DB2 10 for z/OS Query Optimization Update

Terry Purcell
IBM Silicon Valley Lab

Contents

Introduction

IBM® DB2® for z/OS® customers expect improved performance in each release, while maintaining the stability and reliability to which they have grown accustomed from the mainframe environment. And DB2 10 for z/OS continues this theme, with significant attention given to improved plan management, runtime optimizations and new access path choices.

The DB2 optimizer goal is to provide continual improvement in performance for new and existing workloads, while maintaining the stability and reliability of the access path choices on which our customers rely for their traditional workloads. Optimizer theorists and academics will point out the challenges for any query optimizer to guarantee perfection in access path selection. But the reality is, our DB2 for z/OS customers don't want excuses. They need solutions that deliver this reliability in performance without increasing cost.

So how is IBM addressing these challenges? What you will see are the following major themes for enhancements to the DB2 10 for z/OS Optimizer:

- Predicate processing improvements regardless of access path
- New access path choices to improve common query patterns
- Enhancements to plan management

Predicate processing improvements are just that—more efficient predicate evaluation to reduce CPU consumption. New access path choices are provide opportunities for improved performance for query patterns that have been identified as important to our DB2 customers. And finally, Plan Management provides the capability to recover a prior static package upon access path regression and new to DB2 10 is the ability to reuse the prior access path for BIND/REBIND.

This three-pronged approach to enhancing query performance is anticipated to provide a more positive customer experience for the performance of your workloads than any release in recent memory. This is also likely to serve as the model for future DB2 releases to provide improved performance with reduced exposure to regression risk.

Access path management

The reason customers historically embraced static SQL over dynamic is that static is more attractive to a DBA, as its dictionary definition includes the phrase "showing little or no change," while dynamic implies ever-changing.

Just because an SQL or application is dynamic in nature does not mean that the DB2 optimizer will have a problem in its access path selection. There is a benefit to leaving the access path alone if it is already performing well. You have greater control over this for static SQL. Stability is only one consideration. The other consideration is avoiding the overhead of BIND/PREPARE, if there is no desire to explore a new access path.

The topic of access path management encompasses both the reduction in dynamic PREPARE overhead by avoiding PREPARE where possible, and also addressing the potential instability that can occur from large scale BINDs or REBINDs.

Dynamic statement cache enhancements

In a dynamic SQL environment, minimizing the overhead of PREPARE by exploiting the dynamic statement cache is one goal. The dynamic statement cache allows subsequent executions of the same SQL statement to reuse the previously PREPAREd access path, rather than preparing the statement again for every execution.

For effective reuse of a prior execution, the statement must match between executions. This is why applications are encouraged to code the SQL using parameter markers, rather than using literal values.

Not all applications have been coded to use parameter markers, and therefore will require a new PREPARE for each execution unless a recent execution of the same statement has occurred with exactly the same literal values. This is often unlikely.

Literal Replacement

DB2 10 for z/OS introduces the capability to replace the literals with a marker so that queries with literals can now be re-used in the cache. The literals will be replaced with an ampersand, which is similar to, but not the same as, a parameter marker. This is referred to as literal replacement or literal concentration. The idea is that you concentrate all various literal values into a single common ampersand for each predicate.

Repeat executions of the same query can subsequently benefit from a previously cached copy of the query in the dynamic statement, rather than issuing a new PREPARE.

To enable literal concentration, you can perform one of the following:

1. On the client, the PREPARE ATTRIBUTES clause can be coded to include the 'CONCENTRATE STATEMENTS WITH LITERALS' clause.
2. The JCC driver on the client side can be changed to include the keyword "enableliteralReplacement='YES'"
3. Or you can set LITERALREPLACEMENT in the ODBC initialization file in z/OS. This enables all SQL coming into DB2 through ODBC to have literal replacement enabled.

The lookup sequence for the SQL execution is to first lookup the cache for the SQL with literals to see if a prior copy has been cached with the same literals. If found, then the PREPARE is avoided and the prior access path used.

If not found, then the literals are replaced with an ampersand and the lookup is repeated.

If it is found, the matched access path is reused. If not found, then a new PREPARE is issued and the resultant query and access path are stored in the dynamic statement cache.

For transactional workloads with repeated, short, running queries, there is significant benefit to avoiding the overhead of PREPARE for each execution of the query. While replacing a literal with an ampersand or coding a parameter marker can result in reduced PREPARE overhead, it also means that the optimizer cannot take advantage of the literal values to improve its access path decision using FREQVAL or HISTOGRAM statistics.

For example, if statistics show that STATUS='Y' is 99 percent of the data, and STATUS='N' is 1 percent, then a query with WHERE STATUS='N' will recognize that an index may be a good choice for the access path.

However, WHERE STATUS='Y' would be best served with a tablespace scan. Replacing the literal with an ampersand means WHERE STATUS=& cannot take advantage of the frequency statistics.

Provided an efficient access path is chosen for transactional queries, reducing PREPARE overhead should be the goal. Therefore, this is a tradeoff that most DBAs are willing to accept, since PREPARE overhead can be easily observed.

Parameter markers will still provide better performance than the literal replacement technique due to literal replacement potentially requiring an additional cache lookup. There is also additional storage needed for the original predicate attributes such as datatype and length with literal replacement.

Regardless of whether coding parameter markers may be more efficient than literal replacement, the key point is that this enhancement is focused on applications that cannot or did not use parameter markers. Therefore, for transactional workloads, the literal replacement technique is likely to be more efficient than issuing a PREPARE—assuming no access path regressions result from hiding the literal values from the optimizer. Transactional workloads generally comprise many repeated executions of the same query.

In reporting, ad hoc, or query workloads, it is preferable to allow the optimizer to see the literal values for improved access path selection. For those queries, the PREPARE overhead is a small percentage of the overall query cost. Thus, the focus is on providing the optimizer with sufficient information to determine an efficient access path choice rather than short-cutting PREPARE. In general, these workloads do not repeat executions of the same query patterns, so there is minimal benefit to avoiding PREPARE.

Workloads that need both reduced PREPARE overhead and improved information for the optimizer may consider using REOPT(ONCE) BIND option in conjunction with literal replacement. The optimizer will use the set of literals from the first execution of the query for its access path determination. This is a good choice if the query consistently looks for the same values/ranges for the skewed or range predicates.

However, it may not result in the best performance if, for example, the query flips from STATUS='Y' to 'N', because the access path determination will be based upon the first execution.

The target for this enhancement is applications that cannot or have not used parameter markers. Therefore, if a query currently contains a mixture of parameter markers and literal values, then this query will not be eligible for literal replacement—because the application can obviously tolerate parameter markers.

The assumption is that there is an intention to code parameter markers for predicates that change frequently but whose change will not impact the access path, and literals for those for which optimizer would benefit from seeing the literal.

An example would be: WHERE ACCOUNT_NUMBER = ? AND STATUS = 'Y'. There may be millions of distinct ACCOUNT_NUMBERs, and each execution of the query uses a different value, which means the predicate is a good candidate for a parameter marker. The STATUS='Y' predicate may have two possible values, and is likely to be skewed. Therefore, this is a good candidate for a literal value. A query coded this way would not be a candidate for literal replacement because of the mix of parameter markers and literals.

Limitations exist for literal replacements, including lack of support for LIKE predicates. Also, the replaced SQL text with ampersands is not considered valid external syntax. It is an internal representation only. Therefore, externalized copies of the SQL with literals replaced cannot be fed into EXPLAIN or used for statement-level optimization hints or options (which are new to DB2 10).

Access plan stability or plan management.
DB2 9 for z/OS delivered plan management that supported a backup and recovery capability for the access plans of statically bound packages. DB2 10 provides some incremental enhance-ments to the original plan management usability while also introducing some quantum leaps forward in reducing impact of access path regression for static SQL.

It should be noted that one of the most common comments or questions related to the DB2 9 enhancement is regarding the name—plan management. Since it only applies to static packages. The reason for this name is that, in query optimization, "plan" refers to the access plan or access path. Optimizer development therefore does not use the term "plan management" to refer to a plan or package, but the access plan chosen by the optimizer.

Given that DB2 9 has been generally available for more than five years, the retort to this question about the name plan management has improved. To this date, there is no good answer as to why there are no questions about the PLAN_TABLE.

The success of plan management in DB2 9, has resulted in the first enhancement to simply change the zparm default from OFF in DB2 9, to PLANMGMNT=ENABLED in DB2 10. We want you to exploit this backup and recovery capability as an insurance policy for your access paths. Query performance has become mission critical for many of our customers, who cannot tolerate performance regressions either because of service level agreements, or because of the high utilization rates for their mainframe applications.

The zparm PLANMGMT (and associated BIND option) only control the capability for backup/recovery (REBIND SWITCH) of static packages. The zparm does not need to be enabled to support the new DB2 10 plan management and EXPLAIN functions. However, I will repeat the recommendation to enable PLANMGMT as per the DB2 10 default.

A new catalog table SYSIBM.SYSPACKCOPY has been added to hold the metadata for the previous and original copies which was previously only available in the SYSPACKAGE catalog table.

To see information in DB2 9 for the previous/original copies, the user had to REBIND(SWITCH) that copy to become the current and thus populate SYSPACKAGE. This was inconvenient if the requirement was only to view this detail for the saved copies. SYSPACKCOPY is populated during the ENFM process in DB2 10.

Space usage in SPT01 was one concern for DB2 9 customers due to the 64 GB dataset limit for SPT01. DB2 V8 and V9 supported compression of SPT01. Several solutions exist to reduce SPT01 space consumption and avoid the 64 GB limit in DB2 10.

First, SPT01 becomes a PBG (partition by growth) tablespace, and with APAR PM27811, the use of inline LOBs and thus support for the compression of SPT01.

An additional enhancement is the APRETAINDUP REBIND (and REBIND TRIGGER) package option. This option specifies whether an access plan is saved as either an original or previous if the newly generated plan is equivalent. The default is YES—retain the duplicates. Arguably using APRETAINDUP(NO) would result in space savings when using plan management basic or extended (DB2 10 default), because duplicate access plans are not saved.

When using APRETAINDUP(NO) in DB2 10, the existing (current) plan that is being replaced must have been bound in DB2 9 or 10 for the access plan comparison to occur.

That concludes the incremental enhancements to the backup and recovery capability of plan management in DB2 10.

DB2 10 takes Plan Management to the next level

The backup and recovery capability of the DB2 9 plan management was a very welcome addition to REBIND. However, DB2 10 brings significantly more to reduce the risk of access path regression across REBIND. From DB2 9, a new BIND/REBIND resulted in a compressed copy of the PLAN_TABLE rows for each SQL in the package to be saved in the directory/SPT01. The internal representation of the PLAN_TABLE, which DB2 development refers to as the Explain Data Block (EDB), is used as the basis for numerous enhancements to Plan Management—which will be outlined in the upcoming section. What this means however, is that many of these enhancements are only available for BINDs/REBINDs that have occurred in DB2 9 or later.

Many customers associate the recommendation to REBIND as a way to expose their queries to the improvements in the optimizer. But there are numerous situations where a customer may not be ready to expose themselves to access path changes, but REBIND is recommended or forced.

Across a DB2 release, REBIND is recommended to take advantage of the new release runtime structures. While prior release runtime structures are tolerated in the new release (This is true for runtime structures from DB2 V6 and later in DB2 10.), optimizations such as SPROCs are disabled. However, during a migration, it is reasonable to assume that system stability is desired from the migration before exposing the application to access path changes, due to concerns about regression.

There is another, often less discussed reason for REBINDing in each new release. And that is that older runtime structures are at greater risk of instability from either an abend or incorrect output. DB2 tolerates prior release runtime structures from V6 onwards in DB2 10, but there is still the question as to whether DB2 was able to test your V6 or V7 runtime structure in DB2 10 testing. Considering the number of possible access paths, and the specific maintenance level when the package was bound, it is difficult to imagine that every combination received test coverage. The bottom line is that it is safer to be more current than two or more releases prior with your REBINDs.

Given that DB2 development encourages a REBIND at least once per release, it is important that we also address the main reason why customers avoid REBIND—which is generally because of the risk of regression.

For this reason, a new parameter has been added for BIND and REBIND that provides the capability to control when a new access path is considered for BIND/REBIND. APREUSE (Access Path Reuse) will attempt to BIND/REBIND using the prior access path as an internal hint to drive the new access path choice. The only supported options are NO/NONE (default and standard BIND/REBIND behavior) or ERROR—whereby the BIND/REBIND will attempt to reuse the prior access path, and if the prior access path cannot be reused, then an error is issued and the BIND/REBIND fails. The level of granularity is on the package, which means a failure of any single SQL to reuse the prior access path, then all SQLs in the package will fail in their reuse.

Another way of looking at APREUSE is that DB2 10 gives you greater control over when you want to expose your application to new access path choices. Rather than being forced to open up new access path choices when REBIND is recommended across migration, or APAR ++HOLD information recommends REBIND, or when a schema change invalidates the package, APREUSE allows you to differentiate REBIND (and BIND) when you do and do not want to expose your application to new access path choices. This is a significant leap forward in providing further stabilization and avoidance of regression for static SQL.

A second additional BIND/REBIND parameter choice —APCOMPARE performs an access path comparison between the prior access path and the newly generated access path. The options are NO/NONE (default and no comparison performed), WARN (warning messages are written to the PLAN_TABLE.REMARKS column to identify access path differences) or ERROR (any access path difference will result in failure of the BIND/REBIND).

Arguably, APCOMPARE(WARN) should be the default for customers performing a BIND or REBIND of their static packages, since this simply externalizes meaningful information to the PLAN_TABLE about changes in the access path. APCOMPARE(ERROR) may be less useful, as it allows a BIND/REBIND to seek a new access path choice, but due to the ERROR option, will fail the BIND/REBIND if a new access path is chosen.

To clarify both APREUSE and APCOMPARE, APREUSE asks BIND/REBIND to attempt to reuse the prior access path. APCOMPARE, without APREUSE, will allow a new access path choice (which is the same as BIND/REBIND prior to APREUSE), but issue PLAN_TABLE messages to identify changes in the access path.

Note: APREUSE or APCOMPARE are only valid for packages bound in DB2 V9 or later. Any use of APREUSE/APCOMPARE on a pre-DB2 9 package will be unable to internally use the EDB for reuse or comparison. However, this will not fail the package which may be misleading to some users.

EXPLAIN enhancements

The theme of plan management is not complete without discussing the externalization of the access path in the PLAN_TABLE and other extended explain tables.

It has been a long-standing requirement from our customers to be able to explain the existing access path for a previously bound package. This is necessary for cases where the prior BIND/REBIND used EXPLAIN(NO), or when the PLAN_TABLE rows are no longer in existence. Issuing a new EXPLAIN will potentially produce a new access path and thus not represent what is currently executing.

Thus, the requirement is to "tell me what I have." And the DB2 solution will be discussed next, and also a solution for the requirement to "tell me what I would get if I performed a BIND/REBIND today."

DB2 10 adds a new option to EXPLAIN — EXPLAIN PACKAGE, for the requirement to "tell me what I have." This allows extraction of the existing access path from the package explain data block EDB). The output from EXPLAIN PACKAGE is inserted into the PLAN_TABLE. No other extended explain tables are populated by EXPLAIN PACKAGE. And it is possible to specify the COPY as the current, previous or original to extract — as can be seen in the syntax diagram in Figure 1.

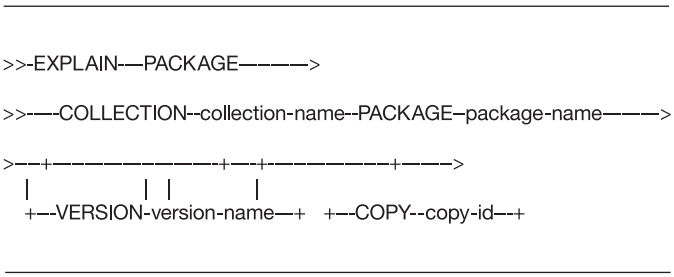

Figure 1. EXPLAIN PACKAGE syntax diagram

In this scenario, the term "EXPLAIN" is really a misnomer. This is actually an "EXTRACT," as an EXPLAIN implied a new access path choice. However, instead of introducing a new EXTRACT keyword, DB2 has piggybacked on the EXPLAIN statement for this enhancement.

As with many of the DB2 10 plan management enhancements that rely on the internal representation of the access path from the EDB, this enhancement is *only* supported if the package is from DB2 9 or later.

If you want to ask the question, "What would the new access path be if I performed a BIND/REBIND today?" Then there is a new EXPLAIN(ONLY) option for BIND/REBIND.EXPLAIN has always had options YES/NO, and the addition of "ONLY" allows this "what if?" question to be easily answered without impacting the existing package. Customers may have accomplished this previously by performing a BIND to a dummy collection or manually explaining the SQL outside the package.

Although it should be noted that extracting the SQL outside of the package and issuing an EXPLAIN may not always be equivalent—as this would become a dynamic EXPLAIN rather than a static explain. A dynamic explain will not consider indexes in a restricted state, whereas a static EXPLAIN will. Also, datatype or length differences between the host variable and predicate column are masked if a dynamic EXPLAIN is performed—and despite the fact that DB2 V8 improved indexability for mismatched datatype/length predicates, there still exist some behavioral differences.

For EXPLAIN(ONLY), the BIND/REBIND is performed, and the chosen access path is written out to the PLAN_TABLE, and finally the BIND/REBIND is rolled back. The PLAN_TABLE entries remain and are flagged with "Y" in the new BIND_EXPLAIN_ONLY column so that customers can determine that these PLAN_TABLE rows—despite the fact that they are associated with a package, came from a BIND/REBIND with EXPLAIN(ONLY) and may not correlate to any current access path in the catalog. Locking/concurrency requirements are the same with EXPLAIN(ONLY) as a standard BIND/REBIND.

An additional parameter is available for BIND/REBIND to perform a syntax check also without creating the actual package. This option, SQLERROR(CHECK), can be used independently of EXPLAIN(ONLY). This SQLERROR(CHECK) BIND/REBIND parameter was specifically targeted to those customers who may have their development and test systems on another DB2 platform, and their only opportunity to validate the package is in their production DB2 for z/OS system.

With all of these new additions to EXPLAIN, it becomes a challenge for users to understand the usage scenario for each EXPLAIN option. Therefore, it is important to summarize the various EXPLAIN usages to help clarify their function.

What is the difference of each EXPLAIN usage?
Options in bold red are new to DB2 10.

- BIND/REBIND with EXPLAIN(YES)
  - Generates a new access path, populates PLAN_TABLE and creates new package
- BIND/REBIND with EXPLAIN(ONLY)
  - Generates a new access path, populates PLAN_TABLE, but does NOT create a new package

- EXPLAIN PLAN (issued in SPUFI/QMF/DSNTEP2 etc)
  - Generates a new access path and populates PLAN_TABLE
- EXPLAIN PACKAGE
  - Does not generate new access path. Extracts existing access path from package and populates PLAN_TABLE.
- EXPLAIN STMTCACHE STMTID/STMTOKEN
  - Does not generate new access path. Extracts existing and populates PLAN_TABLE.

Figure 2. EXPLAIN usage scenarios

As outlined in Figure 2, the following summarizes the key usages of EXPLAIN:
BIND/REBIND with EXPLAIN(YES) is an existing choice, and performs the following:

- Generates a new access path (or attempts to reuse prior if APREUSE(ERROR)
- Populates the PLAN_TABLE
- Creates a new copy of the package

BIND/REBIND with EXPLAIN(ONLY) is a new choice, and performs the following:

- Generates a new access path (or attempts to reuse prior if APREUSE(ERROR)
- Populates the PLAN_TABLE
- Does NOT create a new copy of the package

EXPLAIN PLAN is an existing choice executed from SPUFI/QMF/DSNTEP2, etc., and performs the following:

- Generates a new access path
- Populates the PLAN_TABLE

EXPLAIN PACKAGE is a new choice, and performs the following:

- Does NOT generate a new access path. Extracts the existing access path from the package
- Populates the PLAN_TABLE

EXPLAIN STMTCACHE STMTID/STMTOKEN is an existing choice, and performs the following:

- Does NOT generate a new access path. Extracts the existing access path.
- Populates the PLAN_TABLE

Instance-based statement hints

Optimization Hints (opthints), first delivered in DB2 V6, have been embraced by some customers as a way to override a problem access path choice or stabilize queries to avoid access path change. It is also fair to say that opthints have been avoided by many customers due to their cumbersome nature or due to the challenges for customers to micromanage access paths. The DB2 10 plan management enhancements may reduce the need for hints used to stabilize an access path.

In addition to the plan management enhancements, DB2 10 also improves the infrastructure and usability of opthints, so that customers continue to have a way to override an inefficient access path choice if other more suitable solutions aren't viable.

The first enhancement related to opthints introduces a catalog infrastructure to support a more general form of hints. This is referred to as the access path repository, which holds important query metadata (such as query text), query access paths and other information, such as optimization options.

The repository has three new catalog tables:

- SYSIBM.SYSQUERY is the central table of the access path repository. This table holds one row for each static or dynamic SQL query that is to exploit user-specified hints or options.
- SYSIBM.SYSQUERYPLAN holds the plan hint information for the queries in the SYSIBM.SYSQUERY table.
- SYSIBM.SYSQUERYOPTS SYSIBM.SYSQUERYOPTS holds the option information (if options are specified) for each query in SYSIBM.SYSQUERY.

The original opthints delivered in DB2 V6 have often presented challenges for customers to maintain, because they are tied to a query number. This means for a static SQL, if the application programmer adds or removes lines of code, and the precompiler will generate a new query number for each SQL statement occurring after the code change. DB2 V6 also added the QUERYNO clause that could be added to an SQL statement. This would ensure the same query number is used across application changes. For dynamic SQL, this was the required way to match a statement to a hint. Either for static or dynamic however, altering the SQL is often impractical and thus was often not adopted as a solution.

DB2 10 adds an alternative way to associate a statement and a hint—using the query text. Similar to the concept of a dynamic statement cache text match, the SQL text is tied to the hint such that a static BIND or dynamic PREPARE will attempt to lookup the statement text to find a matching hint. A hint can therefore be created irrespective of its use for dynamic or static SQL—and thus the hint can be given a global scope or package-level scope.

The hints are stored in the access path repository. The PLAN_TABLE isn't going away however. Instead, there is now an alternate method for looking up the hint which makes it simpler for dynamic SQL and more stable for application changes in static SQL.

While tying the hint to the statement text means that changes in query numbers will not affect the hint, changes to the SQL statement will result in the match failing.

To take advantage of the new hints process, the following steps should be followed:

- Enable the OPTHINTS zparm
- Populate the user table DSN_USERQUERY_TABLE with the query text
  - Insert from SYSPACKSTMT (static) or DSN_STATEMENT_CACHE_TABLE (dynamic) to ensure that the correct DB2 representation of the SQL text is used.
- Populate PLAN_TABLE with the corresponding hints
  - **Note**: Choose any arbitrary QUERYNO. QUERYNO must match between PLAN_TABLE and DSN_USERQUERY_TABLE. Duplicate query numbers in the PLAN_TABLE may result in difficulty matching the PLAN_TABLE and DSN_USERQUERY_TABLE rows.
- Execute the new command BIND QUERY to integrate the hint into the repository.
- The next package BIND/REBIND or dynamic PREPARE can pickup hint.

To remove the query from the repository, use the FREE QUERY command.

Instance-based (or statement-level) options

The same infrastructure that allows an access path hint to be matched to the statement text is also extended to allow statement level scope for a small number of BIND parameters and zparms, namely:

- REOPT
- STARJOIN enablement and number of tables qualified for STARJOIN
- Parallelism enablement and number of degrees

This has been another long-standing customer requirement to provide improved granularity for the REOPT BIND option. Once a query is identified that would benefit from REOPT(ALWAYS), it is common that the customer does not want the overhead of REOPT for all other statements in the package, but only the identified query. The previous recommendation has been to separate out the targeted query into a separate package which is often impractical.

The steps to implement these statement level options are similar to that for statement level hints—minus the PLAN_TABLE input:

- Enable the OPTHINTS zparm
- Populate the user table DSN_USERQUERY_TABLE with the query text
  - Insert from SYSPACKSTMT (static) or DSN_STATEMENT_CACHE_TABLE (dynamic) to ensure that the correct DB2 representation of the SQL text is used.
  - **Note**: Choose any arbitrary QUERYNO that does NOT currently exist in the PLAN_TABLE. The reason is that BIND QUERY will first look to find the PLAN_TABLE rows, and if not found, will look at the options in DSN_USERQUERY_TABLE.
- Execute the new command BIND QUERY to integrate the statement level options into the repository.
- The next package BIND/REBIND or dynamic PREPARE can pickup the new options.

Hints and options are mutually exclusive. Therefore at this stage, it is only possible to have either a hint or options for a given statement in SYSQUERY.

Predicate processing and runtime optimizations

The most welcome performance improvements for customers are those that require minimal action or intervention. And the performance enhancements in DB2 10 for z/OS include predicate processing improvements and runtime optimizations that can be exploited within existing access path choices, and also those that offer new choices to the DB2 for z/OS optimizer.

It is well understood by DB2 development that customers are continually pushed to do more with less—whether that is individual DBAs having to manage more DB2 subsystems, or the systems themselves increasing data volumes and throughput without a corresponding increase in available capacity. Thus, improving performance is among the highest priorities for DB2 development.

Improvements to predicate application for IN and OR predicates

DB2 10 delivers several enhancements to IN-list and OR predicate processing.

The first enhancement involves an improvement to the execution performance of long IN-lists and complex AND/OR predicates that are chosen as index screening or stage 1. The improvement comes from DB2 being able to exit the predicate application process as soon as a true or false condition is triggered that allows further processing to be avoided. While DB2 has always been able to stop processing once the row is qualified or disqualified, DB2 10 delivers additional optimizations to traversing the predicate tree.

Unfortunately, there is no way to identify a query as a candidate for this optimization. Which raises the question, "Why is there any detail at all here about this enhancement?" The answer is because of customer questions as to why they have seen some queries achieve CPU reductions without an access path change that would warrant the improvement. And one possible explanation is the aforementioned enhancement.

The next enhancement relates to IN-list predicates that are candidates for index matching. When IN-list predicates are filtering, then matching index access is often desirable using these IN-list predicates. Prior to DB2 10, DB2 could match on the first IN-list (based upon the index key column order) and continue matching on available predicates until a second IN-list was encountered.

Basically, DB2 could only match on one IN-list and any subsequent IN-list predicates would be applied as index screening. In situations where the second IN-list was more filtering, then the performance may not be optimal.

Similarly, when choosing matching IN-list access, DB2 would not choose list prefetch, which is preferred if the index has a poor cluster ratio. Instead, DB2 may choose to match on one less column (without the IN-list) so that list prefetch could be exploited. Thus DB2 would trade improved data I/O performance for less index matching, and instead apply the IN-list as screening.

DB2 10 addresses these issues with IN-list predicates by allowing the optimizer to consider converting the IN-list to a table such that the IN-list processing performs more like a join. This allows matching on multiple IN-lists and also list prefetch to be supported.

However, this IN-list table will only be chosen if matching on multiple IN-lists is chosen, or list prefetch is chosen with matching IN-list access. Otherwise, the previous type of IN-list matching is considered (ACCESSTYPE='N').

In the EXPLAIN output in the PLAN_TABLE, this access to the in-memory table is associated with a new table type 'I' and new access type 'IN'. Figure 3 shows the PLAN_TABLE output for IN-list table access with list prefetch.

```
SELECT *
    FROM T1
    WHERE T1.C1 IN (?, ?, ?);
```

QBNO	PLANNO	METHOD	TNAME	ACTYPE	MC	ACNAME	QBTYPE	TBTYPE	PREFETCH
1	1	0	DSNIN001(01)	*IN*	0		SELECT	*I*	
1	2	1	T1	I	1	T1_IX_C1	SELECT	T	*L*

Figure 3. IN-list table example with list prefetch.

It should be noted that list prefetch will execute once per IN-list element. It is *not* a consolidated list prefetch access for all elements. Therefore, if the IN-list elements are each of a high cardinality, then there may be no benefit to choosing list prefetch.

Another enhancement to IN-list predicate processing is transitive closure support for IN-lists. The National Institute of Standards and Technology defines Transitive Closure as: "An extension or superset of a binary relation such that whenever (a,b) and (b,c) are in the extension, (a,c) is also in the extension."

To put that into a query predicate perspective, if A=B and A=1, then B also =1. DB2 has supported transitive closure for =, <, <=, >, >= and BETWEEN for many releases, but DB2 10 adds support for IN-lists.

Figure 4 demonstrates an example where IN-list predicate transitive closure (PTC) is now possible.

```
SELECT *
FROM T1, T2
WHERE T1.C1 = T2.C1
    AND T1.C1 IN (?, ?, ?)
    AND T2.C1 IN (?, ?, ?) ← Optimizer can generate
                             this predicate via PTC
```

Figure 4. IN-list Predicate Transitive Closure example

Without PTC, DB2 is likely to choose T1 as the leading table in the join sequence because there is only filtering on T1.

By generating an additional predicate AND T2.C1 IN (?,?,?), DB2 may consider the alternate join sequence with T2 as the first (outer) table in the join sequence. This gives DB2 greater opportunity to choose the most efficient access path, regardless of how the query is coded.

OR predicate processing improvements—Online cursor scrolling

We have already discussed some enhancements to predicate processing for IN, and OR predicates, and the improved index matching capabilities for IN-lists. In prior releases of DB2, there has always been a link between OR and IN predicates, with DB2 rewriting simple OR conditions against the same column to become IN-lists. A simple example is; WHERE C1 = 1 or C1 = 2, which is rewritten by DB2 to become WHERE C1 IN (1,2). This hasn't changed in DB2 10.

The DB2 10 enhancements target more complex OR predicates that are not candidates for rewrite to IN-lists. To improve processing for OR predicates, DB2 introduces a new access type — Range-list access. Range-list access refers to a "list of ranges" separated by OR. This is similar to IN-list access in its processing, but without representing the more complex OR conditions as a simplified IN-list predicate.

The two original targets for range-list access are:

- Scrolling/paging SQL–common in CICS and other online web applications whereby the application wishes to fetch the next 'n' rows to fill a screen. This is not to be confused with scrollable cursors which require the application to keep the transaction open for scrolling forward/backward through the result.
- Complex OR predicates against the same columns–whereby the OR predicates are not simple equal predicates that DB2 would convert to an IN, but may include range predicates and/or compound predicates within each OR. The construct is common in SAP and other applications that write predicates in disjunctive normal form.

For both query patterns, the following conditions must be true to support range-list access:

- The OR predicate must refer to a single table.
- Each OR predicate can be mapped to the same index.
- Each OR has at least one matching predicate, given the chosen index.

In this first section, we will discuss the (online) scrolling query pattern. Consider the example in Figure 5:

```
WHERE (LASTNAME='JONES' AND FIRSTNAME>'WENDY')
    OR (LASTNAME>'JONES')
ORDER BY LASTNAME, FIRSTNAME;
```

Figure 5. Cursor scrolling example as a range-list candidate

The above range-list example demonstrates scrolling through the phone book with current cursor position at "JONES, WENDY". To scroll forward from this position, the user may code the WHERE clause predicates as shown in Figure 5. The first OR condition (LASTNAME='JONES' AND FIRSTNAME>'WENDY') requests the remaining "JONES" after the current position. And the second OR condition (LASTNAME>'JONES') requests the rows for the subsequent last names after "JONES".

The new access method can convert this OR predicate into a range-list with two ranges (one range for each OR). Therefore, there will be at most two index probes given the PHONEBOOK index on LASTNAME, FIRSTNAME. The first probe is for LASTNAME='JONES' and FIRSTNAME>'WENDY'. And once FETCHing exhausts all qualified rows from the first probe, the second probe will be issued for LASTNAME > 'JONES'.

These rows appear in the index in ascending order which satisfies the ORDER BY and thus there is no requirement to sort the rows.

Prior to DB2 10, matching index access for this example required multi-index access, list prefetch and a final sort to satisfy the ORDER BY. For this reason, it was common for users to code a redundant predicate "AND LASTNAME >= 'JONES' " to support single matching index access. This is shown in Figure 6.

```
WHERE ((LASTNAME='JONES' AND FIRSTNAME>'WENDY')
    OR (LASTNAME>'JONES'))
AND (LASTNAME >= 'JONES')
ORDER BY LASTNAME, FIRSTNAME;
```

Figure 6. Scrolling example with redundant predicate for matching index access.

In Figure 6, with the added (redundant) predicate, "AND (LASTNAME >= 'JONES') DB2 was able to choose single matching index access. Although prior to DB2 10, the best that DB2 could do was matching on one column using this redundant predicate. And the original predicates were applied as index screening—which means DB2 would position in the index as matching on LASTNAME='JONES', and then have to scan through all FIRSTNAMES from 'A' to 'W' to reach 'FIRSTNAME>'WENDY' using the pre-DB2 10 approach. With range-list, DB2 can use both predicates on LASTNAME and FIRSTNAME to start the initial position in the index at JONES, WENDY.

In DB2 10, the optimizer may choose range-list or any existing access method including multi-index access. It is important to note that the optimizer will make a cost-based decision. One factor that is often unknown to the optimizer is whether the application will FETCH 10 or 20 rows and close the cursor—unless the query has OPTIMIZE FOR 'n' ROWS coded (or FETCH FIRST n ROWS ONLY).

Therefore, it is not guaranteed that range-list will be chosen for this query pattern. There is a second usage case which also fits this scrolling type pattern, but is unrelated to online applications. That is batch restart logic. Range-list access is targeted more towards the online scrolling pattern, and not towards the batch restart usage, which is one reason why DB2 is not more aggressive when choosing range-list, since DB2 does not know whether the same pattern is for an online or batch application. As previously mentioned, the OPTIMIZE FOR or FETCH FIRST clauses are the best way to indicate to DB2 that this is an online query that is requesting a subset of the qualified rows.

OR predicate processing improvements—other range-list usages

Range-list does not only apply to the scrolling type SQL—but any complex OR conditions that can map to a single index.

```
WHERE (LASTNAME='JONES' AND FIRSTNAME='WENDY')
    OR (LASTNAME='SMITH' AND FIRSTNAME='JOHN');
```

Figure 7. Non-scrolling range-list candidate

In Figure 7, range-list can be chosen with each OR leg having two matching columns on the index on LASTNAME, FIRSTNAME. Range-list is also applicable if the index was on FIRSTNAME, LASTNAME, or if the index was only on LASTNAME, or only on FIRSTNAME. As previously mentioned, one requirement for consideration of range-list access is that each OR leg must support matching index access for the same index.

Prior to DB2 10, the optimizer could only choose multi-index access for matching index access due to the OR conditions. And in DB2 10, the optimizer can choose either range-list or multi-index access based upon cost. Since range-list does not support list prefetch, but multi-index access requires list prefetch, it is not unexpected to see range-list chosen for this type of SQL with a high cluster ratio index or if very few rows qualify, and multi-index access chosen for lower cluster ratio indexes.

Range-list EXPLAIN representation

The PLAN_TABLE representation for range-list includes a new access type—ACCESSTYPE='NR'. And one row will appear for each OR condition, since each OR may have a different number of MATCHCOLS. And column MIXOPSEQ (multi-index operation sequence) provides an ordering of the range-list rows.

Figure 8 borrows the range-list example from Figure 5 and reverses the order of the WHERE clause predicates to highlight an interesting nuance of the EXPLAIN output. The simplified EXPLAIN output shows the single predicate (LASTNAME > 'JONES') with MATCHCOLS=1 listed first in the PLAN_TABLE, followed by the MATCHCOLS=2 predicates. This PLAN_TABLE output matches the coding sequence, and in this example, does not represent the order that the values associated with those predicates would appear in the index.

```
WHERE (LASTNAME='JONES')
OR (LASTNAME='JONES') AND FIRSTNAME='WENDY')
ORDER BY LASTNAME, FIRSTNAME;
```

QBlockNo	PlanNo	Accessname	Access_Type	Matchcols	Mixopseq
1	1	1X1	NR	1	1

New access type (NR = IN-List Range) Coding Seq

Figure 8. Range-list EXPLAIN representation.

Range-list, if chosen, will access the index in the sequence that allows a sort to be avoided (if ORDER BY or GROUP BY is coded). The PLAN_TABLE is populated at BIND/PREPARE, and for queries with host variables or parameter markers, it is not known what literal values will be used in the query.

The DB2 implementation of range-list will re-order the OR conditions at runtime based upon the literal values. Thus, it is not possible for BIND/PREPARE to know the order in which these will be executed.

Customers might ask the question, "But for the 'screen-scrolling SQL'—such as the example in Figure 8, you know the order of the execution even with host variables, as it is most MATCHCOLS to least MATCHCOLS. Why don't you order the PLAN_TABLE rows to reflect this?"

First, this takes a very narrow view of the range-list enhancement. As outlined, the enhancement covers more than the screen-scrolling scenario. But more importantly, because DB2 implemented the ordering of execution of the OR predicates at runtime (execution time), BIND/PREPARE is not aware of the ordering. And enhancing DB2 to also provide recognition of the likely order is of minimal value.

The reason is because it is not a guarantee without knowing the literal values—and most SQLs use parameter markers or host variables. DB2 will always order the OR predicates based upon the literals values used to support moving through the index in the direction to support ORDER BY/GROUP BY ordering, or in ascending sequence if no particular order is required.

Therefore, for effective access path analysis, all that is important to understand is that ACCESSTYPE='NR' is chosen AND how many MATCHCOLS for each leg. The order in the PLAN_TABLE will be the sequence it has coded in the SQL. This allows the customer to match the PLAN_TABLE to the SQL. The actual order of execution will be dependent on the literal values used at runtime.

OUTER JOIN Merge and Subquery improvements

In general, materialization is more expensive for the execution of an SQL statement compared with merging a view or table expression. Therefore, in each release, DB2 continues to extend cases where MERGE occurs instead of materialization, and DB2 10 is no exception.

When there are CASE, VALUE, COALESCE, NULLIF or IFNULL expressions on the preserved side of an OUTER JOIN, DB2 will be enhanced to merge the view/table expression. The merge is blocked if it would result in a stage 2 predicate, such as a CASE expression in the ON clause.

Thus, the SQL in Figure 9 will merge the 'A' table expression in DB2 10, while table expression 'B' will continue to materialize. Prior to DB2 10, both table expressions will materialize.

```
SELECT A.C1, B.C1, A.C2, B.C2
FROM T1, (SELECT COALESCE(C1, 0) as C1 ,C2
          FROM T2 ) A            <--table expression 'A' will be Merged
         LEFT OUTER JOIN
              (SELECT COALESCE(C1, 0) as C1 ,C2
               FROM T3 ) B    <-- B will be Materialized
         ON A.C2 = B.C2
WHERE T1.C2 = A.C2;
```

Figure 9. OUTER JOIN merge/materialization example

The second OUTER JOIN merge enhancement involves a view or table expression containing a subquery.

```
SELECT *
FROM T1 LEFT OUTER JOIN
     (SELECT * FROM T2
      WHERE T2.C1 = (SELECT MAX(T3.C1) FROM T3 ) ) TE <--subquery
ON T1.C1 = TE.C1;
```

```
SELECT *
FROM T1 LEFT OUTER JOIN T2                  <-- table expression is merged
ON T2.C1 = (SELECT MAX(T3.C1) FROM T3)  <-- subquery ON-predicate
AND T1.C1 = TT.C1;
```

Figure 10. OUTER JOIN with subquery merge example

Figure 10 provides a SQL example of a subquery on the NULL-supplied table of a LEFT OUTER JOIN. In DB2 10, this table expression (or view) can be merged to the ON clause.

These views and table expressions must only contain a reference to a single table. DB2 will do so by converting the subquery predicate to a "before join" predicate—in the ON clause of a NULL-supplied table. When the table in the table expression is very large or there is significant filtering from the preserved side (left-hand side of a LEFT OUTER JOIN), performance will be improved due to lack of materialization.

Note: Coding a subquery in an ON clause is not permitted. However, DB2 can merge the original table expression and execute as if this was coded.

If the table expression with subquery is on the preserved row table (left-hand side of the table of a LEFT OUTER JOIN), then DB2 would merge the table expression with subquery to the WHERE clause to apply before the join. Coding a subquery in the WHERE clause for a preserved row table (left-side of LEFT OUTER JOIN) is valid syntax, and a user can code this.

Figure 11. Correlated subquery to non-correlated rewrite example

Correlated subquery to non-correlated rewrite

Although not strictly related to materialization, the following subquery rewrite enhancement is likely to be the most common query pattern of any of the previously discussed SQL examples within the merge/materialization topic.

The SQL in Figure 11 seeks to return the most recent transaction for a given ACCOUNTNO (hence the MAX subquery). It is common to see this coded as a correlated subquery.

Although the predicates on ACCOUNTNO in the outer query and the subquery are each indexable, the comparison of A.TRANDATE to the subquery result is stage 2, which means all transactions for a given account must be retrieved from the outer table.

DB2 10 can rewrite this construct to a non-correlated subquery if it is semantically equivalent. Therefore, the subquery will be executed before accessing the outer query block, and the subquery result would become indexable.

DB2 could then choose two matching columns on the outer, and only the desired transaction would need to be accessed.

Stage 2 predicate pushdown

Stage 2 predicates are the most expensive for DB2 to apply. This is a message that has been repeated for many years and customers have been encouraged to rewrite stage 2 predicates to be stage 1 and/or indexable, where possible.

Of course, not all SQL is under the control of developers who heed the recommendation, or the SQL may be application-generated. Also, not all stage predicates are easily rewritten to a more efficient form. So it is clear that DB2 cannot ignore the performance challenge.

DB2 10 enhances predicate application by enabling index manager and data manager (stage 1) to call RDS (stage 2) to evaluate stage 2 predicates. Therefore, these can be potentially applied as index screening before data page access.

Note: These pushed down predicates cannot be applied as index matching. Index on expression (delivered in DB2 9) should be considered for index matching of stage 2 expressions.

This enhancement applies to arithmetic and date-time expressions, scalar built-in functions and CAST operations. Limitations include:

- OR predicates must be able to be applied all at the same stage.
- Access paths involving list prefetch are not candidates for predicate pushdown.
- CASE expressions are not supported.
- IN-list predicates are not supported.

If the query qualifies, then the predicates will be marked in the DSN_FILTER_TABLE under the column—PUSHDOWN. The DSN_FILTER_TABLE is one of the extended explain tables. The DB2 10 Managing Performance manual provides more detail.

Figure 12 provides examples that demonstrate the eligibility for stage 2 predicate pushdown:

Suppose there exists index on (C1,C3)

- WHERE SUBSTR(C1,1,1) = ? ==> index screening
- WHERE SUBSTR(C1,1,1) = ? OR C3 = ? ==> index screening
- WHERE SUBSTR(C1,1,1) = ? OR C4 = ? ==> stage 1
- WHERE SUBSTR(C1,1,1) = ? AND C4 = ? ==> index screening and stage 1
- WHERE SUBSTR(C1,1,1) = ? OR C3 = (SELECT...) ==> stage 2
- WHERE SUBSTR(C1,1,1) = ? AND C3 = (SELECT...) ==> index scr. and stage 2

Figure 12. Stage 2 predicate pushdown examples

Based on Figure 12, where an index exists on C1, C3, prior to DB2 10, the Figure 12 SUBSTR predicate was always stage 2. In DB2 10, the following describes the eligibility of each predicate shown in Figure 12:

- WHERE SUBSTR(C1,1,1) = ?
  - Index screening candidate

- WHERE SUBSTR(C1,1,1) = ? OR C3 = ?
  - Despite the OR predicate, this is an index screening candidate because both sides of the OR can be applied at the same stage—index screening.

- WHERE SUBSTR(C1,1,1) = ? OR C4 = ?
  - This example is *not* an index screening candidate, because the predicate on C4 must be applied on the data row because it is not contained in the index. The compound predicate can however be pushed down to stage 1.

- WHERE SUBSTR(C1,1,1) = ? AND C4 = ?
  - The SUBSTR expression is an index screening candidate because it is not separated by OR.

- WHERE SUBSTR(C1,1,1) = ? OR C3 = (SELECT ….)
  - This is not an index screening candidate because predicates are separated by OR, and the subquery is not a candidate for pushdown. Thus, both predicates remain stage 2.

- WHERE SUBSTR(C1,1,1) = ? AND C3 = (SELECT ….)
  - The subquery is stage 2, but since the predicates are separated by AND, the SUBSTR is an index screening candidate.

This predicate pushdown enhancement does require a REBIND to take effect.

Predicate simplification

Despite the eligibility to pushdown stage 2 predicates to an earlier stage, the fact remains that stage 2 predicates are the least efficient predicates for DB2 to apply. To clarify, it is still true that a stage 1 predicate is more efficient to apply than a stage 2 predicate that is pushed down to stage 1.

In recognition of some important query patterns for customers migrating from other platforms, DB2 V8 and 9 introduced some enhancements to predicate REWRITE that were enabled by zparm PREDPRUNE. Since few customers enabled the zparm, these enhancements do not become available for the majority of customers until DB2 10, where the zparm is removed.

The first enhancement is to remove simple “always false” predicates. The targeted example is demonstrated in Figure 13:

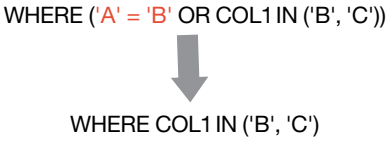

Figure 13. Always False predicate simplification

The “always false” ‘A’=‘B’ predicate renders the entire OR predicate stage 2. In DB2 10 (or V8/9 with zparm PREDPRUNE enabled), the “always false” predicate will be removed, leaving the indexable predicate WHERE COL1 IN (‘B’, ‘C’).

Why would anyone code the original predicate? This query pattern came from a query generator where this construct was used to enable or disable predicates within a common framework to simplify query generation.

For simplicity, assume I have two predicates and the user may wish to search by only one or the other, or both. If the user wanted to search by LASTNAME, the code generator would create the predicates from Figure 14:

```
WHERE ('A' = 'B' OR LASTNAME = 'PURCELL')
AND ('A' = 'A' OR CITY = 'ZZZZZ')
```

Figure 14. Search by LASTNAME example

In the example shown in Figure 14, only the LASTNAME predicate is relevant. For each WHERE clause predicate to be true, one side of the OR must be true. Since 'A'='B' is false, LASTNAME='PURCELL' must be true for the row to qualify. And since 'A'='A' is true, it is irrelevant what the result is for the other side of the OR (CITY='ZZZZZ').

```
WHERE ('A' = 'A' OR LASTNAME = 'ZZZZZZ')
AND ('A' = 'B' OR CITY = 'NEW YORK')
```

Figure 15. Search by CITY example

Thus, to enable the CITY predicate, the query generator would create the predicate structure shown in Figure 15:

Now the LASTNAME predicate has been "disabled," and the CITY predicate "enabled."

If you have read through these predicate pruning examples and have understood this construct, then you would realize that this is another variation of coding a generic SQL to cover all potential search combinations. Other common solutions include coding all predicates as BETWEEN or LIKE predicates, and setting the values to cover the full range if a value is not required—although only the aforementioned query pattern is the target of this enhancement.

And if you are wondering what this means for documented tricks such as "OR 0=1" ? "OR 0=1" is not pruned, although other "always false" equal predicates such as "OR 1=2" are pruned. And since this enhancement only applies to always false equal and IN predicates, any other false conditions such as "OR 1>2" or "OR 0<>0" are not pruned.

Note: This enhancement only applies to literal values, and not parameter markers or host variables, and also does not apply when REOPT is used with parameter markers or host variables.

Removing Unnecessary Tables

Continuing the theme of query simplification and removing redundancy from the query. An additional enhancement under the guise of PREDPRUNE in DB2 V8/9 and enabled by default in DB2 10 is the removal of unnecessary tables in outer joins.

An OUTER JOIN is generally coded because the join relationship between two or more tables is optional. And while an INNER JOIN will only return rows that match across the join, an OUTER JOIN allows rows to be returned even if a match across the join is not found.

So, take the scenario if you were to code a LEFT OUTER JOIN, but not select any columns or apply filtering based upon the result from that optional table. If the join does not introduce duplicates, then that table join was redundant.

Figure 16 shows an example of a redundant or unnecessary table join.

Figure 16. Removing unnecessary tables from OUTER JOINs

Since the query only selects from T1 and no duplicates can be introduced by the join because of the DISTINCT in the query, then the join to T2 is unnecessary and DB2 will prune that table from the query with this new enhancement.

If a DISTINCT is not coded on the query, then the table is only considered redundant if the join columns have a unique index guaranteeing that duplicates cannot be introduced by the join.

Note: If DB2 recognizes this pattern are removes tables from the query, these tables will not appear in the PLAN_TABLE output.

SORT Performance Enhancements

Moving from predicate processing to runtime optimizations related to sort.

sort is often an area of contention in query processing, as sort is required for all workloads from OLTP through to reporting and BI queries. And these queries are all competing for the same BP resources and sort work datasets.

DB2 9 introduced numerous enhancements to the efficiency of sort which have also been extended further in DB2 10.

For a query with ORDER BY + FETCH FIRST n ROWS ONLY, if a sort for ORDER BY cannot be avoided with the use of an index, then it can be inefficient to sort a large result set when only a small number of rows are fetched.

In DB2 9, an in-memory replacement technique is used to achieve the desired order if the result is guaranteed to fit in a 32K page. In DB2 10, this is extended to 128K.

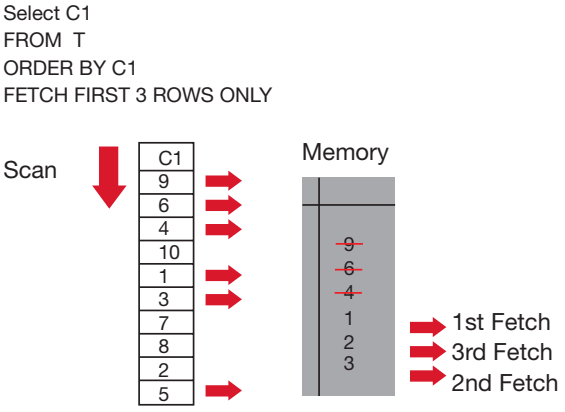

Figure 17. In-memory replacement sort

Figure 17 demonstrates an example of the in-memory replacement sort that was introduced in DB2 9 for FETCH FIRST “n” ROWS ONLY queries that require order (ORDER BY is coded). As the data is scanned, the ordered result is stored in-memory. As a new value is found that deserves to be in the first “n” rows, then this value is swapped in, and the highest stored value swapped out. This continues until all rows are processed by the chosen access path. Finally, the rows are returned in the required order.

To calculate whether this in-memory replacement technique will be used, multiply ”n” times the sort key (for ORDER BY or GROUP BY) + the data length – where ”n” is the value of FETCH FIRST “n” ROWS ONLY.

It should be noted that EXPLAIN does not show which sort technique was used.

DB2 9 also avoids allocating a physical workfile for final sort (for GROUP BY, ORDER BY, DISTINCT) if the number of rows from sort is less than 256 rows and the result can fit in a 32K page. DB2 10 extends this to intermediate sort in many situations.

And finally, GROUP BY queries with less than 32,000 groups will benefit from a hash assist to the input to sort. This allows rows to be hashed to the same location as other keys of the same value upon input to the sort process. Thus, it is more likely that the sort will be able to complete in one MERGE pass—which reduces workfile usage for sort and improves performance.

The above mentioned sort enhancements are considered runtime optimizations, and therefore explain is not aware whether these will take place.

New choices for the query optimizer

As previously discussed, the DB2 optimizer must continually evolve, both with new choices in response to challenging query patterns, and cost model changes in response to the evolution of query workloads and associated performance challenges experienced by existing customers.

Since our customers have grown accustomed to certain optimizer behavior from their existing workloads, then one challenge is to ensure that the gradual evolution of improvements do not greatly disturb this balance.

Minimizing Optimizer Challenges for the optimizer cost model

Query performance regressions are a possibility with any database management system. Fortunately for our DB2 for z/OS customers, their experiences are that regressions represent a very small percentage of their workload.

The reasons as to why cost-based optimization may not always generate the optimal plan include the following:

- Insufficient statistics.
- Unsubstantiated query optimization assumptions due to lack of knowledge of the actual values to be used at execution time.
- Unpredictable runtime resource availability, for example, RID pool usage and other concurrent activity.

In all, the plan picked by purely cost-based optimization may lack some robustness to PREPARE for various scenarios on some queries. To deal with some uncertainties, DB2 10 begins to introduce the concept of risk into the cost estimation process. The optimizer can choose the plan that has the lowest risk associated with it, within the range of access paths that are considered close to the lowest cost.

The simplest example is for the optimizer to answer the question, "How many rows qualify WHERE BIRTHDATE < ?" Since the predicate is a parameter marker or host variable, then the optimizer cannot accurately estimate the selectivity of the predicate until the literal value is known. It is entirely possible that anywhere from 0 to 100 percent of the rows could qualify, depending on the value used at execution time.

But since a majority of queries use parameter markers or host variables, then this type of predicate remains a challenge for the optimizer to estimate accurately. DB2 10 considers the risk of the estimate associated with such a predicate in its cost estimation.

This enhancement is not something that can be controlled by customers, and is an enhancement to the internal optimizer cost model to help choose an access path that has both the lowest cost and lowest risk.

Minimizing risk of RID failure

Another area of risk for execution performance regression is when the optimizer chooses a list prefetch plan, and an inaccurate estimation results in a RID-pool overflow or other RID limit is reached. When this happens, RID access falls back to tablespace scan and it loses all index filtering.

Historically, the DB2 optimizer will perform RID threshold checking as part of query optimization to avoid this runtime performance degradation. However, the optimizer may mistakenly estimate the number of qualified rows, or many concurrent queries may compete for the RID resources. Each of which could cause a limit to be reached.

DB2 10 is enhanced to failover to writing the RIDs to a workfile and continue RID processing rather than falling back to tablespace scan in many situations.

There is a zparm to control the maximum amount of workfile usage for RID processing—MAXTEMPS_RID. However, it is recommended that this zparm be used as a safety net, rather than a general use setting. The recommendation to use the default avoids a scenario of reaching a RID threshold, and failing over to a workfile and continuing processing only to reach the zparm limit and finally fallback to tablespace scan.

Hybrid join already supports incremental RID processing once a RID limit is reached, and DB2 9 Dynamic Index ANDing already supports writing RIDs to workfile instead of falling back to tablespace scan. Therefore, the major targets for this enhancement are list prefetch and multi-index access. There still exist cases where fallback to tablespace scan will occur, such as queries involving column functions (MAX, MIN etc.).

To reduce the incidences of RID pool overflow, DB2 10 also increases the RID pool default (zparm MAXRBLK) from 8 MB to 400 MB.

OPTIMIZE FOR 1 ROW fix

OPTIMIZE FOR 1 ROW (OF1R) has been documented for many releases that it will attempt to choose an access path that avoids a sort in an effort to return the first row quickly.

DB2 development received requirements from DB2 9 and DB2 10 beta customers to strengthen this OF1R sort avoidance behavior based upon seeing some queries choose an access path that sorted when OF1R was coded. This is because the imple-mentation of OF1R encouraged a sort avoidance plan, but still allowed a cost-based decision for a sort path to be chosen if estimated to be efficient.

These requirements led to an enhancement to DB2 10 before GA for the optimizer to block SORT plans if OF1R was coded. Unless of course, no sort avoidance plans exist.

A year after GA, a small number of customers saw access path regressions where OF1R queries switched from matching index access plus sort, to a sort avoidance plan that was less efficient.

For example, DB2 may have chosen a non-matching index scan to avoid the sort instead of the matching index plan that sorted. Non-matching index scan can be an inefficient choice if a large number of rows need to be scanned to find the first row that qualifies against the WHERE clause predicates.

Figure 18 demonstrates this challenge.

```
IDX1 (FIRSTNAME)
IDX2 (LASTNAME, FIRSTNAME)

SELECT *
FROM PHONEBOOK
WHERE FIRSTNAME = ?
ORDER BY LASTNAME, FIRSTNAME
OPTIMIZE FOR 1 ROW
```

Figure 18. OPTIMIZE FOR 1 ROW example.

In Figure 18, should the optimizer choose to match on IDX1 and sort? Or should the optimizer choose non-matching index scan (with index screening on FIRSTNAME) and avoid the sort using IDX2? The answer (unfortunately) is data dependent.

Using the first names of two recent US presidents as an example to demonstrate:

WHERE FIRSTNAME='GEORGE' may return many rows with IDX1 and thus sort this larger number of rows to retrieve the first qualified row by LASTNAME, FIRSTNAME order. But a non-matching index will avoid the sort and find the first 'GEORGE' early in the scan, thus avoiding processing and sorting a large number of rows.

However, FIRSTNAME='BARACK' is likely to scan a large percentage of IDX2 before finding a match due to this being an uncommon first name. This means that IDX1 is a safer choice for this example because it will match to find all occurrences of FIRSTNAME='BARACK' quickly, and since a small number of rows are likely to qualify, the sort will be efficient.

From this customer experience, we learned that we have two potential users of OF1R:

1. Customers who used OF1R to guarantee sort avoidance (and/or list prefetch, etc.) for targeted queries.
2. Customers who may have used OF1R more pervasively. This may be because it was adopted as a site standard for their online transactions, or to solve a one-time query performance issue that has now gone, or by programmers inadvertently copying SQL that included the OF1R clause.

APAR PM56845 adds zparm OPT1ROWBLOCKSORT which controls OF1R behavior. The default is DISABLE, meaning that the optimizer attempts to choose an access path that avoids a sort if it is estimated to be cost effective in returning the first row quickly. However, the optimizer is free to consider plans that require a sort. Setting the zparm to ENABLE disables plans that require a sort, and chooses the lowest cost plan that avoids a sort. DISABLE is more consistent with the behavior of OF1R prior to DB2 10.

Extending VOLATILE TABLE usage

Another option widely used by customers to influence a particular optimizer behavior is the VOLATILE table attribute. VOLATILE table support was added in DB2 V8 based upon a requirement from SAP to support their Cluster tables. The requirement was to always preference index access over a tablespace scan, and guarantee that the data rows would be accessed in the index sequence, which meant no list prefetch.

Many customers have true volatile tables—where the data volumes grow and shrink continually—making it difficult to collect a representative and reliable set of RUNSTATS data.

However, these true volatile data tables do not always fit the SAP model for VOLATILE. The limitation on list prefetch would cause index access to be avoided if the only matching index plan requires list prefetch.

DB2 10 extends the VOLATILE table support to the general use case, without impacting the SAP case. To do this, if a table has only one index, and that index is unique, then DB2 10 will continue to follow the original SAP cluster table rules. Since SAP cluster tables only have one index—and that index is defined as unique, then this rule in DB2 10 preserves the SAP-specific behavior.

If the table has more than one index (or the only index is non-unique), then DB2 uses the NPGTHRSH zparm rules which prioritize matching index access over tablespace scan or non-matching index scan. With NPGTHRSH, list prefetch is permitted.

Note: The goal of both VOLATILE (at the table level) and NPGTHRSH (at the subsystem level) is to choose the index with the most matching predicates. If multiple indexes have the same high matching columns, then these indexes compete on cost. There is still a benefit to attempting to collect representative statistics on VOLATILE tables because this will be used in the optimizer cost-based decision.

Index INCLUDE columns

While not strictly related to the optimizer, the following enhancement can improve query performance if used correctly, and can also simplify the choices available to the optimizer. The enhancement is called index INCLUDE column support, which has been a long-standing requirement from customers.

It is very common to see customers create indices to support index-only access. However, for cases where you add columns to a unique index, then prior to DB2 10, the customer is required to keep two indices. One index supports the uniqueness of the business rule, and the other supports index-only access.

DB2 10 adds support for “include columns” to be added to a unique index.

In Figure 19 below, assume that IDX1 guarantees the uniqueness of C1, and IDX2 is created to provide index-only access for some queries:

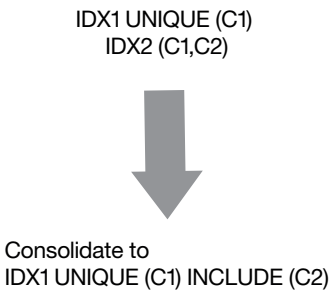

Figure 19. Index INCLUDE columns.

In DB2 10, you can alter IDX1 to add C2 as an "INCLUDE" column, and then drop IDX2. Alternatively, you can create a new index as IDX3 UNIQUE (C1) INCLUDE (C2). And then drop the original indexes — IDX1 and IDX2. Regardless of the method used, the goal as demonstrated in Figure 19 is to consolidate the two existing indexes into one.

Columns that are included cannot be matching columns for queries and cannot provide ordering for GROUP BY or ORDER BY. However, since the preceding columns are unique, matching on all columns preceding the included columns will guarantee one row or less is returned.

Therefore, while some queries may see a reduction in matching columns if there are predicates on all columns of the larger index, there is not expected to be any measurable performance difference.

Note: The true motivation for this enhancement is to reduce the number of indices on a table where additional indices have been added for index-only. Fewer indices can improve INSERT/UPDATE/DELETE performance, utility performance, reduce space and potentially improve buffer pool hit ratios as fewer indices are competing for BP resources.

This comment should seem obvious, but adding more columns to an index increases the size of the index. Adding include columns may degrade the performance of queries that were previously using the original (smaller) index.

Improving parallelism efficiency and removing limitations

The introduction of zIIP processors has increased the motivation for many customers to exploit parallelism. This is in addition to the traditional reason for parallelism to reduce the elapsed time of long running queries. DB2 10 continues the theme from prior releases of reducing parallelism limitations, and also introduces enhancements to improve distribution of work across child tasks.

Removing parallelism limitations

In previous releases, when multi-row fetch is used, parallelism is disabled for the last parallel group in the top-level query block for many queries.

For example, for a simple query SELECT * FROM TABLE, if multi-row fetch is used, then parallelism is disabled. This restriction forces customers to choose between multi-row fetch and parallelism. Alternatively, if the customer does attempt both, then DB2 may choose to introduce a final SORT, if possible, so that parallelism and multi-row fetch can co-exist.

DB2 10 removes this restriction, but only if the query explicitly contains the FOR FETCH/READ ONLY clause. The restriction still exists for an ambiguous cursor (a query that is not explicitly a read-only query).

Another parallelism restriction that is removed in DB2 10 is when the parallel group contains a workfile. In many situations, parallelism was disabled when a workfile was contained within a parallel group. But in DB2 10, the workfile can be shared across parallel child tasks. This enhancement only applies to CPU parallelism, and does not extend to FULL OUTER JOINs.

Effectiveness of query parallelism

Once parallelism is chosen for a query, it is often a challenge for DB2 to ensure that the data is distributed evenly across each parallel child task. To effectively reduce the overall elapsed time of a query using parallelism, it is important that each child task executes approximately the same amount of work, otherwise the elapsed time is dictated by the single longest running task.

The key ranges for each child task are decided at BIND/PREPARE time based on statistics such as LOW2KEY, HIGH2KEY, and/or histogram statistics. It is assumed by the optimizer that the data is uniformly distributed throughout the range of LOW2KEY and HIGH2KEY, unless histogram statistics exist to provide more detail about the data distribution throughout the range.

This makes DB2 too dependant on the availability and accuracy of the statistics. Since histograms are not generally collected by customers, the problem of uneven distribution of the parallel child tasks can be all too common.

This challenge of uneven distribution of the parallel child tasks is most evident when DB2 uses the index key ranges or IN-list elements to cut the parallel degrees. When parallelism degrees are cut based on page ranges or partition boundaries, the work is often more evenly distributed. Key range partitioning is used, in general, when the access path is driven by an available index—which is often desirable if this index also provides other predicate filtering.

```
SELECT *
FROM MEDIUM_TABLE M, LARGE_TABLE L
WHERE M.C2 = L.C2
AND M.C1 BETWEEN CURRENT DATE – 90 DAYS AND CURRENT DATE;
```

Figure 20. Sample query for determining parallel ranges

Given the query in Figure 20, DB2 will consider the LOW2KEY and HIGH2KEY of the date column C1, and distribute the keys across the number of degrees. At execution time, each parallel child task operates against its defined range of keys. But if the data is not evenly distributed across those key values, then some parallel tasks may be processing fewer rows than other tasks—or zero rows.

Dynamic Record Range Partitioning

DB2 10 introduces dynamic record range partitioning to help address the problem of uneven distribution of the parallel child tasks. In dynamic record range partition, DB2 introduces a sort into the access path so that the exact number of rows and key values are known at execution time. The resultant rows from the sort will be evenly divided across the parallel child tasks for subsequent join operations.

This division of work doesn't have to be on the key boundary, unless it is required to support GROUP BY or DISTINCT ordering. Record range partitioning is therefore dynamic because partitioning is no longer based on the key ranges decided at BIND/PREPARE time.

Instead, the key ranges are based on the number of composite records and the number of work elements (parallel child tasks). All the problems associated with key partitioning, such as the limited number of distinct values, lacking of statistics, data skew or data correlation, are bypassed and the composite side records are distributed evenly based upon the actual number or rows sorted at query execution.

This sort is not free however, and therefore the cost of the sort is taken into consideration by the optimizer in its cost-based access path decision.

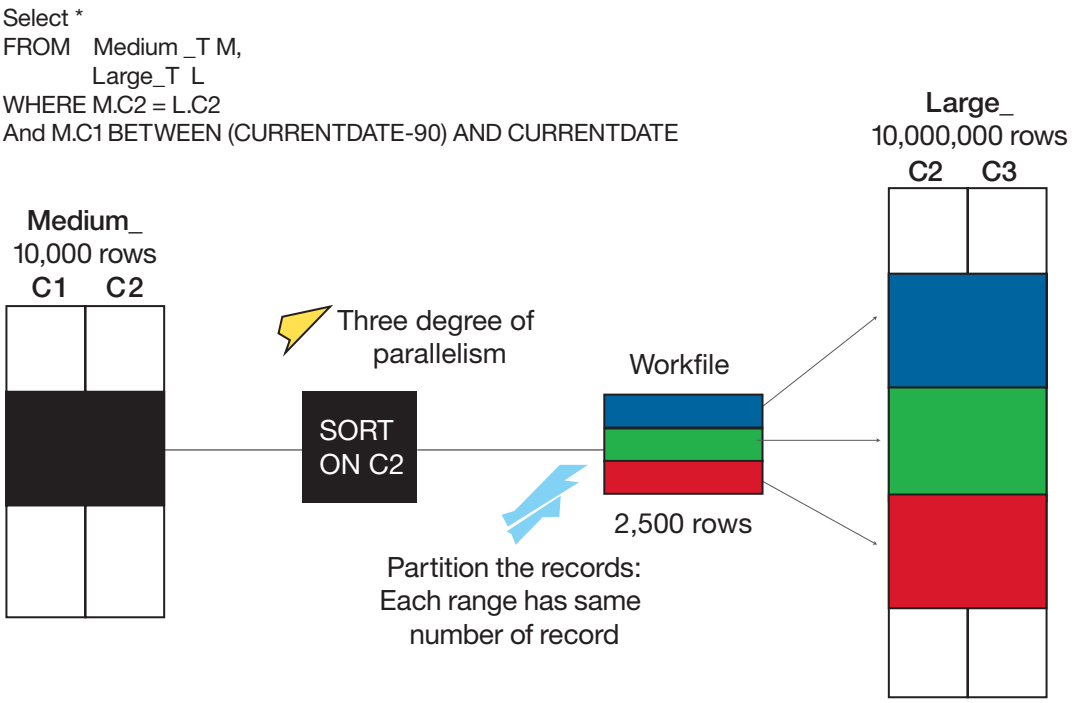

Figure 21. Even distribution of work with dynamic record range partitioning.

Figure 21 provides an example of how the output of the sort is evenly distributed between the parallel child tasks, such that each child task processes the same number of rows for the next join step.

While this enhancement overcomes data skew or uneven distribution of work on the driving table in a join operation, it is still possible that subsequent joins introduce a further unevenness to the number of rows being processed.

And as previously mentioned, dynamic record range partitioning is a new cost-based decision for the optimizer when parallelism is enabled for a query.

Straw model parallelism

The second solution in DB2 10 to deal with this uneven distribution challenge is straw model parallelism. The concept behind straw model is the DB2 will break up the access into more work elements than there are concurrent parallel degrees. And therefore with straw model, there is an opportunity to ensure that no single task is monopolizing the work.

For straw model, zparm PARAMDEG still drives the number of concurrent parallel degrees, but since more work elements are created, there is a queue of elements waiting. As each child task completes, it takes the next work element from the queue and begins processing.

```
Select *
FROM  Medium _TM
WHERE M.C1 BETWEEN 20 AND 50
```

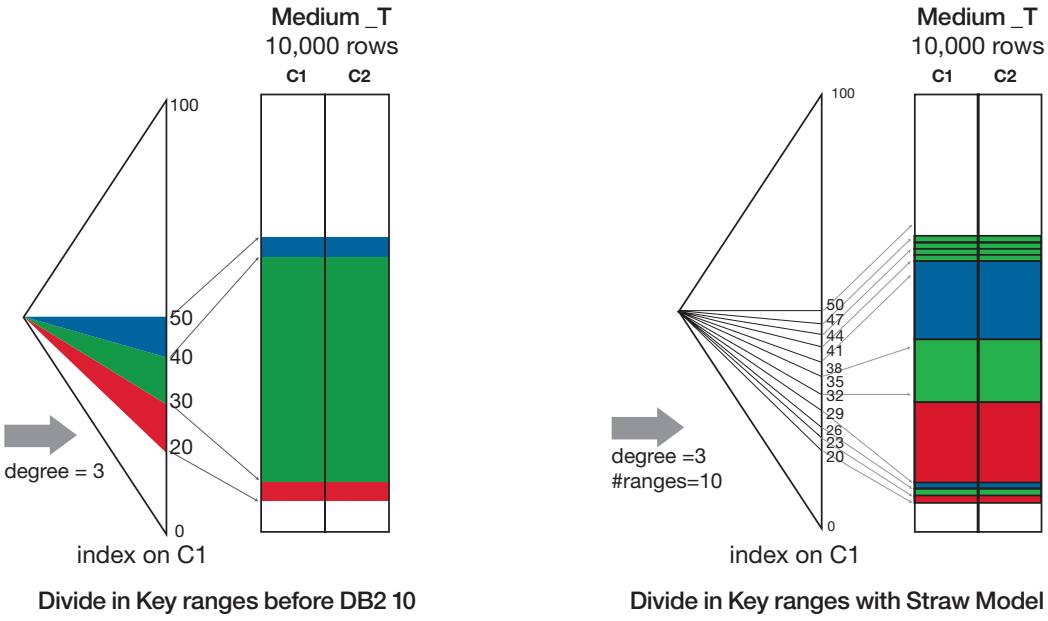

Figure 22. Straw model parallelism example.

Figure 22, assume PARAMDEG is 3. The left side of the figure shows the work distributed without the benefit of straw model, and the right side using straw model. For the straw model example, DB2 has chosen to create 10 work elements, with PARAMDEG=3 dictating the number that actually execute concurrently.

Note: the number of work elements is influenced by factors such as the number of partitions if parallelism cuts on page ranges, or the number of keys if parallelism cuts on key ranges.

Thus, for the straw model example in Figure 22, tasks 1 to 3 will process the first three work elements. If child task #2 completes first, it will take the fourth work element to begin processing. Next, if task #3 completes, it will take the fifth work element, and so on, until all work elements are complete.

This straw model process allows parallelism to cut the work into a finer degree of granularity. If the work is not evenly distributed, then the shorter running tasks will complete quicker and begin on the next element in the queue.

Cutting to a finer degree of granularity increases the likelihood that more tasks will share the work, and avoid the situations where one task processes all of the rows and other child tasks process zero rows.

Both straw model and dynamic record range partitioning are new choices available to the optimizer in DB2 10 for z/OS.

Improving the inputs to the query optimizer

While a discussion on RUNSTATS is not technically an optimizer topic, the RUNSTATS utility is the method to capture the catalog statistics that the optimizer uses for access path selection. And therefore, any enhancements to RUNSTATS may ease the burden associated with the statistics collection process and also improve the stability of access path choices.

When discussing RUNSTATS and the optimizer, it is common to hear the question, "When is the optimizer going to take advantage of real-time statistics (RTS)?" The answer is: in DB2 10.

But what needs an explanation is the other questions that also arise, such as, "When, or can I, stop running RUNSTATS and rely on RTS?" This question implies some misunderstanding of what information RUNSTATS and RTS can both provide to the DB2 optimizer.

In simple terms, RTS provides volume information, for example, how many INSERTS/UPDATES/DELETES have occurred since the last REORG or RUNSTATS, and for indexes, how many pseudo-deleted index entries and near/far leaf pages.

While some of the optimizer decisions are based on object size (because it can cost more to access a one million row table compared with a one hundred row table), the optimizer needs to determine the selectivity of WHERE/ON clause predicates to estimate the cost associated with accessing each object. This means it needs column cardinalities (COLCARDF) and frequencies/histograms and more. And RTS does not provide this information.

Since the optimizer still relies on RUNSTATS, DB2 10 also includes improvements to the usability and performance of the RUNSTATS utility.

Optimizer Validation with Real Time Statistics (RTS)

If RTS doesn't provide all of the information necessary for the optimizer, how does the optimizer use RTS in V10?

The simple answer is that RTS is being used as a "sanity check" for certain exception conditions.

The situations where the optimizer will validate the catalog statistics against RTS include:

- Catalog shows that the table, or the qualified partitions are empty,
- or if the table is marked as VOLATILE
- or the table qualifies for NPGTHRSH (NPAGESF < NPGTHRSH zparm).

In the above scenarios, DB2 will read the RTS tables during static BIND/REBIND or dynamic PREPARE to validate the number of rows in the table. This value is then used in the optimizer's access path selection.

Note: Since RTS was integrated into the DB2 catalog in DB2 9 NFM, customers migrating to DB2 10 from DB2 9 are able to exploit optimizer validation with RTS in DB2 10 CM9. Customers migrating from DB2 V8 must wait until DB2 10 NFM before the optimizer can exploit RTS.

In addition to reading RTS, DB2 10 adds a further validation for WHERE clause predicates using a probe of index non-leaf pages for exception conditions. If a WHERE clause predicate is estimated by the optimizer to qualify zero rows, and there exists an index that would support matching index access, then DB2 will also probe the index non-leaf pages during BIND/PREPARE to validate the predicate estimate.

This index probing only applies if the optimizer has the literal value at BIND/PREPARE to use for the index probe. This means the query must contain literals rather than host variables or parameter markers, or the query uses the REOPT BIND parameter. RTS validation however, does not require the literal values or REOPT to validate the number or rows in the table or partition.

This predicate and table size validation is externalized in a new EXPLAIN table called the DSN_COLDIST_TABLE.

RUNSTATS Problem Summary and automation

Understanding what RUNSTATS options to collect, and when to collect them, is a complex task. Answering the question, "When to collect statistics?" has been made easier with the stored procedure—DSNACCOX.

But the second question, "What to collect?" still remains a challenging task without the benefit of some tooling. Fortunately, IBM has the Data Studio Statistics Advisor as a standalone tool that can analyze a query or a workload and determine which statistics would benefit.

In addition to the above solutions that are already available, DB2 10 has taken steps to improve the automation of statistics collection.

The DB2 10 solution for *automating statistics maintenance* is through a set of stored procedures that can monitor the need for statistics collection and schedule the execution of RUNSTATS.

The DB2 10 for z/OS Managing Performance Guide contains a section titled, Automating statistics maintenance. This section provides more detailed information, including the steps required to setup the monitoring.

This process will issue RUNSTATS alerts for out-of-date statistics, missing and conflicting database statistics. However, it should be noted that this process is not determining tailored RUNSTATS for your query workloads, which is the goal of a statistics advisor tool.

Once the required statistics are identified, then executing RUNSTATS through this automated process can collect those statistics through the exploitation of statistics profiles which are discussed in the next section.

Note: The target for these stored procedures is to simplify integration with tooling provided by various vendors. However, this does not preclude a customer from integrating this into their existing statistics collection process.

In addition to RUNSTATS automation, there are additional DB2 10 enhancements that attempt to help with RUNSTATS complexity and cost.

RUNSTATS simplification

The first enhancement related to RUNSTATS simplification is related to the KEYCARD option becoming the default for index RUNSTATS—regardless of whether it is explicitly specified or not. This is in response to the ongoing recommendation by IBM for customers to collect KEYCARD to provide more accurate information to the optimizer about the intermediate multi-column cardinalities for indexes with three or more columns.

The general recommendation for statistics has been to collect RUNSTATS TABLE(ALL) INDEX(ALL) KEYCARD, and now with KEYCARD defaulted, the recommendation can be simplified to RUNSTATS TABLE(ALL) INDEX(ALL). A generalized recommendation for additional FREQVAL, HISTOGRAM or COLGROUP statistics is more complicated however. The requirement for these additional statistics is based on the predicates from an individual query or workload. This is why tooling is often necessary to assist in the identification of this requirement.

Once additional statistics requirements are identified, then DB2 10 allows you to create an individual statistics profile for each table such that the execution of RUNSTATS can simply use the option USE PROFILE to collect these targeted statistics for a table.

Figure 23 provides some examples of the statistics profile options in DB2 10.

RUNSTATS options to SET/UPDATE/USE a stats profile

- Integrate specialized statistics into generic RUNSTATS job
  - RUNSTATS … TABLE tbl COLUMN(C1)… SET PROFILE
    Alternatively use SET PROFILE FROM EXISTING STATS
  - RUNSTATS … TABLE tbl COLUMN(C5)… UPDATE PROFILE
  - RUNSTATS … TABLE tbl USE PROFILE

Figure 23. RUNSTATS profile options

The statistics profile options include the ability to:

- SET PROFILE,
- UPDATE PROFILE,
- DELETE PROFILE,
- and USE PROFILE.

When included in a RUNSTATS execution, SET and UPDATE do not physically execute the RUNSTATS job to collect statistics. They simply create or update the profile.

One nice feature of this statistics profile is the ability to SET PROFILE FROM EXISTING STATS whereby DB2 will create the statistics profile based upon that statistics that exist in the catalog for this table. This can be beneficial if you have collected FREQVAL, HISTOGRAM and/or COLGROUP at different stages in the past but do not know all of the options that have been used in the past.

UPDATE PROFILE can merge the existing profile information with the new options when you wish to add new options to a profile. If you wish to truly replace the profile, then it is preferable to DELETE PROFILE followed by a new SET PROFILE.

The only concern with this approach is that deleting a profile does not delete existing statistics. Leaving statistics in the catalog that are no longer being recollected will cause those statistics to become stale. Therefore, an additional step is to manually delete those statistics from the catalog that are no longer part of a profile.

The statistics profiles are not without their own usability challenges. When USE PROFILE is specified, all tables specified in the RUNSTATS job must have a profile. Therefore, if you have 10,000 objects to collect statistics on, and only 50 have specialized RUNSTATS requirements, then you cannot integrate the 9950 default RUNSTATS tables with the statistics profile tables. The current implementation of statistics profiles does however, simplify the RUNSTATS syntax when issuing a single table RUNSTATS.

RUNSTATS performance

The other important enhancement to RUNSTATS is in the area of performance. The existing SAMPLE option of RUNSTATS can reduce CPU cost because it only performs the CPU-intensive cardinality calculation on the percentage specified by the sample keyword. However, with SAMPLE, all 100 percent of the rows are still read by the RUNSTATS job. DB2 10 adds the option to sample at the page/row level— which means a reduction in the rows that RUNSTATS will read. This can have a more significant improvement in the CPU and elapsed time performance of RUNSTATS.

When given the option to choose a percentage, the most obvious question is, "What percentage to use?" DB2 provides the TABLESAMPLE SYSTEM AUTO option, which allows DB2 to choose the correct percentage based on the table size. When specifying the AUTO option, tables with less than 500,000 rows will use a 100 percent sample, and after 500,000 rows, the percentage will scale down from 100 percent to a low value of 10 percent. RUNSTATS will adjust the percentage with the AUTO option, or you can override with a specific numeric value.

Being more aggressive with a lower percentage SAMPLE value can reduce the RUNSTATS cost. However, there is some risk of being too aggressive since RUNSTATS is unable to estimate the values for the rows that are skipped. For this reason, DB2 implements a lower value of 10 percent.

TABLESAMPLE only applies to single-table tablespaces and does not apply to LOB tablespaces. Indices do not exploit this page and row-level sampling.

Summary

DB2 10 for z/OS is a significant release for query performance and optimization. It acknowledges the increased focus our customers are placing on reducing total cost of ownership while maintaining the stability and reliability of their mainframe environments.

While prior releases of DB2 have brought new optimizer access path choices, this additional focus on predicate and runtime optimizations, and plan management, should help achieve the goal of improved performance with lower risk than before. A focus that is expected to continue for future releases of DB2.

IBM Corporation
Software Group
Route 100
Somers, NY 10589

Produced in the United States of America
September 2012

IBM's statements regarding its plans, directions, and intent are subject to change or withdrawal without notice at IBM's sole discretion. Information regarding potential future products is intended to outline our general product direction and it should not be relied on in making a purchasing decision. The information mentioned regarding potential future products is not a commitment, promise, or legal obligation to deliver any material, code or functionality. Information about potential future products may not be incorporated into any contract. The development, release, and timing of any future features or functionality described for our products remains at our sole discretion.

Please Recycle

IMW14646-USEN-00