FETCH or a subsequent OPEN, DB2 returns an error to the user and does not execute the query.
 - **ELIGIBLE:** The query is accelerated if they are eligible for acceleration. DB2 does not use cost information to determine whether to accelerate the queries. Queries that are not eligible for acceleration are executed by DB2. If an accelerator failure occurs while a query is running or if the accelerator returns an error, DB2 returns an error to the application.
 - **ALL:** The query is accelerated if they are eligible for acceleration. DB2 does not use cost information to determine whether to accelerate the queries. Queries that are not eligible for acceleration are not executed by DB2, and an SQL error is returned. If an accelerator failure occurs while a query is running or if the accelerator returns an error, DB2 returns an error to the application.
5. Issue some data change statements for table ATT.

6. Assume the data in partition 1 is out of date and will not be changed in the future. This is a case where we can archive partition 1 to the DB2 Analytics Accelerator.
7. A temporal query can access archived data on the accelerator for table ATT. Based on the value of the bind option GETACCELARCHIVE, or the special register CURRENT GET_ACCEL_ARCHIVE, there are several different behaviors:
 - **NO**: Specifies that if a table is archived on an accelerator server, and a query references that table, the query does not use the data that is archived.
 - **YES**: Specifies that if a table is archived on an accelerator server, and a query references that table, the query uses the data that is archived.

12.1.2 Define an application-period temporal table

The definition of an application-period temporal table is shown in Example 12-1.

Example 12-1 Define an application-period temporal table

```

CREATE DATABASE DBF00101
      STOGROUP SGF00101
      CCSID UNICODE;

CREATE TABLESPACE TUF001A1 IN DBF00101
      USING STOGROUP SGF00101
      Numparts 3
      Segsize 8
      DSSize 1G
      LOGGED
      Freepage 40
      Pctfree 40
      Bufferpool BP8K1
      Locksize ROW
      Lockmax 0
      Compress NO
      Define YES
      Close NO;

CREATE TABLE Policy
      (Bk VARCHAR(4) NOT NULL,
      Eff_Beg DATE NOT NULL,
      Eff_End DATE NOT NULL,
      Client CHAR(4) NOT NULL,
      Type CHAR(3),
      Copay CHAR(3),
      PERIOD BUSINESS_TIME (Eff_Beg, Eff_End),
      PRIMARY KEY (Bk, Client, BUSINESS_TIME WITHOUT OVERLAPS)
      )
PARTITION BY (Eff_End ASC)
      (PARTITION 1 ENDING AT ('1/1/2010') INCLUSIVE,
      PARTITION 2 ENDING ('1/1/2015') INCLUSIVE,
      PARTITION 3 ENDING ('12/31/9999') INCLUSIVE)
IN DBF00101.TUF001A1;

CREATE UNIQUE INDEX IX01_POLICY
      ON Policy
      (Bk, Client, BUSINESS_TIME WITHOUT OVERLAPS)

```

```

        USING STOGROUP SGF00101
        FREEPAGE      0
        PCTFREE       10
        BUFFERPOOL    BP16K0
        DEFINE        NO
        DEFER          NO
        COMPRESS       YES
        CLOSE         NO;

```

Populate data into table Policy.

```

INSERT INTO POLICY
VALUES ('P138','2/1/2004','12/31/9999','C882','PPO','$10');

```

```

INSERT INTO POLICY
VALUES ('P138','2/1/2004','12/31/9999','M990','PPO','$10');

```

```

INSERT INTO POLICY
VALUES ('P138','2/1/2004','12/31/9999','P001','PPO','$10');

```

Update data in table Policy.

```

UPDATE POLICY
FOR PORTION OF BUSINESS_TIME FROM '1/1/2006' TO '12/31/9999'
SET Copay = '$15'
WHERE Client = 'C882' AND
      Bk = 'P138';

```

```

UPDATE POLICY
FOR PORTION OF BUSINESS_TIME FROM '1/1/2015' TO '12/31/9999'
SET Type = 'HMO'
WHERE Client = 'C990' AND
      Bk = 'P138';

```

Delete rows from table Policy.

```

DELETE FROM POLICY
FOR PORTION OF BUSINESS_TIME FROM '1/1/2015' TO '12/31/9999'
WHERE Client = 'P001' AND
      Bk = 'P138';

```

12.1.3 Adding an application-period temporal table into DB2 Analytics Accelerator

In this step, we add an accelerator-shadow table for POLICY to the accelerator, and load data to the table (Figure 12-1 on page 161).

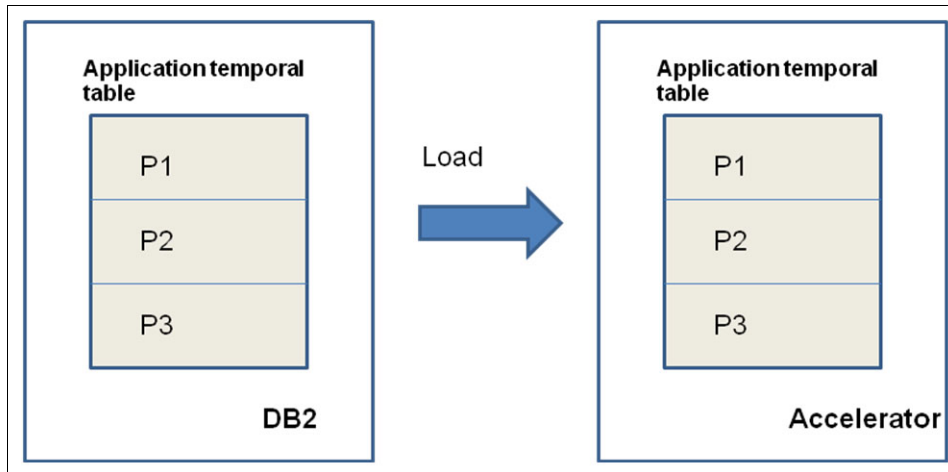


Figure 12-1 Adding tables to an accelerator and loading data

DB2 Analytics Accelerator stored procedures are the administration interface for your accelerators. When you invoke a function from IBM DB2 Analytics Accelerator Studio, the corresponding stored procedure is called. In addition, IBM DB2 Analytics Accelerator stored procedures can be run from the command line or embedded in custom applications. The stored procedures provide functions that are related to tables and accelerators.

The following stored procedures are provided as an interface to the DB2 Analytics Accelerator:

- ▶ **SYSPROC.ACCEL_ADD_TABLES**
Defines tables on an accelerator. The input consists of a list of existing DB2 for z/OS tables, which serve as the basis.
- ▶ **SYSPROC.ACCEL_LOAD_TABLES**
Loads data from the source tables in DB2 into the corresponding tables on an accelerator.

The IBM DB2 Analytics Accelerator Studio could also be used for these operations (Figure 12-2).

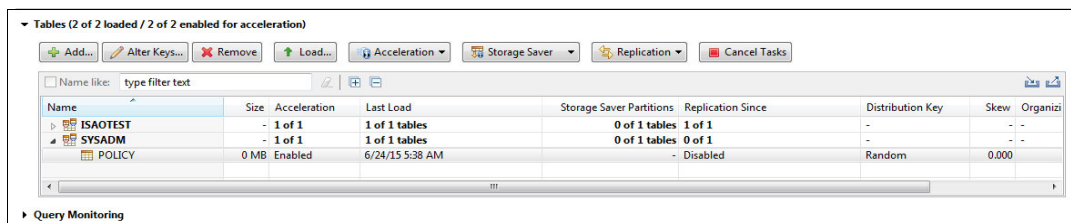


Figure 12-2 IBM DB2 Analytics Accelerator Studio GUI

After loading data to the accelerator-shadow table in the accelerator, the POLICY table on the accelerator has the same data as the POLICY table on DB2.

12.1.4 How are queries processed when data exists on an accelerator?

For the following query, if we set CURRENT QUERY ACCELERATION = NONE, it is executed on DB2 (Figure 12-3):

```
SELECT * FROM SYSADM.POLICY;
```

	BK	EFF_BEG	EFF_END	CLIENT	TYPE	COPAY
1	P138	2006-01-01	9999-12-31	C882	PPO	\$15
2	P138	2004-02-01	2006-01-01	C882	PPO	\$10
3	P138	2015-01-01	9999-12-31	M990	HMO	\$10
4	P138	2004-02-01	2015-01-01	M990	PPO	\$10
5	P138	2004-02-01	2015-01-01	P001	PPO	\$10

Figure 12-3 Results of the query

If we set CURRENT QUERY ACCELERATION = ALL, it is forced to execute on the DB2 Analytics Accelerator. Here are the results, which are the same as in Figure 12-3 because the data on DB2 Analytics Accelerator is the same as in the table on DB2 now (Figure 12-4).

	BK	EFF_BEG	EFF_END	CLIENT	TYPE	COPAY
1	P138	2006-01-01	9999-12-31	C882	PPO	\$15
2	P138	2004-02-01	2006-01-01	C882	PPO	\$10
3	P138	2015-01-01	9999-12-31	M990	HMO	\$10
4	P138	2004-02-01	2015-01-01	M990	PPO	\$10
5	P138	2004-02-01	2015-01-01	P001	PPO	\$10

Figure 12-4 Results of the query

A temporal query can be issued for the application-period temporal table (Figure 12-5):

```
SELECT * FROM SYSADM.POLICY  
FOR BUSINESS_TIME AS OF '1/1/2015';
```

	BK	EFF_BEG	EFF_END	CLIENT	TYPE	COPAY
1	P138	2006-01-01	9999-12-31	C882	PPO	\$15
2	P138	2015-01-01	9999-12-31	M990	HMO	\$10

Figure 12-5 Results of the query

12.1.5 How are data change statements processed?

Next, we insert a new row into table POLICY, and do some inserts and updates (Example 12-2).

Example 12-2 Inserting a new row

```
INSERT INTO POLICY  
VALUES ('P138','2/1/2004','12/31/9999','H001','PPO','$10');  
  
UPDATE POLICY  
FOR PORTION OF BUSINESS_TIME FROM '1/1/2015' TO '12/31/9999'  
SET Type = 'HMO'  
WHERE Client = 'H001' AND  
Bk = 'P138';
```

Then, we want to select data from table POLICY. If we set CURRENT QUERY ACCELERATION = NONE, the query is executed on DB2, and returns the following result (Figure 12-6):

```
SELECT * FROM SYSADM.POLICY ;
```

	BK	EFF_BEG	EFF_END	CLIENT	TYPE	COPAY
1	P138	2006-01-01	9999-12-31	C882	PPO	\$15
2	P138	2004-02-01	2006-01-01	C882	PPO	\$10
3	P138	2015-01-01	9999-12-31	M990	HMO	\$10
4	P138	2004-02-01	2015-01-01	M990	PPO	\$10
5	P138	2015-01-01	9999-12-31	H001	HMO	\$10
6	P138	2004-02-01	2015-01-01	H001	PPO	\$10
7	P138	2004-02-01	2015-01-01	P001	PPO	\$10

Figure 12-6 Results of the query processed on DB2

If we set CURRENT QUERY ACCELERATION = ALL, the query is executed on the DB2 Analytics Accelerator. In this case, the result does not include the data changes that were applied to the table DB2 (Figure 12-7):

```
SELECT * FROM SYSADM.POLICY ;
```

	BK	EFF_BEG	EFF_END	CLIENT	TYPE	COPAY
1	P138	2006-01-01	9999-12-31	C882	PPO	\$15
2	P138	2015-01-01	9999-12-31	M990	HMO	\$10
3	P138	2004-02-01	2015-01-01	M990	PPO	\$10
4	P138	2004-02-01	2006-01-01	C882	PPO	\$10
5	P138	2004-02-01	2015-01-01	P001	PPO	\$10

Figure 12-7 Results of the query processed on the DB2 Analytics Accelerator

If you want to synchronize the data changes on DB2 and DB2 Analytics Accelerator, there are two options:

- ▶ Reload the data on DB2 to the DB2 Analytics Accelerator.
- ▶ Use one of the stored procedures to enable or disable incremental updates for one or more tables on an accelerator.

Note the following stored procedure:

- ▶ SYSPROC.ACCEL_SET_TABLES_REPLICATION

You can start the update of tables with changed data when the tables have been loaded. Start incremental updates by using the **<startReplication>** command of the SYSPROC.ACCEL_CONTROL_ACCELERATOR stored procedure or the corresponding function in IBM DB2 Analytics Accelerator Studio.

12.1.6 Archiving parts of an application-period temporal table to the DB2 Analytics Accelerator

As the data grows, a table gets bigger and bigger. If there are some partitions in which the data is out of date, and will not be changed in the future, those partitions can be archived to the DB2 Analytics Accelerator. Note the following stored procedure:

- ▶ SYSPROC.ACCEL_ARCHIVE_TABLES

The SYSPROC.ACCEL_ARCHIVE_TABLES stored procedure shifts the data of one or more table partitions in DB2 to an accelerator. The purpose of this is to reduce the disk space consumption of tables that are not updated anymore (historical data) on the DB2 side. After the move to an accelerator, the data in the moved partitions must not be updated anymore.

Only tables partitioned by range can be archived in the accelerator. We could also perform these operations by using the IBM DB2 Analytics Accelerator Studio.

Assume that before archiving, there are seven rows in the table POLICY, and that the second row is in partition 1 (Figure 12-8):

```
SELECT * FROM SYSADM.POLICY ;
```

	BK	EFF_BEG	EFF_END	CLIENT	TYPE	COPAY
1	P138	2006-01-01	9999-12-31	C882	PPO	\$15
2	P138	2004-02-01	2006-01-01	C882	PPO	\$10
3	P138	2015-01-01	9999-12-31	M990	HMO	\$10
4	P138	2004-02-01	2015-01-01	M990	PPO	\$10
5	P138	2015-01-01	9999-12-31	H001	HMO	\$10
6	P138	2004-02-01	2015-01-01	H001	PPO	\$10
7	P138	2004-02-01	2015-01-01	P001	PPO	\$10

Figure 12-8 Results of the query before archiving

Assume that the data in partition 1 is out of date. We archive partition 1 to the DB2 Analytics Accelerator (as shown in Figure 12-9). After archiving, partition 1 is empty on DB2.

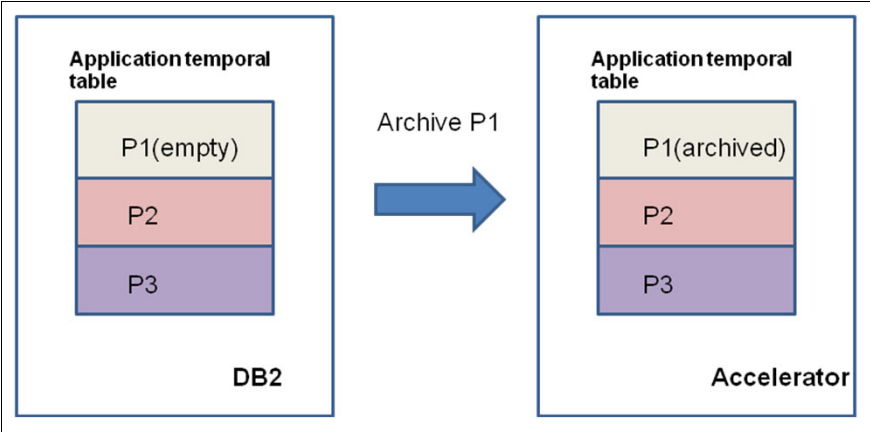


Figure 12-9 Archive partition 1 to an accelerator

The result of running the same query is now only those six rows that are stored on DB2 (Figure 12-10):

```
SELECT * FROM SYSADM.POLICY ;
```

	BK	EFF_BEG	EFF_END	CLIENT	TYPE	COPAY
1	P138	2004-02-01	2015-01-01	M990	PPO	\$10
2	P138	2004-02-01	2015-01-01	P001	PPO	\$10
3	P138	2004-02-01	2015-01-01	H001	PPO	\$10
4	P138	2006-01-01	9999-12-31	C882	PPO	\$15
5	P138	2015-01-01	9999-12-31	M990	HMO	\$10
6	P138	2015-01-01	9999-12-31	H001	HMO	\$10

Figure 12-10 Results of the query after archiving partition 1 to the accelerator

We still could insert, delete, and update data in partitions 2 and 3 (on DB2), but any data change operations on partition 1 (which exists only on the accelerator) are disallowed.

12.1.7 How are queries processed?

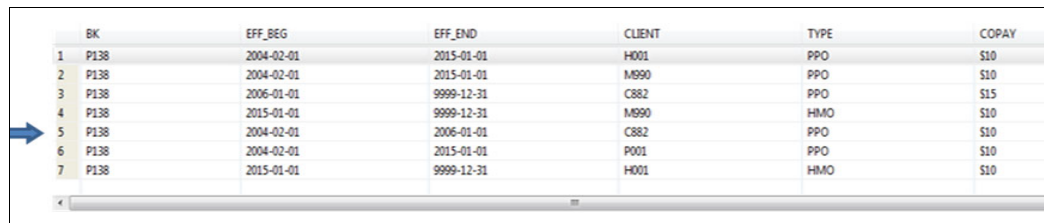
We can control whether to access archived data on an accelerator by setting the value of the bind option GETACCELARCHIVE, and the special register CURRENT GET_ACCEL_ARCHIVE.

- If we do not need to access the archived data, set the value to NO, and queries are executed the same as the queries that were previously demonstrated.
- If we want to query the archived data, set the value to YES. The query also considers the data that is archived (Example 12-3).

Example 12-3 Query the archive data in addition to the active data

```
SET CURRENT QUERY ACCELERATION = ALL;  
SET CURRENT GET_ACCEL_ARCHIVE = YES;  
SELECT * FROM SYSADM.POLICY ;
```

This time, we get seven rows in the results. This includes the rows in partition 1, which exist only on the DB2 Analytics Accelerator where they were archived (Figure 12-11).



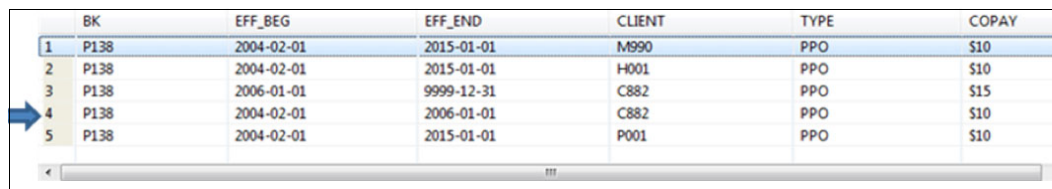
	BK	EFF_BEG	EFF_END	CLIENT	TYPE	COPAY
1	P138	2004-02-01	2015-01-01	H001	PPO	\$10
2	P138	2004-02-01	2015-01-01	M990	PPO	\$10
3	P138	2006-01-01	9999-12-31	C882	PPO	\$15
4	P138	2015-01-01	9999-12-31	M990	HMO	\$10
5	P138	2004-02-01	2006-01-01	C882	PPO	\$10
6	P138	2004-02-01	2015-01-01	P001	PPO	\$10
7	P138	2015-01-01	9999-12-31	H001	HMO	\$10

Figure 12-11 Results of the query that includes both active data and archived data

A temporal query can also be issued for the table, and archived data is considered:

```
SELECT * FROM SYSADM.POLICY  
FOR BUSINESS_TIME FROM '1/1/2005' TO '1/1/2009';
```

The row that is archived is selected as shown in Figure 12-12.



	BK	EFF_BEG	EFF_END	CLIENT	TYPE	COPAY
1	P138	2004-02-01	2015-01-01	M990	PPO	\$10
2	P138	2004-02-01	2015-01-01	H001	PPO	\$10
3	P138	2006-01-01	9999-12-31	C882	PPO	\$15
4	P138	2004-02-01	2006-01-01	C882	PPO	\$10
5	P138	2004-02-01	2015-01-01	P001	PPO	\$10

Figure 12-12 Results of the temporal query

Table 12-1 is the summary of how to execute a query on an application-period temporal table.

Table 12-1 Summary for querying an application-period temporal table that exists on an accelerator

Query	QUERYACCELERATION / CURRENT QUERY ACCELERATION	GETACCELARCHIVE /CURRENT GET_ACCEL_ARCHIVE	Where to execute?	Access archived data?
SELECT... FROM ATT;	NONE	NO	DB2	NO
SELECT... FROM ATT;	ALL	NO	Accelerator	NO

SELECT... FROM ATT;	NONE	YES	ERROR	
SELECT... FROM ATT;	ALL	YES	Accelerator	YES
SELECT... FROM ATT FOR BUSINESS_TIME AS OF ...;	NONE	NO	DB2	NO
SELECT... FROM ATT FOR BUSINESS_TIME AS OF ...;	ALL	NO	Accelerator	NO
SELECT... FROM ATT FOR BUSINESS_TIME AS OF ...;	ALL	YES	Accelerator	YES
SELECT... FROM ATT FOR BUSINESS_TIME AS OF ...;	NONE	YES	ERROR	

12.2 Using the DB2 Analytics Accelerator with system-period temporal tables

System-period temporal tables can be added or loaded to the DB2 Analytics Accelerator. A temporal query against the system-period temporal table can then be offloaded to the DB2 Analytics Accelerator.

The benefits of the DB2 Analytics Accelerator can be leveraged by system-period temporal tables to reduce these items:

- ▶ CPU time
- ▶ Elapsed time
- ▶ Storage

12.2.1 Step-by-step scenario

This scenario consists of the following steps:

1. Assume that customer A has a system-period temporal table STT, and an associated history table STT_HIST. Both tables are partitioned by the sys_end column, and contain three partitions.
2. Create the accelerator-shadow tables on the DB2 Analytics Accelerator for STT (system-period temporal table) and STT_HIST (history table).
3. Load data into the accelerator-shadow tables on the DB2 Analytics Accelerator (Figure 12-13 on page 167).

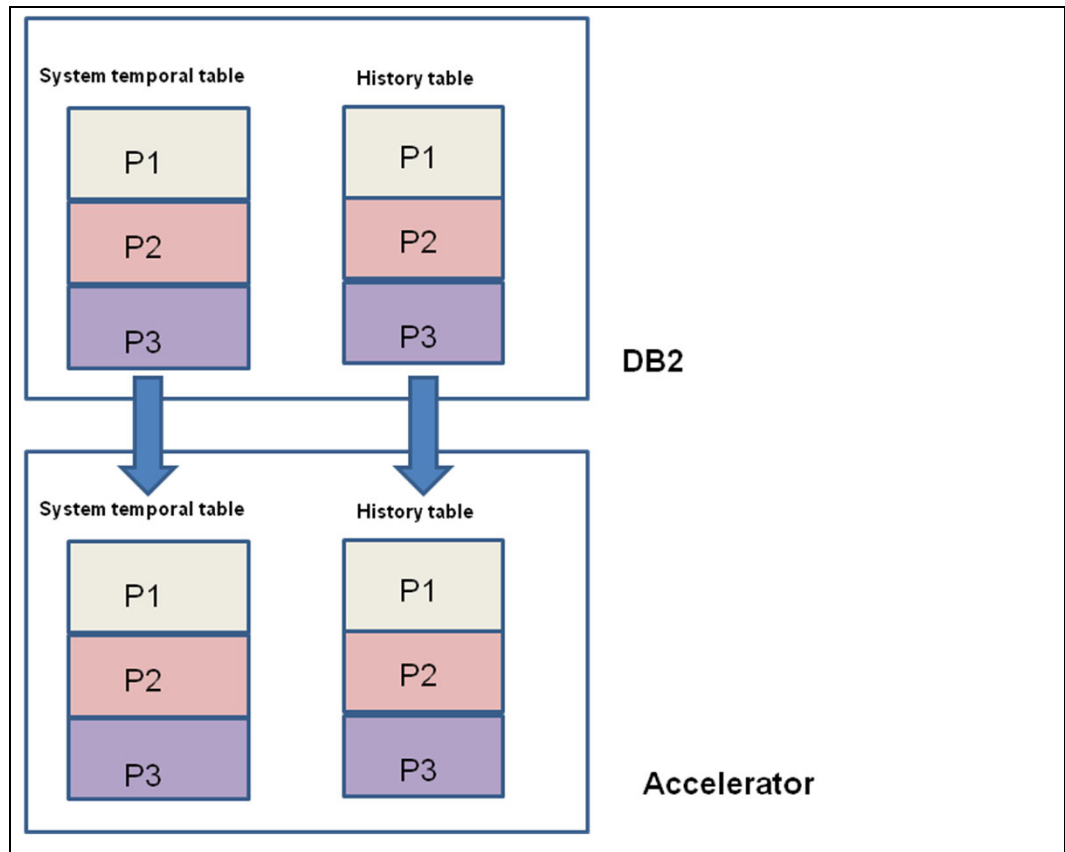


Figure 12-13 Adding tables to the accelerator and loading data

4. A temporal query for table STT can be run on an accelerator. Based on the value of the bind option QUERYACCELERATION or the special register CURRENT QUERY ACCELERATION, there are several behaviors as follows:
 - **NONE:** The query will be executed on DB2.
 - **ENABLE:** The query will be accelerated only if DB2 determines that doing so is advantageous. If an accelerator failure occurs while the query is running or if the accelerator returns an error, DB2 returns an error to the application.
 - **ENABLE WITH FAILBACK:** The query will be accelerated only if DB2 determines that doing so is advantageous. If the accelerator returns an error during the PREPARE or first OPEN for the query, DB2 executes the query without the accelerator. If the accelerator returns an error during a FETCH or a subsequent OPEN, DB2 returns an error to the user and does not execute the query.
 - **ELIGIBLE:** The query will be accelerated if it is eligible for acceleration. DB2 does not use cost information to determine whether to accelerate the queries. Queries that are not eligible for acceleration are executed by DB2. If an accelerator failure occurs while a query is running or if the accelerator returns an error, DB2 returns an error to the application.
 - **ALL:** The query will be accelerated if it is eligible for acceleration. DB2 does not use cost information to determine whether to accelerate the queries. Queries that are not eligible for acceleration are not executed by DB2, and an SQL error is returned. If an accelerator failure occurs while a query is running or if the accelerator returns an error, DB2 returns an error to the application.

5. Issue some data change statements for table STT. The historical data is moved to the history table STT_HIST.
6. The STT_HIST history table grows and grows in size. Assume that the data in partition 1 is out of date and will not be changed in the future. This is a case where we can archive partition 1 of the history table to the DB2 Analytics Accelerator (Figure 12-14).

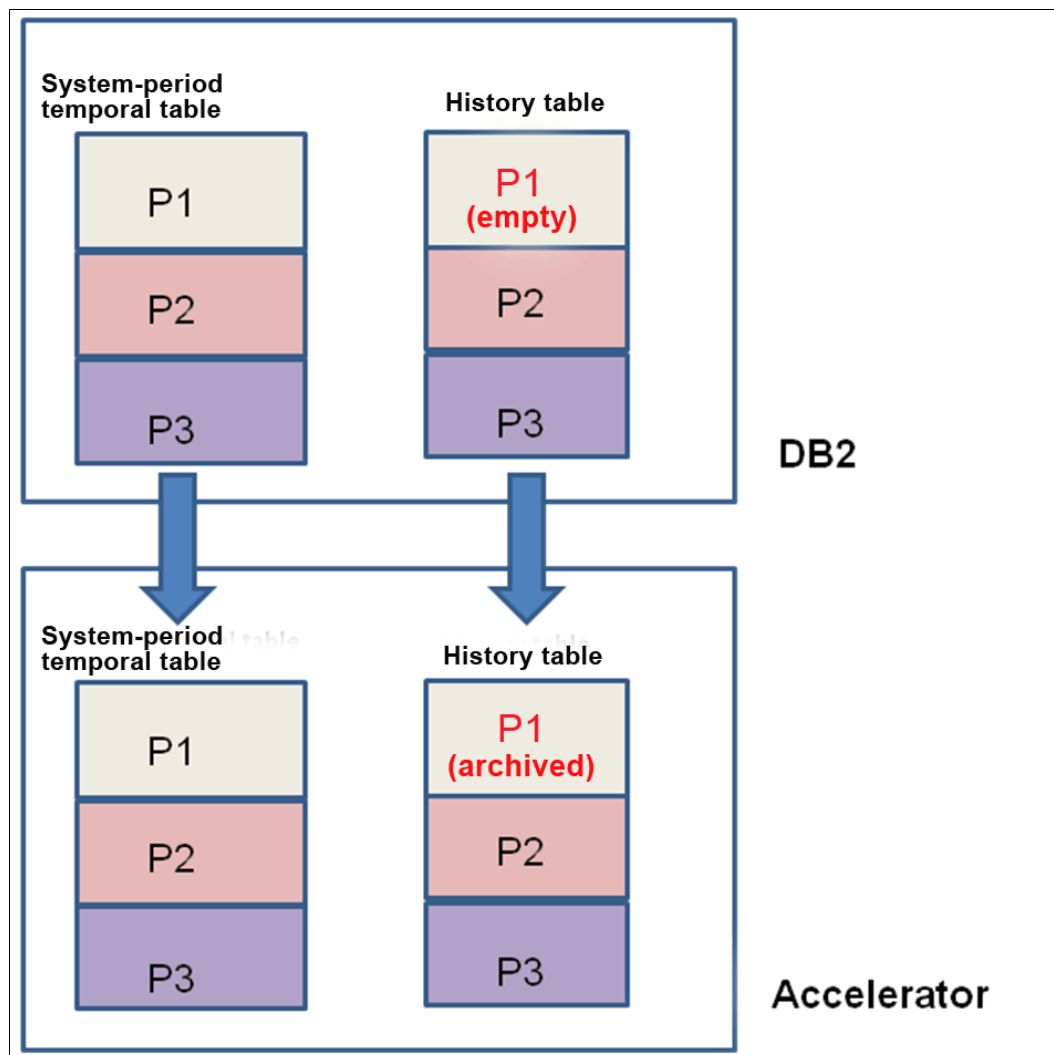


Figure 12-14 Archive partition 1 of the history table STT_HIST to an accelerator.

7. A temporal query can access archived data on the accelerator for the table STT. Based on the value of the bind option GETACCELARCHIVE, or the special register CURRENT GET_ACCEL_ARCHIVE, there are several behaviors as follows:
 - **NO:** Specifies that if a table is archived on an accelerator server, and a query references that table, the query does not use the data that is archived.
 - **YES:** Specifies that if a table is archived on an accelerator server, and a query references that table, the query uses the data that is archived.

12.2.2 How are queries processed when data exists on an accelerator?

Table 12-2 shows a summary for querying a system-period temporal table that exists on the accelerator.

Table 12-2 Summary for querying a system-period temporal table that exists on the accelerator

Query	QUERYACCELERATION / CURRENT QUERY ACCELERATION	GETACCELARCHIVE / CURRENT GET_ACCEL_ARCHIVE	Where to execute?	Access archived data?
SELECT... FROM STT;	NONE	NO	DB2	NO
SELECT... FROM STT;	ALL	NO	Accelerator	NO
SELECT... FROM STT;	NONE	YES	ERROR	
SELECT... FROM STT;	ALL	YES	Accelerator	NO. Because the archived data is for table STT_HIST
SELECT... FROM STT FOR SYSTEM_TIME AS OF...	NONE	NO	DB2	NO
SELECT... FROM STT FOR SYSTEM_TIME AS OF...	ALL	NO	Accelerator	NO
SELECT... FROM STT FOR SYSTEM_TIME AS OF...	ALL	YES	Accelerator	YES
SELECT... FROM STT FOR SYSTEM_TIME AS OF...	NONE	YES	ERROR	



Case study: Combining archive transparency with the DB2 Analytics Accelerator

In this chapter, we continue with the case study in Chapter 8, “Case study: Using archive transparency” on page 109 to exploit the features of the DB2 Analytics Accelerator related to archiving.

This chapter includes the following sections:

- ▶ Overview
- ▶ Enabling an archive-enabled table and archive table for acceleration
- ▶ Querying an accelerator-shadow table
- ▶ Moving archived partitions to the accelerator as an accelerator-archived table
- ▶ Querying both an accelerator-shadow table and an accelerator-archived table

13.1 Overview

In Chapter 8, “Case study: Using archive transparency” on page 109, we created an archive-enabled table and archive table with sample data. We practice how we can enable both of these tables for query acceleration and then move the archive table to the DB2 Analytics Accelerator with the High Performance Storage Saver feature (Figure 13-1).

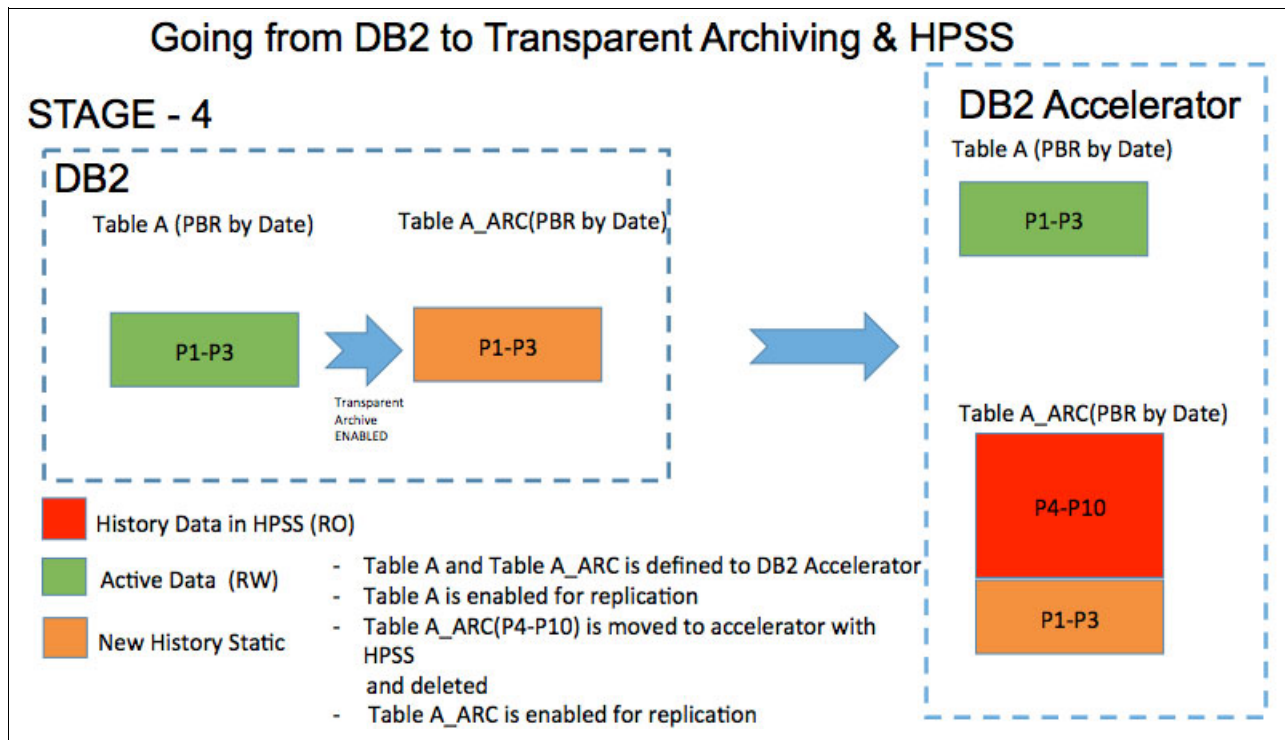


Figure 13-1 Working together: Transparent archiving and the DB2 Analytics Accelerator with High Performance Storage Saver

13.2 Enabling an archive-enabled table and archive table for acceleration

Figure 13-2 on page 173 shows the recent status of two tables (TABLE_A and TABLE_A_ARC) with sample data using IBM Data Studio. TABLE_A contains the active data and TABLE_A_ARC contains the archive data. TABLE_A is an archive-enabled table, and TABLE_A_ARC is an archive table. We'll add these two tables to the DB2 Analytics Accelerator in the next step.

ADMF001.TABLE_A

	PHONENO [INTEGER]	CUSTNO [INTEGER]	COLLDUR [SMALLINT]	DATE [DATE]
1	3171201	2	12	2015-06-14
2	3171427	1	25	2015-06-15

ADMF001.TABLE_A_ARC

	PHONENO [INTEGER]	CUSTNO [INTEGER]	CALLDUR [SMALLINT]	DATE [DATE]
1	3171427	1	10	2015-04-13
2	3171427	1	15	2015-04-15
3	3171201	2	11	2015-04-15
4	3171201	2	11	2015-04-16
5	3171427	1	15	2015-05-15
6	3171201	2	12	2015-05-15

Figure 13-2 Recent status of tables

When you start Data Studio and select the enabled accelerator, you come to the management window. We click **Add** on the Accelerator Management panel as seen in Figure 13-3 and choose the two tables (TABLE_A and TABLE_A_ARC) for the initial load to the accelerator (Figure 13-4 on page 174).

▼ Tables (2 of 2 loaded / 2 of 2 enabled for acceleration) - Refreshing...

Buttons: Add, Alter Keys..., Remove, Load..., Acceleration, Storage Saver, Replication, Cancel Tasks

Name	Size	Acceleration	Last Load	Replication Since	Distribution Key	Skew	Organizing Keys	Organized
ISAOTEST	-	1 of 1	1 of 1 tables	1 of 1	-	-	-	-
SYSADM	-	1 of 1	1 of 1 tables	0 of 1	-	-	-	-

Figure 13-3 Click Add

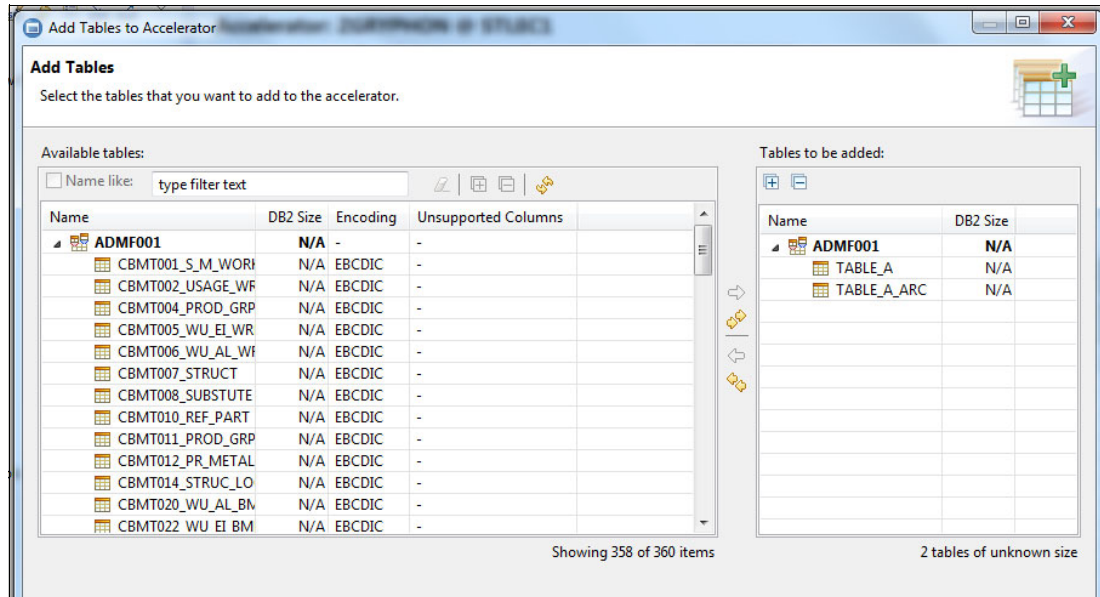


Figure 13-4 Adding tables to the accelerator

After the two tables are added to the accelerator, they are initially put into “*Initial load pending*” status.

If we look at the details of the two tables by clicking **Load**, as seen in Figure 13-5, partitions 1 and 2 of TABLE_A are empty, and partition 3 has data, which is active data. Partitions 1 and 2 of TABLE_A_ARC contain archived data, and partition 3 is empty.

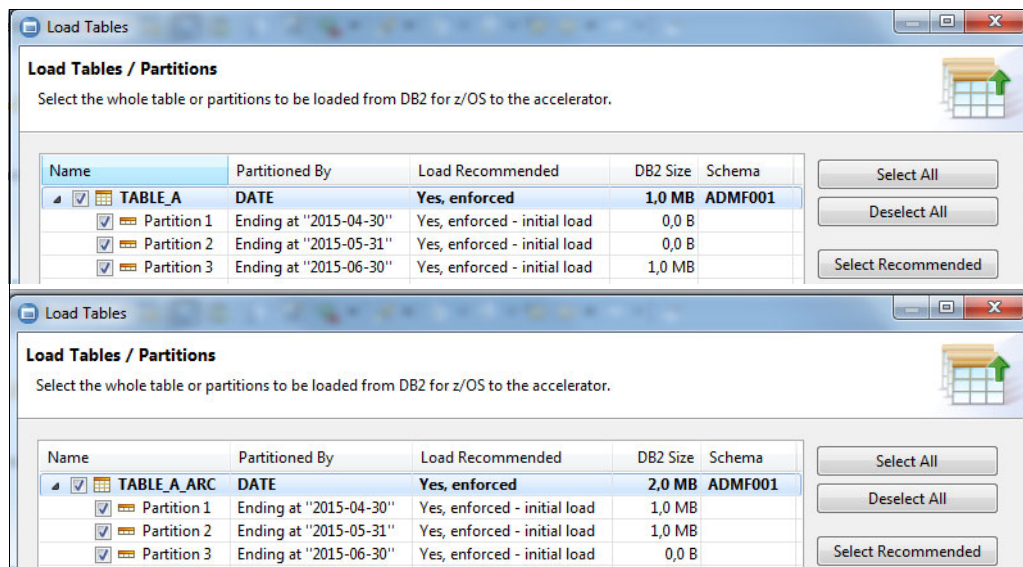


Figure 13-5 How the partitions look before loading data to the accelerator

When the tables are loaded into the accelerator, they are enabled and appear as in Figure 13-6 on page 175.

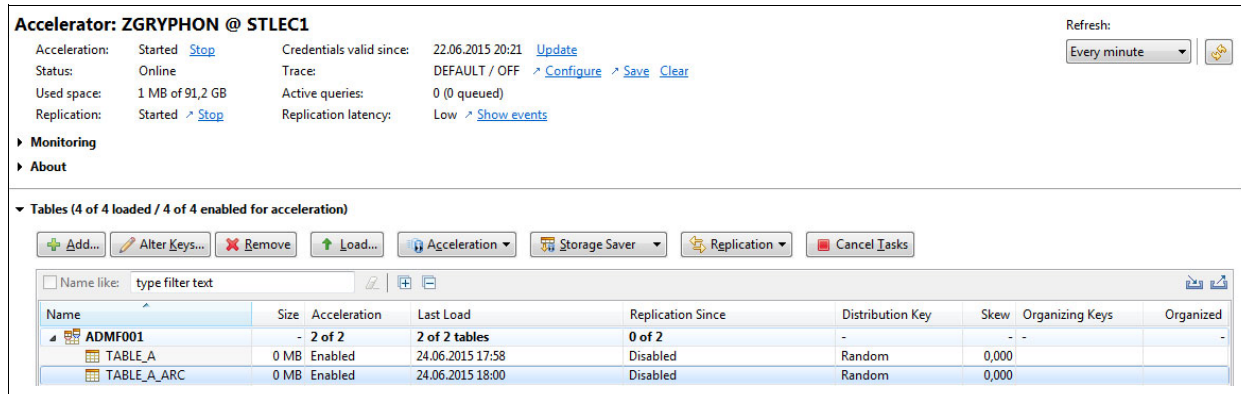


Figure 13-6 After initial load is completed and the tables are enabled for replication

At this point, the contents of TABLE_A (archived-enabled table) and TABLE_A_ARC (archive table) are the same on both DB2 and the accelerator, and both tables are referred to as *accelerator-shadow tables*.

13.3 Querying an accelerator-shadow table

When TABLE_A (archived-enabled table) and TABLE_A_ARC (archive table) are added to the accelerator, we refer to them as *accelerator-shadow tables*. An accelerator-shadow table exists both in DB2 and in the accelerator. The table on the accelerator contains all or a subset of the columns in the DB2 table. After the table is defined to the accelerator, you can load data into the table on the accelerator by copying data from the original DB2 table to the corresponding table on the accelerator. When the data is loaded to the accelerator, we can enable query acceleration for this table by using the CURRENT QUERY_ACCELERATION special register.

In this example, we access an accelerator-shadow table and retrieve data only from the table on the accelerator. The SYSIBMADM.GET_ARCHIVE built-in global variable is set to Y to access archived data. The CURRENT QUERY_ACCELERATION special register is set to ALL to execute queries in the accelerator if they are eligible for acceleration.

- FOR READ ONLY is added to the query.

FOR READ ONLY: Without the READ ONLY clause, the query was not allowed.

- Query is routed to the accelerator.
- Archive-enabled table and the archive table are accessed.

The result contains only rows from the accelerator-shadow table that exist on the accelerator (Example 13-1).

Example 13-1 Access an accelerator-shadow table on the accelerator

```
SET CURRENT QUERY ACCELERATION = ALL ;
SET SYSIBMADM.GET_ARCHIVE='Y' ;
SELECT * FROM ADMF001.TABLE_A FOR READ ONLY ;
```

PHONENO	CUSTNO	COLLDUR	DATE
3171427	1	15	2015-04-15
3171201	2	11	2015-04-16
3171427	1	15	2015-05-15
3171201	2	12	2015-06-14
3171427	1	25	2015-06-15
3171427	1	10	2015-04-13
3171201	2	11	2015-04-15
3171201	2	12	2015-05-15

DSNE610I NUMBER OF ROWS DISPLAYED IS 8
DSNE616I STATEMENT EXECUTION WAS SUCCESSFUL, SQLCODE IS 100

As you see in Figure 13-7, Data Studio's Query Monitor window, which keeps the history of completed SQL statements the query is executed in the accelerator. The top query, which is already highlighted, shown in the SQL Text column, is our query that is executed.

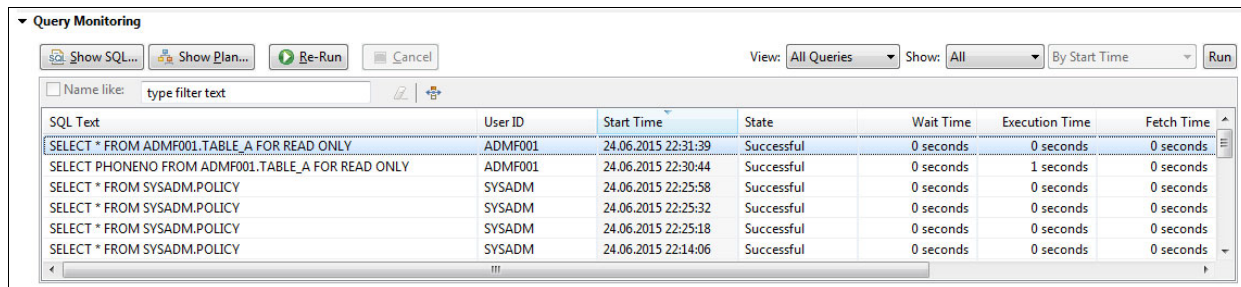


Figure 13-7 Query Monitor window

13.4 Moving archived partitions to the accelerator as an accelerator-archived table

In this part of the case study, partitions 1 and 2 of TABLE_A_ARC are moved to the accelerator by using the storage saver function. The data in these two partitions is deleted from DB2 and these partitions are placed in a status of PRO. TABLE_A_ARC, which was an accelerator-shadow table when its contents were the same on DB2 as in the accelerator, now becomes an *accelerator-archived table*. An accelerator-archived table is a table on the accelerator for which partitions of a DB2 table, or the entire table, are archived on the accelerator.

When you click **storage saver**, you'll see the warning message in Figure 13-8 on page 177. It warns that after the data is moved to the accelerator, it no longer resides within DB2. Therefore, any operation that is not supported by the accelerator, such as modifying data or schemas, is no longer possible for the archived partitions.

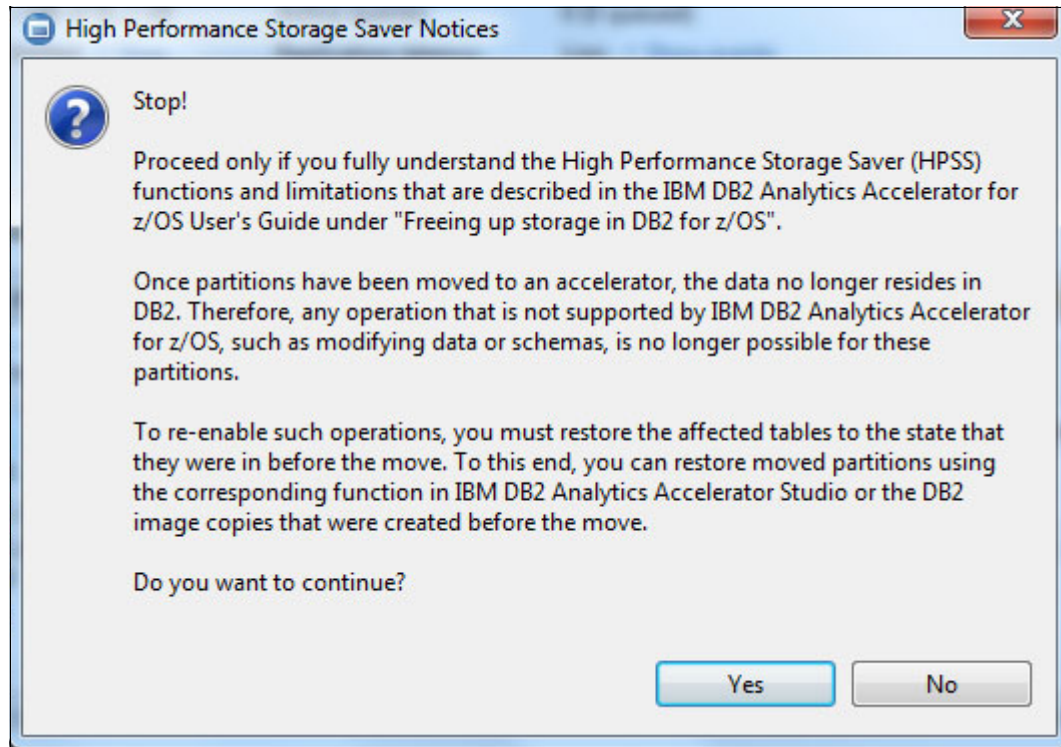


Figure 13-8 Storage Saver warning

After we click **Yes**, the next window is to select the required partitions for archiving. Those partitions are moved as explained in 10.3, "Online data archiving" on page 139. As seen in Figure 13-9, partitions 1 and 2 are selected for archiving.

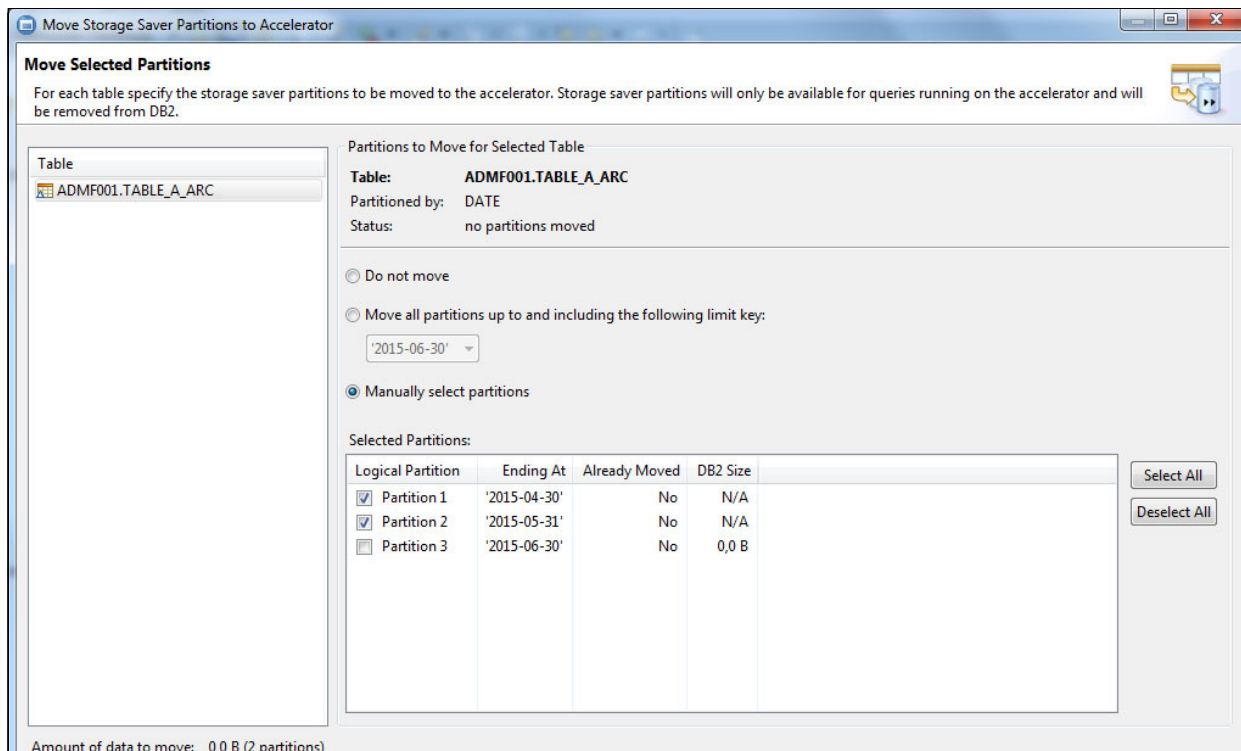


Figure 13-9 Partition selection for archiving

When the archive process is completed, the status of the partitions selected turn to PRO status (Example 13-2).

Example 13-2 Database status after archive

NAME	TYPE	PART	STATUS
PHONE	TS	0001	RW
-THRU		0003	
PHONE	TS		
PHONEH	TS	0001	RW, PRO
-THRU		0002	
PHONEH	TS	0003	RW
PHONEH	TS		

13.5 Querying both an accelerator-shadow table and an accelerator-archived table

Now we have TABLE_A both in DB2 and in the accelerator, which is enabled for transparent archiving. Partition 3 of TABLE_A_ARC is both in DB2 and the accelerator. Partitions 1 and 2 of TABLE_A_ARC have been moved to the accelerator and removed from DB2. TABLE_A is an accelerator-shadow table and TABLE_A_ARC is an accelerator-archived table.

Examples illustrate how the SYSIBMADM.GET_ARCHIVE built-in global variable affects whether archive data is accessed for a query against an archive-enabled table, and how the CURRENT QUERY ACCELERATION and CURRENT GET_ACCEL_ARCHIVE special registers affect whether a query is executed in an accelerator.

Based on these facts, we tested some access combinations. Table 13-1 summarizes the combinations illustrated in the examples.

Table 13-1 Examples

Description	Example 3	Example 4	Example 5
GET_ARCHIVE built-in global variable			
CURRENT QUERY ACCELERATION special register	ALL	ALL	NONE
CURRENT GET_ACCEL_ARCHIVE special register	YES	NO	NO
Where the query is executed	Accelerator	Accelerator	DB2
Archive-enabled table accessed? (archive-enabled table is also an accelerator-shadow table)	YES	YES	YES

Description	Example 3	Example 4	Example 5
Archive table accessed? (archive table is also an accelerator-archived table)	YES	YES	YES
Archived partitions from the archive table that reside on the accelerator accessed?	YES	NO	NO (because query was executed on DB2)

Example 3

In this example, we access an accelerator-archived table. The SYSIBMADM.GET_ARCHIVE built-in global variable is set to Y to access archived data. The CURRENT QUERY ACCELERATION special register is set to ALL to execute queries in the accelerator if they are eligible for acceleration. The CURRENT GET_ACCEL_ARCHIVE special register is set to YES to include data archived on an accelerator server when a query references an accelerator table. Note the following actions:

- ▶ Query is routed to the accelerator.
- ▶ Archive-enabled table and the archive table are accessed.
- ▶ Accelerator-archived table is accessed for the data (partitions) archived on the accelerator.

The result contains all of the data that is retrieved from the accelerator (Example 13-3).

Example 13-3 Access an accelerator-archived table with the CURRENT GET_ACCEL_ARCHIVE special register set to YES

```

SET CURRENT QUERY ACCELERATION = ALL ;
SET SYSIBMADM.GET_ARCHIVE='Y' ;
SET CURRENT GET_ACCEL_ARCHIVE = YES ;
SELECT * FROM ADMF001.TABLE_A FOR READ ONLY ;
-----+-----+-----+-----+-----+-----+-----+
      PHONENO      CUSTNO  COLLDUR  DATE
-----+-----+-----+-----+-----+-----+-----+
      3171427          1        15  2015-04-15
      3171201          2        11  2015-04-16
      3171427          1        15  2015-05-15
      3171201          2        12  2015-06-14
      3171427          1        25  2015-06-15
      3171427          1        10  2015-04-13
      3171201          2        11  2015-04-15
      3171201          2        12  2015-05-15
DSNE610I NUMBER OF ROWS DISPLAYED IS 8
DSNE616I STATEMENT EXECUTION WAS SUCCESSFUL, SQLCODE IS 100

```

Example 4

In this example, we also access an accelerator-archived table. The SYSIBMADM.GET_ARCHIVE built-in global variable is set to Y to access archived data. The CURRENT QUERY ACCELERATION special register is set to ALL to execute queries in the accelerator if they are eligible for acceleration. The CURRENT GET_ACCEL_ARCHIVE special register is set to NO so that data archived on an accelerator server is not accessed when a query references an accelerator table. Note the following:

- ▶ Query is routed to the accelerator.
- ▶ Archive-enabled table and the archive table are accessed.
- ▶ Accelerator-archived table is not accessed.

The result contains only rows from the archive shadow copy of the archive-enabled table. The query is executed in the accelerator. Because GET_ACCEL_ARCHIVE = NO, only rows from TABLE_A are retrieved (Example 13-4).

Example 13-4 Access an accelerator-archived table with the CURRENT GET_ACCEL_ARCHIVE special register set to NO

```

SET CURRENT QUERY ACCELERATION = ALL ;
SET SYSIBMADM.GET_ARCHIVE='Y' ;
SET CURRENT GET_ACCEL_ARCHIVE = NO ;
SELECT * FROM ADMF001.TABLE_A FOR READ ONLY ;
-----+-----+-----+-----+-----+-----+
      PHONENO      CUSTNO  COLLDUR  DATE
-----+-----+-----+-----+-----+-----+
      3171201          2        12  2015-06-14
      3171427          1        25  2015-06-15
DSNE610I NUMBER OF ROWS DISPLAYED IS 2
DSNE616I STATEMENT EXECUTION WAS SUCCESSFUL, SQLCODE IS 100

```

Example 5

In this example, we also access an accelerator-archived table. The SYSIBMADM.GET_ARCHIVE built-in global variable is set to Y to access archived data. The CURRENT QUERY ACCELERATION special register is set to NONE so queries are not executed in the accelerator. The CURRENT GET_ACCEL_ARCHIVE special register is set to NO, but the value doesn't matter because the CURRENT QUERY ACCELERATION special register is set to NONE, disabling query acceleration. Note the following:

- ▶ Query is routed to DB2. The query is not executed in the accelerator.
- ▶ Archive-enabled table and archive table are accessed.
- ▶ Accelerator-archived table is not accessed.

The result contains only rows from the archive-enabled table and associated archive table. The query is executed in DB2. Because partitions 1 and 2 were moved to the accelerator (and only exist on the accelerator), those partitions are empty in DB2 and there is nothing that can be retrieved from them for the query (Example 13-5).

Example 13-5 Access the accelerator-archived table with the CURRENT QUERY ACCELERATION special register set to NONE

```

SET CURRENT QUERY ACCELERATION = NONE ;
SET SYSIBMADM.GET_ARCHIVE='Y' ;
SET CURRENT GET_ACCEL_ARCHIVE = NO ;
SELECT * FROM ADMF001.TABLE_A FOR READ ONLY ;
-----+-----+-----+-----+-----+-----+
      PHONENO      CUSTNO  COLLDUR  DATE
-----+-----+-----+-----+-----+-----+
      3171201          2        12  2015-06-14
      3171427          1        25  2015-06-15
DSNE610I NUMBER OF ROWS DISPLAYED IS 2
DSNE616I STATEMENT EXECUTION WAS SUCCESSFUL, SQLCODE IS 100

```

Glossary

accelerator-only table

An accelerator-only table is a table for which the data exists only on the DB2 Analytics Accelerator. An accelerator-only table is automatically created in the accelerator when the CREATE TABLE statement is issued on DB2 with the IN ACCELERATOR clause.

accelerator-shadow table

An accelerator-shadow table exists both in DB2 and in the accelerator. The table in the accelerator contains all or a subset of the columns in the DB2 table.

accelerator-archived table

An accelerator-archived table is a table on the accelerator for which partitions of a DB2 table, or the entire table, are archived on the accelerator. When the data is archived on the accelerator, the original DB2 data is deleted. The table space for a DB2 table that is associated with an archived accelerator table is set to a permanent read-only state so that the original DB2 table can no longer be changed.

application period

A pair of columns with application-maintained values that indicate the period of time when a row is valid. See also *application-period temporal table*.

application-period temporal table

A table that includes an application period. See also *application period*, *bitemporal table*.

archive table

A table that is used by the database manager to store rows deleted from the associated archive-enabled table.

archive-enabled table

A table for which deleted rows are stored in the associated archive table.

begin column

In a system period or an application period, the column that indicates the beginning of the period. See also *period*.

bitemporal table

A table that is both a system-period temporal table and an application-period temporal table. See also *application-period temporal table*, *system-period temporal table*.

end column

In a system period or an application period, the column that indicates the end of the period. See also *period*.

generated column

A column for which the database manager assigns the value. An example of a generated column is an identity column, row change timestamp column, or row-begin column.

generated expression column

A generated column that is defined using an expression.

historical row

A row in a history table. See also *history table*.

history table

A table that is used by the database manager to store the historical versions of the rows from the associated system-period temporal table. See also *historical row*, *system-period temporal table*.

period

In a table, an interval of time that is defined by two datetime columns. A period contains a begin column and an end column. See also *begin column*, *end column*.

row-begin column

A generated column that is defined with the ROW BEGIN clause. The value is assigned whenever a row is inserted into the table or any column in the row is updated. A row-begin column is intended for use as the first column of a SYSTEM_TIME period. See also *generated column*, *row-end column*, *transaction-start-ID column*.

row-end column

A generated column that is defined with the ROW END clause. The value is assigned whenever a row is inserted into the table or any column in the row is updated. A row-end column is intended for use as the second column of a SYSTEM_TIME period. See also *generated column*, *row-begin column*, *transaction-start-ID column*.

system-period temporal table

A table that is defined with system-period data versioning. See also *bitemporal table*, *system-period data versioning*, *history table*.

system-period data versioning

Automatic maintenance of historical data by the database manager using a system period. See also *system period*, *system-period temporal table*.

system period

A pair of columns with system-maintained values that indicate the period of time when a row is valid. See also *system-period temporal table*, *system-period data versioning*.

temporal table

A table that records the period of time when a row is valid.

transaction-start-ID column

A generated column that is defined with the TRANSACTION START ID clause. The value is assigned whenever a row is inserted into the table or any column in the row is updated. A transaction-start-ID column is intended for use in a system-period temporal table. See also *generated column*, *row-begin column*, *row-end column*.

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this book.

Online resources

These websites are also relevant as further information sources:

- ▶ *A Matter of Time: Temporal Data Management in DB2 for z/OS*
http://public.dhe.ibm.com/software/data/sw-library/db2/papers/A_Matter_of_Time_-_DB2_zOS_Temporal_Tables_-_White_Paper_v1.4.1.pdf
- ▶ *Adopting temporal tables in DB2, Part 1: Basic migration scenarios for system-period tables*
http://www.ibm.com/developerworks/data/library/techarticle/dm-1210temporaltable_sdb2
- ▶ *Adopting temporal tables in DB2, Part 2: Advanced migration scenarios for system-period temporal tables*
http://www.ibm.com/developerworks/data/library/techarticle/dm-1210temporaltable_sdb2pt2
- ▶ *Best Practices Temporal Data Management with DB2*
https://www.ibm.com/developerworks/community/wikis/form/anonymous/api/wiki/6aeb9b0-ba6c-4e1c-af2c-4b8b9a140194/page/3af511d6-aab8-4403-9ea5-5bea6cef78df/attachment/4b4a9bce-29b6-4afc-aa54-bd2333da9aa0/media/db2z_temporal_best_practices_2014.pdf

Help from IBM

IBM Support and downloads

ibm.com/support

IBM Global Services

ibm.com/services



SG24-8316-00

ISBN 0738440965

Printed in U.S.A.

Get connected

