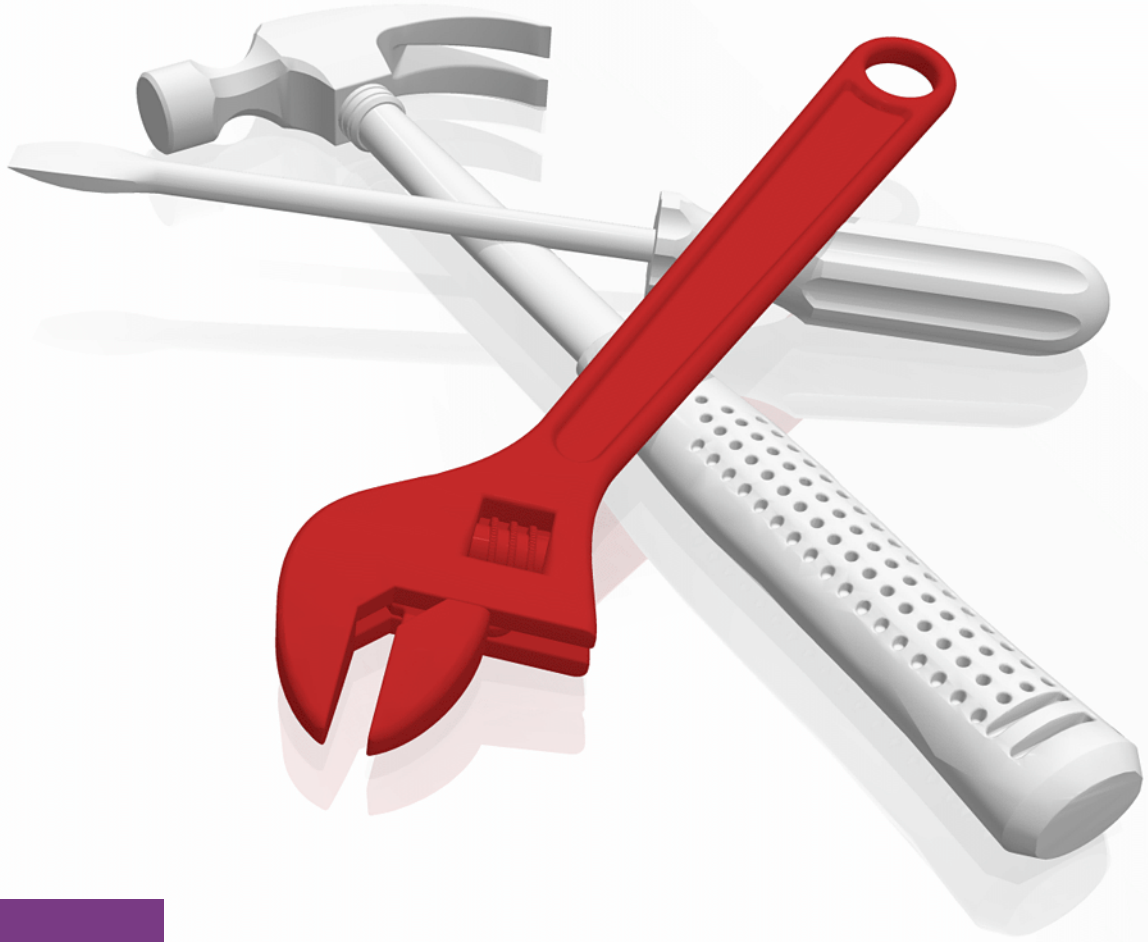
Redbooks
ibm.com/redbooks

DB2 12 for z Optimizer

Terry Purcell



z Systems

IBM

Redpaper

Introduction

There has been a considerable focus on performance improvements as one of the main themes in recent IBM DB2® releases, and DB2 12 for IBM z/OS® is certainly no exception. With the high-value data retained on DB2 for z/OS and the z Systems platform, customers are increasingly attempting to extract value from that data for competitive advantage.

Although customers have historically moved data off platform to gain insight, the landscape has changed significantly and allowed z Systems to again converge operational systems with analytics for real-time insight. Business-critical analytics is now requiring the same levels of service as expected for operational systems, and real-time or near real-time currency of data is expected. Hence the resurgence of z Systems.

As a precursor to this shift, IDAA brought the data warehouse back to DB2 for z/OS and, with its tight integration with DB2, significantly reduces data latency as compared to the ETL processing that is involved with moving data to a stand-alone data warehouse environment. That change has opened up new opportunities for operational systems to extend the breadth of analytics processing without affecting the mission-critical system and integrating near real-time analytics within that system, all while maintaining the same z Systems qualities of service.

Apache Spark on z/OS and Linux for System z also allow analytics in-place, in real-time or near real-time. Enabling Spark natively on z Systems reduces the security risk of multiple copies of the Enterprise data, while providing an application developer-friendly platform for faster insight in a simplified and more secure analytics framework.

How is all of this relevant to DB2 for z/OS? Given that z Systems is proving again to be the core Enterprise Hybrid Transactional/Analytical Processing (HTAP) system, it is critical that DB2 for z/OS can handle its traditional transactional applications and address the requirements for analytics processing that might not be candidates for these rapidly evolving targeted analytics systems. And not only are there opportunities for DB2 for z/OS to play an increasing role in analytics, the complexity of the transactional systems is increasing. Analytics is being integrated within the scope of those transactions.

DB2 12 for z/OS has targeted performance to increase the success of new application deployments and integration of analytics to ensure that we keep pace with the rapid evolution of IDAA and Spark as equal partners in HTAP systems.

This paper describes the enhancements delivered specifically by the query processing engine of DB2. This engine is generally called the optimizer or the Relational Data Services (RDS) components, which encompasses the query transformation, access path selection, run time, and parallelism.

DB2 12 for z/OS also delivers improvements targeted at OLTP applications, which are the realm of the Data Manager, Index Manager, and Buffer Manager components (to name a few), and are not identified here.

Although the performance measurement focus is based on reducing CPU, improvement in elapsed time is likely to be similarly achieved as CPU is reduced and performance constraints alleviated. However, elapsed time improvements can be achieved with parallelism, and DB2 12 does increase the percentage offload for parallel child tasks, which can further reduce chargeable CPU for analytics workloads.

This IBM® Redpaper™ contains the following sections:

- Why read further
- The performance focus in DB2 12 for z/OS
- UNION ALL and outer join performance
- Runtime optimizations
- Predicate optimizations
- User-Defined table function predicate optimizations
- Optimizer cost model enhancements
- RUNSTATS and optimizer-generated statistics profiles
- Static and dynamic plan stability
- Summary

Why read further

If you are interested in understanding the overall performance focus and the performance expectation of DB2 12 for z/OS, read the next section. If you are interested in a high-level overview of each enhancement, skip to each section and view the examples that are provided for most enhancement topics. For a thorough understanding, read all of the sections.

This overview is written in a manner that allows DB2 database administrators, developers, and consultants with a basic understanding of SQL performance to logically determine both how and why each enhancement will benefit the identified SQL. As each enhancement is explained, there is generally an overview of how DB2 operated in prior releases and how DB2 12 provides a benefit. And, despite IDAA and Spark with z Systems being referenced in the introduction, the focus is returning to z Systems for true real-time analytics, so there is no further discussion about these topics. Instead, the overview discusses the DB2 12 enhancements to the RDS components for improved performance.

Employees and users of competitors to DB2 for z/OS can also benefit from reading this overview, including Hadoop-based SQL engines and cloud database providers. These competitive systems typically lack the maturity of performance-focused features of DB2 for z/OS. Therefore, understanding how and why DB2 has addressed performance challenges in CPU-constrained environments can help those competitors to identify opportunities for their customers. Increasing the knowledge of our competitors encourages new innovation among all database vendors, which benefits all of our customers and the industry as a whole.

The performance focus in DB2 12 for z/OS

The introduction referred to the importance of bringing analytics closer to the operational system, but also to the increased complexity of modern OLTP systems. The complexity and thus opportunity for improvements associated with analytics workloads might be understood, but the complexity associated with OLTP systems requires more explanation.

Complex SQL can mean anything related to an increased number of tables being joined, number of rows being processed by the query, and duplicate removal expressions. The goal for DB2 development in DB2 12 for z/OS was to identify common SQL constructs from ERP applications and query generators, and to identify the bottlenecks that might affect the performance of existing operational systems as more work is deployed to DB2 for z/OS.

The following is a list of the key focus areas or SQL constructs targeted in DB2 12 for z/OS, with more detail provided in the following sections, including an explanation of other interesting query patterns targeted for improvement:

- Complex views, table user-defined functions (or stand-alone queries), or both containing:
  - UNION ALL
  - Outer joins
  - Stage 2 join expressions
  - CASE expressions, user-defined functions, CAST expressions, or scalar functions

Additionally, the following areas were targeted as bottlenecks that might inhibit workload scalability for complex operational or analytics processing in DB2 for z/OS:

- Sort and workfile processing
- Prepare cost and frequency
- I/O performance

This is not an exhaustive list. Later sections cover the major enhancements targeted at analytics and complex OLTP workloads. Enhancements related to traditional (simpler) OLTP workloads are also included in DB2 12 for z/OS, but not included in this overview.

Performance expectations

As outlined in the introduction, a major goal in DB2 12 for z/OS is to ensure that DB2 is well positioned to handle the demands of traditional workload growth and support new opportunities in HTAP. Although the performance improvements can benefit existing workloads, as you see in this section, the benefits vary based on the amount of exposure that customers have to the SQL constructs or bottlenecks that were targeted in this release.

In DB2 11 for z/OS, we reported performance improvements for our internal workloads that exceeded the results of any prior DB2 release, with some workloads showing 30 - 40% CPU reductions as highlighted in Figure 1.

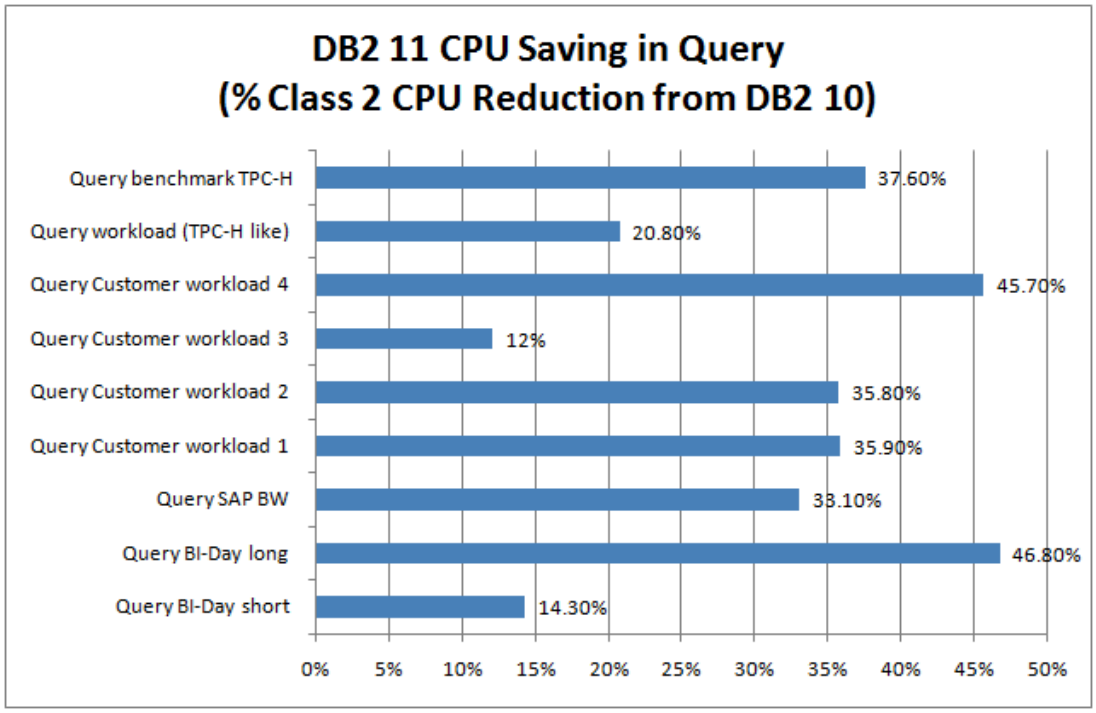


Figure 1 Internal DB2 workload results from DB2 11 for z/O

During DB2 12 for z/OS development, we added additional workloads to our internal measurements that represented some of the new workload growth opportunities in complex transactional processing and real-time analytics in DB2. And as outlined in Figure 2, the performance results vary significantly for each of the measured workloads. Figure 1 compares DB2 10 to DB2 11 results, and the DB2 12 performance is comparing DB2 11 to DB2 12. Therefore, customers not yet on DB2 11 will benefit from considering moving up their migration plans to DB2 11 as a catalyst to taking advantage of DB2 12 for z/OS sooner, given the potential CPU savings and performance improvements available in both releases.

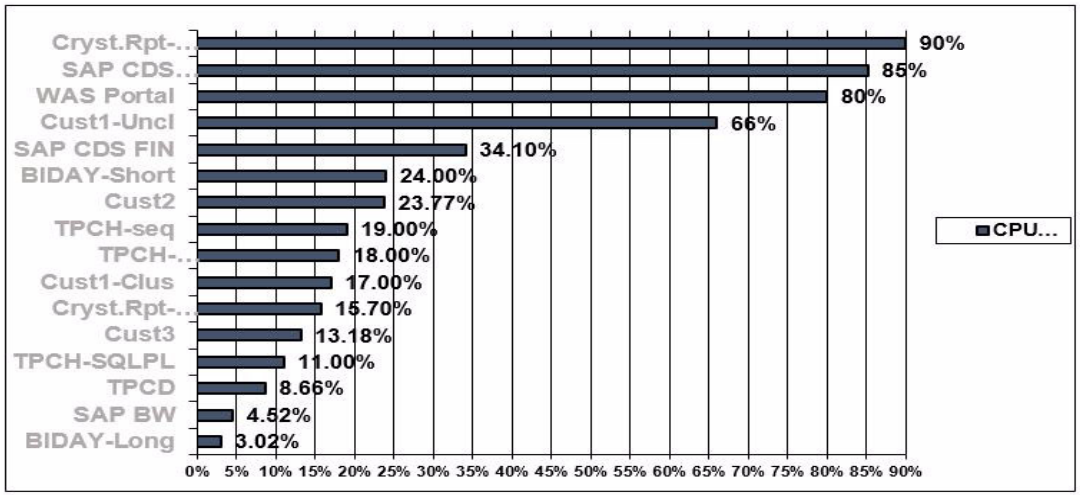


Figure 2 Internal DB2 workload results from DB2 12 for z/OS

Note that these measurements were conducted in an isolated environment to allow accurate performance comparison. Although this isolation does not match customer systems where highly concurrent activity competes for resources, the performance results observed in isolated environments translate to considerable performance improvements in highly concurrent environments.

However, measuring only the DB2 portion of the workload, as represented here, is not capturing application CPU or any other system processing seen in customer environments. However, given the variability of the demonstrated DB2 CPU across these workloads, it does raise questions as to the reasons why some workloads achieved greater improvement than others. See Figure 3 for a summary of performance results for workloads in the 30–90% CPU reduction range.

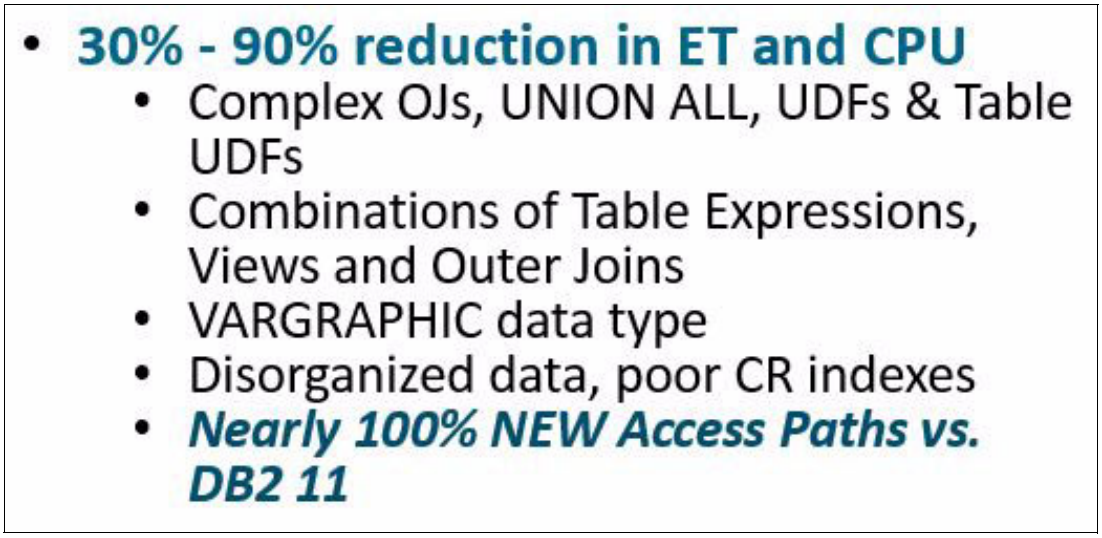


Figure 3 Workload characteristics of workloads in the 30–90% CPU reduction range

The workloads that demonstrate the most significant performance improvements are those that were the primary target for workload growth in DB2 12 for z/OS: UNION ALL and outer joins within either views, table expressions or Table UDFs.

The SAP-specific workloads measured were mostly using the VARGRAPHIC data type, although the non-SAP workloads did not. The CUST1-UNCL workload represents a customer workload that was converted to clustering the data by indexes that were not the indexes used for the joins between tables. This configuration was to represent situations where a workload does not have perfect clustering, or when the clustering have degraded due to regular INSERT/UPDATE/ DELETE activity.

In all these workloads showing large performance improvements, most of the DB2 12 access paths were different from those chosen by the optimizer in DB2 11.

As summarized in Figure 4, the DB2 workloads that were in the range of 15–25% improvement contained queries that involved significant sort and aggregation from GROUP BY or DISTINCT duplicate removal.

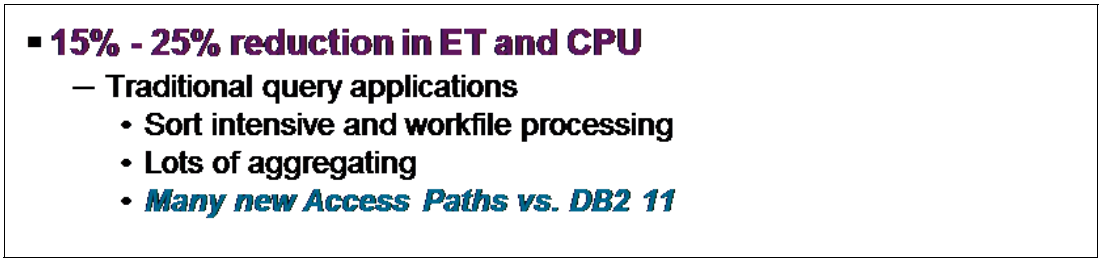


Figure 4 Workload characteristics of workloads in the 15–25% CPU reduction range

The remaining workloads measured achieved less than 15% CPU savings as compared with DB2 11. As noted in Figure 5, these workloads are predominantly scan-based workloads, which are ideal candidates for IDAA due to its highly parallelized architecture. However, it should also be noted that DB2 can improve the elapsed time performance of these workloads with parallelism. Also, z Systems specialty engines (zIIPs) reduce the chargeable CPU costs for these workloads.

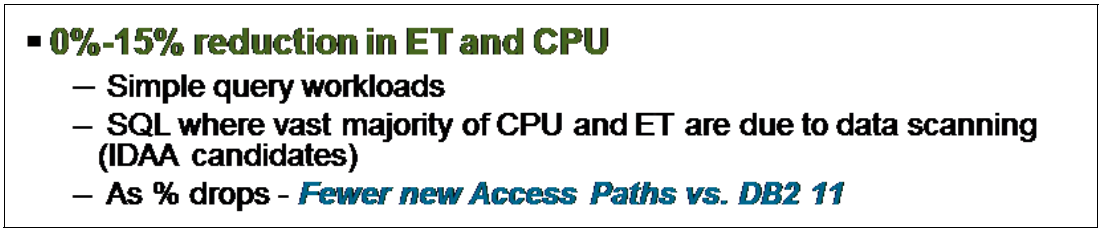


Figure 5 Workload characteristics of workloads in the 0–15% CPU reduction range

In DB2 12 for z/OS, the zIIP offload for parallel child tasks increases to 100% offload from 80% in DB2 11 and prior releases. This change applies to the child tasks of local or distributed applications running on DB2 for z/OS. This enhancement alone has also resulted in performance improvements for parallelism workloads due to the reduction in processing for DB2 to manage the offload percentage that was required before DB2 12.

UNION ALL and outer join performance

Although the UNION ALL construct and outer joins might not always be combined in the same queries, there is significant overlap with the performance challenges of both, so it makes sense to discuss them together. However, not all queries involving UNION ALL or outer joins have performance challenges. The performance opportunities exist where these constructs currently involve materializations where filtering is not applied early in the query execution. Prior releases of DB2 have delivered enhancements for both constructs such that performance can be acceptable for simple use cases.

An outer join is generally coded when the join relationship between tables is optional, whereas an inner join is coded to maintain a mandatory join relationship. The expectation is typically that performance is similar between inner joins and outer joins. Although it might be possible to have similar performance, comparable performance can be challenging because outer joins will return rows even if there isn't a match between both tables, whereas inner joins will only return rows that match between both tables.

Therefore, you may not get the same performance if you simply changed an inner join to become an outer join. In addition, an outer join can dictate that A must be joined before B, while inner joins are free to choose any join sequence.

UNION ALL is used to combine different tables and allow them to be represented as a consistent result. This construct is appearing in many new workloads, including (but not limited to) SAP Core Data Services and IBM WebSphere®. It is also used internally by DB2 for System-period Temporal (available since DB2 10) and Transparent Archive (since DB2 11). The DB2 exploitation exposes more customers to UNION ALL within their queries. Customers are also integrating active and archive tables in their own applications, and using UNION ALL to supplement or alleviate the management challenges of very large table spaces.

Similar to the expectations for outer joins, introducing a UNION ALL infrastructure results in performance expectations similar to that of accessing a single object. This configuration can be far more challenging given the increase in the number of objects involved when UNION ALL is introduced.

In DB2 12 for z/OS, the targeted enhancements for improving UNION ALL and outer join performance to address materialization challenges include:

- Reordering outer join tables to avoid materializations
- Push predicates inside UA legs or outer join query blocks
- Sort of outer to ensure sequential access to inner
- Bypass workfile usage when materialization required
- Trim unnecessary columns and tables
- Push ORDER BY and FETCH FIRST into UA legs

These targeted enhancements are explained in the following sections. You can skip these sections to the UNION ALL and outer join summary, or read through them to gain an understanding of each enhancement and an explanation why each enhancement was targeted in DB2 12.

Reorder outer join tables to avoid materializations

Allowing DB2 to internally reorder the outer join tables within the query overcomes a limitation that occurs when combining outer and inner joins in the same query. In certain instances, DB2 11 and prior will materialize some tables, which can result in local or join filtering not being applied before the materialization. Some customers who have encountered this performance challenge have rewritten their queries to ensure that all inner joins appear before outer joins in the query, if possible.

Rewriting a query is often difficult given the proliferation of generated queries and applications being deployed without thorough performance evaluation. Therefore, minimizing exposure to this limitation in DB2 12 provides a valuable performance boost for affected queries.

Push predicates inside UNION ALL legs or outer join query blocks

The example shown in Figure 6 can be used to demonstrate performance challenges when UNION ALL is combined with an outer join. The original query appears at the top of Figure 6 and appears as a simple two table (left outer) join of T1 to T2.

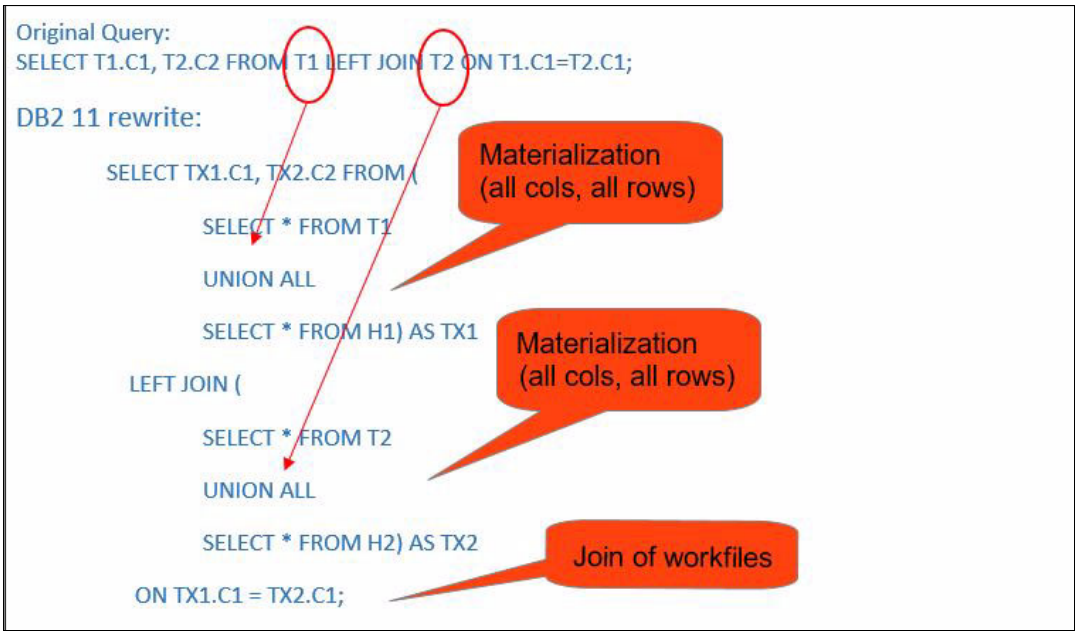


Figure 6 DB2 11 left outer join query with transparent archive tables (or any UNION ALL views)

In this example, both T1 and T2 have archive enabled, which refers to the DB2 11 transparent archive feature. Therefore, DB2 rewrites the query to include the active and archive tables, with T1 circled and the arrow pointing to the first UNION ALL on the left side of the join, and T2 circled with arrow pointing to the second UNION ALL on the right side of the join. This representation would be true of any UNION ALL within a view, where the view definition is replaced within the query where it is referenced.

The performance challenge for this query example is that DB2 executes each leg of the UNION ALL separately and combines the results from each side of the first UNION ALL. DB2 then combines each side of the second UNION ALL before joining the two results together as requested in the outer query block.

In the V11 rewrite of the query, there are no join predicates or any filtering within the UNION ALL legs. The term “combining” means that DB2 returns all columns and all rows from T1 in the first UNION ALL and all columns and all rows from H1. It then materializes those rows to a workfile. The workfile is then sorted in preparation for the join. This process is repeated for the second UNION ALL. All columns and all rows from T2 and H2 are materialized into a workfile and sorted in preparation for the join on column C1 from both workfiles.

The performance of this query depends heavily on the size of the tables that are involved, with very large tables consuming significant CPU and workfile resources to complete.

Figure 7 shows how several of the UNION ALL-focused enhancements in DB2 12 can improve the performance of the query from Figure 6 on page 8. Although the internal rewrite of the tables to the UNION ALL representation remains the same, DB2 12 adds the ability for the optimizer to make a cost-based decision about whether to push the join predicates into the UNION ALL legs.

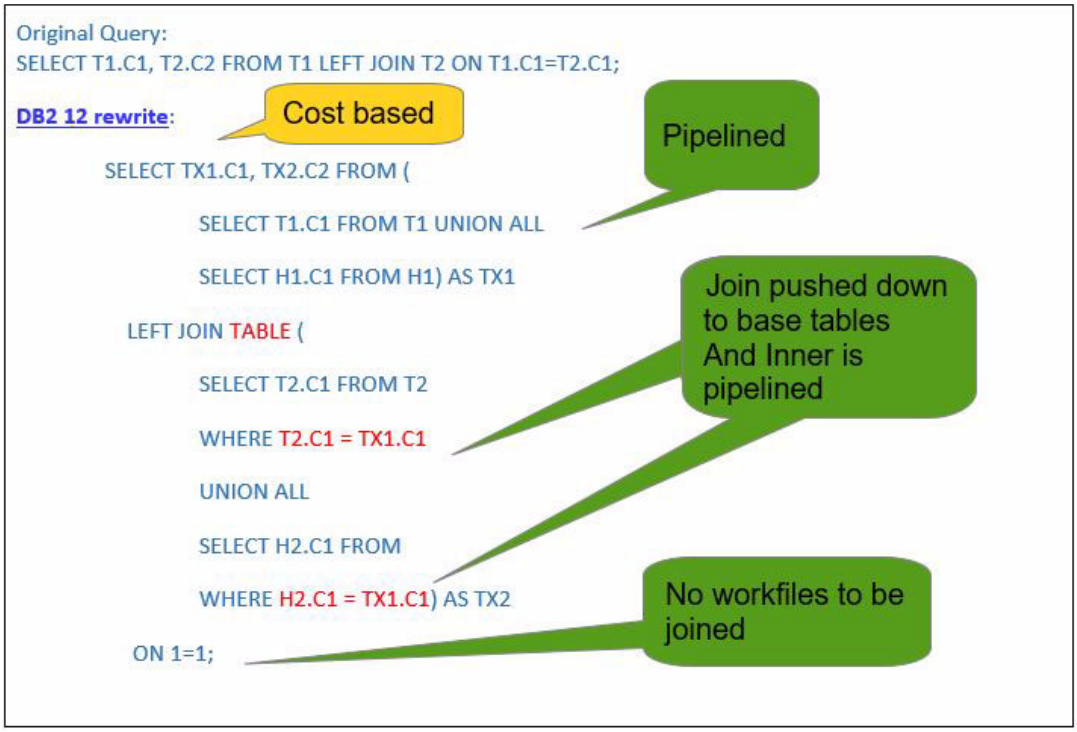


Figure 7 DB2 12 left outer join query with transparent archive tables (or any UNION ALL views)

Figure 7 demonstrates an example where the optimizer chose to push the join predicates inside each UNION ALL leg. The join predicates only occur on the right side of the join because a join is “FROM” the left “TO” the right. The TABLE keyword (highlighted in red) is required externally for this example to be syntactically valid given that pushing the predicates down results in the query appearing as a correlated table expression.

Having the join predicates in each UNION ALL leg allows the join to “look up” each leg of the UNION ALL for every row coming from the outer, rather than sort the full results for the join. By using an additional optimization to reduce workfile usage and materialization,

DB2 12 can “pipeline” the rows from the first UNION ALL (on the left side of the join). Transferring rows from one query block to another can require the results to be written (materialized) to an intermediate workfile as in the DB2 11 example. However, DB2 12 can “pipeline” the result from the first UNION ALL to the join without requiring this materialization.

The result for this query in DB2 12 given the (cost based) join predicate pushdown is that the workfile/materializations are avoided, and available indexes can be used for each leg, as shown in Figure 8.

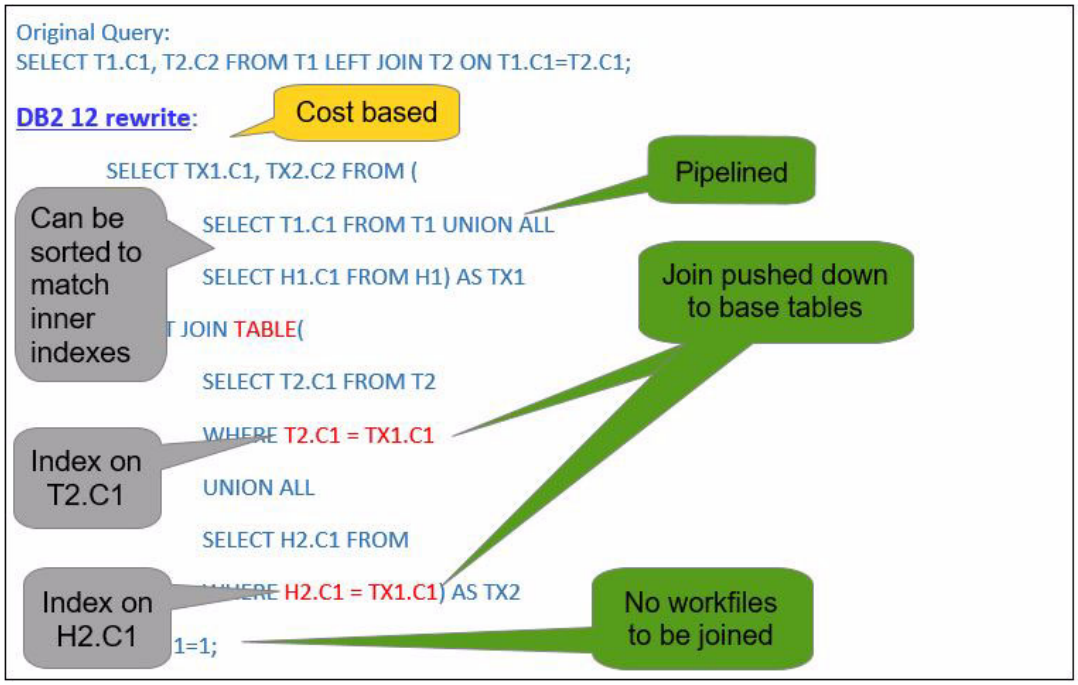


Figure 8 DB2 12 left outer join UNION ALL query with sort of outer

The example in Figure 8 can be expanded to demonstrate more DB2 12 enhancements that can improve performance closer to queries that do not use the UNION ALL infrastructure. If two tables are accessed in a different sequence, then the lookups to the inner (second) table will be random. DB2 can choose to make the access to the inner (second) table sequential by sorting the first table into the sequence of the 2nd. After the sort, both tables will be in the same sequence for the join.

If data from the outer table is accessed in a different sequence than the index used for the join to the inner, DB2 can choose to sort the outer into the sequence of the inner. This technique allows the access to the inner to occur in order. This approach is extended in DB2 12 to joins to UNION ALL views/table expressions, and is a cost-based choice for the optimizer.

Figure 9 demonstrates another variation of the same query where there are no supporting indexes for the join to T2.C1 or H2.C1. DB2 12 allows a sparse index to be built on an individual leg of a UNION ALL. This change applies to one or both legs. It is cost-based and thus can be combined with the other UNION ALL-focused enhancements.

Note that if a sparse index is chosen, you do not need to sort the outer into join order as was depicted in Figure 8 because a sparse index can use hashing if the result can be contained within memory. Thus, there are no concerns regarding random I/O.

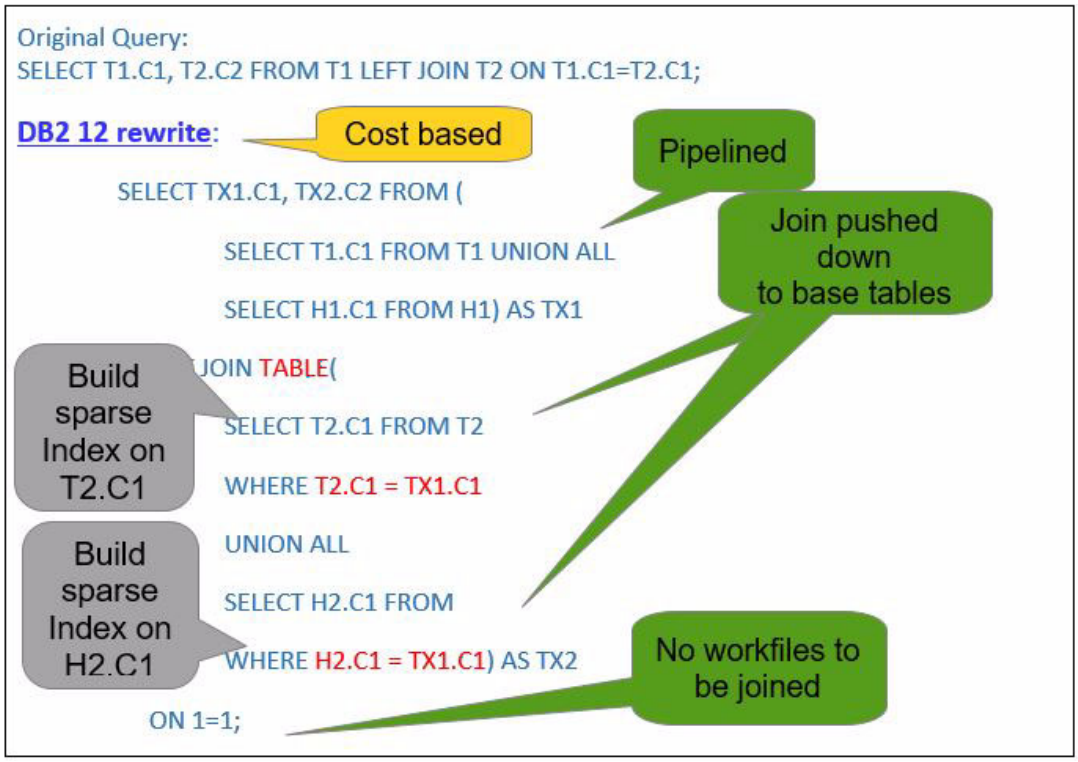


Figure 9 DB2 12 left outer join UNION ALL query without supporting join indexes

Trim unnecessary columns and tables

In the first UNION ALL example in Figure 6 on page 8, it was highlighted that “all columns and all rows” were materialized. The subsequent examples focused on the optimizations to improve the available access path choices to avoid materializing “all rows” of the view/table expression. However, an explanation of the “all columns” was purposefully avoided until a discussion of the enhancement was presented.

The problem is that for materialized views or table expressions where the select list might contain more columns than are needed by the query that is referencing that view or table expression, in DB2 11 and prior when materialization occurs, all columns coded within that view/table expression are included as part of the materialization. DB2 already removes unreferenced columns from the select list if a view/table expression is merged with the referencing query, but not if that is materialized.

DB2 12 provides an enhancement to remove (prune) columns that are not required by the referencing query from the select list of the view/table expression before materialization. This technique has the benefit of reducing the size of the materialized result, and thus reducing the amount of workfile space and sort key size. It can also allow index-only access for access before materialization. Such an enhancement can also apply to join predicate pushdown where materialization can be avoided, as demonstrated in Figure 10 and Figure 11.

```
SELECT P.P_PARTKEY, P2.P_PARTKEY

FROM TPCH30.PART AS P LEFT OUTER JOIN

        (SELECT *

         FROM TPCH30.PART

         UNION ALL

         SELECT *

         FROM TPCH30.PART) P2

 ON P.P_PARTKEY = P2.P_PARTKEY;
```

Figure 10 DB2 11 processing all columns in materialized view/table expression

The query depicted in Figure 10 involves a LEFT OUTER JOIN to a UNION ALL table expression. In the select list for each leg of the UNION ALL is SELECT *, meaning return all columns from that table. However, the referencing SELECT only requires P2.P_PARTKEY (for the SELECT list and ON clause). Given the materialization of the UNION ALL table expression in DB2 11, all columns from PART table (for example, if the table has 10 columns) are accessed and materialized to the workfile. As the outer query is retrieving from that workfile, only P_PARTKEY is retrieved.

The Figure 10 example is repeated in Figure 11 showing that DB2 12 will prune the unnecessary columns and only require P_PARTKEY to be returned from each UNION ALL leg, whereas DB2 11 returned all columns.

```
SELECT P.P_PARTKEY, P2.P_PARTKEY

FROM TPCH30.PART AS P LEFT OUTER JOIN

        (SELECT P_PARTKEY

         FROM TPCH30.PART

         UNION ALL

         SELECT P_PARTKEY

         FROM TPCH30.PART) P2

 ON P.P_PARTKEY = P2.P_PARTKEY; ← Cost based pushdown available to UA legs
```

Figure 11 DB2 12 pruning unreferenced columns with optional join pushdown

If the UNION ALL table expression is materialized, then only P_PARTKEY would be retrieved and written/materialized to workfile compared with all (10) columns in DB2 11. And if P_PARTKEY is indexed, then the optimizer could choose a non-matching index scan in DB2 12 rather than table space scan in DB2 11. Similarly, if the join predicates were pushed down in DB2 12 and matching index access was chosen on P_PARTKEY, then index-only would now be possible because only P_PARTKEY is required.

Another benefit of column pruning is that it can provide extra table pruning opportunities. In DB2 10, an enhancement was delivered to prune outer join tables if no columns were required from that table and it was guaranteed that the table would not introduce duplicates. In DB2 12, this pruning enhancement is extended to outer joins to views/table expressions that are guaranteed not to return duplicates and no columns are required from the view/table expression.

The example in Figure 12 might appear convoluted, but it is a simplified example from an actual ERP workload that will be available on DB2 for z/OS. Before explaining the example, it is valid to ask why you would build a query that references tables from which columns are not actually required. This type of construct is becoming more common as views are built to hide the complexity and present a simplified representation to the user so that they can simply select from a view, rather than writing a complex query themselves.

Similarly for query generators, what is required of the query generator is that the select from the views and the underlying table relationships can be set up or altered without affecting the query generation. Such views can be built on top of each other, as building blocks, for simplicity in constructing these and also to simplify future changes at any level. What this technique means is that the views typically contain more than what you might require, and you get to select only the columns you need, and ultimately that might mean that you also only need a subset of the tables.

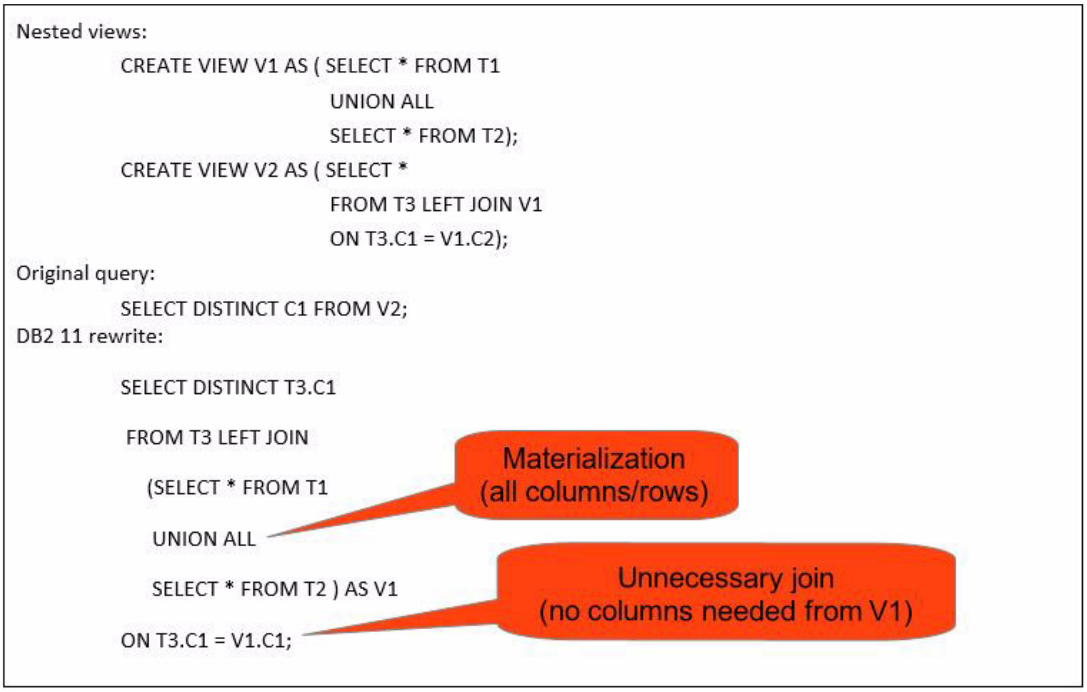


Figure 12 DB2 11 LEFT OUTER JOIN to unnecessary table expression

Figure 12 on page 13 demonstrates an example of how DB2 11 would handle this problem. Similar to prior UNION ALL examples, the UNION ALL retrieves all columns and all rows from T1 and T2 and materialize these to a workfile, which are sorted for the join. T3 is then joined to this materialized result and a sort to remove duplicate C1 values (given the DISTINCT). Because no columns were required from the UNION ALL of T1 and T2, and that the DISTINCT would remove any duplicates introduced, this join is unnecessary.

The DB2 12 rewrite for the Figure 12 on page 13 example is demonstrated in Figure 13. The view definitions and the query against the views are the same between the two figures. What differs is how DB2 12 is able to prune out the table expression that contains the UNION ALL. The result is simply a SELECT DISTINCT only requiring access to T3.

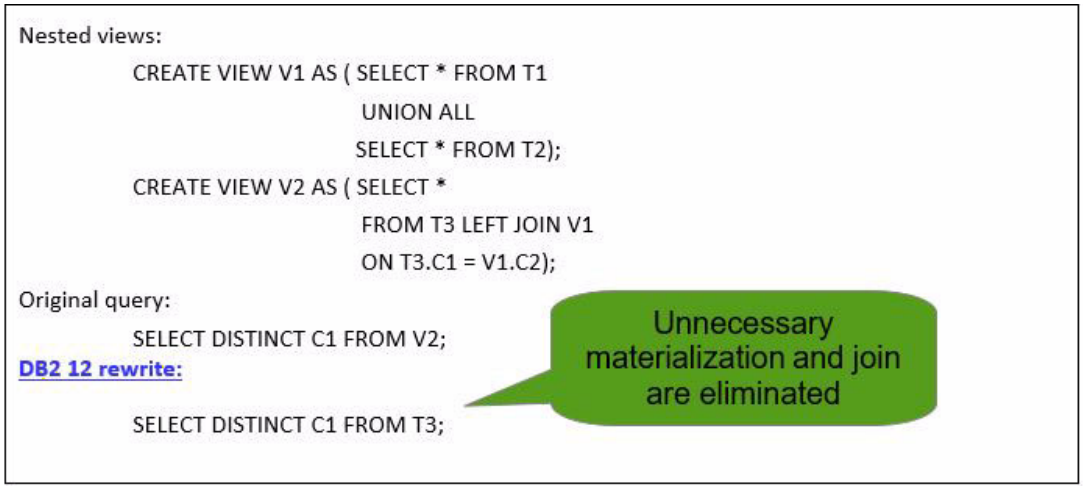


Figure 13 DB2 12 LEFT OUTER JOIN to unnecessary table expression

Push down of ORDER BY and FETCH FIRST

The final DB2 12 enhancement targeted at improving UNION ALL and outer joins with views and table expressions relates to the usage of ORDER BY and FETCH FIRST clauses.

In the example shown in Figure 14, view V1 contains three tables combined with UNION ALL, and then a query against that view includes an ORDER BY 1 (meaning the first column in the select list) and FETCH FIRST 10 ROWS ONLY. Figure 14 shows the view definition, the query, and the merged query in DB2 11.

From a performance perspective, all rows (and columns) from all three UNION ALL legs are retrieved and materialized to a workfile, and then sorted for the ORDER BY. From that sort, the top 10 rows are retrieved. If each of the tables contains 1 million rows, then that materialization and sort involves 3 million rows to retrieve the top 10 rows.

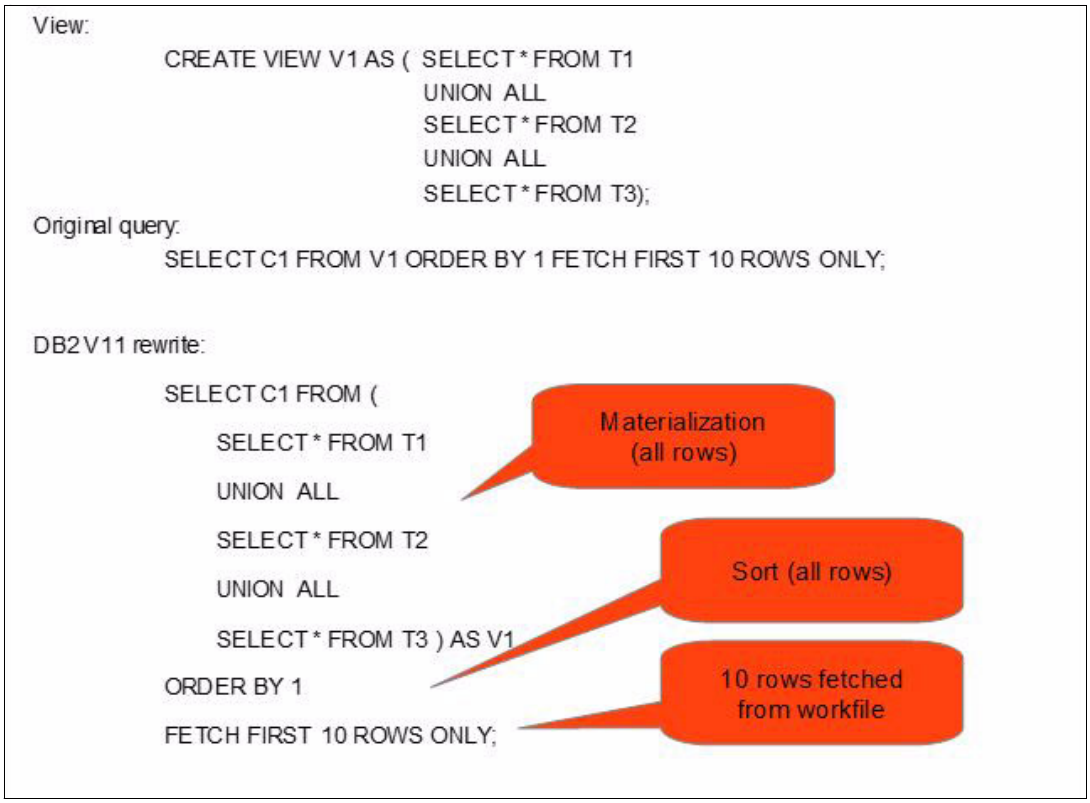


Figure 14 DB2 11 ORDER BY and FETCH FIRST with materialized view

In DB2 12, the ORDER BY and FETCH FIRST clauses are pushed inside each leg of the UNION ALL, as depicted in Figure 15. Also of note as to another difference between Figure 14 on page 15 and Figure 15: In Figure 14 on page 15 (DB2 11), the SELECT list of each UNION ALL leg contains SELECT *, which results in all columns from each table being retrieved and materialized. In Figure 15 (DB2 12), notice that the SELECT * from the view definition has had unnecessary columns pruned, resulting in column C1 only appearing in the merged query in DB2 12.

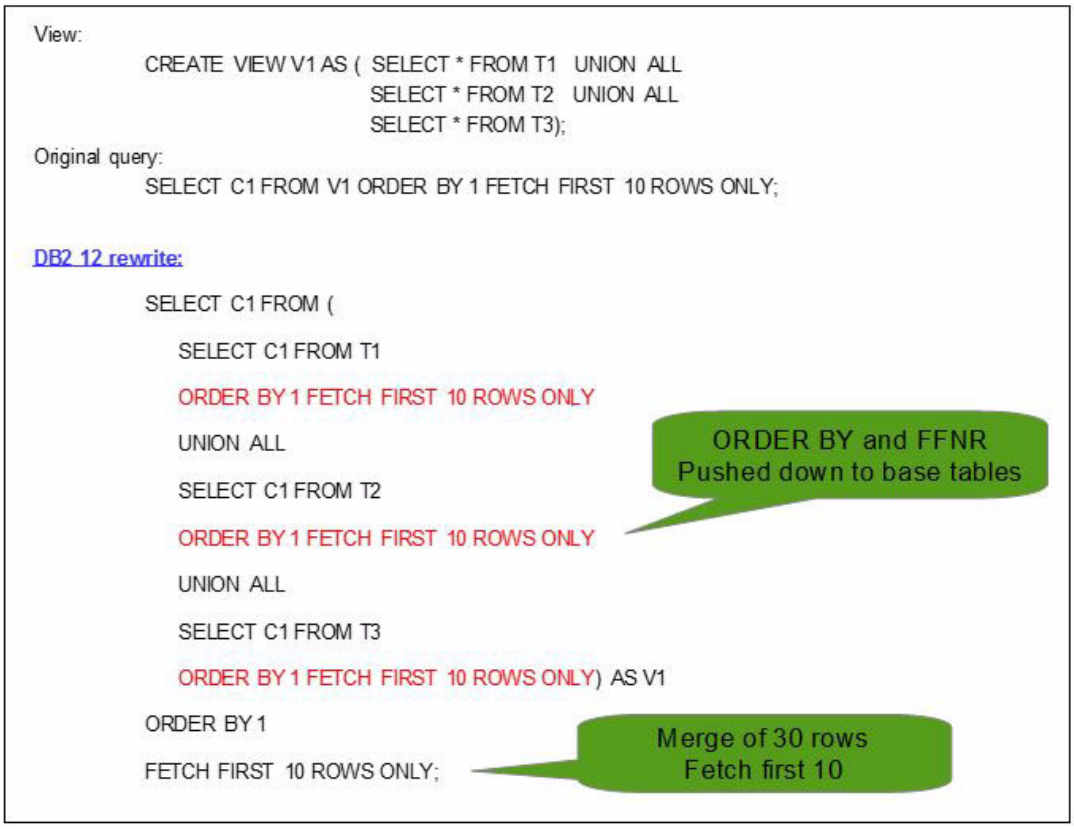


Figure 15 DB2 12 ORDER BY and FETCH FIRST with materialized view

With ORDER BY and FETCH FIRST pushed down to each UNION ALL leg, the DB2 optimizer can now optimize each leg independently. This process includes using an available index to avoid the ORDER BY sort and fetch the top 10 rows without having to retrieve and sort 1 million rows. If no index exists, then sort has in-memory optimizations since DB2 9 for z/OS when the number of rows is guaranteed, as is the case with FETCH FIRST.

Therefore, each leg now can optimize sort or avoid a sort (given an available index) with the ORDER BY and FETCH FIRST on each leg. The final sort merges the three results because it is known that each leg is already in order.

The best-case performance comparison would therefore be DB2 11 sorting 3 million rows, and DB2 12 using indexes in each leg to retrieve access only 10-12 rows total—to retrieve the top 10 rows. Why 12 rows and not 30 rows (3 X FETCH FIRST 10)? The answer is because the DB2 12 enhancement uses a merge process from each leg. Because there are three UNION ALL legs, then each leg will access its first row, and the lowest of those three rows will be returned. That leg will then fetch its next row.

If the top 10 rows all come from one leg, then the maximum number of rows accessed to retrieve the top 10 is 12 rows.

This ORDER BY (and FETCH FIRST) pushdown enhancement is also interesting to demonstrate by using a simpler type of query involving UNION ALL that might appear as a cursor in an OLTP or batch application. Figure 16 shows a simple cursor with UNION ALL and a final ORDER BY, followed by a FETCH that is performed 10 times. Note that there is no FETCH FIRST, and thus the optimizer is not aware of the number of fetches that will occur within the application.

```
DECLARE BB-CURSOR      CURSOR FOR
    SELECT O_ORDERKEY FROM TPCH30.ORDERS
    WHERE O_SHIPPRIORITY < 100
    UNION ALL
    SELECT L_ORDERKEY FROM TPCH30.LINEITEM
    ORDER BY 1 FOR FETCH ONLY                -- Qualifies over 200 million rows.

PERFORM 10 TIMES
    PERFORM 1002-FETCH-CURSOR THRU 1002-EXIT
END-PERFORM

TIMES/EVENTS  APPL(CL.1)  DB2 (CL.2)
------------  ----------  ----------
ELAPSED TIME    0.070273    0.058614
CP CPU TIME     0.002265    0.001554

OPEN             1
FETCH           10
CLOSE            1

BP1    BPOOL ACTIVITY     TOTAL    BP2    BPOOL ACTIVITY     TOTAL
--------------------  --------    --------------------  --------
GETPAGES                     4    GETPAGES                     12

NO SORT WF BP activity
```

Figure 16 DB2 12 UNION ALL in an OLTP-style query

The summarized trace information in Figure 16 shows 1.5 milliseconds of class 2 CPU, 10 fetches, and a total of 16 getpages across BP1 (index) and BP2 (data). Although the DB2 11 results are not shown for this example, the query would qualify over 200 million rows. Because DB2 11 will not push the ORDER BY to each UNION ALL leg, that amount would result in all 200 million rows being sorted and written to workfile, for the application to fetch only 10 rows.

What also is not shown in Figure 16 is the access path chosen in DB2 12, although it can be deduced from the low number of getpages that both legs of the UNION ALL are able to use an index to avoid the ORDER BY sort. And in addition to the ORDER BY pushdown (and FETCH FIRST pushdown when applicable), sort is able to merge the legs across the UNION ALL. What you cannot deduce from this example is how many getpages were required across each leg. This information would require a more detailed getpage trace or separating the objects into more buffer pools. All 10 rows fetched could come from the first leg, the second, or a combination of both.

Summary of targeted UNION ALL and outer join performance enhancements

It was highlighted in the introduction to the UNION ALL enhancements that this construct is appearing more frequently in ERP and customer workloads, and also is the foundation for the recent DB2 features System-Period Temporal and Transparent Archive. The previous sections showed the commitment in DB2 12 to improving UNION ALL performance. This goal is achieved by either avoiding materialization or improving the performance of materialization by piping or merging the materialization result into the next query processing step, and reducing the size of the materialization.

As implied in the examples, where the DB2 11 materialization would process all table rows, avoiding this in DB2 12 could result in only the required rows being processed. This technique results in orders of magnitude performance improvement.

Although most of the examples demonstrated the challenge and then benefit for UNION ALL queries, the DB2 12 optimizations are equally applicable to other materialized views/table expressions with outer joins. Once again, many of the same materialization performance challenges exist for optimal performance of outer join and UNION ALL-based queries. And it is expected that DB2 12 will provide significant performance improvement for these query patterns.

Runtime optimizations

The next DB2 12 target area to be discussed is those enhancements that take advantage of intelligent runtime adaptations, more efficient runtime code path, or both. The predominant DB2 component that delivers the enhancements in this section is the Runtime component within RDS.

As with most of the query performance and optimization enhancements in DB2 12, these are complementary to other features listed. Therefore, the same query can benefit, for example, from a UNION ALL enhancement, a runtime optimization, and several other enhancements listed.

Runtime Adaptive Index

Runtime Adaptive Index extends the existing RID-based access paths List Prefetch and Multi-Index Access, and adds several execution-time enhancements to allow you to adapt at execution time based on index filtering.

The simple example in Figure 17 can be used to highlight the challenge that generic (search) queries place on a query optimizer. Numerous coding techniques are used for this type of SQL. It is commonly used for search screens where the user can fill various fields and select any or all combinations.

Regardless of the technique that is used for the SQL, the premise is the same: The programmer wants to code one SQL that can be used generally. DBAs, however want the programmer to construct dynamic SQL at execution time that is specific to the input request, or to limit the combinations that are possible and code a subset of targeted SQL.

Typically, it becomes the responsibility of the database to address the performance challenges. You can bind the package with the REOPT(ALWAYS) bind option, but this is not always possible if this is coming through DRDA because too many SQLs would use the same package. Similarly, for static SQL, all SQL in the package would be affected.

```
SELECT *
FROM CUSTOMER
WHERE LASTNAME  LIKE ?
  AND FIRSTNAME LIKE ?
  AND ADDRESS   LIKE ?
  AND CITY      LIKE ?
  AND ZIPCODE   BETWEEN ? AND ?
```

Figure 17 Generic search SQL challenge

The SQL in Figure 17 on page 18 is used as a generic search query by “enabling” or “disabling” each predicate based on the user input. For example, if LASTNAME and CITY are entered, then these predicates are populated with their search values, such as LASTNAME LIKE 'SMITH' AND CITY LIKE 'SAN JOSE'. All other predicates are populated with LIKE '%%%%%' (or if the query was coded with BETWEENs for character columns, the predicate would be BETWEEN 'AAAAAAAA' AND 'ZZZZZZZZ') or BETWEEN 0 AND 99999 (for numeric columns).

It is not possible for the DB2 optimizer (or any optimizer) to choose one plan that would best match the filtering predicates because that can change at every execution.

Runtime Adaptive Index is the name that is given to the DB2 12 optimizer (runtime) enhancements that target this type of SQL performance challenge. To describe this, assume that there is one separate index for each column in the WHERE clause, and for this example, the optimizer chose a multi-index plan with each of these five indexes.

At execution time, DB2 run time first looks at the values provided for each predicate to determine whether the predicate is “likely” or “unlikely” to be filtering. LIKE predicates that contain wildcards such as '%' are considered unlikely to be filtering, and BETWEEN or other range predicates that appear to cover the entire range of values are also considered unlikely filtering candidates. At this stage, run time will move unlikely filtering indexes to the end of the multi-index execution sequence.

The run time begins processing the first “likely” filtering index based on the original index operation sequence (identified in the PLAN_TABLE by the MIXOPSEQ column (Multi-Index OPeration SEQuence)) after “unlikely” filtering indexes are moved to the end.

After a certain internal threshold is reached when processing the first filtering index, processing of that index is paused and the other remaining indexes are also searched (up to a similar threshold), which will allow the approximate filtering of each index to be determined. The threshold that is associated with the first index is implemented to ensure that short-running queries are not affected by any cost associated with validating the filtering of other indexes.

After the filtering of each remaining index is determined, the run time can reorder the index execution sequence, discard any non-filtering indexes, or revert to table space scan if there are no indexes that provide sufficient filtering. If an index plan is to continue, then processing of each index continues where it was left off, and in the order of best filtering. Because this process is built on existing multi-index access, each leg is index-only as it is accumulating qualified RIDs that are then intersected with the RID list from each leg. The data rows is then accessed.

Therefore, runtime adaptive index is able to adjust/adapt at execution time to changes in actual filtering without requiring REOPT(ALWAYS) because REOPT would result in a reoptimization of the access path. REOPT(ALWAYS) is a “bind” parameter that indicates at execution time that DB2 should choose a new access path each time the query executes (that is, reoptimize the query). This reoptimization adds overhead to each query execution. This adaptive index solution avoids that reoptimization and thus avoids that extra cost.

At execution time, the system performs a “first” quick peek at the literal values and adjustment of the index execution order or reversion to table space scan based on determination that certain index legs are not filtering. A more accurate filtering estimate is determined at execution after a threshold is reached. This adaptability is applicable to all list prefetch and multi-index access plans.

For multi-index ANDing, adaptive index is able to reorder the indexes from most to least filtering, early-out (discard) of non-filtering indexes, and revert to table space scan earlier than prior releases with insufficient filtering. For multi-index ORing or list prefetch-based plans, this early determination of filtering at two stages (first from peeking at the literals and second after the internal threshold) allows a decision to fall back to table space scan earlier than prior releases if there is no adequate filtering from the selected indexes.

Runtime adaptive index is not only for the generic search SQL challenge. The optimizer is not able to identify the intent of an SQL statement. The adaptive capabilities are applicable any time that the optimizer chose a RID-based plan, such as list prefetch or multi-index access.

The obvious question is whether this capability means that DB2 12 will see a significant increase in multi-index access plans. However, this change is not expected because the optimizer only increases its consideration of multi-index access when the candidate predicates have high uncertainty in their filter factor estimates. Predicate uncertainty was added in DB2 10 to supplement the optimizer's filter factor estimate.

By design, the predicates that have the highest uncertainty in their filtering are range predicates, which could filter anywhere from 0 - 100% of the range based upon the literal values used. Other predicates that can be difficult to estimate include JSON, spatial, and index on expression predicates.

Figure 18 provides a simple example of three range predicates, each with an available index, and filtering with high uncertainty until execution time when the literal values are known. This type of query becomes a good candidate for multi-index access and its adaptive capabilities in DB2 12.

```
SELECT *
FROM TAB1
WHERE COL1 < ?
  AND COL2 < ?
  AND COL3 < ?;
INDEXES: IX1(col1), IX2(col2), X3(col3)
```

Figure 18 SQL with high predicate uncertainty

Performance measurements for adaptive index can range from 20% CPU saving for list prefetch or multi-index ORing when failover to table space scan is the most appropriate choice, and orders of magnitude performance improvement for multi-index ANDing with a mix of filtering and non-filtering indexes. No measurable regression was found when the existing multi-index plan had appropriate filtering.

Internal runtime and complex expression optimization

In each recent DB2 release, an effort was made to improve the execution performance of expensive operations by using machine code generation. Historically, SPROCs (Select PROCedures and other xPROCS – also referred to as fast column processing) are the most notable example of machine code generation in DB2.

Customers rely on SPROCs for their noticeable reduction in CPU after static SQL undergoes a REBIND in each new release. Some more recent examples of machine code generation enhancements include being (non-matching) IN-list and OR predicate evaluation in DB2 10, DECFLOAT data type processing, and SUBSTR scalar function evaluation in DB2 11.

Because optimizations based on machine code generation are not displayed externally in the explain output, it can leave customers questioning where a performance improvement came from or how to predict that such an improvement is possible. The latter is more difficult because there is no way to predict it, but answering the question after a performance improvement is seen is easier by explaining what improvements are available in each release.

DB2 12 applies the technique of machine code generation to the following items:

- CASE statements
- Extending the DB2 11 SUBSTR enhancements to all data types
- External data type conversions for DATE-based data types

An additional internal optimization is block fetch for moving qualified rows across Data Manager (stage 1) and RDS (stage 2). This data is recorded in the DSN_DETCOST_TABLE when it is determined that the optimizer has enabled internal block fetch. This process involves internally moving blocks of rows (similar to the external multi-row fetch) rather than row by row.

The next runtime optimization is referred to as expression sharing, and is best described by the example in Figure 19.

When an expression is coded within a view or table expression, and a referencing select is merged with that view/table expression, then that outer reference is replaced with the expression. Figure 19 provides an example of a SELECT from a table expression (TX) that references an arithmetic expression four times. When the referencing SELECT is merged with the table expression, the result is that expression is repeated four times (once for each outer reference).

```
SELECT SUM(C1), AVG(C1), MAX(C1), MIN(C1)
 FROM (SELECT DECIMAL(C1,9,2) * 0.75/DECIMAL(C2,9,2) AS C1
       FROM T) AS TX
Is merged to:
SELECT SUM(DECIMAL(C1,9,2) * 0.75/DECIMAL(C2,9,2))
      ,AVG(DECIMAL(C1,9,2) * 0.75/DECIMAL(C2,9,2))
      ,MAX(DECIMAL(C1,9,2) * 0.75/DECIMAL(C2,9,2))
      ,MIN(DECIMAL(C1,9,2) * 0.75/DECIMAL(C2,9,2))
FROM T
```

Figure 19 DB2 11 expression merge example

From a performance perspective, that also means that in DB2 11, the expression is run four times.

In DB2 12, when the merge occurs, DB2 recognizes that the expression is repeated. At execution time, the expression is executed only once and the result shared across the four references. This technique only applies to expressions where DB2 merges the query. If the query is written with multiple occurrences of the same expression, then DB2 does not detect and share the expression.

The term expression is often used to reference any type of arithmetic, scalar function, or user-defined functions (UDFs). Of these expressions, UDFs are often the most expensive given the flexibility in what can be contained within, including native SQL that references other tables or external UDFs that contain an external application program.

DB2 12 also enhances the performance of deterministic UDFs by supporting caching of the UDF result given the same inputs. This enhancement is achieved with a hash table that is entirely managed by DB2, which means there is no user control. For a UDF that is defined as DETERMINISTIC and NO EXTERNAL ACTION, DB2 builds a hash key from the input variables and stores the output result.

Each UDF invocation first looks up the cache to see whether those values have already been processed. DB2 monitors the cache at regular intervals. If there have been no situations where a cached value has been found, then the cache is disabled to avoid performance regression. The cache only persists within a single query execution, and is not shared by other queries running the same UDF or for the next execution of the same SQL.

Sort, workfile, and sparse index improvements

Sort and workfile usage are areas in DB2 where there is continual focus on improving performance and reducing resource consumption. The workfile buffer pool and data sets can be a point of contention for workload scalability because OLTP, batch, and other queries can all converge on the same resources. This reason is why DB2 has continually sought to improve sort avoidance, optimize sort processing, and have smaller sorts that are processed in-memory.

Materializations also converge on the workfile buffer pool and data sets, and thus the reduction in materializations for UNION ALL and outer join queries can reduce this contention significantly for candidate workloads. The (ORDER BY and) FETCH FIRST pushdown was referenced also as one of the optimizations related to UNION ALL. It was mentioned in that section that pushing these down to the UNION ALL leg could result in DB2 taking advantage of other new or existing optimizations.

FETCH FIRST (when combined with DISTINCT, GROUP BY, or ORDER BY) has had numerous sort performance improvements since DB2 9. These improvements are made possible because DB2 is given precise knowledge about how many rows need to be sorted and returned.

Sort has previously implemented an in-memory replacement technique for sort with FETCH FIRST if the result is guaranteed to be in-memory.

Partial sort-avoidance for FETCH FIRST

Although DB2 is already able to avoid many sorts with an index that provides the required order, DB2 12 introduces the concept of partial sort-avoidance with FETCH FIRST when order is provided for the leading columns only.

In the example provided in Figure 20, there is an index on C1, and the query has an ORDER BY C1, C2 with FETCH FIRST 10 ROWS ONLY. Because the ORDER BY contains more columns than the available index, a sort cannot be avoided. However, in DB2 12, the optimizer can recognize that, if INDEX1 on C1 is chosen, then rows are to be passed into sort in C1 order. Therefore, the sort can stop fetching rows after the order for both C1 and C2 can be guaranteed to be within the number of rows requested by FETCH FIRST.

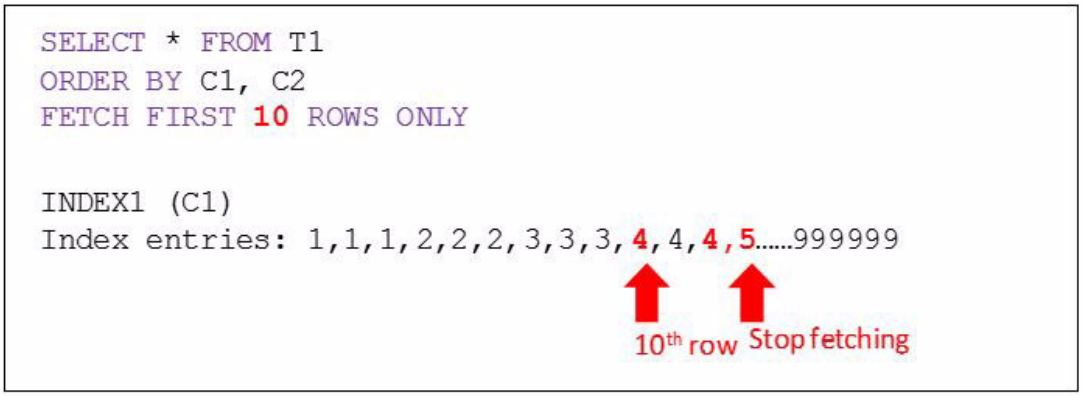


Figure 20 DB2 12 partial sort-avoidance with FETCH FIRST

Given the index entries in Figure 20, the 10th row in C1 order is highlighted. After sort fetches the 10th row (which corresponds to the FETCH FIRST 10 ROWS ONLY), then sort only needs to fetch until the next change of C1 value.

In this example, the 13th row has value 5, and this signals to sort that this 13th row is not needed because the top 10 rows in C1, C2 order are guaranteed to be found in the first 12 rows. Those 12 rows can be sorted and the top 10 returned as requested by the query. Contrast this process to DB2 11, where all rows would need to be fetched and read by sort when a sort cannot be avoided.

Extensions to sort avoidance for OLAP functions

Sort avoidance is also extended to OLAP functions that combine PARTITION BY and ORDER BY. Although DB2 11 already supports sort avoidance, if an index matches the ORDER BY clause with an OLAP function (such as RANK), that did not apply for sort avoidance when PARTITION BY was involved. The example provided in Figure 21 highlights an appropriate index that could be used in DB2 12 to avoid the sort for this SQL statement.

```
SELECT L_SUPPKEY, L_SHIPDATE,
       RANK() OVER(PARTITION BY L_SUPPKEY
                   ORDER BY L_SHIPDATE) AS RANK1
FROM LINEITEM;

CREATE INDEX SK_SD_1 ON LINEITEM(L_SUPPKEY, L_SHIPDATE);
```

Figure 21 Sort avoidance for OLAP functions with PARTITION BY clause

Sort space reductions and in-memory exploitation

When a sort cannot be avoided, then using memory to process the sort or reducing the length of the sort can result in that sort being contained in-memory or at a minimum reducing the amount of workfile resources that are needed to complete the sort.

DB2 11 avoids the final write to workfile for the last sort in the query if that result can be contained in the sort tree. In DB2 12, this feature is extended to many intermediate sorts, such as sorts for joins.

In DB2 11, the sort tree size is limited to 32,000 nodes. Although the amount of memory for sort can be controlled and therefore increased with zparm SRTPOOL, this limit of 32,000 nodes remained. However, in DB2 12, sort allows up to 512,000 nodes for non-parallelism sorts or 128,000 nodes for a sort within a parallel child task. These amounts are still capped by SRTPOOL zparm.

With the ability to contain sorts within the sort tree rather than writing the result to workfile, increasing zparm SRTPOOL is beneficial. This is a per-query zparm allocation that depends on the size of each sort and the number of concurrent queries that dictates how much memory is required across an entire system for sorting.

Increasing the number of nodes of the tree not only has a benefit to DB2 12 of containing a sort in-memory, but GROUP BY and DISTINCT sorts can take further advantage of a larger number of nodes. DB2 9 added hashing support as input to sort for duplicate removal (GROUP BY/DISTINCT), such that duplicates could be collapsed before going through sort. And in DB2 the number of hash entries is tied to the number of nodes of the sort tree.

Therefore, increasing the number of nodes can result in higher cardinality GROUP BY/DISTINCT results, consolidating the groups as rows that are input to sort. This feature can increase the chance that the sort can be contained in memory or at least reduce the amount of workfile space that is required to consolidate duplicates during the final sort merge pass.

Therefore, increasing zparm SRTPOOL can have a greater benefit to improving sort performance in DB2 12 than prior releases. However, increasing SRTPOOL in DB2 11 might not result in that memory being used if the 32,000 limit is reached before the full memory request is allocated. Therefore, DB2 12 can use the memory up to the zparm SRTPOOL value that might not have been used previously.

The next sort enhancements relate to reducing the length of the sort row. Reducing the sort row length has an obvious benefit in reducing the amount of memory and workfile space that is needed for sort.

It is common that predicates coded in the WHERE clause are redundantly included in the SELECT list. And any redundancy in the sort key or data row has a negative impact on sort performance and resource consumption. ORDER BY sort already removes columns from the sort key if covered by equals predicates in the WHERE clause. DISTINCT or GROUP BY already removes redundant columns for sort avoidance.

But if a sort is required for DISTINCT or GROUP BY, such redundant columns remain until DB2 12, when they are removed from the sort key as shown by the simple example in Figure 22. In that example, because C1 has an equals predicate in the WHERE clause, all sorted rows are guaranteed to contain that same value, and thus only C2 is needed for the sort.

```
SELECT DISTINCT C1, C2
FROM TABLE
WHERE C1 = ?
```

Figure 22 Redundant columns in sort key

Similarly, in DB2 12, sort avoids duplicating the sort key from the data portion if they are an exact match of each other. That statement opens up an obvious question: Why would sort duplicate the data into the sort key? To answer this question, you need to understand briefly how sort works. Sort operates on a fixed-length concatenated key of the columns required to support the ORDER BY, GROUP BY, DISTINCT sort, or sort for join.

The requested order of the sort key might not match the order of the columns in the SQL's SELECT list. In DB2 12, when the sequence of columns in the sort matches the SELECT list, and contains only fixed-length columns, then sort will not replicate the key. Instead, DB2 12 uses the data portion for the sort. This feature can dramatically reduce the sort row length.

Another goal in DB2 12 is to minimize the impact that larger sorts have on OLTP applications. Using memory for sort helps reduce contention on sort buffer pool and workfiles as a shared resource. Although DB2 9 added the option for sort to use 32 K workfile pages for longer sort rows (greater than 100 bytes), it is still possible for very large sorts of smaller-length rows to dominate the 4 K page buffer pool and workfile data sets.

In DB2 12, many larger sorts of greater than 10,000 rows can now be compacted into the 32 K workfile rather 4 K. This feature might result in an increased requirement for 32 K workfile allocations compared to 4 K, but it does improve separation of longer-running sorts from OLTP. It can improve performance with reduced getpages by using an eight times larger page size.

Sparse Index improvements

Sparse index has continually been enhanced in recent DB2 releases to provide similar support to hash join that is available in most competitive DBMSs. DB2 12 adds some incremental improvements to extend the support of the VARGRAPHIC data type, and also for better memory exploitation.

The first memory-related sparse index enhancement includes improved allocation of memory across multiple usages of sparse index within the same query. In prior releases, there was a high dependency on the optimizer estimate to determine the allocation of memory for each sparse index. In DB2 12, the sort component improves its algorithms to adjust the type of sparse index that is built to optimize memory, and also to reduce getpages when a sparse index must overflow to workfile.

When building a sparse index, sort also attempts to trim the information that must be stored. Similar to sort key length reductions, sparse index can avoid repeating the key from the data if these are equal and of fixed length. And when the key prefix is the same for all rows in the sparse index, the key is truncated to avoid key duplication. And for variable length rows, trailing blanks can be truncated to reduce the amount of memory that is required to contain the sparse index key.

Because sparse index also converges on the sort workfile buffer pool and data sets, any reductions in workfile usage for sparse index has a similar benefit to improved workload scalability by alleviating workfiles as a bottleneck for DB2 applications.

Predicate optimizations

The stage at which DB2 is able to evaluate predicates can have a significant impact on SQL performance. It is well understood that indexable and stage 1 predicates allow filtering to occur at an earlier stage and can use indexes to restrict the search range.

And although limiting the search range can provide orders of magnitude performance over scanning the entire object, if the only filtering comes from a stage 2 predicate, then the performance of that query or application will not be acceptable and deployment of that query or application usually fails. Improving the performance of stage 2 predicates or allowing predicates to become indexable/stage 1 can thereby enable successful application deployments. Applications or queries where there is other filtering from indexable/stage 1 predicates are less affected by stage 2 predicates.

Sort for stage 2 join expressions

The first use case of stage 2 predicates that are targeted in DB2 12 involve stage 2 join predicates. Numerous enhancements in recent releases improve the performance of stage 2 predicates. However, stage 2 join predicates remain as a performance challenge, more specifically stage 2 join predicates on the inner table of a join. Figure 23 contains a simple example of a stage 2 join predicate.

```
SELECT *
FROM T1, T2
WHERE  T1.COL1 = T2.COL1 and
       T1.COL2 = SUBSTR(T2.COL2,1,10)
```

Figure 23 Stage 2 join predicate

DB2 12 can improve the performance of stage 2 join predicates by allowing sort to evaluate the function. This process allows sparse index to be built on the result for the join, which becomes an optimal candidate on the inner table when the result can be contained in-memory, or when there is no other viable index to support the filtering of the join. Alternatively, if the table with the join expression is the outer table of the join, a sort for join order can allow access to the inner table to be performed sequentially.

Stage 2 join predicates are often observed if tables are not designed with consistent data types for joined columns. This situation might occur if applications are integrated later, or if business information is embedded within columns, or if time stamp columns are used within each table and the join is by the consistent date portion of those columns (for example, insert time stamps do not match between two tables). Therefore, DB2 12 can improve performance significantly for these situations without requiring a targeted index on expression to be built.

User-Defined table function predicate optimizations

User-defined table functions (also known as table UDFs, table functions, or TUDFs) were initially targeted to allow an application program to be called from within an SQL statement. This feature provided the flexibility to access non-DB2 objects and represent them as a table within SQL to be joined with DB2 tables.

Inline table UDFs were a further extension to DB2 support, allowing the definitions to contain native SQL. There has been an increase in table functions as an alternative to views because of the capability to create a table function with input parameters, whereas parameterized views are not supported in DB2 for z/OS.

Although prior releases already provided similar merge (and thus materialization avoidance) capabilities for table functions that were syntactically equivalent to views, DB2 12 has improved both the merge of deterministic table functions with input parameters and also improved indexability of input parameters as predicates within the table function, as demonstrated in Figure 24.

Simplified table function definition:

```
CREATE FUNCTION TPCHUDF(in_date1 CHAR(10), in_date2 CHAR(10))…
SELECT O_ORDERPRIORITY, COUNT(*)
FROM ORDERS
WHERE  O_ORDERDATE >= DATE(in_date1)    <<< Was STAGE 2. Now indexable
   AND O_ORDERDATE <  DATE(in_date2)    <<< Was STAGE 2. Now indexable
```

Query referencing table function:

```
SELECT * FROM table(TPCHUDF('1993-07-01', '1993-10-01')) as TAB1;
```

Figure 24 Table function with input variables

VARBINARY indexability improvements

Variable-length binary (VARBINARY) is a more recent data type added to DB2 for z/OS. Increased adoption of any new feature typically clarifies customer usage patterns and therefore identifies opportunities for improvement. DB2 12 adds indexability support for mismatch length comparisons of BINARY and VARBINARY data types, which were previously stage 2.

Comparisons of the same length were already indexable. This feature is depicted in the example in Figure 25 that compares the pre-DB2 12 predicate as stage 2 with a CAST added in DB2 12 to support indexability of mismatched length VARBINARY. Mismatched length BINARY is not shown.

```
SELECT A.C_CUSTKEY, HEX(A.C_NAME)
FROM CUSTOMER A, CUSTNULL B
WHERE A.C_NAME > B.C_NAME;

CUSTOMER.C_NAME: VARBINARY(25) NOT NULL
CUSTNULL.C_NAME: VARBINARY(30)

Pre DB2 12: A.C_NAME > B.C_NAME                              (stage 2)
DB2 12    : CAST(A.C_NAME AS VARYING BINARY(30))>B.C_NAME (Indexable)
```

Figure 25 Indexability for mismatched length VARBINARY

Although many customers do not use VARBINARY or BINARY data types in their environments, improving indexability is important because some DB2 scalar functions return BINARY or VARBINARY results. The improvements to the underlying support of BINARY and VARBINARY indexability were necessary to allow index on expressions to be built on those scalar functions and to be used for matching index access.

Figure 26 provides an example of a scalar function as an index on expression that is now indexable in DB2 12. This example demonstrates the COLLATION_KEY scalar function with a parameter tailored to German.

```
CREATE INDEX EMPLOYEE_NAME_SORT_KEY ON EMPLOYEE
(COLLATION_KEY(LASTNAME, 'UCA410_LDE', 600));

SELECT  *
FROM   EMPLOYEE
WHERE  COLLATION_KEY(LASTNAME, 'UCA410_LDE', 600) = < ?
```

Figure 26 Index on expression example for VARBINARY-based expression

Row permission indexability for correlated subqueries

At this point, there should not be a need to further justify the performance benefit of indexable predicates as compared to stage 2. Instead, this section highlights other examples where indexability has been improved in DB2 12.

Row permissions improve data security at a lower level of granularity and thus become an attractive solution for integrating that security into the database system rather than requiring application control or implemented through views. Efficiency of security validation is paramount, so this enhancement in DB2 12 to resolve indexability of correlated subquery predicates within a row permission that were previously stage 2.

A correlated subquery example of a row permission is shown in Figure 27.

```
CREATE PERMISSION RP1 ON LINEITEM T1 FOR ROWS
WHERE EXISTS
(SELECT 1 FROM ORDER T2
 WHERE T1.ORDERKEY=T2.ORDERKEY) ENFORCED FOR ALL ACCESS
ENABLE
- Insert row into table LINEITEM
- UPDATE LINEITEM SET COMMENT='?' WHERE ORDERKEY = ?;
```

Figure 27 Correlated subquery predicates in a row permission

Optimizer cost model enhancements

Moving through the internal DB2 components where the performance enhancements in DB2 12 are focused, the next area for discussion is the optimizer cost model. Within DB2 development, this is the access path selection (APS) component. The optimizer cost model (access path selection) is responsible for choosing the lowest-cost access path based on the query after query transformation has performed view/table expression merge and predicate transformations (for example, pushdown and rewrites).

The previous section that presented the internal DB2 performance workload results highlighted that most queries and workloads that achieved the largest performance gains were those that chose a new access path in DB2 12 compared with DB2 11. This improvement might be due to these reasons:

- A new access path became available due to a query transformation (such as a predicate rewrite, predicate pushdown, or view/table expression merge).
- A new execution path became available, an existing access path was optimized to be more efficient (DB2 12 Runtime Adaptive Index for example).
- The optimizer cost model has better access to accurate inputs or improves upon its cost formulas.

It would be incorrect to infer from the groupings of topics in this overview that each of the DB2 components operates independently. In fact, each component plays a role in providing an efficient execution for a SQL statement. Query transformations and runtime optimizations have already been described, so we can assume that the optimizer cost model played its role in ensuring an efficient access path based on those enhancements. This section describes the improvements specific to the cost model that take advantage of existing choices.

Extending NPGTHRSH to default statistics

A discussion around zparm NPGTHRSH is unlikely to garner the same immediate understanding compared to a discussion of the VOLATILE table attribute. Both are trying to solve the same problem, which is that there are instances where RUNSTATS collected on an object might be unreliable for one of these reasons:

- That object is early in an application rollout (meaning it is small but likely to grow quickly)
- The object size grows and shrinks regularly, making it difficult to collect representative statistics at the right point in time.

The zparm NPGTHRSH was delivered in DB2 V7 to prefer matching index access over other access paths at the subsystem level. And DB2 12 extends the effectiveness of this zparm.

When zparm NPGTHRSH is set, this zparm value is compared with the number of pages for a table This number is the NPAGESF catalog statistic, or at the partition level, it is the number of pages in the partition that is compared to NPGTHRSH. If NPAGESF is less than NPGTHRSH, then the optimizer prefers matching index access for access to that table, if possible.

NPGTHRSH is disabled by default, although there is at least one major ERP vendor that recommends DB2 for z/OS customers set their default to 10. This setting has been effective for many years. No customer complaint of the wanting a table space scan and DB2 choosing matching index access for a 9 (or less) page table has been received.

In DB2 11 and prior, if statistics were not collected on a table, and thus NPAGESF=-1, DB2 did not use -1 for the NPGTHRSH comparison. Instead, and as documented, -1 becomes 501 for all optimizer costing, including the comparison to NPGTHRSH.

DB2 12 instead allows default stats (-1) to apply the NPGTHRSH rules, if enabled. Enabling NPGTHRSH in DB2 12 also applies the rule for preferring matching index access if only the index being considered has default statistics. This scenario is possible if an index was recently created but statistics are not yet collected.

Although this might be considered a minor change, enabling zparm NPGTHRSH to a small value (even setting to 1) can result in the optimizer avoiding table space scans on very small or empty objects that might simply have statistics that are not representative and are actually larger than the statistics demonstrate.

List prefetch and hybrid join cost improvements

Neither list prefetch nor hybrid join are new to DB2. What is new are improvements to the optimizer to encourage list prefetch and also hybrid join for poorly clustered objects when sort avoidance is not a viable candidate as one of the lower-cost access paths.

The largest complaint with list prefetch is that, if chosen for an online application SQL, list prefetch degrades performance significantly because it accumulates all qualified RIDs before fetching the rows to return to the application. To be more specific, when the query contains an ORDER BY and an index exists to avoid the sort, then list prefetch can be a dangerous access path to choose for this reason for an online window.

The other complaint is if the optimizer underestimates the filtering of a predicate and a large percentage of the table qualifies, then list prefetch is less efficient compared with a table space scan. The introduction of the Runtime Adaptive Index enhancements for multi-index access and list prefetch allows the optimizer to improve the costing associated with list prefetch, multi-index, and hybrid join.

The optimizer now better reflects the improved I/O performance available for such choices when a sort avoidance plan is not one of the viable lower-cost plans.

In addition to the cost model improvements associated with list prefetch, hybrid join support (which uses list prefetch) has been extended as a viable optimizer choice in more situations when parallelism is enabled. Hybrid join has the benefit of more efficiently accumulating list prefetch requests to access the inner table of a join as compared with nested loop join with list prefetch.

Improved filter factor for CURRENT DATE/TIMESTAMP predicates

Range predicates with parameter markers or host variables are among the most difficult predicates for a query optimizer to estimate accurately. It is possible at execution time to specify a range that qualifies anywhere from 0 to 100% for a simple range predicate.

Although the available catalog statistics for a column identifies the ranges that exist within the table, how much of that range qualifies cannot be known until the literal value is supplied at execution time. A predicate ORDER_DATE < ? is more difficult to accurately estimate compared with predicates with literal values such as ORDER_DATE < '2016-01-01' or ORDER_DATE < '9999-12-31'.

DB2 12 supports resolution of predicates involving CURRENT DATE and CURRENT TIMESTAMP to use the actual current value as at the bind/prepare time. This support includes date/time stamp arithmetic that incorporates those special registers. In DB2 11 and prior, ORDER_DATE < CURRENT DATE would use the same filter factor estimate as that of a parameter marker/host variable (ORDER_DATE < ?).

Figure 28 provides predicate examples that resolve the predicate values at bind/prepare time to provide a more accurate filter factor estimation in DB2 12.

```
SELECT * FROM T1
WHERE ORDER_DATE > CURRENT_DATE - 7 DAYS
  AND ORDER_DATE < CURRENT_DATE;
```

Figure 28 Predicates involving date special registers

The obvious question from this enhancement is whether the filter factor estimate will become stale because the values are resolved at bind/prepare, and a static bind might have been performed six months ago and not repeated. Although it is true that the current date six months ago is not the same as the current date today, it is likely that the data in the table has also continued to change over time such that a six-month move in current date also corresponds to six months of additional rows added/updated to the table.

Improved resolution of filter factors at bind/prepare using index probing

Index probing was added in DB2 10 for dynamic prepares with literals (including REOPT(ONCE)) and static binds that use REOPT(ALWAYS). For index probing, DB2 uses the predicate value to probe the index non-leaf pages if a table is volatile or if the original predicate estimate is assumed to qualify zero rows. Index probing allows the optimizer to validate the estimated filter factor based on the data existing in the index. This index probing process also accesses the RTS information for clarity on the current object sizes.

DB2 12 improves the performance of index probing and allows the resultant filter factor to be used more consistently for the benefit of improved cost estimation.

Improved bind/prepare performance with many indexes

Some recent deployments of ERP applications have included a very large number of indexes per table. This configuration is typically based on a design where each function of the application maps to a set of columns of the table. Therefore, which indexes are beneficial to the user depends on the features used.

A query optimizer reads in all available catalog information, including object definitions and statistics for a SQL statement, as input to the bind/prepare process to allow cost comparison of available choices to determine the lowest-cost access path. In DB2 11 and prior, the optimizer would evaluate every available index for each object. This approach presented performance challenges with multi-table joins involving hundreds of indexes per table. Such performance issues have not previously been a concern given that most tables have 10 indexes or fewer.

In DB2 12, the optimizer first evaluates all available indexes for each table, and rank indexes based on the existence of matching predicates, screening predicates, and clustering attributes. Indexes are removed from consideration for the access path if there is no filtering value, and also removed when better filtering indexes are available.

This feature can improve bind/prepare performance when there are many indexes on a table. It also can improve the chances that the index with the most filtering is chosen by the optimizer. Figure 29 provides an example of an SQL statement with 12 indexes on the table. In DB2 11, all 12 indexes would be evaluated by the optimizer throughout the access path selection process. In DB2 12, eight of those indexes would be discarded before beginning the access path selection process.

Figure 29 DB2 12 discarding unnecessary indexes from consideration by the optimizer

RUNSTATS and optimizer-generated statistics profiles

Although the focus is on the optimization and performance enhancements in DB2 12, RUNSTATS provides the most critical input to the optimizer and is therefore an important topic to discuss. This section outlines improvements to RUNSTATS that help customers determine what to collect and integrate statistics collection with their existing maintenance processes.

DB2 12 includes a minor improvement to the clusterratio formula that reduces the impact that unclustered inserts have on the clusterratio of the clustering index. This feature improves the alignment that the clusterratio formula has with the execution behavior of dynamic prefetch, and can extend the life between table space REORGs triggered by stored procedure DSNACCOX based on clusterratio. There is no formal requirement to collect statistics using the new formula before static rebinds or execution of dynamic SQL in DB2 12.

After you determine what statistics to collect, you need to determine how many frequencies to collect. The default is to collect the top 10, but the truth is that only the top 1 might be skewed, or the top 20 or 50. There is no single default that is applicable to all columns and all tables: It is entirely data dependent.

DB2 12 provides the capability to allow RUNSTATS to continue collecting until the data is no longer skewed, with an upper limit of the top 100 values to avoid over-collection. This capability avoids situations where too few values were collected for the optimizer to estimate filtering correctly, and removes the requirement that the DBA specify an appropriate value.

Dynamic statement cache invalidation changes with RUNSTATS in DB2 12. If you run the RUNSTATS utility, then historically, any SQL statements that access those tables/indexes will have their access paths reset. In DB2 12, this “reset” is optional, and can be disabled using the RUNSTATS (INVALIDATECACHE) option.This now becomes optional with RUNSTATS to invalidate the cache by using the INVALIDATECACHE option with the default set to NO. This is a behavior change compared with prior releases of DB2, and is intended to reduce the cost of new prepares when RUNSTATS is run.

Similarly, utilities such as LOAD or REORG will only invalidate statements in the cache if the object is in a pending state before the utility was run. The theory behind these behavior changes is that it is expected that a REORG or RUNSTATS is not necessarily run because of a bad access path, and thus, those utilities run for other purposes should not disrupt the existing access path. RUNSTATS with UPDATE NONE REPORT NO is still an effective method to flush statements from the dynamic statement cache.

Another RUNSTATS enhancement in DB2 12 is further integration of the DB2 optimizer recommendations with RUNSTATS. In DB2 11, the optimizer would externalize to a catalog table and optionally to an explain table, with RUNSTATS recommendations based on the identification of missing statistics or conflicting statistics that can benefit the SQL being bind/prepared/explained. In DB2 11, an extra manual step was required to convert those recommendations into RUNSTATS statements to be run.

In DB2 12, the RUNSTATS PROFILE will automatically be updated when new recommendations are made by the optimizer. If a profile does not already exist, then a new profile is created that merges the existing statistics in the catalog with the new recommendations.

To enable the usage of the optimizer recommendations, the customer simply needs to specify USE PROFILE, which is now supported for inline stats in DB2 12. Therefore, all methods of DB2 RUNSTATS collection are now supported for using profiles. Figure 30 provides a visual overview of the RUNSTATS integration steps with the optimizer in DB2 11 and DB2 12.

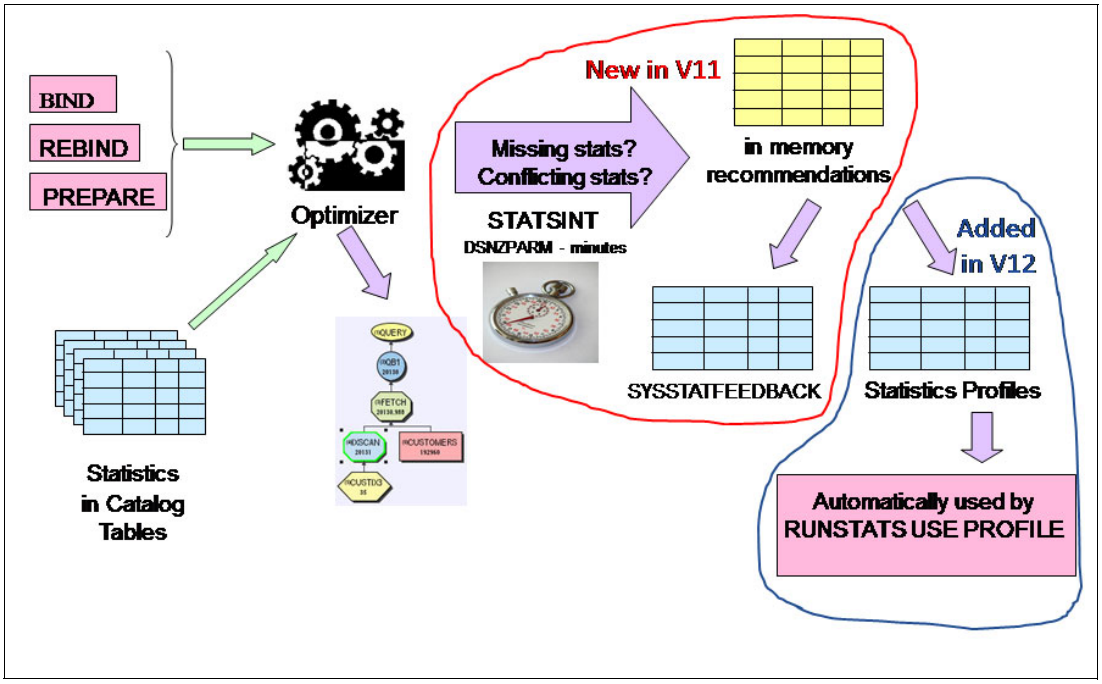


Figure 30 Optimizer externalization of missing statistics

Updates to a profile trigger a recommendation from DSNACCOX that RUNSTATS should be run for that table. These enhancements are a significant step forward in RUNSTATS simplification and integration with existing maintenance processes.

Static and dynamic plan stability

Although the delivery of performance improvements is critical for ensuring that application performance continues to meet service-level agreements and user expectations, customers also want stability and reliability of that performance. And if you have been reading all of the prior sections in this DB2 12 optimizer overview, you would see the attention to all aspects of improving query performance, from RUNSTATS integration with the optimizer for ensuring appropriate inputs to access path selection, query, and predicate transformations.

These features mean that the optimizer has the best available choices, cost model enhancements, and runtime improvements. These have been consistent themes through the recent DB2 releases.

However, complementary to this has been the focus since DB2 9 on allowing fallback to a prior access path after a static REBIND, or to reuse a prior access path for a new BIND/REBIND from DB2 10. These static plan stability (or plan management) features have alleviated concerns for customers across DB2 release migrations when rebinding many static packages and ensuring minimal risk to their business from performance regression.

DB2 12 delivers further usability features to static plan stability, and also extends the benefit of basic stability to dynamic SQL that static SQL has always enjoyed.

Static plan stability usability improvements

Being able to switch to a prior good access path or reusing a prior good access path are both valuable features. However, management of those prior copies should not be disruptive to currently active applications.

DB2 12 supports selectively FREEing either of the PREVIOUS or ORIGINAL copies individually. The ORIGINAL is intended to be the preferred copy that stays consistent across multiple REBINDs. However, this ORIGINAL copy will become stale over time and should be updated after performance of the current access path is considered to be the new standard for that package.

This step requires manual intervention to manage that ORIGINAL copy, which is still true in DB2 12. However, the ability to selectively FREE only the ORIGINAL copy is an improvement over prior releases, where all prior copies would need to be FREEd (both ORIGINAL and PREVIOUS) to allow the ORIGINAL to be repopulated at the next REBIND. You can also FREE only invalid copies in DB2 12. And most importantly, FREEing these inactive copies (either ORIGINAL or PREVIOUS) can occur without affecting the active copy.

Although this change seems minor, if you consider that DB2 11 would require exclusive access to the current package to manage those inactive copies, it is a simple but important availability enhancement to allow this to occur without disrupting a production application.

A further usability enhancement to the reuse (APREUSE) of a prior access path is the option to specify directly either the ORIGINAL or PREVIOUS copy as input to that reuse. In DB2 11, the only option is to specify the current copy as input.

Reusing either of the inactive copies would require a two-step process to first SWITCH the inactive copy to current, and then to REBIND with APREUSE. This two-step process introduces the risk that an auto-bind could overwrite that current bind if the inactive copy was invalid. DB2 12 avoids this risk by allowing the REBIND with APREUSE from the inactive copy in one step.

Dynamic plan stability

The basic premise that static SQL helps ensure stability in performance is well understood. This feature provides obvious value given the critical nature of most workloads that customers run on DB2 for z/OS. The time between static rebinds could be weeks or years, during which time there is satisfaction that performance of that access path is consistent. The life expectancy of a dynamic SQL is instead the time the SQL remains in the dynamic statement cache. Stability is only guaranteed within that time until that query next exits and enters the cache. This time frame can be from minutes to days.

DB2 12 allows you to stabilize dynamic SQL and store the SQL and cache structures in the DB2 catalog similar to static SQL. Upon entry to the cache, DB2 can load the existing structures from the catalog for stabilized SQL rather than preparing a new access path. This process provides improved reliability in the access path, but also can significantly reduce the cost and improve performance of the first entry of the SQL into the cache.

Heavy “cache-load” periods such as the beginning of the business day will see improved performance from prepare avoidance. Internal DB2 testing demonstrates 70–97% CPU reduction (of the prepare cost) for loading stabilized SQL into the cache compared with a new prepare, with more complex SQL achieving the more significant savings.

Stabilizing dynamic SQL in this manner also provides benefits across future DB2 release migrations, in application of DB2 maintenance, in recycling DB2, across members of a data sharing group, and also after DB2 utility execution. All these events result in the potential for a new access path today, whereas dynamic plan stability can preserve the access path across each of these events. However, it should be noted that rebind capability or the switch and reuse capabilities are not yet available to dynamic SQL.

Dynamic plan stability is considered to be one of the most significant SQL performance enhancements in DB2 12 because of the benefit to dynamic SQL OLTP applications and repeating analytics SQL.

Summary

If you have read the Query Performance and Optimization overview from DB2 10 and DB2 11 for z/OS, you would note that DB2 12 for z/OS provides even more enhancements targeted at the increased demands that customers are placing on their operational systems. And similar to the focus in these recent DB2 releases, most of the performance enhancements delivered do not require user intervention, other than rebind for static SQL.

The customer focus on real-time analytics requires that the analysis occurs close to the operational data. Otherwise, it is incredibly difficult to deliver real-time insights. And for DB2 for z/OS workloads, that means that customers want that analysis to occur with systems that are tightly integrated with z Systems or within the DB2 for z/OS system itself. DB2 12 for z/OS is ready, targeting performance improvements that allow real-time analytics to occur within the scope of a transaction. It does so by focusing on critical inhibitors to workload scalability and enhancements to important SQL constructs exhibited in known analytics workloads.

IBM Z is at the core of Enterprise HTAP systems, and although DB2 for z/OS has traditionally handled the transactional workloads, those transactional requirements are increasing in complexity, and DB2 integration with the analytics is paramount for true HTAP. When customers are ready to gain real-time insight from their operational systems, DB2 12 for z/OS is available to meet the challenge.

Authors

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

Terry Purcell is a Senior Technical Staff Member with the IBM Silicon Valley Lab, where he is lead designer for the DB2 for z/OS Optimizer. Terry has 25 years of experience with DB2 in SQL Performance and optimization as a customer, consultant, and DB2 developer. Terry is an IBM Master Inventor with more than 25 U.S. patents for database search technology.

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

Jason Alpers
Brian Baggett
Tom Beavin
Patrick Bossman
Frank Bower
Ashish Chaudhari
Po Chih Chen
Jason Cu

Tammie Dang
Dengfeng Gao
Michelle Guo
Pamela Ham
Peng Huang
Whitney Huang
Maggie O Jin
Tejaswini Karra
Peter Kuang
Allan Lebovitz
Wei Li
Elma Lui
Andrei Lurie
Bruce McAlister
Paul McWilliams
Jerry Mukai
Khoa Pham
Kendrick Ren
Gary Seto
Martina Simova
Cynthia Suo
Jasmi Thekveli
Ishan Tsai
Lingyun Wang
Li Xia
Ying Zeng
Xiao Min Zhao
DB2 for z/OS Development at IBM Silicon Valley Lab

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

Stay connected to IBM Redbooks

- Find us on Facebook:

  http://www.facebook.com/IBMRedbooks
- Follow us on Twitter:

  http://twitter.com/ibmredbooks

- Look for us on LinkedIn:

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

- Explore new IBM 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

Notices

This information was developed for products and services offered in the US. This material might be available from IBM in other languages. However, you may be required to own a copy of the product or product version in that language in order to access it.

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, MD-NC119, Armonk, NY 10504-1785, US

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 jurisdictions 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 provide in any way it believes appropriate without incurring any obligation to you.

The performance data and client examples cited are presented for illustrative purposes only. Actual performance results may vary depending on specific configurations and operating conditions.

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.

Statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

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 actual people or business enterprises 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. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation, registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the web at “Copyright and trademark information” at http://www.ibm.com/legal/copytrade.shtml

The following terms are trademarks or registered trademarks of International Business Machines Corporation, and might also be trademarks or registered trademarks in other countries.

DB2®	Redpaper™	z/OS®
IBM®	Redbooks (logo)  ®	
Redbooks®	WebSphere®	

The following terms are trademarks of other companies:

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

REDP-5445-00

ISBN 0738456128

Printed in U.S.A.