bmcsoftware

Best Practices for DB2 on z/OS Performance

A Guideline to Achieving Best Performance with DB2

Susan Lawson and Dan Luksetich
www.db2expert.com
and
BMC Software

September 2008

YL&A®
www.ylassoc.com

Contacting BMC Software

You can access the BMC Software website at http://www.bmc.com. From this website, you can obtain information about the company, its products, corporate offices, special events, and career opportunities.

United States and Canada

Address	BMC SOFTWARE INC 2101 CITYWEST BLVD HOUSTON TX 77042-2827 USA	**Telephone**	713 918 8800 or 800 841 2031	**Fax**	713 918 8000

Outside United States and Canada

Telephone	(01) 713 918 8800	**Fax**	(01) 713 918 8000

Restricted rights legend

Customer support

You can obtain technical support by using the BMC Software Customer Support website or by contacting Customer Support by telephone or e-mail. To expedite your inquiry, see “Before contacting BMC.”

Contents

About this book

This book contains general information about improving performance for DB2 on z/OS and specific information about how BMC products help improve performance.

Susan Lawson and Daniel Luksetich of YL&A wrote much of the general material, and BMC supplied the product-specific text.

Susan Lawson and Daniel Luksetich of YL&A cite the following references.

Category	Document
GC18-9846	IBM DB2 9 Installation Guide
SC18-9840	IBM DB2 9 Administration Guide
SC18-9854	IBM DB2 9 SQL Guide and Reference
SC18-9851	IBM DB2 9 Performance Monitoring and Tuning Guide

For more information on any BMC products, refer to the product documentation or www.bmc.com.

bmcsoftware

Chapter 1

How DB2 manages performance

By Susan Lawson and Daniel Luksetich, YL&A Associates

The DB2 optimizer

The optimizer is the component of DB2 that determines the access plan to be used. During the preparation (static bind or dynamic prepare) of an SQL statement, the SQL compiler is called on to generate an access plan. The access plan contains the data access strategy, including index usage, sort methods, locking semantics, and join methods.

To predict performance or troubleshoot performance problems, we need to understand what the optimizer is doing in order. Here we look at the various access paths and components evaluated by the optimizer when determining the plan for data access.

Access paths

Access paths are determined by the optimizer at bind time or at run time. Many factors go into the optimizer's access path choice. The following discusses some of the access paths the optimizer may choose to access your data. It also discusses how to determine if it is the best path or if we should provide help, such as better indexes, to encourage a better access path.

Table space scan

A table space scan (sometimes referred to as a relational scan) is an access path where DB2 has chosen to scan the data and not use an index. This type of access path is most often used for one of the following reasons:

- A matching index scan is not possible because an index is not available or no predicates match the index columns.
- A high percentage of the rows in the table are returned. An index is not really useful in this case, because it would need to read all the rows.
- The indexes have matching predicates with low cluster ratios and are therefore efficient only for small amounts of data.
- Access is through a created global temporary table (indexes are not allowed on these types of tables).

Limited partition scan and partition elimination

DB2 can limit the number of partitions scanned for data access or eliminated for the process of partition pruning. The query must provide the leading columns of the partitioning key in order to tell DB2 which partitions are to be scanned or eliminated.

Index access

Index access occurs when the DB2 optimizer has chosen to access the data via one or more of the given indexes. You can have single index access or multiple index access. Multiple index access typically occurs when there are compound predicates in a query connected by “AND” and / or “OR”. DB2 can use different indexes, or the same index multiple times to match combinations of predicates. The RIDs of the qualifying index entries are then intersected (AND) or unioned (OR) before the table space is accessed.

The number of columns that match in the index will determine if you are using a non-matching index scan or a matching index scan.

Increasing the number of matching columns can help query performance.

If all the columns needed for the query can be found in the index, and DB2 does not access the table, then the access is consider index only.

Prefetch

Prefetching is a method of determining in advance that a set of data pages is about to be used and then reading the entire set into a buffer with a single asynchronous I / O operation. The optimizer can choose three different types of prefetch: list, sequential, or dynamic.

Nested loop join

A nested loop join can be used for both inner and outer join types. During a nested loop join, DB2 scans the composite (outer) table. For each row in that table that qualifies by satisfying the predicates, DB2 searches for matching rows of the new (inner) table. DB2 concatenates any rows it finds in the new (inner) table with the current row of the composite table. If no rows match the current row, DB2 discards the current row if it is an inner join. If it is an outer join, DB2 concatenates a row of null values. Stage 1 and stage 2 predicates can eliminate unqualified rows before the physical joining of rows occurs.

Nested loop join is often used when:

- The outer table is small.

- The number of data pages accessed in the inner table is small.
- Predicates with low filter factors reduce the number of qualifying rows in the outer table.
- An efficient, highly clustered index exists on the join columns of the inner table (or DB2 can dynamically create a sparse index on the inner table, and use that index for subsequent access).

A nested loop join repetitively accesses the inner table. DB2 scans the outer table once and accesses the inner table as many times as the number of qualifying rows in the outer table. Therefore, this join method is usually the most efficient when the values of the join column passed to the inner table are in sequence and the index on the join column of the inner table is clustered, or the number of rows retrieved in the inner table through the index is small. If the joined tables are not clustered in the same sequence, DB2 can sort the composite to match the sequence of the inner table. Accesses to the inner table can use dynamic prefetch.

Hybrid join

The hybrid join method applies only to an inner join. It requires an index on the join column of the inner table. This join method obtains the record identifiers (RIDs) in the order needed to use list prefetch. DB2 performs the following steps during this method:

1. Scans the outer table (OUTER).
2. Joins the outer tables with RIDs from the index on the inner table. The result is the phase 1 intermediate table. The index of the inner table is scanned for every row of the outer table.
3. Sorts the data in the outer table and the RIDs, creating a sorted RID list and the phase 2 intermediate table. The sort is indicated by a value of Y in column SORTN_JOIN of the plan table. If the index on the inner table is highly clustered, DB2 can skip this sort; the value in SORTN_JOIN is then N.
4. Retrieves the data from the inner table, using list prefetch.
5. Concatenates the data from the inner table and the phase 2 intermediate table to create the final composite table.

The hybrid join method is often used if non-clustered index is used on the join columns of the inner table or if the outer table has duplicate qualifying rows.

Merge scan join

A merge scan join requires one or more predicates of the form TABLE1.COL1 = TABLE2.COL2, where the two columns have the same data type, length, and null attributes. If the null attributes do not match, the maximum number of merge join columns is one. The exception is a full outer join, which permits mismatching null attributes.

DB2 scans both tables in the order of the join columns. If no efficient indexes on the join columns provide the order, DB2 might sort the outer table, the inner table, or both. The inner table is put into a work file; the outer table is put into a work file only if it must be sorted. When a row of the outer table matches a row of the inner table, DB2 returns the combined rows.

A merge scan join is often used if

- The qualifying rows of the inner and outer table are large, and the join predicate does not provide much filtering; that is, in a many-to-many join.
- The tables are large and have no indexes with matching columns.
- Few columns are selected on inner tables. This is the case when a DB2 sort is used. The fewer the columns to be sorted, the more efficient the sort is.

Star join

The star join method is the access path used in processing a star schema. A star schema is composed of a fact table and a number of dimension tables that are connected to it.

To access the data in a star schema, SELECT statements are written to include join operations between the fact table and the dimension tables; no join operations exist between dimension tables. A query must satisfy a number of conditions before it qualifies for the star join access path. The first requirement is detection of the fact table. Given that the access path objective is efficient access to the fact table, it is important that the fact table is correctly identified.

Read mechanisms

DB2 uses various read mechanisms to access data in an efficient manner, including:

- Dynamic prefetch
- Sequential prefetch
- List prefetch
- Index lookaside

Dynamic prefetch

Dynamic prefetch, a powerful performance enhancer, is the ability of DB2 to dynamically detect when a page set (table space or index space) is being read in a sequential manner. It keeps track of the last 8 pages accessed, and if 5 of them have been determined to be “sequentially available” DB2 launches prefetch readers. Sequentially available pages are not necessarily pages that are next to each other, but they are within the prefetch quantity (typically 32 pages) divided by 2. This information is saved until an application commits if it is bound RELEASE(COMMIT) and saved over a commit if bound RELEASE(DEALLOCATE); therefore, RELEASE(DEALLOCATE) should be the performance choice for batch operations that can take advantage of dynamic prefetch. Remote applications cannot take advantage of RELEASE(DEALLOCATE).

Sequential prefetch

Sequential prefetch is most often used in conjunction with a table space scan. In fact as of DB2 9, sequential prefetch is used only for table space scans. When sequential prefetch is enabled, DB2 will issue consecutive asynchronous I/O operations, reading several pages with each I/O operation. This means that the pages requested will be staged in the buffer pool before they are needed by the application. The quantity of pages read in a single prefetch I/O operation depends upon a number of factors, including the buffer pool size, page size, and the VPSEQT setting of the buffer pool.

List prefetch

List prefetch is typically used when the optimizer determines that access will be via an index, but the cluster ratio of the index indicates that there can be random reads against the table space. As opposed to potentially reading several table space pages randomly, or perhaps even redundantly, DB2 will collect the index entries that match the supplied predicates, sort the RIDs of the qualifying entries by page number, and access the table space in a sequential or skip sequential manner.

Index lookaside

Index lookaside is a process where DB2 looks to the next page in the index for the requested key instead of traversing the entire index. This can result in a great deal of get page reduction if there is a sequential access pattern to the index keys. This performance enhancer is a definite bonus when accessing an inner table in a nested loop join, and the inner table index used is in the same key sequence as the outer table. As with sequential detection, index lookaside is dependent upon information being held in memory (last page accessed). When an application commits and is bound RELEASE(COMMIT), the information is lost. An application bound RELEASE(DEALLOCATE) retains the index lookaside. Remote applications always lose the information across commits.

Locking

DB2 uses transaction locking (via the IRLM—Internal Resource Lock Manager), latches, and other non-IRLM mechanisms to control concurrency and access of SQL statements and utilities. These mechanisms associate a resource with a process so that other processes cannot access the same resource when it would cause lost updates and access to uncommitted data. Generally, a process will hold a lock on manipulated data until it has completed its work. This way it ensures that other processes do not get hold of data that has been changed but not committed. Another use is for repeatable read, when an application needs to reread data that must still be in the same state as it was when it was initially read. There are also user options to avoid locking and allow for access to uncommitted data, and system settings and mechanisms to provide for lock avoidance when it would not cause a data integrity problem.

From a performance viewpoint, we want to minimize the amount of locks that we take. From an availability standpoint, we want to minimize the duration of those looks. A lock takes about 400 CPU instructions and 540 bytes of memory. Excessive locking can be expensive, especially when using a mechanism such as row level locking.

Lock avoidance

It is important that our application get lock avoidance if possible. Lock avoidance DB2 reduces the overhead of always locking everything. DB2 will check to be sure that a lock is probably necessary for data integrity before acquiring a lock. Lock avoidance is critical for performance. Its effectiveness is essentially controlled by application commits.

Page and row locking can be avoided at execution time by letting the system use lock avoidance mechanisms. DB2 can test to see if a row or page has committed data on it. If it does, then no lock is required on the data at all. Lock avoidance is valid only for read-only or ambiguous cursors and requires a combination of events and settings to occur. First, the statement must be read-only or an ambiguous cursor and have the proper isolation level and the appropriate CURRENTDATA setting. The best setting for maximum lock avoidance is using an isolation level of CS, either as a bind parameter or on the individual SQL statement, and CURRENTDATA set to NO as a bind option. (This is the default in DB2 9.)

In “Locking and concurrency” on page 74 we will look at more details and best practices for providing the best concurrency for our applications.

Sorting

DB2 can perform sorts on the behalf of several access paths selected for an SQL statement. At startup, DB2 allocates a sort pool in the private area of the DBM1 address space. DB2 uses a special sorting technique called a tournament sort. During the sorting processes, it is not uncommon for this algorithm to produce logical work files called runs, which are intermediate sets of ordered data. If the sort pool is large enough, the sort completes in that area. It is good practice to not make this pool too small. The large portion of the sort pool is kept in real memory above the 2 GB bar.

More often than not, the sort cannot complete in the sort pool and the runs are moved into the DB2 work files that are used to perform sorts. These runs are later merged to complete the sort. When the DB2 work files are used for holding the pages that make up the sort runs, you could experience performance degradation if the pages get externalized to the physical work files because they must be read back in later to complete the sort.

You need to have several, large sort work files, if possible on separate volumes. With each release of DB2, re-evaluate the size / number to make sure it is suitable for your use because each release tends to add additional users.

Avoiding sorts

You can use a number of techniques to avoid sorts.

- GROUP BY

  A sort can be avoided for a GROUP BY if there are columns in an index used for access that match the GROUP BY columns. The index can be a unique or non-unique index, and DB2 can prune index columns if there are equals predicates provided. This means that if there are three columns in the GROUP BY clause, for example EMPNO, WORKDEPT, and SEX, and a predicate in the query like WHERE SEX='F', then DB2 can eliminate the SEX column from the grouping operation and use an index on EMPNO, WORKDEPT to avoid a sort.

- ORDER BY

  As with GROUP BY, ORDER BY can avoid a sort if an index is chosen that matches the ORDERY BY columns for single table access, or for the first table accessed in a join. Like with GROUP BY, DB2 can do ORDER BY pruning and still use an index if leading index columns are matched to single values.

- DISTINCT

  DISTINCT can use a unique or non-unique (new as of DB2 9) index to avoid a sort, much in the same way as GROUP BY. Prior to DB2 9, DISTINCT could only use a unique index to avoid a sort, and so a GROUP BY on all columns in the SELECT list may be more effective in avoiding the sort.

- INTERSECT, UNION, and EXCEPT

  A sort could be avoided if there is a matching index with the unioned / intersected / excepted columns, or if DB2 realizes that the inputs to the operation are already sorted by a previous operation. If duplicates are not possible, use UNION ALL, INTERSECT ALL, or EXCEPT ALL to avoid a sort. UNION cannot avoid a sort.

DB2 catalog statistics

The optimizer uses several statistics in the DB2 catalog for access path optimization. Without correct statistics, it is impossible for the optimizer to choose the best access path to your data. Depending on the data values and your predicates, you need to collect various statistics. Catalog statistics that are out of date and do not reflect the current condition of the data in tables is the number one cause of access path problems and poorly performing SQL.

Cardinality statistics

Basic cardinality statistics are essential for access path determination. Collect cardinality statistics for all columns used in predicates subsystem wide, or better yet on every column in every table. Cardinality statistics reflect the number of rows within a table and the number of distinct values for a column. Having these numbers is critical for the optimizer to calculate accurate filter factors for predicates. The filter factors are used to make critical decisions about access path selection.

Frequency distribution statistics

Many performance problems can be attributed to skewed distribution of data. The optimizer assumes that the data is uniformly distributed. Therefore, frequency distribution statistics are needed on columns where there is a skew in the data values that may cause poor decisions by the optimizer. As of DB2 V8, frequency distribution statistics can be collected on all columns (index, non-index, groups, etc.). The DB2 system catalog can hold any number of values from columns in which these statistics have been gathered, along with the frequency of the occurrence of the value. These type of statistics are useful for queries that have the following characteristics:

- Dynamic or static SQL with embedded literals
- Static SQL with host variables and bound REOPT(ALWAYS)
- Dynamic SQL with parameter markers bound REOPT(ALWAYS), REOPT(AUTO) or REOPT(ONCE)
- Static or dynamic SQL with host variables or parameter markers against nullable columns that cannot be null

If you have no statements with these characteristics, you don’t need to gather these types of statistics for query performance reasons.

Histogram statistics

Histogram statistics enable DB2 to improve access path selection by estimating predicate selectivity from value-distribution statistics that are collected over the entire range of values in a data set. This helps filtering estimation when certain data ranges are heavily populated and others are sparsely populated.

DB2 chooses the best access path for a query based on predicate selectivity estimation, which in turn relies heavily on data distribution statistics. Histogram statistics summarize data distribution on an interval scale by dividing the entire range of possible values within a data set into a number of intervals.

DB2 creates equal-depth histogram statistics, meaning that it divides the whole range of values into intervals (called quantiles) that each contain about the same percentage of the total number rows.

Use histogram statistics for the same types of statements that can take advantage of distribution statistics, and when there are a lot of range predicates or a need to cover the entire range of values and the distribution statistics were not gathered for every value.

Buffer pools

Buffer pool configuration can have an affect on query performance. Buffer pools can provide memory for frequently used pages and can help speed up query performance. The optimizer can be influenced by the virtual buffer pool definitions. The access path selection process *may* be affected; for example, inner and outer table selection for joins can be influenced by which table fits best into memory. For more on buffer pool tuning refer to Chapter 4, “Buffer pool tuning.”

Parallelism

The use of DB2 CPU parallelism can help reduce elapsed time for long running queries. Queries with the following characteristics will be able to take advantage of parallelism:

- Table space scans and index scans
- Joins
- Nested loop
- Merge scan
- Hybrid without sort on new tables
- Sorts
- Aggregate functions
- Long running, read-only queries, both static and dynamic SQL, from both local and remote sites

Parallelism will not be considered in only a few places:

- Queries that materialize views
- Queries that perform materialization because of nested table expressions
- Queries that perform a merge scan join of more than one column
- Queries that use direct row access

Parallelism generally should not be used if a system is already CPU-constrained, because parallelism would only add to the problem in most (but not all) situations. Sometimes getting that long-running hog query out faster will return CPU resources back to the critical tasks faster - it just depends!

CPU parallelism cannot be used when a cursor is defined WITH HOLD, because this cursor's use is potentially interrupted by a commit, which causes a stoppage in processing. The parallelism could have caused many resources to be held ineffectively if this was allowed.

Keep in mind that adding parallelism to poorly tuned queries is simply wasting resources, and you will probably not see any benefit from parallelism. Short running queries (queries that complete sub-second) are usually not going to see a great benefit from parallelism. But how often are long-running queries separated from short-running queries? If you are trying to get the benefits from parallelism and not cause unnecessary overhead where it does not belong, consider this type of granularity of processing. For example, you could separate the long-running queries into a separate package and BIND it DEGREE(ANY) in a different collection and then use the SET CURRENTPACKAGE statement to switch between it and a program bound with DEGREE(1) for shorter queries that are better to let run sequentially.

Parallelism is measured by the number of concurrent parallel processes, which is known as the degree of parallelism. This is determined in a multi-phase optimization at bind time. This degree can be seen in the ACCESS_DEGREE column in the PLAN_TABLE from the results of an Explain. The actual degree of parallelism at run-time may be different because the buffer pool size / sequential area are not taken into consideration during the BIND process along with other factors (listed below). You may experience degree degradation, or parallelism may not be used at run time.

Monitor the buffer pool attributes during execution (IFCID 202) to ensure the size and thresholds use the settings you were anticipating and were not ALTERed (IFCID 201). You also want to monitor the actual degree at run time and see how that compares to the degree determined at bind time. This is the only way to view the actual degree used at run time, through viewing IFCID 221. This IFCID also contains the QUERYNO of the query that correlates to the QUERYNO in the PLAN_TABLE. If you built a table to hold the trace data from this IFCID, you could join it on the QUERYNO column of the PLAN_TABLE to have a nice report comparing the degree

chosen at bind time with the degree chosen at run time. You could even compose a query joining those two tables to the DSN_STATEMNT_TABLE to verify the accuracy of the data to the reality. Other useful information in this IFCID includes why a particular degree was chosen, or more important, why the degree was degraded:

- 0 = degree at run was the same as bind
- 1 = host variables
- 2 = no ESA sort support
- 3 = cursor used for update or delete
- 4 = parallel group is empty
- 5 = MVS/ESA enclave services not available
- 6 = SPRMMDEG value was applied

Look at the parallel group elapsed time (IFCID 222) to evaluate if you are getting the response time improvements you were hoping to achieve with parallelism. If you are not getting any benefit, it may be best to turn it off. Remember that when you bind DEGREE(ANY), you create two access paths that must be stored in the EDM pool, thus increasing the storage needed.

Predicates and SQL tuning

Understanding how DB2 interprets and processes predicates is critical in understanding how to take you SQL tuning beyond just trying to add an index to a table.

Types of DB2 predicates

DB2 has two major components that process the predicates in your queries and subsequently filter the data. Where and how the queries are processed can make a huge difference in the elapsed and CPU time consumed during query processing. It is easy to remember what predicates are stage 1 and stage 2 by just remember some simple guidelines. In stage 1, DB2 knows about tables and indexes. In stage 2, DB2 does such things as sorting and the processing of functions.

All of the possible predicate types are covered in great detail in the IBM DB2 manuals (*DB2 9 Performance Guide* and *DB2 V8 Administration Guide*). Examples are covered here.

Stage 1 indexable

One simple example of a stage 1 predicate is a predicate of the form:

```
COL op value
```

value can be a literal, parameter marker, or host variable. If the stage 1 predicates coded in an SQL statement match leading columns of the index, they are called stage 1 matching predicates. Stage 1 matching predicates can be critical to performance because they are the only predicates that can actually physically limit the amount of data retrieved from an index, and subsequently the data. The operator can be an =, >, <, >=, or <=.

Range predicates using a greater than, less than, or a BETWEEN can be a stage 1 indexable predicates, but additional stage 1 predicates that can match columns after the range predicate in the index will not match. These predicates, however, can still be applied as index screening predicates.

Stage 1 index screening

Index screening predicates are stage 1 predicates that did not match, but are still used during index access to filter index entries before the table is accessed. The reason they are not matching is that there may be columns in the index before the predicate column on which no predicate was written. Another situation may be when there is a range predicate coded on a column before other columns of the index. Index screen predicates do not limit the range of data that has to be read from the index. They can, however, be useful in avoiding in filtering before table access in situations in which you can't match on all columns.

Other stage 1

Stage 1 predicates that are neither index matching or screening are still processed in the stage 1 engine. The reason the predicates are not matching or screening might be simply because the columns they are coded against are not in an index. Stage 1 predicates are still processed before stage 2 predicates, and will therefore use less CPU. While matching predicates are processed by DB2 in the order of the index columns, other predicates are processed in a certain order. The best advice when coding predicates that are not going to match on an index is to code them in what you believe will be the most restrictive first. One example of a stage 1 predicate that is non-indexable would be of this form.

```
COL <> value
```

The DB2 manuals clearly document all stage 1 non-indexable predicate forms.

Stage 2

Stage 2 predicates are applied after DB2 has moved data from the stage 1 to the stage 2 component of the processor. Typically a predicate is stage 2 if something has to be done to a column before the predicate can be evaluated. One example is if a function has to be applied to a column:

```
Function(COL) = value
```

Because functions are evaluated in the stage 2 component, the data for the COL in this predicate has to be moved to stage 2 and the function will be evaluated before the comparison can be made to the value. Because the stage 2 component does not access indexes and table directly, stage 2 predicates eliminate the possibility of index access, as well as the filtering of data prior to stage 2. In many situations it is possible, and often very beneficial, to rewrite stage 2 predicates as stage 1.

Stage 3

Stage 3 predicates are not a DB2 database concept. They represent predicates that are coding within the application program layer. In many cases, applications are written so that data is returned from DB2 to the application, and then for some reason the application discards some or all of the data. We like to call this stage 3 filtering. It is just about the most expensive thing you can do, and should be avoided. Going back and forth to DB2 is very expensive, and any filtering should be driven into the database as much as possible. Any stage 2 predicate is always better than a stage 3 predicate.

Combining predicates

When predicates are combined by linking them with AND and OR, the stage at which the predicates are evaluated may be affected. Any two predicates (simple or compound) connected with an OR will result in both predicates being evaluated at the highest stage predicate. For example, if a stage 1 predicate is connected to a stage 2 predicate with an OR, both predicates are non-indexable stage 2 predicates. If a stage 1 predicate is connected to a stage 1 non-indexable predicate, both are stage 1 non-indexable.

Boolean term predicates

Boolean term predicates are important to the indexability of predicates. A Boolean term predicate is, very simply, a predicate that when evaluated to false, makes the entire WHERE clause false. In that way, a Boolean term predicate evaluating false can result in the evaluation of other predicates being avoided. DB2 can only match on single indexes using Boolean term predicates. In other words, if your query contains no Boolean term predicates, the best index access that can be achieved would be a non-matching index access or multi-index access.

In some situations, you can introduce redundant Boolean term predicates to a query to achieve desired index access. Assuming an index on COL1 COL2, the following example is a typical scrolling cursor with no Boolean term predicates:

```
WHERE COL1 > :H1
OR (COL1 = :H1 and COL2 > :H2)
```

The addition of a redundant Boolean term predicate can get you the desired index access:

```
WHERE COL1 > :H1
OR (COL1 = :H1 and COL2 > :H2)
AND COL1 >= :H1
```

Predicate transitive closure

DB2 can introduce its own redundant predicates to queries by applying something called predicate transitive closure. So, under the premise of transitive closure, if A=B and B=C, then surely A=C. DB2 uses this to generate redundant predicates for join operations. If there is a join between two tables on a column A, and there is a local filtering predicate coded against column A of one table, then DB2 will generate the exact same predicate against column A of the other table. This is done so that local filter factors can be calculated against both tables, and in the case of an inner join, DB2 can then go to the table that qualifies the least number of rows first. This can result in substantial costs savings during inner joins.

Keep in mind that some predicates, such as LIKE, IN, and subqueries do not qualify for transitive closure. If you want transitive closure for these types of predicates, you must code the redundancy yourself. Also, in some cases of query tuning the redundant predicate generated by DB2 may not be desired. Changing the predicate to a LIKE, IN, or subquery can eliminate the redundancy.

Coding Efficient SQL

By Peter Plevka, BMC Software

SQL performance tuning basics

Before you can improve performance, you must analyze your SQL and understand what the application needs to do. You may need to make performance trade-offs based upon the cost of resources and the ability to test new applications.

The following questions will help you determine what options are available for tuning and the priority of each option:

- Is the SQL static or dynamic? Many application development tools today generate dynamic SQL; most of the time, you do not see the SQL prior to its execution.
- What schema is involved in the application? Can you make changes to the schema (for example, adding or removing indexes)?
- Have statistics been collected on the application objects that are referenced by the SQL? The DB2 optimizer, which is invoked when SQL gets bound to a package or gets dynamically prepared, relies on statistical information about the database objects that are accessed by the particular SQL. Even the best-written SQL will not perform well when bound without up-to-date statistics in the DB2 catalog.

You may find that one SQL statement is causing a major performance problem. In general, the 80/20 rule applies – 20% of your SQL statements probably cause 80% of your performance problems. Therefore, it is important to keep the overall DB2 workload perspective in mind when tuning a single SQL statement. Also, be sure to balance the benefits of tuning opportunities with the amount of time and effort to make the changes. This workload perspective is important to ensure that performance tuning is carried forward on the highest priority applications, and on the applications that show the greatest opportunity for resource savings.

Performance data

Tuning SQL statements is an art form, but the art is based upon concrete facts and figures. Performance data is particularly important. SQL performance tuning data is derived from several sources:

- Execution metrics from monitors or traces
- Explain data
- Object statistical data

Let’s look at each of these in detail.

Execution metrics

To make intelligent decisions, be sure that you have actual runtime statistics for a high percentage of the total DB2 workload. Collect execution statistics in both development and test systems. The earlier you find a performance problem, the cheaper it will be to change the application.

Execution information should have detailed data on the use of the resources that can be controlled – CPU, I/O, and memory. DB2 collects metrics and publishes them through the Instrumentation Facility Interface. While the overhead of collecting high-level metrics is small, low-level metrics are more expensive and require evaluation before collection. A good way to reduce the cost of collecting execution statistics is to set thresholds based on the resource consumption of the SQL workload.

Collect detailed I/O information, including aggregated counts and total elapsed times for all I/O activities. DB2 is an I/O engine, but applications often select everything from a particular table and apply filtering to the data within the application logic. You should never do this, because it is very expensive for your application. Let DB2, not your application, filter the data. Collecting I/O data for application objects by object name is expensive but useful. Any time DB2 spends executing I/O processing for an SQL statement is time spent waiting for data.

After collection, the only overhead for these metrics is the cost of the storage media to maintain the data for whatever historical period is required. When you have historical data, you can compare resource consumption on DB2 before and after a change to your application program or SQL has been applied. This verification can be helpful, so consider keeping the data for a certain amount of time.

Explain data

DB2 thoroughly documents the input to the optimizer that chose the access path for retrieving the result set for each SQL statement. This information is stored in DB2 tables if they exist at the conclusion of the Bind/Rebind/Mini-Bind processes.

Object statistical data

DB2 provides tables in the catalog to maintain statistical information about the data that resides in DB2 tables and indexes. If this statistical information is not present (the values in these columns are -1), the DB2 optimizer assumes default values. The vast majority of the time, these default values will not generate an optimal access path at execution time. To avoid this situation, be sure to run RUNSTATS whenever the quantity structure of your DB2 tables change (such as mass deletes or loads)

Optimizer basics

The optimizer determines what it believes is the optimal means to access the required data for each SQL statement. The choices that the optimizer makes for access path selection are extremely important. The optimizer uses several sources of input, including the number and speed of available CPUs, buffer pool configuration, object statistics from the catalog, SQL statement text, and schema definition (for example, whether an index exists on a table).

For static SQL, the optimizer chooses an access path at Bind / Rebind time. The access path is stored in the DB2 directory database and is used for all subsequent executions of that SQL statement. At Bind / Rebind time, specify EXPLAIN(YES) and DB2 will externalize all the access path information to a set of tables (PLAN_TABLE) commonly referred to as the Explain tables. If you save this information for a significant period of time allows, you can use it to compare “before and after” access path statistics when you change the SQL and you need to verify the change.

For dynamic SQL, the optimizer chooses an access path at execution time unless there is a dynamic statement cache. If there is a dynamic statement cache, the access path is stored with the statement in the cache. Subsequent executions of the same statement will use the same access path, which makes execution of dynamic SQL much cheaper for subsequent executions.

Many dynamic SQL statements have different literals specified for predicates. Develop some mechanism to treat similar SQL statements as equivalent. Parse the text and evaluate the different pieces of each statement to determine equivalency.

DB2 access path choices

The following list shows the potential access paths for DB2, listed in the order of minimum to maximum resource usage when accessing objects with greater than 10 million rows in at least one table referenced in the statement:

Minimum resource usage	Direct row access using ROWID
	One fetch index scan using MIN,MAX
	Unique matching index scan using a predicate value
	Matching index scan only
	Non-matching index scan only
	Matching index cluster scan
	Matching random index scan
	Multiple matching index scan using AND and OR
	Non-matching cluster index scan
	Segmented table space scan

	Non-segmented table space scan (in parallel or sequential)
Maximum resource usage	Non-matching random index scan

Tuning options

Now that we have analyzed the DB2 workload, collected execution statistics and chosen access path information, it's time to look how you want to tune your SQL. You can improve SQL performance in multiple ways:

- Change the SQL
- Change the schema
- Collect up-to-date object statistics

Let's analyze each of these options to determine the conditions under which each one would be viable.

Change the SQL

The first option for better performance is to change the SQL, but is also the most complex option. Consider these areas for possible performance improvements:

- Analyze the predicate (WHERE clause)
- Write to the available indexes
- Optimize the sequence of table access (multi-table SQL)

Analyze the predicate (WHERE clause)

Good qualifying predicates reduce the resource requirements of an SQL statement. Evaluate predicates for the factors that affect resource usage, including whether the predicate is indexable, whether it is Stage 1 or Stage 2, and the order of predicate evaluation.

Examine the Explain tables to determine if an index has been selected for the predicate. The *DB2 Version 9.1 for z/OS Administration Guide* lists the rules that DB2 applies to determine whether a predicate is indexable. You can apply these rules during development, or you can apply these rules when you find predicates in the Explain Tables that are marked non-indexable. Depending on your requirements, not all predicates in a query will be indexable. Knowledge of the database schema will help you determine when you should make a predicate indexable by changing the SQL -- and whether it will be worthwhile to make the change.

The DB2 Explain tables identify each predicate as being Stage 1 or 2. In general, Stage 2 predicates consume more resources and perform worse than Stage 1 predicates. When you find predicates that are classified Stage 2, consult the DB2 Version 9.1 for z/OS Administration Guide for list of the rules that DB2 uses to determine whether a predicate is Stage 1 or 2. To improve performance, particularly CPU usage, change Stage 2 predicates to Stage 1 – but ensure that the change will not affect the results of the SQL statement.

The order of predicate processing is important because the order enables DB2 to eliminate non-qualifying rows from the result set as soon as possible and avoid further CPU and I/O processing. Within the same query, elimination of non-qualifying rows from one predicate may reduce the number of rows evaluated by future predicates. This will reduce CPU usage and possibly I/O processing.

DB2 has three criteria for determining the order of predicate processing. Predicates are ordered for processing based on the following criteria:

1. Indexable predicates
2. All other Stage 1 predicates
3. Stage 2 predicates

With each of these, the following rules impose additional order:

1. Equal predicates
2. Range and IS NOT NULL predicates
3. All other predicates

Write to the available indexes

Predicate analysis leads into the topic of indexes. You may find a predicate that is not indexable because the column does not exist in any index. Consider writing predicates to take advantage of existing indexes where possible before considering adding additional indexes. Before you add an index, evaluate the impact that index will have to the application as a whole compared to the one SQL statement that will benefit from the index.

Optimize the sequence of table access (multi-table SQL)

When an SQL statement requires data from more than one table (such as joins or subselects), the order of access is extremely important. When accessing multiple tables, use the most restrictive table access first and the least restrictive table last. This allows DB2 to eliminate as many rows as possible as early in the process as possible – and reduce CPU and I/O usage.

Change the schema

Another way to improve performance is to change the schema. Indexing is important. To determine the optimal index structure for an application, look at your application design requirements (for example, what will the end-user be looking or searching for?) and execution statistics that show which table columns are used in predicates.

DB2 Version 8 added choices for index design. DB2 9 introduced more options, including the compressed index, a method to reduce the amount of I/O when accessing an index in DB2. Primary indexes can be unique or not unique, and clustered or not clustered. Non-partitioning secondary indexes (NPSI) have always existed in DB2, but with Version 8 and later you can choose a data-partitioned secondary index (DPSI). The DPSI provides for secondary indexes through table-based partitioning.

Because DPSIs provide IBM utilities with enhanced partition independence, they increase the availability of applications during REORG and LOAD utility executions. A DPSI is generally a poor choice in access path selection. A DPSI can be efficient if the predicate is referring only to the columns in the DPSI. This situation will force a scan of the index for each partition in the table unless you are using partition screening. To avoid this problem, implement a process to examine the use and value of a DPSI.

Index design is affected by many factors, including:

- **Predicate usage**. Index design must be based on predicate usage. While designing the database schema correctly from the beginning is always the best policy, it is sometimes not an option. To improve performance for applications that were not designed optimally, it can be beneficial to create a CRUD matrix using actual execution data. To create this matrix, examine the columns in the application and estimate the number of times the data will be ‘c’reated, ‘r’ead, ‘u’pdated, and ‘d’eleted The CRUD matrix collects and analyzes predicate usage for each table column. This data provides a solid basis for the analysis of an existing index structure and the possible benefit of making changes to or adding indexes.

- **Read vs. update frequencies**. The predicate data can be aggregated to determine how often updates would occur for an existing index or a potentially new index. Having a lot of update activity (update/delete/insert SQL) for each additional index will add to the overall cost of your application. Index pages can become disorganized just like data pages. This leads to increased I/O activity, greater resource utilization, and decreased performance -- all reasons why databases are reorganized. By the way, it is more important to reorganize your indexes than your table spaces.

- **Correlated columns**. The predicate data can find columns that are used together in SQL statements. If these columns are used often and accessed together, they become good candidates to participate in a multiple column index.

- **Impact on maintenance**. Depending upon the application, some tables may be refreshed on a regular basis with a LOAD utility, or some tables and/or indexes may be reorganized on a regular basis to improve performance. Consider these activities when creating an index strategy for the application. You may need to make a trade-off between performance of the SQL and the performance of the maintenance utilities.

- **Column values**. Columns have a certain number of distinct values. Frequency statistics, which are generated by the RUNSTATS utility, provide the optimizer with significant opportunities for improved performance. When these columns appear in equality predicates, the optimizer can make use of these frequency statistics to accurately estimate filter factors and make a better index choice as a result.

Collect up-to-date object statistics

One way to tune SQL performance is to collect statistics with the RUNSTATS utility for the application objects. Object statistics may be the most overlooked SQL tuning tool. The statistics collected and stored in the DB2 catalog provide the optimizer with a better look at the objects being accessed and provide the optimizer with a much better calculation of the cost for each possible access path. In the rare instance whereDB2 chooses an access path that is resource intensive, look at the object statistics to determine if they exist and if they are up to date.

Conclusion

DB2 applications provide tremendous value to your organization, but you may be able to make them perform even better. Find problems early before they cause bigger problems. Several tools are available to help you find the problem. Consider adding or altering indexes to get better performance. Wherever possible, build automation into your processes for collecting or analyzing real-time execution metrics. Test potential solutions and measure the improvements. Don’t forget to document your work so that you can get credit for improving performance and therefore saving your organization time and money.

Chapter 3

Index Analysis and Tuning

When and how to design indexes

Indexes support many purposes within the database. They can be used to enforce a primary key rule or a unique rule. They can be used in support of foreign keys. They can be used in support of table partitioning (formerly a requirement, now an option). They can also be used to improve the performance of certain queries.

Smart index design

The ultimate in proper index design is to create only the number of indexes required, and to have indexes that can serve multiple purposes. There are always trade-offs to the design decisions you make. Most of the time the trade-off will be trading the responsibility of the enforcement of a business rule from the database to the application. The “holy grail” of index design would be one index per table. This one index would support the primary key, foreign key, partitioning, clustering, and the data access. While coming up with such a design is difficult, it should at least be an inspiration.

One of the best things to do for performance when migrating from logical to physical design is to abandon any object or generated keys that were created as generic primary keys. A lot of designs involve the creation of a generic key for a table object. In some cases these keys are useful, but in most situations they are just placeholders. A more performance oriented design involves the use of meaningful business keys as primary keys. This instantly eliminates an index for the generic primary key and uses the meaningful business key as the primary key in its place.

Meaningful business primary keys can be propagated as foreign keys to child tables; the child tables use the columns of the parent primary key as the first part of the child primary key. These keys can also be used for clustering. Because it is likely that the parent and child tables will be read together as part of a transaction, the common cluster will help reduce I/O times.

From a performance perspective, the only time a generated generic key should be used as a primary key is when the business keys become too large and unmanageable.

Partitioned and non-partitioned indexes

Partitioning is a useful way to improve availability, concurrency, and the manageability of large tables. With indexes defined upon a partitioned table, decide a whether it is appropriate to create a partitioning index, a data partitioned secondary index (DPSI), or a non-partitioned secondary index (NPSI).

Partitioning indexes are indexes that are partitioned, and contain the partitioning key columns as the leading columns of the index. These indexes are useful for data access, primary / foreign key support, and clustering. If the goal is to control the table and access to the table with the lowest number of indexes, then this is a good choice.

There may be a variety of reasons for partitioning a table for availability and concurrency, such as by geographic region, hour of the day, customer number range, or year. A DPSI is a partitioned index without the leading columns matching the partitioning key. These indexes are excellent for availability. However, the partitioning key is required as part of the SQL predicate to avoid accessing every index partition in a query. Otherwise, the query will probe every index partition.

An NPSI is the best choice for performance when queries will not be supplying the partitioning key for data access. The NPSI avoids having to probe all partitions of a DPSI. However, an NPSI can become very large and therefore more difficult to manage. High REORG times, and the potential for a problem with one partition impacting the entire NPSI can impact availability.

Index impacts

One of the easiest ways to improve query performance is to build indexes that match the predicates within a query. However, there are always trade-offs to database design, and this holds true for indexes. Unless you have a completely read-only table that is not too big, you simply cannot afford to build an index on every column combination. Every secondary index that is added to a table introduces a random reader for inserts, deletes, and updates to key values. Every insert and delete (and some updates) causes an I/O against the secondary index to add and remove keys. Typically, the secondary index is not in the same cluster of the table data, and that can result in many random I/Os to get index pages into the buffer pool for the operation. Thus an insert to a table with a secondary index will actually have additional random reads. Therefore, it is extremely important to understand the frequency of execution of all statements in an application. Is your application heavy read with rare inserts and deletes? Then perhaps you can afford more indexes on the table. Are there a lot of changes to data? Then perhaps fewer indexes should be created.

In addition to execution of inserts and deletes, weigh the frequency of certain select statements and the need for indexes to support them. If an SQL statement execution results in a table space scan, perhaps it is not a bad thing if the statement is executed very infrequently. The bottom line is that tables often have way too many indexes, and their importance has to be carefully considered.

Avoiding too many indexes

Having a balance between the application access requirements and the number of indexes is important. Here are some recommendations for avoiding too many indexes.

- **Avoid generic keys.** Generic keys make for extremely flexible database designs. They simplify the propagation of keys, and they keep key sizes small. That's great for a really flexible database design. However, generic keys almost always increase the randomness of data access in an application because every table is clustered differently if clustering is on the primary key. Generic keys also may result in an increase in the number of indexes created because data access is not always by the generic key.

- **Use meaningful partitioning keys**. If you are partitioning data, balance the need for partitioning with a proper partitioning key and index design. Try to use a partitioning key that satisfies the partitioning design and can also be used as part of a partitioning index, and perhaps even a clustering index. The last thing you ever want to do is partition on a meaningless random key to distribute I/O. This completely antiquated concept only introduces more I/O, maintenance, and cost to your application.

- **Have an index serve multiple purposes**. Make the consideration of indexes part of the physical database design. By making certain sacrifices, you can use one index to control partitioning, clustering, primary and foreign key, and the primary data access point for the table. Serving the business processes with the smallest number of indexes can only positively affect performance.

Tuning existing indexes

If you decide to have an index to support an application process, it is important to be sure it is used and is as optimal as possible.

Evaluating index usage

The first step to index tuning is to determine which indexes exist and how they are being used. The DB2 system catalog provides the first source for information about the current indexes defined. You can query the following tables to determine which indexes are in the system:

- SYSIBM.SYSINDEXES
- SYSIBM.SYSKEYS

- SYSIBM.SYSINDEXPART
- SYSIBM.SYSINDEXSPACE

Only the SYSINDEXES and SYSKEYS tables are really needed to determine the logical organization of the index to answer questions such as whether the index is partitioned, clustering, what the key columns and key column sequence are, and whether the index is unique.

Once the existing indexes are determined, it is important to figure out whether they are actually being used. This is not always an easy task. Once again, the primary tool is the system catalog. The SYSIBM.SYSPLANDEP and SYSIBM.SYSPACKDEP tables show which indexes are being used for static SQL statements. However, they do not provide the complete picture. Indexes used in support of foreign key relationships do not show up in access plans for inserts and updates. The SYSIBM.SYSFOREIGNKEYS table can be joined to the SYSIBM.SYSINDEXES and SYSIBM.SYSKEYS tables to determine the indexes that support foreign keys. Finally, dynamic SQL statements may or may not be using indexes. These statements can be captured by a variety of techniques, including using the EXPLAIN STMTCACHE ALL statement, running a performance trace, or looking at an online monitor. Once these statements are captured, they can be EXPLAINed and their index usage evaluated.

Gathered statements and their index usage can be evaluated using the EXPLAIN facility to determine the effectiveness of the indexes. Evaluating the indexes can be a lengthy process, but in many cases it is beneficial to understanding index usage and eliminating useless indexes. It can also be useful in finding index redundancy, and perhaps consolidating some of those indexes.

Determining if another index is useful

The easiest thing to do is to add an index in support of a problem query. However, before doing this, some careful evaluation may help overall system performance. Perhaps the new index can be avoided.

First, determine if the index is really needed. How often is the query run? How much data does it have to read compared to how much data it returns? If the query does not run that often, or if it does not filter much data, perhaps a table scan is not such a bad access path. If the requirements demand more, consider an index. Carefully examine the indexes that exist on the table. Is the required index similar to one that already exists, and can that index be modified to satisfy both the current and desired access paths? Perhaps access to the table can be achieved via a path through another table and an available index there. Maybe additional predicates can be coded to allow an existing index to be used. Benchmark each available choice to determine its effectiveness.

If a new index must be created, it is important to measure the full impact of the new index. If it saves CPU for the query it was built for, does it add CPU to processes that perform insert and delete operations against the table? In other words, measure the system-wide impact of a new index.

Index column sequence

It is a traditional notion with index design that you place high cardinality columns first to reduce the number of index pages that must be examined. While this technique may save CPU, it may actually introduce more randomness to the index access path, and potentially more I/O wait time. Low cardinality columns should not necessarily be avoided as high order index columns, especially if their use results in dual use indexes, such as one that matches the partitioning and provides the primary key.

Tuning indexes with BMC solutions

BMC SQL Performance for DB2 helps you diagnose DB2 SQL performance problems and write applications that perform much more efficiently, thereby reducing the overall cost for running the application. BMC SQL Performance for DB2 integrates the functionality of BMC SQL Explorer for DB2 (plan analyzer capability and access path compare) and BMC APPTUNE for DB2 (SQL monitor capability including Explain analysis) into a single offering that helps you optimize performance and availability by tuning the application SQL or physical database design.

BMC SQL Performance for DB2 includes the Index component to ensure that existing indexes are optimally structured to support the production applications. The Index Component extends the capability of BMC APPTUNE object analysis by collecting and reporting on column usage data for SQL statements. It automatically collects and displays actual access counts for each unique SQL statement (table and index, and predicate usage frequencies). For example, you can generate a report, which shows how many distinct SQL statements have used a particular column in any kind of predicate, or ORDER BY clause. This type of information tells you if statements access non-indexed columns or how changes to existing indexes affect other statements in your workload. Other table and index reports provide quick access to listings of the most-used objects based on get page volume or index access ratio.

The Index component also extends the capability of the Explain function by comparing access paths after making changes to simulated indexes in a cloned database.

A “what-if?” index analysis lets you model changes to indexes. The Index component provides on-demand, dynamic data collection of index dependencies and catalog statistics.

BMC SQL Performance for DB2 enables you to obtain accurate, real-time performance information about DB2 indexes. Because the Index component presents data at the object level, you can review the index access data to evaluate the performance of your SQL and identify candidates for index improvements.

Another important factor in index tuning is to understand if indexes are used at all, and how expensive particular indexes are to your business. Performance Advisor technology in BMC SQL Performance for DB2 provides a performance management database that serves as warehouse for performance trending and analysis of how performance changes over time. Packaged reports tell you which indexes are not used or just rarely used.

Buffer pool tuning

This chapter presents the following topics:

How buffer pools work

Buffer pools are areas of storage above the 2 GB bar that temporarily store pages of table spaces or indexes. When a program accesses a row of a table, DB2 places the page containing that row in a buffer. When a program changes a row of a table, DB2 must write the data in the buffer back to disk, typically either at a checkpoint or a write threshold.

They way buffer pools work is fairly simple by design, but tuning these simple operations that make a significant impact to the performance of our applications. The data manager issues get page requests to the buffer manager who—hopefully—can satisfy the request from the buffer pool instead of having to retrieve the page from disk. We often trade CPU for I/O in order to manage buffer pools efficiently. Buffer pools are maintained by subsystem, but individual buffer pool design and use should be by object granularity and, in some cases, also by application.

Initial buffer pool definitions are set at installation/migration are often hard to configure at this time because the application process against the objects is usually not detailed at installation. But regardless of what is set at installation time, we can use the ALTER command any time after the install to add and delete buffer pools, resize the buffer pools, or change any of the thresholds. The buffer pool definitions are stored in the boot strap data set (BSDS) and we can move objects between buffer pools via an ALTER INDEX/TABLESPACE and a subsequent START/STOP command of the object.

We can have up to 80 virtual buffer pools. This allows for up to:

- 50 4K page buffer pools
- 10 32K page buffer pools
- 10 8K page buffer pools
- 10 16K page buffer pools

Buffer pool performance factors

There are several factors to be considered with DB2 buffer pools. For best performance, consider good object separation along with size and write thresholds. Of course, we must monitor and measure the health of the buffer pool and be sure it meets our processing needs.

Buffer pool size

The maximum size usable for all buffer pools is 1 TB; the maximum size for a single buffer pool is 1 TB. This is not realistic because our machines are still limited on memory; for example, the z990 has a limit of 256 GB of real storage. Be aware of the storage available for buffer pools and use it wisely. Buffer pool sizes are determined by the VPSIZE parameter, which determines the number of pages to be used for the virtual pool. The virtual pool has an upper limit of 1 TB for all buffer pools or 1 TB for a single buffer pool. However, we are still limited by our hardware.

Set the buffer pool to a size that seems valid, and monitor the actual usage through the pages written into buffer pool in writes per second and then take that times the number of seconds pages should stay resident. This will give you the number of pages to start with.

Example:

```
2100 writes/sec * 3 seconds = 6300 pages (minimum, start with 6300
pages)
```

The hardest part with this formula is to determine the page residency time. This is extremely dependent on the objects in the buffer pool and how the objects are being accessed by the various applications. The question you should start with is “how quickly is this page referenced by the application?”

One traditional rule for determining whether a buffer pool is sized correctly is to increase the size of the buffer pool until no more hit ratio improvements occur and / or paging problems are not occurring. That is a valid rule but is it optimal? Not really.

Pages used in the buffer pools are processed in two categories: random (pages read one at a time) or sequential (pages read via prefetch). The pages are queued separately: LRU – Random Least Recently Used queue or SLRU – Sequential Least Recently Used Queue. The percentage of each queue in a buffer pool is controlled via the VPSEQT parameter (Sequential Steal Threshold). This influences the sizing of the buffer pool because some randomly accessed objects that are not re-referenced might just need a small buffer pool, whereas sequentially access objects with high reference rates may benefit from a large buffer pool with a high sequential threshold.

Buffer pool hit ratios

A buffer pool hit ratio is defined as the percentage of times the page was found in the pool. This is determined by the GETPAGES and pages read statistics.

```
(GETPAGES - Pages Read) / GETPAGES
```

The pages read are the synchronous I / Os plus all pages read by prefetch, list prefetch, and dynamic prefetch.

This can be calculated for any or every application execution from the accounting 101 record data for random access, sequential access, or both. Be very cautious when using a monitor that displays the buffer pool hit ratio. Find out what calculation is behind the hit ratio being shown on your particular performance monitor or reports since you may have a problem and not know it because the monitor shows a good hit ratio.

NOTE

BMC MAINVIEW for DB2 continuously monitors all buffer pools and their performance characteristics. BMC MAINVIEW for DB2 can send an alarm whenever there is an exceptional situation with one of your buffer pools, before this situation becomes critical to your application performance.

BMC Pool Advisor for DB2 takes the performance management of DB2 storage to the next level. BMC Pool Advisor continuously monitors DB2 workload and automatically adjusts buffer pool sizes and threshold depending on how the workload changes. Furthermore, BMC Pool Advisor for DB2can reconfigure your buffer pools and move table spaces and index spaces to buffers that better fit the performance attributes of those objects, thus leading into better hit ratios, less I/O, and less CPU consumption.

A negative hit ratio can happen when prefetch pages are brought into the buffer pool and are never referenced. The pages may not get referenced because the query ended or the prefetch pages were stolen for reuse before they could be used and had to be retrieve from disk again. We have seen one other situation where sequential detection kept turning on dynamic prefetch, and very few of the pages were used from each prefetch stream.

Tuning by the buffer pool hit ratio alone is not a guarantee of good performance. It is helpful only if we have done a good job with object separation, knowing how objects are used (i.e., whether the pages are referenced). In addition, a high buffer pool hit ratio can be reflective of extreme rereading of the same data. This can be really deceptive and misleading, and so make sure to understand the hit ratio relative to the number of SQL statements issued—all relative to the number of transactions being processed.

Page externalization

Pages are externalized to disk when the following occurs:

- Deferred write threshold (DWQT) reached
- Vertical deferred write threshold (VDWQT) reached
- Data set is physically closed or switched from R/W to R/O
- DB2 takes a checkpoint
- QUIESCE (WRITE YES) utility is executed
- Page is at the top of LRU chain and an update is required of the same page by another process

It is desirable to control page externalization via DWQT and VDWQT buffer pool thresholds for best performance and avoid surges in I/O. We do not want page externalization to be controlled by DB2 system checkpoints because too many pages would be written to disk at one time, causing I/O queuing delays, increased response time, and I/O spikes. During a checkpoint, all updated pages in the buffer pools are externalized to disk and the checkpoint is recorded in the log (except for the buffer pool used for the work files).

DWQT

Let's look at the buffer pool thresholds that help control writes. The DWQT, also known as the horizontal deferred write threshold, is the percentage threshold that determines when DB2 starts turning on write engines to begin deferred writes (32 pages/asynchronous I/O). The value can be from 0 to 90% with a default of 30%. When the threshold is reached, write engines (up to 600 available) begin writing pages to disk. You can run out of write engines if the write thresholds are not set to keep a constant flow of updated pages being written to disk. If this happens occasionally, it is okay; if this occurs daily, you have a tuning opportunity. DB2 turns on these write engines, basically one vertical pageset, one queue at a time, until a 10% reverse threshold is met. The Statistics report provides a WRITE ENGINES NOT AVAILABLE indicator when DB2 runs out of write engines.

When setting the DWQT threshold, a high value improves the hit ratio for updated pages, but increased I/O time when deferred write engines begin to externalize pages to disk. If you use a low value to reduce I/O length for deferred write engines, it will increase the number of deferred writes. Set this threshold based on the referencing of the data by the applications.

If you set the DWQT to zero so that all objects defined to the buffer pool are scheduled to be written immediately to disk, DB2 actually uses its own internal calculations for exactly how many changed pages can exist in the buffer pool before it is written to disk. 32 pages are still written per I/O, but it will take 40 dirty pages (updated pages) to trigger the threshold so that the highly re-referenced updated pages, such as space map pages, remain in the buffer pool.

VDWQT

The DWQT threshold works at a buffer pool level for controlling writes of pages to the buffer pools, but for a more efficient write process you can control writes at the page set / partition level. You can control this with the VDWQT, which is the percentage threshold that determines when DB2 starts turning on write engines and begins the deferred writes for a given data set. This helps to keep a particular page set / partition from monopolizing the entire buffer pool with its updated pages. The value is 0 to 90% with a default of 5%. The VDWQT should always be less than the DWQT.

A good rule of thumb is that if less than 10 pages are written per I / O, set the VDWQT to 0. You may also want to set it to 0 to consistently externalize the data to disk. It is normally best to keep this value low to prevent heavily updated page sets from dominating the section of the deferred write area. You can specify both a percentage and a number of pages (0 - 9999). You must set the percentage to 0 to use the number specified. Set to 0,0 and system uses MIN(32,1%) for good for consistent I / O.

If you set the VDWQT to zero, 32 pages are still written per I / O, but it will take 40 dirty pages (updated pages) to trigger the threshold so that the highly re-referenced updated pages, such as space map pages, remain in the buffer pool.

It is a good idea to set the VDWQT using a number rather than a percentage because if someone increases the buffer pool, more pages for a particular page set can occupy the buffer pool and this may not always be optimal or what you want.

When looking at any performance report, showing the amount of activity for the VDWQT and the DWQT, you want to see the VDWQT being triggered most of the time (VERTIC.DEFER.WRITE THRESHOLD), and the DWQT much less (HORIZ.DEFER.WRITE THRESHOLD). There are no general ratios since that would depend on both the activity and the number of objects in the buffer pools. The bottom line is that you want to control I / O by the VDWQT, with the DWQT watching for and controlling activity across the entire pool and in general writing out rapidly queuing up pages. This will also help limit the amount of I / O that the checkpoint would have to perform.

DB2 checkpoints are controlled through the DSNZPARM CHKFREQ. You can set the number of log records written between DB2 checkpoints (200-16,000,000) and minutes between checkpoints (1-60). You may need different settings for this parameter depending on the workload; for example, you may want it higher during batch processing. In very general terms, DB2 should take a checkpoint about every 5 - 10 minutes during online processing, or some other value based on investigative analysis of the impact on restart time after a failure. There are two real concerns for how often we take checkpoints:

- The cost and disruption of the checkpoints
- The restart time for the subsystem after a crash

Hundreds of checkpoints per hour is definitely too many, but the general guideline (3 to 6 checkpoints per hour) is likely to cause problems in restart time, especially if it is used for batch update timing. After a normal quiesce stop, there is no work to do. The real concern for checkpoints is for DB2 abend situations or when a MODIFY IRLM,ABEND command is issued, which will also cause DB2 to have to be restarted.

The costs and disruption of DB2 checkpoints are often overstated. While a DB2 checkpoint is a tiny hiccup, it does not prevent processing from proceeding. However, if you are not taking checkpoints often enough, especially with large buffer pools and high write thresholds (such as the defaults), it can cause enough I/O to make the checkpoint disruptive. The situation is corrected by reducing the amount written and increasing the checkpoint frequency—which yields much better performance and availability. If the write thresholds (DWQT/VDQWT) are doing their job, there is less work to perform at each checkpoint.

Even if the write thresholds (DWQT/VDQWT) and checkpoints are set properly, we could still see an unwanted write problem if the log data sets are not properly sized. If the active log data sets are too small, active log switches will occur often. When an active log switch takes place, a checkpoint is taken automatically. Therefore, logs could drive excessive checkpoint processing, resulting in constant writes. This would prevent us from achieving a high ratio of pages written per I/O because the deferred write queue would not be allowed to fill as it should.

Strategies for assigning buffer pools

Many organizations take a generic approach to buffer pool assignment: catalog and directory in BP0, work files in BP7, and two more buffer pools – one for indexes and one for table spaces. While this is a good start, it is not enough for best performance. We need several buffer pools to improve performance of selected page sets for critical objects. More buffer pools allow for more granular tuning and monitoring. By developing a strategy for proper sizing and object placement, we can reduce application response time, optimize usage of memory resources, and save CPU through I/O reduction.

The example below shows for how to start break objects into buffer pools based upon their type of usage by the applications. Each of these buffer pools has its own unique settings, and the type of processing may differ between the batch cycle and the online day. Generic breakouts are shown for this example; actual definitions (VPSIZE, VPSEQT, VDWQT, DWQT) would be much more finely tuned and less generic:

BP0 Catalog and directory - DB2 only use
BP1 Work files (DSNDB07)
BP2 Code and reference tables - heavily accessed
BP3 Small tables, heavily updated - transaction tables, work tables
BP4 Basic tables
BP5 Basic indexes

BP6 Special for large clustered, range scanned table
BP7 Special for random search table and full index
BP8 Vendor support objects
BP9 Derived tables and “saved” tables for ad-hoc
BP10 Staging tables (edit tables for short-lived data)
BP11 Staging indexes (edit tables for short-lived data)
BP12 Non-shared data (in data sharing environment)

Let’s take a look at a few of the special buffer pools and how their objects should be tuned based upon how the objects are used.

The work file (DSNDB07) table space must be separated into its own buffer pool. It has its own unique set of characteristics unlike any of the other buffer pools. All of the pages at one time could be dirty, updated, and referenced. If pages get externalized and read back in they can be stolen. Another myth that we need to dispel is the fact that if is **not** 100% sequential. Due to some of the processing that takes place in the work files (sort / merge, sparse indexes for non-correlated subqueries), some of the processing in this buffer pool may be random. In fact, in certain environments, it may be very random. The VPSEQT could start at 90% and must be monitored to determine if it should be lower. The VDWQT can start at 90% because we want to keep the pages around (especially when sorting). DSNDB07 pages are **not** written to disk at checkpoint, and we never want this buffer pool to hit DM Critical threshold. However, there may be some problems with these settings. If we set the VDWQT and the DWQT to 90%, and there is a slow-down in the actual writing of updated pages to a particular disk, the queues would get long, the I/Os would not be completing fast enough, and the queue of updated pages in the pool would past through the 90%, 95%, 97.5% thresholds (SPTH, DMTH, IWTH) and things could get exceedingly ugly.

A short example of the importance of tuning this buffer pool would be where we rapidly adjusted several buffer pool parameters, but paid special attention to the buffer pool for DSNDB07 and reduced warehouse queries from hours to minutes.

The hit ratio for this buffer pool needs to be calculated a bit differently because some SQL statements utilize a special GETPAGE request which reserves an empty buffer even before the page is brought in from disk. During the sort input processing a GETPAGE will be issued for each empty work file page without a read I/O. Because of this fact, when we calculate the hit ratio for the buffer pool supporting the work files we will divide the number of GETPAGEs in half.

The hit ratio calculation for the work file buffer pool is as follows:

```
((GETPAGES/2) - PAGES READ) / (GETPAGES)/2)
```

If the work files are not isolated to their own buffer pool then it would be impossible to calculate the hit ratio due to the unique nature of the work file usage of the buffer pool.

Another special buffer pool is the one for code, decode, reference and lookup tables it also has its own set of unique characteristics. These tables (and their indexes) should be placed in their own buffer pool.

We want to pin these objects in memory because they are often small and highly re-referenced. They are also generally read-only. A couple of buffer pool tuning techniques we will want to use are to first size the buffer pool large enough to fit all of the code and reference tables and their indexes, or at least sets of constantly used ones. Then turn on the FIFO (first in first out) queuing in so that we can kill the CPU overhead for LRU management. There is no need to manage a queue for objects that will be totally memory resident. Then set the VPSEQT to 0 and kill the overhead for sequential prefetch management, since the objects will be accessed randomly.

We have discussed all of the buffer pool thresholds and how to break objects out into separate buffer pools based on usage. Now just give some thought to the fact that you could have multiple sets of thresholds for each buffer pool based upon usage. This usage could vary between day time (on-line) processing, night time (batch) processing, week end (summary report) processing and so on.

An -ALTER BUFFER POOL command could be issued before each change in processing to properly configure the buffer pool for the new upcoming usage.

It's understandable that many organizations do not go to this level of detail due to the effort need to initially make these breakouts and then the effort needed to manage them. This is where we can call on our vendors to help make this easier and more automated.

Managing buffer pools with BMC solutions

DB2 uses significant amounts of storage to service user requests for application data. In many cases the more storage made available to DB2 the better. However, over-allocating these storage pools can have a negative impact on overall z/OS performance. Under-allocating these same storage pools can have a negative impact on DB2 application performance. Optimizing the use of these pools to minimize z/OS system overhead while providing optimal response time is important for many DB2 shops.

BMC Pool Advisor for DB2

BMC Pool Advisor for DB2 addresses this challenge by continuously monitoring DB2 storage pool usage, weighing the priorities for system-level performance against application performance goals. Workload fluctuations are often unpredictable, and adjusting DB2 pools dynamically can have a significant positive impact on performance. BMC Pool Advisor for DB2 collects DB2 storage pool data on an

ongoing basis and analyzes usage in context to the entire DB2 subsystem and overall z/OS performance. Based on this analysis, BMC Pool Advisor for DB2 can automatically adjust the size and thresholds of DB2 storage pools to accommodate changes in the workload and available system resources.

BMC Pool Advisor for DB2 provides you with expert assistance in performance tuning and problem resolution, as well as some automated resource management capabilities. BMC Pool Advisor for DB2

- collects and analyzes DB2 data about buffer pool, EDM pool, dynamic statement caching, sort pool, RID pool, and group buffer pool use
- makes recommendations concerning storage resource allocation, object configuration, and various DB2 parameter settings
- automatically manages these resources on a real-time basis for best performance across workload fluctuations

BMC Pool Advisor for DB2 looks at storage requirements across the system and makes recommendations based on total storage resource requirements, so that an increase in one resource does not cause a shortage in another. When real storage is constrained, BMC Pool Advisor for DB2 attempts to balance resource use and, under severe conditions, prevents increases to any storage resources.

Buffer pools

BMC Pool Advisor for DB2 can process large lists of buffer pools and data objects. It accurately groups data types by considering more performance characteristics including:

- type of object (table or index)
- degree of sequential access versus random access
- activity level (number of get pages)
- object size
- update rate
- optimum working set size for best hit rates
- priority

The Configuration Advisor calculates a score that represents the overall fitness of the current configuration for the measured attributes of the page sets. You can submit an analysis request that evaluates the mix of pools and objects and recommends changes to the assignment of objects to pools and the attributes of the pools. You can accept and implement those changes or modify and resubmit the analysis request.

Using this iterative and interactive approach, you can reach a configuration that meets the needs of your objects, while fitting within acceptable limits of complexity and resource use.

You can run the Configuration Advisor on a single subsystem or member. In a data sharing environment, BMC Pool Advisor for DB2 collects and analyzes data from all members and recommends changes to the assignment of objects to pools based on the workload for the entire data sharing group.

Group buffer pools

BMC Pool Advisor for DB2 continuously monitors and evaluates critical group buffer pool performance metrics and configuration values, alerting you to potential problems and providing advice on prevention and resolution.

Efficiency values provide an overall status for each group buffer pool. You can use these values to compare the relative performance between the different group buffer pools in the data sharing group, making it possible to identify over-and under-allocated resources.

Read / write ratios help determine the predominant access method for each pool, helping you to group similarly accessed page sets to specific group buffer pools and increase their operational efficiency. For example, the number of available directory and data entries are monitored to prevent castouts and cross-system invalidation.

BMC Pool Advisor for DB2 recognizes potential storage resource shortages before they result in critical event failures, and advises you on how to make changes to your group buffer pool configuration.

Subsystem tuning

This chapter presents the following topics:

The DB2 subsystem itself has many opportunities for improving performance. This includes managing several areas of memory and subsystem wide thresholds/limits. Here we will take a look a few areas of memory critical to the successful performance of our applications as well as some thresholds that could affect our overall application.

EDM pools

The Environmental Descriptor Manager (EDM) pool is made up of five components, each of which is in its own separate storage area both above and below the 2 GB bar, and each contains many items including the following:

- EDM RDS pool (1 below the bar and 1 above the bar) - EDMPOOL
  - CTs – cursor tables (copies of the SKCTs)
  - PTs – package tables (copies of the SKPTs)
- EDM skeleton pool (above the bar) - EDM_SKELETON_POOL
  - SKCTs – skeleton cursor tables
  - SKPTs – skeleton package tables
- EDM DBD cache (above the bar) - EDMDBDC
  - DBDs – database descriptors
- EDM statement cache (above the bar) - EDMSTMTC
  - Skeletons of dynamic SQL for CACHE DYNAMIC SQL

If the size of first three EDM pools is too small, you will see increased I/O activity on the table spaces that support the DB2 directory.

Our main goal for these EDM pools is to limit the I/O against the directory and catalog. If the pool is too small, you will see increased response times due to the loading of the SKCTs, SKPTs, and DBDs. Also, if your EDM pool is too small, you will see fewer threads being used concurrently because DB2 knows that it does not have the appropriate resources to allocate/start new threads.

By correctly sizing the EDM pool (EDMPOOL, EDM_SKELETON_POOL and EDMDBDC), you can avoid unnecessary I/Os from accumulating for a transaction. If a SKCT, SKPT or DBD must be reloaded into the EDM pool, this is additional I/O. This can happen if the pool pages are stolen because the EDM pool is too small. Pages in the pool are maintained on an LRU queue, and the least recently used pages get stolen if required. Efficiency of the EDM pool can be measured with the following ratios:

- CT requests versus CTs not in EDM pool
- PT requests versus PTs not in EDM pool
- DBD requests versus DBDs not in EDM pool

You would like to see at least 80% hit ratio for objects found in the EDM pool, if not more.

By correctly sizing the EDMSTMTC cache, you can help performance for dynamic SQL statements that are utilizing the cache.

RID pool

The Row Identifier (RID) pool is used for storing and sorting RIDs for operations such as:

- List prefetch
- Multiple index access
- Hybrid joins
- Enforcing unique keys while updating multiple rows

The optimizer looks at the RID pool for list prefetch and other RID use. The full use of RID pool is possible for any single user at run time. If not enough space is available in the RID pool, run-time RID-related failures can result in a table space scan. For example, if you want to retrieve 10,000 rows from a 100,000,000 row table and no RID pool is available, a scan of 100,000,000 rows will occur, at any time and without external notification. The optimizer assumes that physical I/O will be less with a RID large pool.

The default size of the RID pool is 8 MB unless specified. The size can be set from 16 KB to 100,000 MB. If the size is set to 0, DB2 will not use it and this would prevent list prefetch and multiple index access operations from occurring. The size is set with an installation parameter. The RID pool is created at start up time, but no space is allocated until RID storage is actually needed. It is then allocated above the 16 MB line in 16 KB blocks as needed, until the maximum size you specified on installation panel DSNTIPC (MAXRBLK DSNZPARM) is reached. There are a few guidelines for setting the RID pool size. You should have as large a RID pool as required because it is a benefit for processing and you never want it to run out of space.

You can use three statistics to monitor for RID pool problems:

- RIDs over RDS limit
- RIDs over DM limit
- Insufficient pool size

RIDs over RDS limit

This is the number of times list prefetch is turned off because the RID list built for a single set of index entries is greater than 25% of the number of rows in the table. If this is the case, DB2 determines that instead of using list prefetch to satisfy a query it would be more efficient to perform a table space scan, which may or may not be good depending on the size of the table accessed. Increasing the size of the RID pool will not help in this case. This is an application issue for access paths and needs to be evaluated for queries using list prefetch. In other words, it is not a RID problem, but a SQL tuning issue. A lot of times, the types of queries that result in RDS limit failures are multiple-use queries that can be split into multiple queries within an application with if-then logic and no RID processing for list prefetch or hybrid join.

There is one very critical issue regarding this type of failure. The 25% threshold is actually stored in the package / plan at bind time, therefore it may no longer match the real 25% value, and in fact could be far less. It is important to know what packages / plans are using list prefetch, and on what tables. If the underlying tables are growing, then rebinding the packages / plans that are dependent on it should be rebound after a RUNSTATS utility has updated the statistics.

RIDs over DM limit

This occurs when over 28 million RIDs (DB2 limit) were required to satisfy a query. The consequences of hitting this limit can be fallback to a table space scan. You have options to control this:

- Fix the index to filter more RIDs.
- Add an additional index better suited for filtering.
- Force list prefetch off and use another index. (OPTIMIZE for 1 row may do the trick here.)
- Rewrite the query.
- Maybe it just requires a table space scan.

Please note that increasing the size of the RID pool will not help this problem.

Insufficient pool size

This indicates that the RID pool is too small. In this case, you need to increase the size of the RID pool.

If you have a RID pool that the optimizer has determined to be too small for RID list to be built for a query, then it will use an alternative access path. Be sure that the RID pool is the same size between test and production, or you could be seeing different results in your EXPLAIN output as well as in your query execution.

Sort pool

At startup, DB2 allocates a sort pool in the private area of the DBM1 address space. DB2 uses a special sorting technique called a tournament sort.

DB2 sorts are performed in two phases – the initialization phase and the merge phase. The initialization phase is where DB2 builds ordered sets of “runs” (intermediate sets of ordered data) from the given input. In the merge phase, DB2 merges the runs.

During the sorting processes, it is not uncommon for this algorithm to produce logical work files called runs. If the sort pool is large enough, the sort completes in that area. More often than not, the sort cannot complete in the sort pool and the runs are moved into work files (DSNDB07). These runs are later merged to complete the sort. When the work files are used for holding the pages that make up the sort runs, you could experience performance degradation if the pages get externalized to the physical work files since they must be read back in later in order to complete the sort.

Make sure that you assign an adequately sized sort pool to your system and/or avoid really large SQL sorts in programs that need to execute fast. If any overflow to the work files it is reported in IFCID (traces), your performance monitor can report on these events.

NOTE

BMC Pool Advisor for DB2 will report on the overall sort pool efficiency and any failures to provide adequate sort size, and it will recommend changes to the sort pool size, depending on the workload.

The sort pool size defaults to 2 MB unless specified. It can range from 240 KB to 128 MB and is set with an installation DSNZPARM SRTPOOL. The larger the sort pool (sort work area) is, the fewer sort runs are produced. If the sort pool is large enough, the buffer pools and work files will not be used. If buffer pools and DSNDB07 are not used, the better performance will be due to less I/O. We want to size sort pool and work files large because we do not want sorts to have pages being written to disk. All work files used for the sort are considered dirty, protected pages in the buffer pool. If a page is written to disk during the sort and is read back in, it is considered clean and unprotected. This could lead to a poorly performing sort process.

Miscellaneous DSNZPARMs

The DB2 subsystem is managed by several DSNZPARMs. You can change the majority of these DSNZPARMs without an outage. Here we look at a few DSNZPARMs used to control threads and tracing abilities.

Thread management

Threads take up storage in DB2 and need to be managed as more and more processes are requesting threads. Several DSNZPARMs control the number and type of threads we have.

CTHREAD

CTHREAD specifies the maximum number of allied threads that can be allocated concurrently. Allied threads are threads that are started at the local subsystem. It is specified on the MAX USER field on the install panel – DSNTIPE. The default is 200; you can set it as high as 2000.

Allied threads can be the following:

- TSO user (DSN command or a DB2 request from DB2 QMF)
- Batch job (DSN command or a DB2 utility)
- An IMS region that can access DB2
- An active CICS transaction that can access DB2
- A utility (each utility uses one thread, plus one thread for each subtask)
- A connection from users of CAF and RRSAF

The total number of threads accessing data that can be allocated concurrently is the sum of the CTHREAD value and the MAXDBAT value and cannot be more than 2000 in total. When the number of users who are attempting to access DB2 exceeds the number you specify, excess plan allocation requests are queued and could have an effect on performance and throughput. In the majority of situations, the actual total amount of real and virtual storage determines the maximum number of threads that DB2 can handle.

Parallelism can also have an affect on the total number of threads. DB2 utilities each use a minimum of one thread, plus an additional thread for each subtask. Therefore, a single utility might use many threads. Specify a thread value accordingly to accommodate parallelism within utilities. You may want to consider using a value that is higher than the default value.

MAXDBAT

MAXDBAT specifies the maximum number of database access threads (DBATs) that can be active concurrently in the DB2 subsystem. The default value is 200 and can be set up to 1999. As mentioned before, the total number of threads accessing data concurrently is the sum of field CTHREAD and MAXDBAT, with a total allowable of 2000. If a request for a new connection to DB2 is received and MAXDBAT has been reached, the resulting action depends on whether ACTIVE or INACTIVE is specified for CMTSTAT DSNZPARM. If the option is ACTIVE, the allocation request is allowed but any further processing for the connection is queued waiting for an active database access thread to terminate. If the option is INACTIVE, the allocation request is allowed and is processed when DB2 can assign an unused database access thread slot to the connection.

CMTSTAT

CMTSTAT specifies whether to make a thread active or inactive after it successfully commits or rolls back and holds no cursors. A thread can become inactive only if it holds no cursors, has no temporary tables defined, and executes no statements from the dynamic statement cache. If you specify ACTIVE, the thread remains active. This provides the best performance but consumes system resources. If your installation must support a large number of connections, specify INACTIVE.

INACTIVE has two different types. The first is an inactive DBAT where the thread remains associated with the connections, but the thread's storage utilization is reduced as much as possible. The only problem here is this still potentially requires a large number of threads to support a large number of connections. The other type is an inactive connection that uses less storage than an inactive DBAT. In this case, the connections are disassociated from the thread. The thread is allowed to be pooled and reused for other connections, new or inactive. This provides better resource utilization because there are typically a small number of threads that can be used to service a large number of connections.

It is often recommended to use CMTSTAT to allow DB2 to separate the DBAT from the connection (at commit time) and reuse the DBAT to process others' work while the connection is inactive. The only issue is that this cuts an accounting record when the connection becomes inactive. You may want to consider rolling up accounting data by using the ACCUMACC and ACCUMUID DSNZPARMs to control this.

ACCUMACC

ACCUMACC allows DB2 to accumulate accounting records by end user for DDF and RRSAF threads. If this is set to NO, DB2 writes an accounting record when a DDF thread is made inactive or when sign-on occurs for an RRSAF thread. The maximum limit is 2065535. The number specified will be the number of occurrences aggregated per record. DB2 writes an accounting record every *n* occurrences of the "end user" on the thread. The default is 10.

ACCUMUID

ACCUMID provides the aggregation criteria for DDF and RRS accounting record roll-up. A value of 0-10 tells DB2 how to aggregate the records. This aggregation is based on three values: end user ID, transaction and workstation. These values must be provided by the application. These values can be set by DDF via 'Server Connect' and 'Set Client' – SQLSETI calls, by RRSAF threads via RRSAF SIGNON, AUTH SIGNON and CONTEXT SIGNON, or by properties in Java programs when using the Java Universal Driver. The values are as follows:

- 0 = End user ID, transaction, workstation (default) – nulls accepted
- 1 = End user ID
- 2 = End user transaction name
- 3 = End user workstation name
- 4 = End user ID, end user transaction name – nulls accepted
- 5 = End user ID, end user workstation name – nulls accepted
- 6 = End user transaction, end user workstation – nulls accepted
- 7 = End user ID, end user transaction name, end user workstation name – all non-null
- 8 = End user ID, end user transaction name – all non-null
- 9 = End user ID, end user workstation name – all non-null
- 10 = End user transaction name, end user workstation name – all non-null

For example, if the thread is from workstation MYWRKSTN, the end user ID of NULL this thread would qualify for roll-up if ACCUMID is 5 (end user ID, end user workstation name – nulls accepted). It would not qualify for roll-up if ACCUMID is 9 (end user ID, end user workstation name – all non-null).

The threads that do not qualify for roll-up write individual accounting records. Any thread with all key values set to null will not qualify for roll-up, and an individual accounting record is written.

Tracing parameters

You can use tracing parameters to record activities in the DB2 subsystem, including the recording of accounting data, statistics data, and audit data.

Accounting and statistics

It is important to have the appropriate DB2 accounting and statistics classes activated to continuously gather information about your DB2 subsystem and its activity. The following DSNZPARMs turn on the appropriate classes:

- **SMFACCT**. As a general recommendation, have SMF account class 1,2,3,7, and 8 turned on.

- **SMFSTAT**. - It is also recommended to have SMF statistics class 1,3,4 (1 and 3 at a minimum) selected during normal execution.

These classes use a small amount of overhead, but they are used by the majority of DB2 shops and usually do not cause an issue. Any other trace classes should not be running at all times because they can cause excessive overhead if run for long periods. Measuring the impact of setting these traces at a very active site demonstrated an approximate 4-5% overhead. Various vendor online performance monitors typically set these traces anyway, and tests have shown that adding these traces to SMF when already set for an online monitor, or vice versa, resulted in no additional overhead. The benefits of setting these traces almost always outweigh the costs, as accounting and statistics trace data forms the basis of solid reactive and proactive performance tuning. When executing other traces, it is wise to set the trace to only the IFCIDs necessary for the appropriate performance analysis or problem diagnosis. Run these traces for only short periods of time.

NOTE

BMC MAINVIEW for DB2 uses SMS accounting and statistic traces to collect and display comprehensive information on each DB2 subsystem component. Alarms can be defined for each collected element, which will help you to identify bottleneck situations before they become a problem for the application.

AUDITST

Use the AUDITST parameter to start the DB2 audit trace to provide a primary audit trail for DB2 activities. The trace can record changes in authorization IDs for a security audit, record changes that are made to the structure of data (such as dropping a table), data values (such as updating or inserting records), or record audit of data access (selection of data).

The default is NO, but you can set it to YES or a list of trace classes. There is overhead with using the AUDIT feature, so use it only in exceptional cases.

Once initiated, the audit trace creates records of all actions of certain types and sends them to a named destination. The information obtained by the audit trace records includes:

- The ID that initiated the activity, the LOCATION of the ID that initiated the activity (if the access was initiated from a remote location)
- The type of activity and the time the activity occurred
- The DB2 objects that were affected
- Whether access was denied

- Who owns a particular plan and package

You enable a particular table to be audited via the CREATE or ALTER TABLE DDL using the AUDIT (NONE, ALL, CHANGES) option.

DB2 catalog

The DB2 catalog and directory act as central repositories for all information about objects, authorizations, and communications regarding the support and operations of DB2.

The catalog is composed of several DB2 tables and can be accessed via SQL. The catalog contains details about DB2 objects obtained from the data definition language (DDL) when an object is created or altered, or from data control language (DCL) when an authorization is granted on an object or group of objects. The DB2 catalog also contains information about communications with other DB2 and non-DB2 databases through the use of the communications database (CDB), which contains information about VTAM and TCP/IP addresses.

Take care of the catalog. It should have frequent reorganizations and RUNSTATS, using the same guidelines you would use for application table spaces and indexes. Add your own indexes to the catalog as needed.

Locking and contention

A balance must be achieved for maximum concurrency of all processes. DB2 has many controls to allow e maximum concurrency while maintaining the integrity of our data. These range from the parameters we bind our programs with to options of the DDL for the creation of objects, to subsystem-level parameters.

DB2 uses locks to control concurrency within a database, that is, to manage simultaneous access to a resource by more than one user (also referred to as serialization). Locks prevent access to uncommitted data, which prevents updates from becoming lost (part of data integrity) and allows a user to see the same data, without the data ever changing, within a period of processing called a commit scope. From a performance standpoint, everything done in DB2 has a trade-off. In locking, there is a trade-off between concurrency and performance, as more concurrency comes at a higher cost of CPU usage due to lock management. There are also cases where DB2 will override the locking strategy designed, because processes hold locked resources exceeding the site-established thresholds. However, only certain combinations of parameters cause this to occur.

The following subsystem parameters are critical for achieving high concurrency and availability in the DB2 subsystem. It is important to set the parameters according the needs of the applications running in the subsystem.

IRLMRWT

IRLMRWT represents the number of seconds that a transaction will wait for a lock before a time out is detected. The IRLM uses this for time-out and deadlock detection. Most shops take the default of 60 seconds, but in some high-performance situations where detection must occur sooner (so the applications are not incurring excessive lock wait times), it is been set lower (10-20 seconds). If time-outs occur often, review and tune the application so that it does not hold locks longer than necessary.

NUMLKTS

NUMLKTS represents the maximum number of locks on an object. If you are turning off lock escalation (LOCKMAX 0 on the table space), you need to increase this number, and be sure to commit often. If you are using LOCKMAX SYSTEM, the value here will be the value for lock escalation. See “Locking” on page 15 for more information.

NUMLKUS

NUMLKUS represents the maximum number of page or row locks that a single application can have held concurrently on all table spaces. This includes data pages, index pages, and rows. If you specify 0, there is no limit on the number of locks. Be careful with 0 because if you turn off lock escalation and do not commit frequently enough, you could run into storage problems (DB2 uses 540 bytes for each lock).

URCHKTH

In our high volume, high performance, high availability environments, it is imperative that commits be performed. Commits help us with the following:

- Lock avoidance
- Concurrency
- Restart
- Rollback / recovery
- Utility processing
- Resource release

URCHKTH helps us find long running units of work that are not committing. It specifies the number of checkpoints that should occur before a message is issued identifying a long-running unit of work. Consistent checkpoint intervals enhance the identification of long-running units of work.

The result will be a message written to the console. Don't forget, the message is just the identification...now you must get the commits in the application!

LRDRTHLD

It is critical that applications commit frequently to release resources. LRDRTHLD, also known as the long-running reader threshold, can proactively identify reader threads that have exceeded the user-specified time limit threshold without COMMIT. DB2 writes an IFCID 313 record each time the time interval is exceeded. When a non-zero exists in the LRDRTHLD DSNZPARM, DB2 records the time a task holds a read claim; when the specified number of minutes is passed, the record is written. Valid values for LRDRTHLD are 0 (default; disables it) and 1 - 1439 minutes.

It is recommended to turn this on because even readers can cause availability issues. A long-running reader who is not committing is preventing utility processing from doing drains. Long-running readers, especially those who may compete with utility processing, need to be committing.

SKIPUNCI

SKIPUNCI allows you to skip uncommitted inserts. It specifies whether statements ignore a row that was inserted by a transaction (other than itself) and that has not yet been committed or aborted. This applies only to statements running with row-level locking and isolation-level read stability or cursor stability. If the value is YES, DB2 behaves as though the uncommitted row has not yet arrived and the row is skipped. If NO (default) is specified, DB2 waits for the inserted row to be committed or rolled back. DB2 processes the row as a qualifying row if the insert commits; DB2 moves on to find another row if the insert is rolled back.

If a transaction performs one or more inserts and spawns a second transaction, specify NO for SKIP UNCOMM INSERTS if the first transaction needs the second transaction to wait for the outcome of the inserts. Using YES offers greater concurrency than the default value of NO.

Managing DB2 subsystems with BMC solutions

While DB2 on z/OS delivers significant value, it can consume a large percentage of mainframe resources. A poorly tuned DB2 system affects overall response time and can ultimately lead to costly hardware upgrades. A finely tuned DB2 system helps to reduce the overall cost of ownership for DB2 on z/OS while delivering better service to end-users.

BMC System Performance for DB2

BMC System Performance for DB2 helps you address DB2 performance challenges and improve staff productivity by tuning your DB2 system dynamically and automatically as workloads change. BMC System Performance for DB2 provides the following benefits:

- improves application performance by providing intelligent real-time management and tuning of DB2 system resources and parameters that can adversely affect performance
- reduces end-user response time
- minimizes costs associated with DB2
- optimizes the use of DB2 storage
- increases staff productivity by providing advisor technology
- avoids system problems and downtime by automatically detecting and correcting system problems
- maintains performance consistency with constantly adjusting application workloads

BMC System Performance for DB2 includes the following components:

- BMC MAINVIEW for DB2 (including the BMC CATALOG MANAGER Browse function)
- BMC Pool Advisor for DB2
- BMC OPERTUNE for DB2

BMC System Performance for DB2 provides comprehensive reports that let you monitor all aspects of DB2 from one central view. You can hyperlink from that view to reports about specific data. The report set combines the reporting abilities of BMC MAINVIEW for DB2 and BMC Pool Advisor for DB2 with a supplemental set of comprehensive reports on all aspects of DB2.

BMC MAINVIEW for DB2

DB2 on z/OS, while delivering significant value, consumes a large percentage of mainframe resources in most customer environments. A poorly tuned DB2 system can affect overall response time and ultimately lead to costly hardware upgrades. A finely tuned DB2 system helps to reduce the overall cost of ownership for DB2 on z/OS while delivering better service to end-users. This makes DB2 performance management solutions easy to justify.

BMC MAINVIEW for DB2 provides comprehensive, user-friendly monitoring and management tools. BMC MAINVIEW for DB2 samples DB2 activities to identify degradation, issue early warnings, and enable automation to manage exceptions before they become problems. It manages any number of DB2 systems across multiple locations to enable you to keep track of the complete DB2 environment or focus on a specific group, which is essential for data sharing.

BMC CATALOG MANAGER for DB2

BMC CATALOG MANAGER for DB2 provides highly productive methods for creating and managing your DB2 databases. Using an ISPF interface, BMC CATALOG MANAGER for DB2 provides interactive access to catalog information and application data with simple-to-use menus, panels, and online Help.

Using BMC CATALOG MANAGER for DB2, you interact with the catalog by performing actions on specific objects. You do not need to have complete knowledge of DB2 structures or SQL syntax because BMC CATALOG MANAGER for DB2 maintains database structures and constructs the necessary SQL statements. You choose when and how to execute these statements. The created SQL can be saved, edited, and reused for other tasks.

One of the most useful functions of BMC CATALOG MANAGER for DB2 is its ability to generate lists of DB2 catalog objects, both for queries and for executing commands against the listed items. The ability to execute action commands against list items offers powerful administrative support in your DB2 environment.

You can use the data editing functions to create, edit, or browse data in a table or view without leaving BMC CATALOG MANAGER for DB2. The necessary data manipulation language (DML) statements are built for you automatically while you work within the familiar ISPF interface. The data editing function enables you to copy data from one table or view into another table or view. In many cases, this feature prevents the need to run load and unload utilities.

With BMC CATALOG MANAGER for DB2, you can execute the following types of SQL statements and DB2 commands and submit utility jobs interactively:

- SQL statements
- DB2, DB2 DSN, and DB2 utility commands
- BMC Software utility commands
- BMCSTATS and BMC SQL Explorer for DB2 commands
- user-written commands

The execution of commands and statements requires minimum input—usually one command verb. You do not need to know the syntax of the eventual SQL, command, or utility statement because BMC CATALOG MANAGER for DB2 constructs the required DB2 syntax from information in the selected line of a list, installation defaults, and user-specific defaults.

BMC Pool Advisor for DB2

BMC Pool Advisor for DB2 continuously monitoring DB2 storage pool usage, weighing the priorities for system level performance against application performance goals. BMC Pool Advisor for DB2 collects DB2 storage pool data on an ongoing basis and analyzes this usage in context to the entire DB2 subsystem and overall z/OS performance. Based on this analysis BMC Pool Advisor for DB2 can automatically adjust the size and thresholds of DB2 storage pools to adjust for changes in the workload and available system resources.

EDM pool

BMC Pool Advisor for DB2 attempts to make the EDM pool big enough to contain all frequently used objects in addition to the largest infrequently-referenced objects without the need for I/O.

If the EDM pool is sized correctly, there should be

- fewer SQL statement failures
- more concurrently active threads
- better overall system performance
- fewer wasted resources due to over allocation
- fewer unnecessary delays resulting from physical I/Os to load objects from disk

BMC Pool Advisor for DB2 accomplishes this by constantly monitoring use of the EDM pool and making recommendations for size changes when increases in activity necessitate, keeping the pool operating at optimum levels. If BMC OPERTUNE for DB2 is also installed, the EDM pool size can be adjusted automatically, with no user intervention.

RID and sort pools

Since DB2 version 8, RID and SORT pools are allocated above the 2 GB bar. Large sort pools can provide an in-memory sort to your SQL without the need to access the temporary database for I/O. BMC Pool Advisor for DB2 monitors the RID and SORT pools continuously and can spot problems before they become critical. BMC Pool Advisor for DB2 can warn you of these situations. If BMC OPERTUNE for DB2 is installed, it can dynamically alter the pool allocations to prevent failures. For more information, see "Sorting" on page 16 and "RID pool" on page 55.

See "Managing buffer pools with BMC solutions" on page 49 for more information.

BMC OPERTUNE for DB2

BMC OPERTUNE for DB2 lets you assign resources to match system requirements, whether the workload is light, average or heavy. Workload changes can be tuned to eliminate under- or over- allocation of resources, and resource allocations can be changed every 30 minutes without a DB2 outage. You can change performance parameters in real time, thus avoiding DB2 outages. BMC OPERTUNE for DB2 also provides operational assists that help in managing DB2 threads, utilities, and log configurations.

Chapter 6

Designing databases and applications for performance

This chapter presents the following topics:

Object creation guidelines

This chapter discusses basic guidelines for creating objects in DB2 for z/OS and provides guidelines that you may want to consider for high volume/high performance databases and applications.

Table space guidelines

Table spaces are objects that consist of one or more data sets that are used to store DB2 data. Tables are created in table spaces, and you can have one or many tables in a table space, depending on the type of table space defined. There are five types of table spaces: simple, segmented, partitioned, universal, large object (LOB), and XML. Only simple and segmented table spaces allow for multiple tables. Simple table spaces can no longer be created in DB2 9. You should be moving toward universal table spaces; in some cases, to use certain DB2 9 features, you will be required to.

- **Simple table spaces.** These table spaces can contain one or more tables; however, the rows of the different tables are not kept on separate pages, which can cause concurrency and space usage issues. You can no longer create these in DB2 9; however, existing ones will still be supported.

- **Segmented table spaces**. Segmented table spaces can contain one or more tables, and the space is divided into same-size segments. Each segment can contain rows from only one table.

- **Partitioned table spaces**. A partitioned table space divides the data into several separate data sets (partitions). There can be only one table in a partitioned table space. You can have up to 4096 partitions of up to 64 GB each. Each partition can have different characteristics, such as volume placement and free space.

- **Universal table spaces**. A universal table space is created as a range-partitioned table space, or a partition-by-growth table space, depending upon the setting in the CREATE TABLESPACE statement. Like partitioned table spaces, there can be up to 4096 partitions. Many features in DB2 9 require universal table spaces.

- **LOB table spaces**. LOB table spaces are required to hold large object data. LOB table spaces are associated with the table space that holds the logical LOB column.

- **XML table spaces**. XML table spaces are required to hold XML data that is stored in XML data type columns. Unlike LOB table spaces, the XML table spaces are created implicitly by DB2 when an XML column is defined. LOB table spaces can be explicitly defined.

Referential integrity

DB2 has the ability to enforce the relationships between tables within the database engine. This is generally a very good thing to do for the following reasons:

- Moves data intensive work into the database

  Enforcing referential integrity (RI) is a data intensive process. Data intensive processing belongs inside the database engine. Taking advantage of database enforced RI is the right performance decision. It means that less data needs to be transmitted to and from an application to enforce these rules.

- Relieves the application of responsibility

  If the rules are coded in the database, application programmers don't have to put the logic to enforce the rules into their application code. This can decrease programming time. In addition, the SQL to enforce the rules is a lot easier to code than the equivalent application logic, and the rules are documented in the DB2 system catalog.

- Is centrally enforceable

  Because RI rules enforced by the database are attached directly to the related objects, they are enforced no matter how many different applications are updating the database objects. This centralizes the logic and eliminates the need to duplicate code among more than one application, which could lead to inconsistencies.

It is very important for performance to make sure that all foreign keys are backed by indexes. Otherwise, performance of such things as deletes against parent tables can be severely impacted. From a performance perspective, you should never define database-enforced RI on code values. Code value control should be done at the application layer with cached code values. Finally, if you are going to use database-enforced RI, you need to have application design controls in place that avoid doing the enforcement in the application layer anyway. This is redundant code that requires more database calls, and can become extremely expensive. Put the RI in the database, and let the database do the work!

Free space

The FREEPAGE and PCTFREE clauses help improve the performance of updates and inserts by allowing free space to exist on table spaces or index spaces. Performance improvements include improved access to the data through better clustering of data, less index page splitting, faster inserts, fewer row overflows, and a reduction in the number of reorganizations required. Trade-offs include an increase in the number of pages (and therefore more storage needed), fewer rows per I/O and less efficient use

of buffer pools, and more pages to scan. As a result, it is important to achieve a good balance for each individual table space and index space when deciding on free space, and that balance will depend on the processing requirements of each table space or index space. When inserts and updates are performed, DB2 will use the free space defined, and by doing this it can keep records in clustering sequence as much as possible. When the free space is used up, the records must be located elsewhere, and this is when performance can begin to suffer. Read-only tables do not require any free space, and tables with a pure insert-at-end strategy generally don't require free space. Exceptions to this would be tables with VARCHAR columns or tables using compression that are subject to updates due to the fact that records could get relocated.

The FREEPAGE amount represents the number of full pages inserted between each empty page during a LOAD or REORG of a table space or index space. The trade-off is between how often reorganization can be performed versus how much disk can be allocated for an object. FREEPAGE should be used for table spaces so that inserts can be kept as close to the optimal page as possible. For indexes, FREEPAGE should be used for the same reason, except improvements would be in terms of keeping index page splits near by the original page instead of being placed at the end of the index. FREEPAGE is useful when inserts are sequentially clustered.

PCTFREE is the percentage of a page left free during a LOAD or REORG. PCTFREE is useful when you can assume an even distribution of inserts across the key ranges. It is also needed in indexes to avoid all random inserts causing page splits.

The best index strategy would be to have zero for FREEPAGE and a PCTFREE value that accommodates all inserts between index reorganizations with no page splits. Page splits can be quite expensive and can limit concurrency, and so a free space and reorganization strategy that eliminates or severely reduces index page splits can be a huge performance gain!

Column data types

Data types specify the attributes of the columns when creating a table. When the database design is being implemented, any of the DB2 supplied data types can be used. Data is stored in DB2 tables that are composed of columns and rows. Every DB2 table is defined by using columns. These columns must be one of the built-in DB2 data types or user-defined data types. Every DB2-supplied data type belongs to one of these three major categories:

- Numeric
- String (binary, single byte, double byte)
- Date-time

There are also more specialized data types, such as row ID values and XML values.

You need to choose the best data type for your data for best storage and performance. You need to consider how you plan to process the data. For example, if you are doing math on values, they need to be a numeric data type. You also need to be careful with varying length data types (VARCHARs) since they can have additional overhead, such as logging and row relocation.

Clustering indexes

The clustering index is NOT always the primary key. It generally is not the primary key but a sequential range retrieval key and should be chosen by the most frequent range access to the table data. Range and sequential retrieval are the primary requirement, but partitioning is another requirement and can be the more critical requirement, especially as tables get extremely large. If you do not specify an explicit clustering index, DB2 will cluster by the index that is the oldest by definition (often referred to as the first index created). If the oldest index is dropped and recreated, that index will now be a new index and clustering will now be by the next oldest index.

We have seen many problems where an access path in test was exactly the opposite in production and this was simply because the indexes on the tables were created in a different sequence. In this case, no clustering index was defined, so the range retrieval index in test was different from the range retrieval index chosen in production.

Highly clustered secondary indexes should be constructed in certain situations, although this generally is not possible and can take considerable effort in physical design. But consider an index that has a high number of duplicates. If a scan is required for a set of like data, and the index is highly unclustered, the amount of random I/O will be high because index driven random data access will be used. This will contribute to a slower response. Index-only access would generally be chosen by the optimizer if the data required was completely contained within the index. That might be a design option in this case, but generally that is not possible. There are often methods we can use when picking columns for this type of high clustered non-clustering index that would match more closely to the actual ordering of the data in the table. This would allow the index to be selected by the optimizer as an access path, reducing the amount of I/O. However, it is not intuitive and not easy, but could be necessary for performance.

When processing an INSERT, DB2 will choose the defined clustering index for insertion order. If a clustering index has not been explicitly defined, DB2 will use the current oldest surviving index. Some call this the first index created, but contrary to popular belief, the default index chosen for insert processing is not the first index created on the table nor the index with the lowest object identifier (OBID). For example, if a table had indexes A, B, C, and D, and they had been defined in that order, and none was defined as the clustering index, index A would be chosen for inserts. If index A was dropped and redefined, it would become the “newest” index on the table, and index B would now be used for inserts. During the execution of the REORG utility, DB2 will choose the index to use for clustering the data in the same manner. Always explicitly define a clustering index on any table.

A partitioned index is not required to be the clustering index of a partitioned table. You can choose any index for your clustering index, regardless of the partitioning.

It is important to note that DB2 uses the clustering index to direct inserts into a table space. Therefore, when distributing data across a table space via a random key, an appropriate amount of free space will be required to maintain cluster of sequential readers or multiple rows of child data for a parent key. For random inserts free space becomes an issue if there are hot spots of significant inserts in certain areas of the table space. These hot spots could quickly fill up, resulting in exhaustive searches for free space that progressively get worse over time. In these situations, a good reorganization strategy is necessary, or perhaps a sequential insert strategy, is more appropriate.

You can achieve a sequential insert strategy turning clustering off. In DB2 V8, set PCTFREE and FREEPAGE to 0 with MEMBER CLUSTER; in DB2 9, set APPEND YES. These techniques alter the insert algorithm for a table space so that new inserts are placed at the end. This can result in dramatic improvements in insert performance for high insert applications. However, data access time can increase, as well as the need for reorganizations, so carefully consider and test these settings.

Locking and concurrency

It is important to understand locking and concurrency are critical concepts because they are critical to application performance. We will look at the most important aspects of DB2 locks.

Lock sizes

Locks can be taken on various objects in DB2. The size of the lock determines how much of the total object will be locked. Several objects can be locked in DB2, determining the lock size:

- Non-LOB data:
  - Table space: Lock the entire table space
  - Table: Only segmented table spaces allow for an individual table to be locked
- Partition: Lock only a partition
- Page: Lock an individual page in a table
- Row: Lock an individual row in a table
- LOB data: LOB locks
- XML data: XML lock

Table space and table locks are the most encompassing lock, and of course this allows for the least amount of concurrency between users. Table locks in a simple table space may lock data in other tables since the rows can be intermingled on different pages. In a segmented table space, all of the pages with rows from a single table will be locked and will not affect other tables in the table space.

For partitioned table spaces or universal table spaces, DB2 can choose to lock only a partition of the table space, when they are accessed. DB2 takes locks on the individual partition instead of the entire table space and does not escalate locks to the table space level, but rather to the partition level.

DB2 may not be able to take partition level locks under the following conditions:

- Plan / package bound ACQUIRE(ALLOCATE)
- Table space defined LOCKSIZE table space
- The LOCK TABLE statement without PART option

Page locking is usually the lock size of choice for best concurrency. This allows DB2 to only lock a single page of data, whether it is a 4 K, 8 K, 16 K or 32 K page.

DB2 supports row-level locking; if applications are experiencing a lot of concurrency problems, it might be considered. However, do not use it as a fix for what could be a physical design issue or an application design issue. Use it when a page of data is simultaneously needed by multiple applications and each user's interest is on different rows. If the interest is on the same row, row-level locking buys you nothing. Use row-level locking only if the increase of the cost (concurrency and wait-time overhead) of locking is tolerable and you can definitely justify the benefit.

LOB locks

Large objects (LOB) locks have different characteristics from regular locks. There is no concept of row or page locking with a LOB; LOB locking is not at all like the traditional transaction-level locking. Because LOBs are in an associated object, concurrency between the base table and the LOB must be maintained at all times, even if the base table is using uncommitted read (UR). A LOB lock still needs to be held on the LOB to provide consistency and, most importantly, to maintain space in the LOB table space.

Locks on LOBs are taken when they are needed for an INSERT or UPDATE operation and released immediately at the completion of the operation. LOB locks are not held for SELECT and DELETE operations. If an application uses the uncommitted read option, a LOB lock might be acquired, but only to test the LOB for completeness. The lock is released immediately after it is acquired.

XML locks

The last type of lock is the XML lock. DB2 stores XML column values in a separate XML table space (similar to LOBs). An application that reads or updates a row in a table that contains XML columns might use lock avoidance or obtain transaction locks on the base table. When an XML column is updated or read, the application might also acquire transaction locks on the XML table space and XML values that are stored in the XML table space. A lock that is taken on an XML value in an XML table space is called an XML lock. In addition to the XML locks, page locks are acquired on pages in the XML table space. During insert, update, and delete operations, X page locks are acquired; during fetch processing, including uncommitted read and lock avoidance, S page locks are acquired. In summary, the main purpose of XML locks is for managing the space used by XML data and to ensure that XML readers do not read partially updated XML data.

Concurrency

For an application to achieve maximum concurrency, consider these items. First, design the application to hold locks for as short a time as possible by pushing DML toward the end to the unit of work. By doing DML in a predetermined sequence, and committing often, we can avoid deadlocks. This is an important responsibility of the applications and should always be considered. The database itself only has a few options to help concurrency or deadlock problems because the majority of locking problems lie in the applications.

Consider the following DB2 features when finding the best options for concurrency:

- **Partitioning**. If you partition table spaces in a meaningful fashion that corresponds to how you process data, you can run processing in parallel with separate processes each hitting their own partition or sets of partitions. Running against separate partitions eliminates the potential for locking conflicts between these processes. It also can be very beneficial in reducing cross system read-write interest in a data sharing environment.

- **Clustering**. Keeping related tables in a common cluster can improve concurrency by reducing the amount of random access to these related objects. Changes can follow a predictable pattern, reducing the number of locks taken as well as the randomness of those locks.

- **Uncommitted read.** Reading uncommitted is one of the most effective ways to improve concurrency, and is widely used with great success. Most concerns revolve around reading data that may eventually be rolled back. Most of these concerns center around the updating of data that was read before being rolled back. A solid optimistic locking strategy usually alleviates these concerns.

- **Optimistic locking support.** Optimistic lock is a technique where, upon update the WHERE clause of the update includes predicates on one or more columns in addition to the primary key for the table. These additional columns could be either a special update timestamp or all non-key columns. This technique assures that an update is made to a row that has not changed since the row was previously read. This obviously requires additional application responsibility to place these additional predicates in the WHERE clause and to include the update timestamp in the SET clause of the update.

DB2 9 provides for some assistance in the area of optimistic locking with the introduction of the row change timestamp. This automated timestamp column can be referenced in SQL statements to retrieve or test the value, and is automatically updated whenever a row is updated. This relieves the application of some, but not all, of the responsibility of optimistic locking.

Guidelines for databases and applications

Consider these recommendations for creating a database for maximum concurrency during design—before the tables and table spaces are physically implemented because to change after the data is in use would be difficult.

- Use segmented or universal table spaces, not simple.
  - — Will keep rows of different tables on different pages, so page locks only lock rows for a single table. If you are on DB2 9, simple table spaces cannot be created, but may still exist. You should migrate them to segmented.
- Use LOCKSIZE parameters appropriately.
  - — Keep the amount of data locked at a minimum unless the application requires exclusive access.
  - — However, be careful with row level locks due to overhead.
- Consider spacing out rows for small tables with heavy concurrent access by using MAXROWS =1.
  - — Row-level lock can be used to help with this but the overhead is greater, especially in a data sharing environment

- Use partitioning where possible.
  - Can reduce contention and increase parallel activity for batch processes.
  - Can reduce overhead in data sharing by allowing for the use of affinity routing to different partitions.
  - Locks can be taken on individual partitions.
- Use data partitioned secondary indexes (DPSIs).
  - Promotes partition independence and less contention for utilities.
- Consider using LOCKMAX 0 to turn off lock escalation.
  - In some high-volume environments, this may be necessary. NUMLKTS will need to be increased, and the applications must commit frequently.
- Use volatile tables.
  - Reduces contention because an index will also be used to access the data by different applications that always access the data in the same order.
- Have an adequate number of databases.
  - Reduce DBD locking if DDL, DCL and utility execution is high for objects in the same database.
- Use sequence objects.
  - Provides for better number generation without the overhead of using a single control table.

There are many recommendations for lessening the amount of locks taken by the application and for best concurrency.

- Have commits in the applications.
  - Proper commit strategies allow you to reduce contention, achieve lock avoidance, reduce rollback time for an application, reduced elapsed time for system restart, and allow other processes to operate.
- Bind with appropriate parameters—UR or CS.
- Use uncommitted read where appropriate.
  - Best to use at statement level.

- Close all cursors as soon as possible.
  - — Allow for locks and resources to be freed.
- Access tables in the same order.
  - — This prevents the applications from deadlocking.
- Code retry logic for deadlocks.
  - — Don’t fail immediately; the lock may get released.
- Consider using SKIP LOCKED DATA, if applicable.
  - — Query skips data that is incompatibly locked.
- Use caution with multi-row inserts with positioned updates and deletes.
  - — They can expand the unit of work.
- Use optimistic locking.
  - — Test whether the underlying data source column has changed by another transaction since the last read operation. Define a column in the table with FOR EACH ROW ON UPDATE AS ROW CHANGE TIMESTAMP and use the time stamp for comparison during the update.

Bind options

Several bind options influence the locking and concurrency of an application, including

- ISOLATION
- CURRENTDATA
- ACQUIRE and RELEASE

ISOLATION

You can set varying levels of program isolation.

- **CS (Cursor Stability)**. This level of program isolation can hold a lock on the row or page (depending on the lock size defined) only if the cursor is actually positioned on that row or page. The lock will be released when the cursor moves to a new row or page, but the lock will be held until a commit is issued if changes are being

made to the data. This option allows for the maximum concurrency for applications that are accessing the same data, but cannot allow uncommitted reads. In DB2 9, this is the default.

- **RS (Read Stability)** holds locks on all rows or pages qualified by stage 1 predicates for the application until the commit is issued. Non-qualified rows or pages, even though touched, are not locked, as with the repeatable read (RR) option. Uncommitted changes of other applications cannot be read, but if an application issues the same query again, any data changed or inserted by other applications will be read, because RS allows other applications to insert new rows or update rows that could fall within the range of the query. This option is less restrictive but similar in function to RR.

- **RR (Repeatable Read)** holds a lock on all rows or pages touched by an application program since the last commit was issued, whether or not all those rows or pages satisfied the query. It holds these locks until the next commit point, which ensures that if the application needs to read the data again, the values will be the same (no other process could update the locked data). This option is the most restrictive in terms of concurrency of applications and is the default until DB2 9.

- **UR (Uncommitted Read)** tells DB2 not to take any locks and to ignore other locks and allow read access to the locked data. It allows an application to run concurrently with other processes (except mass deletes and utilities that drain all claim classes).

CURRENTDATA

Another bind option is CURRENTDATA. In DB2 version 8 or earlier, to set this to NO to allow for lock avoidance. In DB2 9, the default is NO.

ACQUIRE and RELEASE

Two other bind options that affect locking include the ACQUIRE and RELEASE parameters. The table space, partition, and table lock duration are determined by the BIND - ACQUIRE and RELEASE parameters. The ACQUIRE parameter determines when table, table space, or partition locks are taken. ALLOCATE says when the first SQL statement is issued, the maximum required lock is taken on all the objects in the plan or package (not available in DB2 9). USE says when the SQL statement is issued, the required lock is taken on the objects involved in the SQL statement (Only option in DB2 9).

RELEASE determines when table, table space, or partition locks are released. DEALLOCATE indicates that this will happen at the end of the program, and COMMIT indicates this will happen at COMMIT. These are very powerful choices, especially for concurrency and performance. COMMIT says that once the process commits it will not come back again, and various object locks and other information saved in memory should be destroyed. Then when the process comes back, all these locks and storage must be recreated. This can be expensive for transactions that

operate repeatedly against the same objects. In these situations a DEALLOCATE setting can have a dramatic impact on performance because it dramatically reduces these operations. However, DEALLOCATE decreases concurrency for such operations as commands, database changes via DDL, and utilities for maintenance operations performed by DBAs. So, coordinate these maintenance activities with application processing to take advantage of DEALLOCATE. RELEASE(DEALLOCATE) is ignored for remote connections.

Reduce I/O contention

High I/O rates are typically the result of a database design that did not consider the major processes that were to be accessing the database. You can design for reduced I/O and reduce I/O once an application has gone production. However, these efforts do take time for careful analysis, and subsequently time is money. In other words, you're going to have to test design ideas. Let the database tell you how to design the database.

During the physical database design, it is extremely important to understand the major processes that access the database, which database tables are accessed, and how they are accessed. Typically, tables that are accessed together should be clustered in the same sequence. This can be one of the highest factors in reducing I/O contention. Do the common clustering in support of the most active process, especially for large scale batch processing. Common clustering can encourage performance enhancers such as sequential detection and list prefetch. You can test the effectiveness of this design with simple programs or scripts that simulate the predicted access.

When implementing your database, the last thing you want to do is to partition objects in such a way as to randomize the distribution of data. With modern DASD devices, the data is typically distributed inside the DASD subsystem, and all that a random partitioning key will do is introduce random I/Os into your application. If your DASD subsystem does not have substantial quantities of cache, or does not have parallel access volumes (PAVs), consider placing tables and indexes in separate SMS storage groups. Otherwise, place your faith in the DASD subsystem. These days, I/O rates greater than 7 ms should not be considered acceptable. We are usually quite happy at 4 ms or less per synchronous I/O.

If an existing system is experiencing high I/O response times due to high I/O rates (and not poor DASD response), you have several options to reduce the I/O and improve performance. Collect comprehensive accounting reports organized by application. Hopefully, identifiers such as authorization ID, correlation name, or workstation ID are available and can be used to associate and group the DB2 accounting data by application. Once the accounting data is organized, examine the application with the highest synchronous I/O rates. Examine SQL rates: Is the application issuing too many SQL statements? Capture and analyze SQL statements to determine which have access paths that are introducing random I/Os (typically

joins and correlated subqueries). This is a good opportunity to review clustering of commonly accessed tables, adding indexes, or perhaps forcing the materialization of out of cluster tables (via nested table expression with a GROUP BY) to get them into cluster with other tables for queries that process large amounts of data.

Coding efficient SQL

The best thing you can do to improve the performance of your database application is to not call the database. Every time you make a call to DB2, you are crossing address spaces (even for remote calls) - and that costs money. When you design applications, use some good common practices to help achieve low cost and high performance.

Avoid calling the database

Generic application design results in generic performance. There are tremendous pushes to object bases and service oriented designs, and it makes a lot of sense. These techniques lead to fast application development, great code reuse, and extremely adaptable applications. All these things are great for the business, but typically terrible for performance. With some thought, however, performance objectives can be met.

Object design is great, but some extra analysis can lead to processes that draw from multiple objects (thus multiple tables). These designs may benefit from specialized objects that join tables and thus reduce the number of calls to the database. In other situations object refresh, which will lead to another call to the database, can be avoided. Also, there is no reason to read an object to verify it before updating it. You can simply perform an update directly utilizing an optimistic locking approach to avoid any clashes.

Cache infrequently updated frequently accessed objects

Code tables most certainly should be cached locally for code lookup. With the lower costs of memory on small application servers, it can really pay off to cache more than just code values. On highly active systems, base reference tables that are infrequently updated can be cached for significant performance gains. In some cases, setting a refresh cycle of an hour or so can eliminate millions of SQL calls.

Do not code programmatic joins

Are you reading table A and then reading table B in the same unit of work? Do these two tables have a relationship? If so, read both tables in one statement. A SQL join is almost always more efficient than a programmatic join - possibly 30% to 60% faster. This may mean setting up specialized objects, but we're talking about a performance design. One more note about programmatic joins is that when you code a

programmatic join, you dictate the table access sequence. When DB2 receives a join statement, it can make a decision as to which table to access first, and this is a great performance advantage. If you hard code that access sequence in your program, the database has no chance.

Get only the data you need

Don't retrieve data unless you need it. Get only the columns you need. Each additional column retrieved adds more CPU.

Filter data in your SQL statement

Don't be afraid to put any kind of predicate in your SQL statement. Just don't filter data in your application. Even stage 2 predicates in the SQL are better then returning the data in your program just to let it fall on the floor. The SQL language is extremely powerful and has a lot of flexibility. Plus, no matter how badly you code the statement at least the filtering logic is in the statement, it is exposed to the database and to a DBA that can help tune it. If it's in your application code, then you're pretty much on your own.

Process data in your application

If you are looking for great flexibility of your code, nothing beats putting everything in a SQL statement. You can plug a massive SQL statement into SPUFI, a COBOL batch program, a remote Java application, or into a script on a web server. That's real power and flexibility! However, it can be really expensive. While DB2 does a great job at efficiently filtering the data via predicates, it does not do as good a job processing the data. Therefore, if you need to modify the data coming out of the SQL, do it in your application. What if you need the data back in a certain sequence, and DB2 can't avoid a sort? You are probably better off sorting in your program. Do you need decisions made on the data coming out? You could code a CASE expression in the SELECT list of the SQL statement, but it will be cheaper to do it in your program.

Take advantage of advanced features for data-intensive processes

A data-intensive process is one in which a change in a value in one table affects the data in another table. Advanced database features allow data-intensive logic to stay in the database, including database enforced referential integrity, triggers, check constraint, and stored procedures. Enforced RI and trigger are the most powerful.

Use triggers and enforced RI wisely to avoid excessive calls from the application to the database. Think about this: if you insert a row in a child table, do you want to read the parent row yourself or let the database do it for you? If the database does it for you, that's one less call you need to make. The real trick is staying committed and

informed so that when you use database-enforced RI you don't enforce the RI rules in the application layer anyway. Triggers can be a real smart choice for RI that cannot be enforced in the database. Triggers allow you to extend the functionality of the database by reacting to any changes to the data in tables. If your application changes table A, does data in table B need to change as well? Then you should be better off having a trigger make that change instead of making the multiple calls in the database. Just remember that triggers are a synchronous process and not the best choice for things like replication. However, putting data-intensive business logic in them can be a significant performance gain.

Designing stored procedures for performance

Stored procedures can be a performance gain or a performance bust. The advantage of using stored procedures basically comes in two forms:

- To improve performance over multiple database calls across a network
- To allow for legacy data access via a remote SQL call

Stored procedures are expensive. That is, for external stored procedures, the call results in a trip from the DB2 address space to a stored procedure address space. Any SQL calls made from inside the stored procedure goes from the stored procedure address space to the DB2 address space. All of these cross memory calls can get expensive. For this reason, placing single SQL statements into stored procedures is a performance detriment. The real performance gain is to place multiple SQL calls, including business logic, inside the stored procedure. Send the stored procedure a request and have it deliver a result. This will replace multiple remote calls from the application with a single call to the stored procedure, and improved performance.

Stored procedures are an excellent way to cheaply access legacy resources, such as VSAM files. You can avoid several application layers by simply making a call to the stored procedure. Use of global temporary tables and cursors returned from the stored procedure can let you return data in a relational manner back to your program.

While the SQL procedure language is available in DB2 version 8, it is translated into a C program and runs external to the database engine. DB2 9 introduces internal SQL procedures that eliminate the external calls and dramatically improve performance of SQL language procedures. However, you should still be coding multiple SQL calls and business logic inside an internal SQL procedure if you want a performance improvement over a group of SQL calls. You don't want one SQL call per procedure.

Design for performance with BMC solutions

BMC provides comprehensive solutions for managing physical database and application design.

Managing physical database design

When creating a physical database design for performance, we need to understand how to manage DDL and how to manage changes in the schemas.

As DB2 applications become more structurally complex and more mission critical, the need to add and modify data structures, as well as the need for change management, increases significantly. In addition, the growing complexity of the DB2 environment has made the change process itself more difficult and costly. Without an effective change management tool, DBAs find that defining data structure and managing changes are complex, tedious, resource intensive, and error-prone tasks.

BMC Database Administration for DB2 is a set of integrated tools designed to work together to help you easily manage your most complex DB2 for z/OS environments. It enables you to do more less time with a greater degree of quality.

The BMC Database Administration for DB2 solution consists of the following products:

- BMC CATALOG MANAGER for DB2
- BMC CHANGE MANAGER for DB2
- BMC COPY PLUS for DB2
- BMC LOADPLUS for DB2
- BMC UNLOAD PLUS for DB2
- BMC SNAPSHOT UPGRADE FEATURE

BMC CHANGE MANAGER for DB2 is an effective and safe change management tool that works with DB2 objects across multiple subsystems. It removes the complexity of physical design. It can be integrated with a data modelling tool (which generates DDL based on a physical database model), or you can use the intuitive ISPF-based user interface to define your database objects yourself, and let BMC CHANGE MANAGER for DB2 handle the correct syntax, order of definition and execution of the DDL. Using compare and baseline features, you can perform synchronization or versioning of the database schemas and implement changes without disrupting local modifications. When coupled with BMC high-speed utilities, BMC CHANGE MANAGER for DB2 performs faster changes and data migrations.

BMC CATALOG MANAGER for DB2 facilitates the day-to-day tasks associated with administering DB2 environments. It features highly productive methods for creating and managing DB2 schemas ad hoc. BMC CATALOG MANAGER for DB2 lets you create test data for your newly created tables and execute all kinds of DB2 utilities against your database in order to manage and maintain your schema. Using an ISPF-based interface similar to BMC CHANGE MANAGER for DB2, the product provides interactive access to catalog information through easy-to-use menus, dialog panels, and online Help.

Managing DB2 application design

In addition to the physical database design tasks solved with BMC Database Administration for DB2, BMC provides other solutions to understand locking and concurrency issues or to find a better way to code more efficient SQL.

BMC SQL Performance for DB2

BMC SQL Performance for DB2 helps you diagnose DB2 SQL performance problems and write applications that perform much more efficiently, thereby reducing the overall cost for running the application. BMC SQL Performance for DB2 integrates the functionality of the following products and technologies into a single offering that helps you optimize performance and availability by tuning the application SQL or physical database design:

- BMC APPTUNE for DB2 (SQL monitor capability including Explain analysis)
- BMC SQL Explorer for DB2 (plan analyzer capability and access path compare)
- Index component
- Performance Advisor technology

BMC APPTUNE for DB2

BMC APPTUNE for DB2 is an application performance and resource analysis product that is used to gather and display data from a single SQL statement or a set of SQL statements. The gathered data provides valuable information about performance and resource usage of DB2 applications.

Unlike other DB2 performance aids that rely on expensive DB2 SQL traces to gather data, BMC APPTUNE for DB2 collects all relevant performance measures in real time for every SQL statement (static or dynamic) executed in one or more DB2 subsystems. It summarizes the collected data and stores it for analysis. This method of collecting data provides detailed information on the performance and resource usage of DB2 applications while avoiding the costly overhead and large volumes of data associated with other DB2 performance aids.

BMC APPTUNE for DB2 combines the analysis of SQL statements with the analysis of the objects that are accessed by the statement. You can identify the most heavily accessed DB2 tables and indexes, and you can analyze table and index get page activity, buffer pool efficiency, and I/O response time by subsystem, buffer pool, database, and data set (table space and index space).

BMC APPTUNE for DB2 can collect data from all of the DB2 subsystems or data sharing members on each z/OS image across the sysplex. This allows you to analyze collected data by individual DB2 subsystem or member, or to analyze data aggregated from multiple DB2 subsystems, or whole data sharing groups—all from a single TSO session.

BMC APPTUNE for DB2 includes a user interface and analysis capabilities that have been engineered for completeness and ease of use. In most cases, you need only three keystrokes to find the most resource intensive SQL in your workload, and start to tune that SQL statement. In other words, BMC APPTUNE helps you to find the needle in the haystack.

BMC APPTUNE for DB2 provides a variety of reports that allow you to “zoom” or “expand” to multiple levels of performance data to get the answers you need to solve a performance problem. You can display the reports in a traditional, numeric presentation or an easy-to-understand, graphical presentation. Most reports are available in online and batch formats.

To facilitate the tuning of SQL statements, BMC APPTUNE for DB2 includes an integrated, robust functionality that can explain dynamic or static SQL statements, then interpret and summarize the results in an easy-to-read and understandable format. Expert rules are applied to the result of such an explain, and you receive suggestions on where to start your tuning activities.

BMC SQL Explorer for DB2

BMC SQL Explorer for DB2 is an SQL plan analysis tool that enables you to solve performance problems resulting from inefficient SQL statements using bad access paths.

Application performance and availability can be significantly affected by minor changes in either application or DB2 data structures. DB2 optimizer changes in access path selections can occur because of subtle statistical changes in the catalog statistics or schemas, resulting in degraded application and transaction performance.

SQL statements, catalog statistics, and data structure definitions can all have a major impact on throughput and response time. Often, changes in the access path selection are not detected until the application has been moved into production and the application either performs poorly or is unavailable.

BMC SQL Explorer for DB2 addresses the following areas of concern:

- Plan table information in DB2 is cryptic and requires expert knowledge to understand.
- DBAs must control application performance within their change-management procedures, because ultimately they have to correct the DB2 performance problem.
- During application design reviews, areas of responsibility can be clouded so that quantified objective information regarding change recommendations and impact on performance is not readily available.
- Application developers want more involvement in performing application-specific tuning.
- New releases of DB2 for z/OS continue to change and improve the DB2 optimizer and the SQL language. Expertise is required to stay current on these changes. The expert rule base contained in BMC SQL Explorer is continually being evaluated by the BMC development team to reflect these changes.
- The analysis of each SQL statement in each application can be time-consuming, requires expert knowledge of DB2 for z/OS, and is usually performed in reaction to performance problems. Any attempt to rewrite SQL statements without expert knowledge is usually a trial-and-error process.

You can tailor the expert rules used in BMC SQL Explorer for DB2 and BMC APPTUNE for DB2 to your particular environment or technical audience. For example, you can modify rules to enforce installation standards and to detect SQL statements that should be avoided in certain circumstances.

When writing or rewriting SQL statements in ISPF Edit sessions, you can use the BMC SQL Explorer for DB2 edit macro to identify and correct performance problems before you even compile and bind your program. This process reduces the risk of implementing new or changed SQL into production.

BMC SQL Explorer for DB2 supports you in identifying access path changes and changes in SQL statements for whole applications. You can integrate the BMC SQL Explorer for DB2 batch compare component into your lifecycle management solution (which you use to move source code and other application components from one environment to another; for example, mass test to production). This component compares the new access path in the DBRM or in the PLAN_TABLE with the current (old) one, to identify which statements will change in the access path. You can add automation, based on the value of the return code of the batch compare step, to avoid binds or inform DBAs about the exception situations.

Index component

BMC SQL Performance for DB2 includes the Index component to ensure that existing indexes are optimally structured to support the production applications. The Index component extends the capability of BMC APPTUNE object analysis by collecting and reporting on column usage data for SQL statements. It automatically collects and displays actual access counts for each unique SQL statement (table and index, and predicate usage frequencies). For example, you can generate a report that shows how many distinct SQL statements have used a particular column in any kind of predicate or ORDER BY clause. This tells you if statements access non-indexed columns or how changes to existing indexes affect other statements in your workload. Other table and index reports provide quick access to listings of the most-used objects based on get page volume or index access ratio.

The Index component also extends the capability of the Explain function by comparing access paths after making changes to simulated indexes in a cloned database.

A “what-if?” index analysis lets you model changes to indexes. The Index component provides on-demand, dynamic data collection of index dependencies and catalog statistics.

BMC SQL Performance for DB2 enables you to obtain accurate, real-time performance information about DB2 indexes. Because the Index component presents data at the object level, you can review the index access data to evaluate the performance of your SQL and identify candidates for index improvements.

Performance Advisor technology

Because DB2 has many moving parts and can produce a mountain of performance data, solving performance problems can seem like looking for a needle in a hay stack. Performance data includes high-level statistics and accounting data that can be captured at a relatively low cost. BMC MAINVIEW for DB2 collects and summarizes DB2 accounting and statistics data produced by DB2 traces and stores this data in a set of DB2 tables. BMC APPTUNE for DB2 collects low-level data about SQL statements and object access.

Performance advisor technology turns data into intelligence, and intelligence into actions.

The BMC DB2 performance management strategy is evolving to include Performance Advisor technology, which will drive down cost of ownership while delivering high performance and availability. Performance Advisor technology adds advisory and analysis features to performance data that has been collected with BMC products—to automatically identify and resolve problems.

Performance Advisor technology takes this detail data, correlates and consolidates it, and externalizes the data into DB2 tables where it can be analyzed using BMC products or other tools. For example, capturing DB2 access path information at DB2 bind time provides a valuable baseline for later analysis.

While performance data is critical, performance advisors provide the intelligence and automation required to identify and resolve performance problems. You can maintain historical data and use it for batch reporting. Automation is a key feature of the performance advisor, with the ability to resolve problems as they are occurring without direct intervention from a person.

BMC SQL Performance for DB2 version 6.2 and later provides Performance Advisor technology that will evaluate DB2 object performance and determine when a reorganization could be beneficial to solve a performance problem. If BMC Database Performance for DB2 is installed, it can execute the reorganization automatically

Reorganization strategies

DBA trends

The DBA trends nowadays for maintenance today seem to follow one of two opposing strategies: reorganize everything or reorganize nothing. Neither of these is the best strategy. Reorganizing everything means you may be spending CPU on objects that do not need reorganized. On the other hand, some people choose not to reorganize anything when needed but rather do everything once a year. Why such a generic approach? Time and people. We simply do not have the time to evaluate every object to determine if reorganizations are needed and to get those scheduled accordingly.

Tuning potential with reorganizations

Highly organized data can provide you several benefits, especially to sequential processes. Reorganizations provide you the following:

The following occurs during a reorganization of a table space:

- The data in the table space and the corresponding indexes defined on the table in the table space are also reorganized
- The space of dropped tables is reclaimed (if it was not reclaimed before)
- Free space is provided
- New allocation units become effective
- Segments are realigned and the rows of the tables are rearranged in clustering-index sequence (except for simple table spaces with multiple table)
- Overflow pointers are eliminated
- The corresponding indexes are rebuilt from scratch
- Updates the version number of each row to the current version number

The following occurs during the reorganization of an index:

- Fragmented space is reclaimed and it can improve access performance.
- Free space is provided
- Index levels are potentially reduced

To reorganize or not to reorganize: that's the question

While we know that most shops have way to generic standards on reorganizations, the question is when is the best time to reorganize? This question can only be answered by gathering and evaluating proper statistics. Here are some basic guidelines for determining when to run a reorganization.

Step 1: Collect statistics

To determine if a reorganization is needed, you must have current statistics about your data. Use the RUNSTATS utility to gather statistics about your data and determine if a reorganization is needed. You could also use the statistics that are automatically gathered by Real Time Statistics (RTS) to see how your data has changed and if a reorganization may be necessary. Both options are described below.

RUNSTATS

Execute RUNSTATS on a frequent basis on tables that have a large number of updates, inserts, or deletes. For tables with a great deal of insert or delete activity, you may decide to run statistics after a fixed period of time or after the insert or delete activity. The data gathered from RUNSTATS will help you develop a reorganization strategy. You can choose to run statistics without updating the catalog so that you do not risk any access path changes. This can give you a preview of the organization of the data without the potential to affect access paths.

Because RUNSTATS with update can influence access path, especially for dynamic SQL, it is important to understand your database objects, as well as the expected access paths to those objects. While you want the database to know the condition of your data you may not be able to afford a broken access path for a critical application. It may be best to be able to predict the access path change and decided whether or not it's a good thing. This requires a capture of statistics without update, and propagation of statistics to a test environment to check the access paths. You can do this with a home-grown process or a vendor tool. Another choice is to automate RUNSTATS on non-critical objects and control and test the RUNSTATS for critical objects.

NOTE

BMC DASD MANAGER for DB2 provides a DB2 statistics collection utility called BMCSTATS. This batch-driven utility runs outside of DB2, collects more statistics, and uses less time and fewer CPU resources.

The general recommendation is that if you don't understand your access paths and data then you should run RUNSTATS on a regular basis and put your faith in the DB2 optimizer. If you have critical concerns about your transactions then a better understanding of expected access paths and careful controlling of RUNSTATS (perhaps never even running it after the desired paths are achieved) is the way to go.

Real time statistics

The DB2 Real Time Statistics (RTS) facility lets DB2 collect data about table spaces and index spaces and then periodically write this information into two catalog tables. The statistics can then be used by user-written queries or programs, the DB2-supplied stored procedure DSNACCOR, or the Control Center to make decisions for object maintenance (i.e., REORG, RUNSTATS, COPY).

DB2 is always generating real-time statistics for database objects. It keeps these statistics in virtual storage and calculates and updates them asynchronously upon externalization.

Two catalog tables hold the statistics:

- SYSIBM.SYSTABLESPACESTATS
- SYSIBM.SYSINDEXSPACESTATS

DB2 populates these tables with one row per table space, index space, or partition.

Some statistics collected that may help determine when a REORG is needed include space allocated; extents; number of inserts, updates, or deletes (singleton or mass) since the last REORG or LOAD REPLACE; number of unclustered inserts, number of disorganized LOBs, and number of overflow records created since the last reorganization.

Statistics are also gathered on indexes. Basic index statistics include total number of entries (unique or duplicate), number of levels, number of active pages, space allocated, and extents. Statistics that help to determine when an index reorganization is needed include time when the last REBUILD, REORG, or LOAD REPLACE occurred. Statistics also report the number of updates, deletes (real or pseudo, singleton or mass), and inserts (random and those that were after the highest key) since the last REORG or REBUILD. These statistics are, of course, very helpful for determining how your data physically looks after certain processes (e.g., batch inserts) have occurred so you can take appropriate actions if necessary.

An example of using the RTS statistics to determine if a reorganization is needed is below. It shows how to monitor the number of records inserted since the last REORG or LOAD REPLACE that are not well-clustered with respect to the clustering index. Ideally, "well-clustered" means the record was inserted into a page that was within 16 pages of the ideal candidate page (determined by the clustering index). You can use the SYSTABLESPACESTATS table value REORGUNCLUSTINS to determine whether you need to run REORG after a series of inserts.

```
SELECT NAME
FROM SYSIBM.SYSTABLESPACESTATS
WHERE DBNAME = 'DB1' and
   ((REORGUNCLUSTINS*100)/TOTALROWS)>10
```

Step 2: Select/Exclude objects

Object selection is an important part of the reorganization process, especially for highly active databases. Object availability requirements, as well as object priorities, all play a roll.

Reorganizations can be executed with full availability to the applications, but there are some limitations especially when it comes to object switching and drains. Therefore, some reorganizations may have to be excluded from any automated selection process, and managed manually during coordinated limited outage windows.

Index reorganizations are typically far more valuable than table space reorganizations, as well as being faster and easier to perform. Disorganized indexes can play a major role in application performance problems for random access, sequential access, and inserts (as well as some updates to index keys). Index reorganizations should take a priority over table space reorganizations, and table space reorganizations should not be performed just because, but rather on the associated indexes that really need it.

Table space reorganizations become important for sequentially read objects or table spaces with high volumes of inserts. Favor these spaces over others when prioritizing your selection process.

Step 3: Analyze thresholds

Establish some general guidelines for when a table space should be reorganized by using statistics for the clustering index:

- Any data set behind the table space has multiple extents
- CLUSTERRATIO < 90 percent
- (NEARINDREF + FARINDREF) / CARD > 10 percent
  - Can be caused by updates to variable character fields
- FAROFFPOS / CARD > 5 percent
- NEAROFFPOS / CARD > 10 percent
- DBD growth after successive drops/recreates in a table space

Some index space indicators for REORG include the following:

- Review the LEAFDIST, LEAFNEAR and LEAFFAR columns in the SYSINDEXPART catalog table
- Large numbers in the LEAFDIST column indicate that there are several pages between successive leaf pages, and using the index will result in additional overhead

  — In this situation, DB2 may turn off prefetch usage

- PSEUDO_DEL_ENTRIES / CARDF is greater than 10%

Reorganization strategies with BMC solutions

To ensure that DB2 database performance, and therefore availability, is at adequate levels, you must be able to forecast growth and analyze performance trends to determine which table spaces and indexes require maintenance. Then you must balance the need for object maintenance (such as reorganizations) with the impact that the required downtime has on business operations.

DB2 performance degrades over time. When objects become disorganized, this affects DB2 application performance and your ability to meet service level objectives. Disorganized data decreases database efficiency in the following ways:

- In sequential processing, more I / Os are required to retrieve disorganized data than are needed to retrieve data items that are physically adjacent to one another.
- The retrieval inefficiency caused by the disorganization slows DB2 application response time and decreases user productivity.
- A disorganized database wastes DASD space.

The problem becomes more difficult over time. You must manage more and more objects with the same resources, application complexity increases, and availability requirements from the business continues to go up.

What are your choices?

Without automation, you must spend valuable time analyzing and interpreting the large amount of information gathered, and then creating corrective actions for problems discovered.

Taking action to improve database performance, such as running traditional reorganizations, often means that your applications are not available during the process. Online reorganizations are used widely, but are still costly when executed on database objects which do not really require a reorganization. With shrinking batch windows and requirements for access to data 24 hours a day, 7 days a week, finding time and CPU power to reorganize data and perform other maintenance is difficult.

You could schedule reorganization jobs of your objects, whether they need it or not. This is a very expensive and time-consuming approach, and it would not work in any environment that requires continuous high availability, or one that is running at 100% CPU.

BMC Database Performance for DB2

BMC Database Performance for DB2 simplifies the complex, critical tasks that are required to maintain optimal database performance. It enables you to manage larger maintenance workloads and still maintain data availability. With BMC Database Performance for DB2, you can

- Perform maintenance only on objects that require it.
- Reorganize objects without impacting DB2 availability.
- Control and increase collection of DB2 object statistics without impacting application access to the objects.
- Generate batch or online analysis reports to do capacity planning and growth analysis.
- Automate the detection and correction of exception conditions.

BMC DASD MANAGER PLUS for DB2

BMC DASD MANAGER PLUS for DB2 integrates all of the statistics collection and analysis tools that you need to tune a DB2 system for optimal performance and storage usage. By using BMC DASD MANAGER PLUS for DB2, you can gather and store statistics for physical objects in a DB2 production environment, analyze trends, estimate space requirements, monitor changes in the database, and automate utility generation.

BMC DASD MANAGER PLUS provides the following functionality:

- generates BMC Software and IBM utilities and commands
- collects and manages statistics, including improved BMCSTATS capabilities at much lower CPU cost as the RUNSTATS utility.
- analyzes statistical trends using data displays and graphs
- sets thresholds for statistics and reporting exceptions
- sets thresholds and corrective actions for generating utilities automatically
- reports on events, statistics, and exceptions
- estimates space requirements for new and existing objects from statistics

BMC REORG PLUS for DB2

BMC REORG PLUS for DB2 efficiently reorganizes DB2 data. Advanced techniques and additional functions allow BMC REORG PLUS for DB2:

- reduces costs of reorganizing DB2 data because fewer CPU cycles and EXCPs are used
- runs outside of the DBM1 address space so that the reorganization does not interfere with your regular SQL processing within DB2
- increases availability of DB2 data because the time needed to reorganize the data is reduced
- improves DB2 performance by allowing more frequent reorganizations because of reduced reorganization costs and elapsed times

Online reorganizations

BMC Database Performance for DB2 enables online reorganizations. Performing an online reorganization provides the following benefits:

- allows full read / write (RW) access to DB2 data during the reorganization
- delivers significantly improved data availability to meet growing 24 x 7 requirements

- operates in a nondestructive manner, which allows you to easily make the objects available without having to recover

- provides full data availability for batch applications that use the BMC APPLICATION RESTART CONTROL (AR/CTL) for DB2. The AR/CTL suspend-and-resume feature eliminates outages for DB2 objects.

Automation Component

The BMC Database Performance for DB2 Automation component provides a graphical user interface (GUI) console. The console enables IT staff of all experience levels to execute and automate database maintenance tasks by using a common interface.

Using the Automation component, you can expand your ability to manage increasing maintenance workloads and improve database performance. The Automation component enables you to:

- execute and automate database maintenance tasks, like reorganizations, quickly and effectively by using the console

- automate the completion of routine database maintenance, such as performing object analysis on a regular basis

- automating reorganizations based on thresholds that you define, during time periods that you define

- quickly perform daily tasks to keep your databases running at peak performance For example, you can look for potential problems before they affect database performance by viewing the exceptions list and interactively registering reorganization candidates to fix problems

- list current candidates to see the objects and actions that are registered in the automation component repository.

- status of jobs to see the following information:

  — in-progress work
  — pending work in the processing queue
  — jobs that have failed and need to be restarted
  — completed work

The intelligent automation capabilities of the Automation component can complete the following processing, freeing staff to concentrate on other tasks:

- register problem objects in the repository for automated corrective actions or maintenance activities
- locate a maintenance window in which to run the processing
- prioritize the execution of corrective actions or maintenance activities within the selected maintenance window
- resolve the exception condition by building and managing the execution of corrective actions in a predefined maintenance window

BMC SQL Performance for DB2 version 6.2 and later provides Performance Advisor technology that will evaluate DB2 object performance and determine when a reorganization could be beneficial to solve a performance problem. If BMC Database Performance for DB2 is installed, it can execute the reorganization automatically

Index

I

J

L

M

N

O

P

R

S

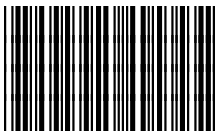
T

U

V

X

Notes



98361