CA Chorus™ for DB2 Database Management

CA Performance Handbook for DB2 for z/OS

Version 03.0.00, Second Edition

CA Technologies Product References

This document references the following CA Technologies products:

- CA Database Analyzer™ for DB2 for z/OS (CA Database Analyzer)
- CA Detector® for DB2 for z/OS (CA Detector)
- CA Index Expert™ for DB2 for z/OS (CA Index Expert)
- CA Insight™ Database Performance Monitor for DB2 for z/OS (CA Insight DPM)
- CA Plan Analyzer® for DB2 for z/OS (CA Plan Analyzer)
- CA Rapid Reorg® for DB2 for z/OS (CA Rapid Reorg)
- CA SQL-Ease® for DB2 for z/OS (CA SQL-Ease)
- CA Subsystem Analyzer for DB2 for z/OS (CA Subsystem Analyzer)

Contact CA Technologies

Contact CA Support

For your convenience, CA Technologies provides one site where you can access the information that you need for your Home Office, Small Business, and Enterprise CA Technologies products. At http://ca.com/support, you can access the following resources:

- Online and telephone contact information for technical assistance and customer services
- Information about user communities and forums
- Product and documentation downloads
- CA Support policies and guidelines
- Other helpful resources appropriate for your product

Providing Feedback About Product Documentation

If you have comments or questions about CA Technologies product documentation, you can send a message to techpubs@ca.com.

To provide feedback about CA Technologies product documentation, complete our short customer survey which is available on the CA Support website at http://ca.com/docs.

Documentation Changes

The following documentation updates have been made since the first edition of this documentation:

- Legal Notices (see page 2)—Updated to reflect public documentation legal disclaimer.

Contents

Chapter 1: Introducing DB2 Performance

Using This Handbook

This handbook helps DB2 performance tuners to complete the following tasks:

- Design databases for high performance.
- Understand how DB2 processes SQL and transactions, and tune those processes for performance.

For database performance management, the philosophy of performance tuning begins with database design and extends through the database and application development and production implementation.

Key Topics

This guide includes the following key topics:

- Fundamentals of DB2 performance.
- Descriptions of features and functions that help you manage and optimize performance.
- Methods for identifying DB2 performance problems.
- Guidelines for tuning static and dynamic SQL.
- Guidelines for the proper design of tables and indexes for performance.
- DB2 subsystem performance information, and advice for subsystem configuration and tuning.
- Commentary for proper application design for performance.
- Real-world tips for performance tuning and monitoring.

Audience

The primary audience for this handbook is physical and logical database administrators.

Because performance management has many stakeholders, additional users may benefit from this information, including application developers, data modelers, and data center managers. For successful performance initiatives, DBAs, developers, and managers must cooperate and contribute to performance management. There are no right ways or wrong ways exist to tune performance. However, trade-offs are common as people decide about database design and performance.

Are you designing for ease of use, minimized programming effort, flexibility, availability, or performance? As you make these choices, consider organizational costs and benefits and whether they are measured in the amount of time and effort by the personnel involved in development and maintenance, response time for end users, or CPU metrics.

This handbook assumes that you have a good working knowledge of DB2 and SQL. This book helps you build good performance into the application, database, and the DB2 subsystem. This book provides techniques to help you monitor DB2 for performance and to identify and tune production performance problems.

About DB2 Performance

When one of the following events occurs, DB2 performance becomes a focal point:

- An online production system does not respond within the time specified in a service level agreement (SLA).
- An application uses more CPU (and thus costs more money) than is desired.

Most of the efforts involved in tuning during these situations revolve around fire fighting, and when the problem is solved, the tuning effort is abandoned. Performance tuning can be categorized as follows:

- Designing for performance (application and database)
- Testing and measuring performance design efforts
- Understanding SQL performance and tuning SQL
- Tuning the DB2 subsystem
- Monitoring production applications

An understanding of DB2 performance, even at a high level, can help ensure the performance of applications, databases, and subsystems.

Measuring DB2 Performance

Performance design and tuning are vital because users want accessible, accurate, and secure data.

Strong performance is a balance between expectations and reality. For effective performance tuning, work with each stakeholder to identify and understand their expectations. After identifying and understanding the expectations of the stakeholders, you realize the full benefits of performance tuning.

How you define strong DB2 database and application performance centers on two areas. First, it means that users do not need to wait for the information that they require to do their jobs. Second, it means that the organization does not spend too much money running the application. The methods for designing, measuring, and tuning for performance vary from application to application. The optimal approach depends upon the type of application.

Your site may use one or all of the following factors to measure performance:

- User complaints
- Overall response time measurements
- Service level agreements (SLAs)
- System availability
- I/O time and waits
- CPU consumption and associated charges
- Locking time and waits

More Information:

EXPLAIN Facility and Predictive Analysis (see page 79)

DB2 Performance Problem Root Cause

When the measurements that you use at your site warn you of a performance problem, you face the challenge of finding the root cause. The following list details common root causes of performance problems:

- Poor database design
- Poor application design
- Catalog statistics that do not reflect the data size, organization, or both
- Poorly written SQL

- Generic SQL design
- Inadequately allocated system resources, such as buffers and logs

In other situations, you may be working with packaged software applications that cause performance problems. In these situations, it may be difficult to tune the application, but it is possible to improve performance through database and subsystem tuning.

More Information:

Understand and Tune Your Packaged Applications (see page 143)

Designing for Performance

Teams can optimize an application for high availability, ease of programming, flexibility and reusability of code, and performance. The most thorough application design must consider all of these areas. If a design team commits the following errors, major performance problems can occur:

- Overlooks performance
- Considers it late in the design cycle
- Fails to understand, test, and address performance before production

Designing an application for performance can help avoid many performance problems after the application has been deployed. Proactive performance engineering can help eliminate the redesign, recoding, and retrofitting during implementation to satisfy performance expectations or alleviate performance problems. Proactive performance engineering also lets you analyze and stress test the designs before you implement them.

Using the techniques and tools in this handbook can turn performance engineering research and testing from a lengthy process inhibiting development to one that can be performed efficiently before development.

Many options exist to index and table design for performance. These options center on the type of application that you are implementing, specifically the type of data access (random versus sequential, read versus update, and so on). In addition, the application has to be properly designed for performance. The design process requires an understanding of the application data access patterns, as well as the units of work (UOWs) and inputs. Considering these points and designing the application and database accordingly are the best defense against performance problems.

More Information:

Designing Tables and Indexes for Performance (see page 59)
EXPLAIN Facility and Predictive Analysis (see page 79)

Identifying Performance Problems

How do we find performance problems? Are SQL statements in an application causing problems? Does the area of concern involve the design of our indexes and tables? Or, perhaps our data has become disorganized, or the catalog statistics do not accurately reflect the data in the tables? It can also be an issue of dynamic SQL, and poor performing statements, or perhaps the overhead of preparing the dynamic SQL statements.

When it comes to database access, finding and fixing the worst performing SQL statements is the top priority. However, how does one prioritize the worst culprits? Are problems caused by the statements that are performing table space scans or the ones with matching index access?

Performance is relevant, and finding the least efficient statement is a matter of understanding how that statement is performing relative to the business need that it serves. This handbook can help you prioritize site-specific performance needs.

Fixing the Performance Problems

After you identify a performance problem, determine the best approach to address it. The problem could be updating catalog statistics, tuning an SQL statement, or adjusting parameters for indexing, table space, or subsystem.

With the proper monitoring in place, you can identify, understand, and fix the problem.

More Information:

Understand and Tune Your Packaged Applications (see page 143)
Designing Tables and Indexes for Performance (see page 59)
Application Design and Tuning for Performance (see page 105)
Tuning Subsystems (see page 125)
Tuning Tips (see page 153)

Chapter 2: SQL and Access Paths

DB2 offers you a level of abstraction from the data because you do not have to know the following information:

- Where the data physically exists
- Where the data sets are
- Which is the access method for the data
- What index to use

All that you should know is the name of the table and the columns you are interested in. You can quickly build applications that access data using standardized language that is independent of any access method or platform. This design enables development of applications, portability across platforms, and all of the power and flexibility that comes with a relational or object-relational design.

The DB2 Optimizer

The DB2 optimizer performs the following functions:

- Accepts a statement
- Checks for syntax
- Checks the DB2 system catalog
- Builds access paths to the data

The DB2 optimizer uses catalog statistics to attach a cost to the possible access paths and then chooses the cheapest path. Understanding the DB2 optimizer, the DB2 access paths, and how your data is accessed helps you adjust the DB2 performance.

The DB2 optimizer is responsible for interpreting your queries and determining how to access your data in the most efficient manner. However, it can use only the best of the available access paths. The DB2 optimizer does not know what access paths are possible that are not available. The DB2 optimizer can deal only with the information that you provide. This information is primarily stored in the DB2 system catalog, which includes the basic information about the tables, columns, indexes, and statistics. The DB2 optimizer does not know about the indexes that could be built, or anything about the possible inputs to our queries (unless all literal values are provided in a query), input files, batch sequences, or transaction patterns. Therefore, an understanding of the DB2 optimizer, statistics, and access paths is important, but the design of applications and databases for performance is also important.

The DB2 optimizer interprets your queries and determines the most efficient method to access your data. The DB2 optimizer can use access paths that are only available when determining the most efficient way to access your data. If other access paths exist but are not available, these access paths are not considered.

The DB2 optimizer can use only the information you provide when determining access paths and interpreting queries. The primary location for this information is the DB2 system catalog, which includes the basic information about tables, columns, indexes, and statistics. However, the DB2 optimizer does not know about indexes that could be built, possible inputs to queries, input files, batch sequences, or transactions patterns unless you specifically provide this information.

Importance of Catalog Statistics

Catalog statistics include information about the amount of data and the organization of that data in DB2 tables. The DB2 optimizer selects access paths to the data based on cost, so catalog statistics are critical to proper performance. Maintaining accurate, up-to-date statistics is critical to performance.

The DB2 optimizer uses various statistics to determine the cost of accessing the data in tables. The type of query (static or dynamic), use of literal values, and run-time optimization options dictates which statistics are used. The cost that is associated with various access paths affects whether indexes are chosen, or which indexes are chosen, the access path, and table access sequence for multiple table access paths.

In addition to catalog statistics and database design, the DB2 optimizer uses other information to calculate and choose an efficient access path. The following information is used to determine the access path:

- Model of the central processor
- Number of processors
- Size of the RID pool
- Various installation parameters
- Buffer pool sizes

Ensure that you are aware of these factors as you tune your queries.

Following are the three types of DB2 catalog statistics:

- Cardinality
- Frequency Distribution
- Histogram

Each type provides different details about the data in your tables, and DB2 uses these statistics to determine the best way to access the data dependent on the objects and the query.

Cardinality Statistics

Cardinality statistics reflect the number of rows in a table or the number of distinct values for a column in a table. These statistics provide the main source for the filter factor. *Filter factor* is a percentage of the number of rows that are expected to be returned for a column or a table. DB2 uses cardinality statistics to determine which of the following methods to use to access data:

- Whether access is using an index scan or a table space scan
- Which table to access first when joining tables

DB2 needs an accurate estimate of the number of rows that qualify after applying various predicates in your queries to determine the optimal access path. When multiple tables are accessed, the number of qualifying rows that are estimated for the tables can also affect the table access sequence. Column and table cardinalities provide the basic, but critical information for the DB2 optimizer to make these estimates.

Example: EMP Table Access

When DB2 has to choose the most efficient way to access the EMP table, this access depends on the following factors:

- Size of the table
- Cardinality of the SEX column
- Any index that might be defined on the table

DB2 uses the catalog statistics to determine the filter factor for the SEX column. In this case, 1/COLCARDF, COLCARDF is a column in the SYSIBM.SYSCOLUMNS catalog table. The resulting fractional number represents the number of rows that are expected to be returned based on the predicate. DB2 uses this filter factor number to decide whether to use an index (if available) or table space scan.

Example: EMP Table Access Path

If the cardinality of the SEX column is three (male, female, and unknown), the table has three unique occurrences of a value of SEX. In this case, DB2 determines a filter factor of 1/3, or 33 percent of the values. Consider that the cardinality of the table, as reflected in the CARDF column of the SYSIBM.SYSTABLES catalog table, is 10,000 employees. Then, the estimated number of rows that are returned from the query is 3,333. DB2 uses this information to decide which access path to choose.

Consider that you are using the predicate in the following query in a join to another table. The filter factor that is calculated will be used not only to determine the access path to the EMP table, but it could also influence which table will be accessed first in the join.

```
SELECT EMPNO
FROM   EMP
WHERE  SEX = :SEX
```

Example: Column Groups

Statistics can also be gathered in groups of columns. This method is called *column correlation statistics*. This method is important because the DB2 optimizer can use only the information it is given. The following command shows an example that uses this method:

```
SELECT E.EMPNO, D.DEPTNO, D.DEPTNAME
FROM   EMP E
INNER JOIN
       DEPT D
ON     E.WORKDEPT = D.DEPTNO
WHERE  LASTNAME   = :LAST-NAME
```

Example: Filter Factor

Imagine that the amount of money that someone is paid is correlated to the amount of education they have received. For example, an intern with one year of college is not paid as much as an executive with an MBA. However, if the DB2 optimizer is not given this information, it has to multiply the filter factor for the SALARY predicate by the filter factor for the EDLEVEL predicate. This calculation may result in an exaggerated filter factor for the table, and it could negatively influence the choice of an index or the table access sequence of a join. For this reason, it is important to gather column correlation statistics on any columns that might be correlated. More conservatively, you may want to gather column correlation statistics on any columns that are referenced together in the WHERE clause. The following command shows an example that uses this method:

```
SELECT E.EMPNO
FROM   EMP E
WHERE  SALARY  > :SALARY
AND    EDLEVEL = :EDLEVEL
```

Frequency Distribution Statistics

Frequency distribution statistics reflect the percentage of frequently occurring values. You can collect the information about the percentage of values that occur most or least frequently in a column in a table. These frequency distribution statistics can dramatically impact the performance of the following types of queries:

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

Note: DB2 can collect distribution statistics using the RUNSTATS utility.

In some cases, cardinality statistics are not enough. With only cardinality statistics, the DB2 optimizer has to assume that the data is evenly distributed. For example, consider the SEX column from the previous queries. The column cardinality is 3. However, it is common that the values in the SEX column will be *M* for male or *F* for female, with each representing approximately 50 percent of the values. The value *U* occurs only a small fraction of the time. This example is called *skewed distribution*.

Skewed distributions can have a negative influence on query performance if DB2 does not know about the distribution of the data in a table, the input values in a query predicate, or both. In the following query:

```
SELECT EMPNO
FROM   EMP
WHERE  SEX = :SEX
```

Likewise, if the following statement is issued:

```
SELECT EMPNO
FROM   EMP
WHERE  SEX = 'U'
```

Most of the values are M or F, and DB2 has only cardinality statistics available, then the same formula and filter factor applies. In such situations, the distribution statistics can pay off.

If frequency distribution statistics for the SEX column are gathered in the previous examples, the following frequency values in the SYSIBM.SYSCOLDIST table may occur (simplified in this example):

VALUE	FREQUENCY
M	.49
F	.49
U	.01

If DB2 has this information, and if the following query is issued:

```
SELECT EMPNO
FROM   EMP
WHERE  SEX = 'U'
```

DB2 can determine that the value U represents about one percent of the values of the SEX column. This additional information can have dramatic effects on the access path and table access sequence decisions.

Histogram Statistics

Histogram statistics summarize data distribution on an interval scale and span the entire distribution of values in a table.

Histogram statistics divide values into quantiles. The *quantile* defines an interval of values. The quantity of intervals is determined using a specification of quantiles in a RUNSTATS execution. Thus, a quantile represents a range of values within the table. Each quantile contains approximately the same percentage of rows. This percentage can be more than the distribution statistics in that all values are represented.

Histogram statistics improve on distribution statistics because they provide value-distribution statistics that are collected over the entire range of values in a table. Collecting statistics over the entire table goes beyond the capabilities of distribution statistics because distribution statistics are limited to only the most or least occurring values in a table.

To query on the middle initial of a person's name, the initials are most likely in the range of A to Z. If you used only cardinality statistics, any predicate referencing the middle initial column would receive a filter factor of 1/COLARDF or 1/26. If you have used distribution statistics, any predicate that used a literal value could take advantage of the distribution of values to determine the best access path. That determination would depend on the value being recorded as one of the most or least occurring values. If it is not, the filter factor is determined based on the difference in frequency of the remaining values. Histogram statistics eliminate this guesswork.

If histogram statistics are calculated for the middle initial, they would appear similar to the following values:

QUANTILE	LOWVALUE	HIGHVALUE	CARDF	FREQ
1	A	G	5080	20%
2	H	L	4997	19%
3	M	Q	5001	20%
4	R	U	4900	19%
5	V	Z	5100	20%

The DB2 optimizer has information about the entire span of values in the table and any possible skew. This information is especially useful for the following range predicates:

```
SELECT EMPNO
FROM   EMP
WHERE  MIDINIT BETWEEN 'A' and 'G'
```

Access Paths

DB2 can use the following access paths when accessing the data in your database:

- Tablespace Scan
- Limited Partition Scan
- Index Access
- List Prefetch
- Nested Loop Join
- Hybrid Join
- Merge Scan Join
- Star Join

The access paths are exposed using the EXPLAIN facility. You can invoke the EXPLAIN facility using one of these techniques:

- EXPLAIN SQL statement
- EXPLAIN(YES) BIND or REBIND parameter
- Visual EXPLAIN
- Optimization Service Center

Table space Scans

You can use the DB2 table space scan to search an entire table space to satisfy a query. The type of table space determines what is scanned. Simple table spaces will be scanned in their entirety. Segmented and universal table spaces will be scanned for only the table specified in the query.

Note: An R in the PLAN_TABLE ACCESSTYPE column indicates a table space.

A table space scan is selected as an access method when:

- A matching index scan is not possible because no index is available or no predicates match the index column.
- DB2 calculates that a high percentage of rows is retrieved, so any index access would not be beneficial.
- The indexes that have matching predicates have a low cluster ratio and are not efficient for large amounts of data that the query is requesting.

Limited Partition Scan and Partition Elimination

DB2 uses partitioning key values to eliminate partitions from a table space scan. DB2 uses the column values that define the partitioning key to eliminate partitions from the query access path. You must code queries explicitly to eliminate partitions. This partition elimination can apply to table space scans and index access.

For example, if the EMP table is partitioned by the EMPNO column, the following query would eliminate any partitions not qualified by the predicate:

```
SELECT EMPNO, LASTNAME
FROM   EMP
WHERE  EMPNO BEWTEEN '200000' AND '299999'
```

If an index was available on the EMPNO column, DB2 can choose an index-based access method. DB2 could also choose a table space scan, but use the partitioning key values to access only the partition in which the predicate matches.

If the partitioning key values are supplied in a predicate, DB2 can also choose a secondary index for the access path and can still eliminate partitions using those values. For example, in the following query DB2 can choose a partitioned secondary index on the WORKDEPT column and can eliminate partitions based on the range provided in the query for the EMPNO:

```
SELECT EMPNO, LASTNAME
FROM   EMP
WHERE  EMPNO BEWTEEN '200000' AND '299999'
AND    WORKDEPT = 'C01'
```

Index Access

DB2 uses indexes on your tables to perform these actions:

- Access the data based on the predicates in the query
- Avoid a sort in support of the use of the DISTINCT clause
- ORDER BY and GROUP BY clauses
- INTERSECT and EXCEPT processing

DB2 matches the predicates in your queries to the leading key columns of the indexes that are defined against your tables. The MATCHCOLS column of the PLAN_TABLE indicates this match. If the number of columns matched is greater than zero, the index access is considered as a matching index scan. If the number of matched columns is equal to zero, the index access is considered as a non-matching index scan. In a non-matching index scan, all of the key values and their record identifiers (RIDs) are read.

Note: A non-matching index scan is used if DB2 can use the index to avoid a sort when the query contains any of the following clauses:

- ORDER BY
- DISTINCT
- GROUP BY
- NTERSECT
- EXCEPT

The matching predicates on the leading key columns are equal (=) or IN predicates. This match would correspond to an ACCESSTYPE value of either I or N, respectively. The predicate that matches the last index column can be an equal, IN, NOT NULL, or a range predicate (<, <=, >, >=, LIKE, or between).

For example, the following query assumes that the EMPPROJACT DB2 sample table has an index on the PROJNO, ACTNO, EMSTDATE, and EMPNO columns:

```
SELECT EMENDATE, EMPTIME
FROM   EMPPROJACT
WHERE  PROJNO = 'AD3100'
AND    ACTNO IN (10, 60)
AND    EMSTDATE > '2002-01-01'
AND    EMPNO = 000010
```

In the previous example, DB2 chooses a matching index scan matching on the PROJNO, ACTNO, and EMSTDATE columns. DB2 does not choose a match on the EMPNO column because the previous predicate is a range predicate. However, the EMPNO column can be applied as an index screening column. Because the EMPNO column is a stage 1 predicate, DB2 can apply it to the index entries after the index is read. Although the EMPNO column does not limit the range of entries that are retrieved from the index, it can eliminate the entries as the index is read. This elimination reduces the number of data rows that have to be retrieved from the table.

If all of the columns specified in a statement can be found in an index, DB2 may choose index only access. The value Y in the INDEXONLY column of the PLAN_TABLE indicates the index only access.

DB2 can access indexes using multi-index access. An ACCESSTYPE of M in the PLAN_TABLE indicates the multi-index access. A multi-index access involves reading multiple indexes or the same index multiple times, gathering qualifying RID values together in a union or intersection of values and then sorting them in data page number sequence. In this way, a table can still be accessed efficiently for more complicated queries.

With multi-index access, each operation is represented in the PLAN_TABLE in an individual row. These steps include the matching index access (ACCESSTYPE = MX or DX) for regular indexes or DOCID XML indexes, and then unions or intersections of the RIDs (ACCESSTYPE = MU, MI, DU, or DI).

For example, the following SQL statement could use multi-index access if an index exists on the LASTNAME and FIRSTNME columns of the EMP table:

```
SELECT EMPNO
FROM   EMP
WHERE  (LASTNAME = 'HAAS' AND FIRSTNME = 'CHRISTINE')
OR     (LASTNAME = 'CHRISTINE' AND FIRSTNME = 'HAAS')
```

With the previous query, DB2 could possibly access the index twice, union the resulting RID values together (due to the OR condition in the WHERE clause), and then access the data.

Virtual Buffer Pools

You can have up to 80 virtual buffer pools. This feature allows up to any one of the following buffer pool sizes:

- Fifty 4-KB page buffer pools BP0 - BP49
- Ten 32-KB page buffer pools BP32K - BP32K9
- Ten 8-KB page buffer pools
- Ten 16-KB page buffer pools

The physical memory available on your system limits the size of the buffer pools, with a maximum size for all buffer pools of 1 TB. It does not use additional resources to search a large pool versus a small pool. If you exceed the available memory in the system, the system begins swapping pages from physical memory to disk. This swapping can severely affect performance.

What EXPLAIN Tells You

The PLAN_TABLE is the key table for determining the access path for a query. The PLAN_TABLE provides the following critical information:

METHOD

Indicates the joint method, or whether an additional sort step is required.

ACCESSTYPE

Indicates whether the access to is through a table space scan or through index access. If it is by index access, the specific type of index access is indicated.

MATCHCOLS

Indicates the number of columns that are matched against the index if an index access is used.

INDEXONLY

Indicates that the access required can be served by accessing the index only, and avoiding any table access when a value of Y is in this column.

SORTN####, SORTC####

Indicates any sorts that may happen in support of a UNION, grouping, joint operation, and so on.

PREFETCH

Indicates whether the prefetch can play a role in the access.

By manually running EXPLAIN and examining the PLAN-TABLE, you can retrieve the following information:

- Access path
- Indexes that are used
- Join operations
- Any sorting that occurs as part of your query

If you have additional EXPLAIN tables (for example, those created by Visual Explain or the Optimization Service Center), those tables are populated automatically. This is done by using those tools or by manually running EXPLAIN. If you do not have the remote access that is required from a PC to use the tools, you can also query those tables manually.

The tables provide detailed information about such entities as predicates stages, filter factors, eliminated partitions, parallel operations, detailed cost information, and so on.

Note: For more information about tables, see the *DB2 Performance Monitoring and Tuning Guide* (DB2 9).

List Prefetch

List prefetch is used for access to a moderate number of rows when access to the table is by non-clustering or clustering index when the data is disorganized. List prefetch is less useful for queries that process large quantities of data or small amounts of data.

The preferred method of table access is by using a clustered index. When an index is defined as clustered, DB2 tries to keep the data in the table in the same sequence as the key values in the clustering index. When the table is accessed using the clustering index and the data in the table is organized, the access to the data in the table typically is sequential and predictable. If accessing the data in a table using a non-clustering index or the clustering index with a low cluster ratio, the access to the data can be random and unpredictable. In addition, the same data pages could be read multiple times.

When DB2 detects that a non-clustering index or a clustering index that has a low cluster ratio is used to access a table, DB2 may choose the list prefetch index access method instead.

Note: List prefetch is indicated in the PLAN_TABLE with a value of L in the PREFETCH column.

List prefetch accesses the index using a matching index scan of one or more indexes and collects the RIDs into the RID pool. RID pool is a common area of memory. The RIDs are sorted in ascending order by the page number, and the pages in the table space are retrieved in a sequential or a skip sequential way.

If DB2 determines that the RIDs to be processed will take more than 50 percent of the RID pool when the query is executed, list prefetch is not chosen as the access path. If DB2 determines that more than 25 percent of the rows of the table must be accessed, list prefetch can terminate during the execution. These situations are known as *RDS failures*. During *RDS failures,* the access path changes to a table space scan.

Nested Loop Join

The nested loop join is the access method that DB2 has selected when joining two tables together and you do not specify join columns. A nested loop join repeatedly accesses the outer table as rows are accessed in the inner table. The nested loop join is most efficient when a few rows qualify for the inner table. The nested loop join is a good access path for transactions that are processing little or no data.

Note: Nested loop join is indicated using a value of 1 in the METHOD column of the PLAN_TABLE.

In a nested loop join, DB2 scans the outer or first table accessed in the join operation using a table space scan or index access. For each qualifying row in that table, DB2 searches for matching rows in the inner or second table accessed. DB2 concatenates any rows that it finds in both tables.

An index that supports the join on the outer table is important. For the best performance, that index should be a clustering index in the same order as the inner table. In this way, a join operation can be a sequential and efficient process.

DB2 can also dynamically build a sparse index on the outer table when no index is available on that table. If DB2 determines one of the following facts, it may sort the inner table:

- The join columns are not in the same sequence as the outer table.
- The join columns are not in an index.
- The join columns are in a clustering index where the table data is disorganized.

Hybrid Join

When DB2 detects that the inner table access is a non-clustering index or a clustering index in which the cluster ratio is low, the hybrid join access method is used.

Note: The hybrid join is indicated with a value of 4 in the METHOD column of the PLAN_TABLE.

In a hybrid join, DB2 scans the outer table using a table space scan or an index. DB2 then joins the qualifying rows of the outer table with the RIDs from the matching index entries. DB2 creates a RID list for all the qualifying RIDs of the inner table. DB2 sorts the outer table and RIDs creating a sorted RID list and an intermediate table. DB2 then accesses the inner table using the list prefetch and joins the inner table to the intermediate table.

Hybrid join can outperform a nested loop join when the inner table access is a non-clustering index or clustering index for a disorganized table. The hybrid join is better if the equivalent nested loop join would be accessing the inner table in a random fashion and accessing the same pages multiple times. Hybrid join takes advantage of the list prefetch processing on the inner table to read all the data pages only once in a sequential or skip sequential manner. The hybrid join is also better when the query processes more than a tiny amount of data or less than a very large amount of data.

Because the list prefetch is used, the hybrid join can experience RID list failures. RID list failures result in one of the following actions:

- A table space scan of the inner table on initial access to the inner table.
- A restart of the list prefetch processing on subsequent accesses.

If you are experiencing RID list failures in a hybrid join, try to encourage DB2 to use a nested loop join.

Merge Scan Join

Merge scan join is used as the table access method when DB2 detects one of the following facts:

- Many rows of the inner and outer table qualify for the join.
- The tables have no indexes on the matching columns of the join.

Note: A merge scan join is indicated with a value of 2 in the METHOD column of the PLAN_TABLE.

In a merge scan join, DB2 scans both tables in the order of the join columns. If no efficient indexes are providing the join order for the join columns, DB2 sorts the outer table, the inner table, or both tables. The inner table is placed into a work file. If the outer table has to be sorted, it is placed into a work file. DB2 then reads both tables and join the rows together using a match/merge process.

Star Join

DB2 employs a star join when it detects that the query is joining tables that are a part of a star schema design of a data warehouse.

Note: The star join is indicated with the value of S in the JOIN_TYPE column of the PLAN_TABLE.

Read Mechanisms

DB2 can apply read mechanisms to enhance performance when accessing your data. These performance enhancers can have a dramatic effect on the performance of your queries and applications.

These read mechanisms are as follows:

- Sequential Prefetch
- Dynamic Prefetch and Sequential Detection
- Index Lookaside

Sequential Prefetch

When your query reads data in a sequential manner, DB2 elects to use the sequential prefetch to read the data ahead of the query requesting it. DB2 launches asynchronous read engines to read data from the index and table page sets into the DB2 buffers. This prefetch loads the data pages into the buffers in expectation of the query accessing them.

The size of the buffer pool that is used determines the maximum number of pages that are read by a sequential prefetch operation. When the buffer pool is smaller than 1000 pages, the prefetch quantity is limited due to the risk of filling up the buffer pool. If the buffer pool has more than 1000 pages, 32 pages can be read with a single physical I/O.

Note: A value of S in the PREFETCH column of the PLAN_TABLE indicates the sequential prefetch.

Dynamic Prefetch and Sequential Detection

DB2 can perform the sequential prefetch dynamically at execution time. This method is called *dynamic prefetch*. This method uses a technique called *sequential detection* to detect the forward reading of table pages and index leaf pages. DB2 tracks the pattern of pages that is accessed to detect sequential or near sequential access. The most recent eight pages are tracked, and data access is determined to be sequential if more than four of the last eight pages are page-sequential.

Note: A page is considered page-sequential if it is within one half of the prefetch quantity for the buffer pool.

Five of eight sequentially available pages activate dynamic prefetch. If DB2 determines that the query or application is no longer reading sequentially, it can deactivate the dynamic prefetch. If a query is likely to get dynamic prefetch at execution time, DB2 indicates in an access path.

Note: A value of D in the PREFETCH column of the PLAN_TABLE indicates the dynamic prefetch.

The area of memory that is used to track the last eight pages that are accessed is destroyed and rebuilt for RELEASE(COMMIT). So, dynamic prefetch is dependent on the bind parameter RELEASE(DEALLOCATE). Because RELEASE(DEALLOCATE) is not effective for remote connections, sequential detection and dynamic prefetch are less likely for remote applications.

Index Lookaside

Index lookaside is a performance enhancer that is active only after an initial access to an index. For repeated probes of an index, DB2 checks the most current index leaf page to see if the key is found on that page. If DB2 finds the key, DB2 does not read from the root page of the index down to the leaf page, but it uses the values from the leaf page. This method can affect performance because getpage operations and I/Os can be avoided.

Index lookaside depends on the query reading index data in a predictable pattern. This happens for transactions that are accessing the index using a common cluster with other tables. Index lookaside is dependent on the RELEASE(DEALLOCATE) bind parameter.

Sorting

Sorting is used to support certain clauses in your SQL statement or in support of certain access paths.

DB2 can sort in support of the following functions:

- ORDER BY clause
- GROUP BY clause
- To remove duplicates (DISTINCT, UNION, EXCEPT, or INTERSECT)
- In join or subquery processing

Sorting in an application is always faster than sorting in DB2 for the small result sets. However, by placing the sort in the program, you remove some of the flexibility, power, and portability of the SQL statement, along with the ease of coding.

Avoiding a Sort for an ORDER BY

DB2 can avoid a sort for an ORDER BY by using the leading columns of an index to match the ORDER BY columns. If the columns are the leading columns of the index that are used to access a single table or the leading columns of the index to access the first table in a join operation. DB2 uses *order by pruning* to avoid a sort.

As an example, in this query, we have an index on WORKDEPT, LASTNAME, and FIRSTNME. DB2 avoids this sort if it uses the index on those three columns to access the table.

```
SELECT *
FROM   EMP
WHERE  WORKDEPT = 'D01'
ORDER BY WORKDEPT, LASTNAME, FIRSTNME
```

DB2 can also avoid a sort in one of the following situations:

- When any number of the leading columns of an index matches the ORDER BY clause.
- When the query contains an equals predicate on the leading columns ORDER BY pruning.

In this example, the following queries avoid a sort if the index is used:

```
SELECT *
FROM   EMP
ORDER BY WORKDEPT

SELECT *
FROM   EMP
WHERE  WORKDEPT = 'D01'
ORDER BY WORKDEPT, LASTNAME
```

Avoiding a Sort for a GROUP BY

A GROUP BY can use any of the following methods to avoid a sort:

- Unique index
- Duplicate index
- Full or partial index key

If the index on WORKDEPT, LASTNAME, and FIRSTNME is used, you can avoid each of the following statements:

```
SELECT WORKDEPT, LASTNAME, FIRSTNME, COUNT(*)
FROM   EMP
GROUP BY WORKDEPT, LASTNAME, FIRSTNAME

SELECT WORKDEPT, COUNT(*)
FROM   EMP
GROUP BY WORKDEPT

SELECT WORKDEPT, FIRSTNME, COUNT(*)
FROM   EMP
WHERE  LASTNAME = 'SMITH'
GROUP BY WORKDEPT, FIRSTNAME
```

Avoiding a Sort for a DISTINCT

A DISTINCT can avoid a sort by using a unique index only if that index is used to access the table. You can use a GROUP BY on all of the columns in the SELECT list to take advantage of the additional capabilities of GROUP BY to avoid a sort.

As an example, the following two statements are identical:

```
SELECT DISTINCT WORKDEPT, LASTNAME, FIRSTNME
FROM   EMP

SELECT WORKDEPT, LASTNAME, FIRSTNME
FROM   EMP
GROUP BY WORKDEPT, LASTNAME, FIRSTNME
```

Avoiding a Sort for a UNION EXCEPT or INTERSECT

DB2 can avoid a sort for an EXCEPT or INTERSECT. DB2 cannot avoid a sort for a union in one of the following situations:

- If there is a matching index with the UNIONed, intersected, and excepted columns.
- If the inputs to the operation are already sorted by a previous operation.

DB2 cannot avoid a sort for a union, but if duplicates are possible change each to a UNION ALL, EXCEPT ALL, or INTERSECT ALL.

Parallelism

DB2 uses parallel operations to access tables and indexes. This configuration impacts performance for queries that process large quantities of data across multiple table and index partitions. The response time for these data-or processor-intensive queries can be reduced.

The two types of query parallelism are as follows:

- Query I/O parallelism
- Query CP parallelism

I/O parallelism manages concurrent I/O requests for a single query. I/O parallelism can fetch data using these multiple concurrent I/O requests into the buffers and significantly reduce the response time for large I/O bound queries.

Query CP parallelism breaks a large query into smaller queries and runs the smaller queries in parallel on multiple processors. This type of parallelism can also significantly reduce the elapsed time for large queries.

You can enable query parallelism by using the DEGREE ANY parameter of a BIND command or by setting the value of the CURRENT DEGREE special register to the value of ANY. Because there is overhead to the start of the parallel tasks, ensure that you use query parallelism for larger queries. For small queries, query parallelism can be a performance detriment.

Chapter 3: Predicates and SQL Tuning

To write efficient SQL statements and tune SQL statements, you need a basic understanding of how DB2 optimizes SQL statements and how it accesses your data.

When writing and tuning SQL statements, filter as much as possible as early as possible. The most expensive thing you can do is to travel from your application to DB2. Filter as close to the indexes and data as possible while processing in DB2.

Understand how to reduce repeat processing. The most efficient SQL statement is the statement that never executes. Do only what is necessary to complete the job.

DB2 Predicates

Predicates in SQL statements are classified. These classifications dictate how DB2 processes the predicates and how much data is filtered during the process. These classifications are as follows:

Stage 1 and Stage 1 Indexable

The DB2 stage 1 engine understands your indexes and tables and can use an index for efficient access to your data. Only a stage 1 predicate can limit the range of data accessed on a disk.

Stage 2

The stage 2 engine processes functions and expressions. But, it is unable to access data in indexes and tables directly. Data from stage 1 is passed to stage 2 for further processing. So, stage 1 predicates are more efficient than stage 2 predicates. Stage 2 predicates cannot use an index, and thus cannot limit the range of data retrieved from a disk.

Stage 3

A stage 3 predicate is processed in the application layer. Filtering is performed after the data is retrieved from DB2 and processed in the application. Stage 3 predicates are the least efficient.

The *IBM DB2 Performance Monitoring and Tuning Guide* contains lists of the various predicate forms, and whether they are stage 1 indexable, stage 1, or stage 2.

Stage 1 Indexable

Stage 1 indexable predicates offer the best performance for filtering. However, just because a predicate is listed as stage 1 indexable does not mean that the predicate is used for an index or processed in stage 1. Many other factors must also be in place. Following are the first and second points that determine if a predicate is stage 1:

- The syntax of the predicate.
- The type and length of constants used in the predicate.

If the predicate, which is classified as a stage 1 predicate, is evaluated after a join operation, it is a stage 2 predicate. All indexable predicates are stage 1, but not all stage 1 predicates are indexable.

You can use stage 1 indexable predicates as predicates to match the columns of an index.

Example of a Stage 1 Predicate:

col op value

col

Indicates a column of a table.

op

Indicates an operator such as =, >, <, >=, <=, and so on.

Value

Indicates an expression that does not contain a column from the table (a non-column expression).

Predicates that contain BETWEEN, IN (for a list of values), and LIKE (without a leading search character) can also be stage 1 indexable. When an index is used, the stage 1 indexable predicate provides the best filtering. Because the stage 1 indexable predicate can actually limit the range of data accessed from the index. Try to use stage 1 matching predicates whenever possible. Assuming that an index exists on the EMPNO column of the EMP table, the predicate in the following query is a stage 1 indexable predicate:

```
SELECT LASTNAME, FIRSTNME
FROM   EMP
WHERE  EMPNO = '000010'
```

Other Stage 1

Just because a predicate is stage 1 does not mean that it can use an index. Some stage 1 predicates are not available for index access. These predicates (again, the best reference is the chart in the IBM manuals) are generally of the form *col NOT op value*, where *col* is a column of a table, *op* is an operator, and *value* represents a non-column expression, host variable, or value. Predicates containing NOT BETWEEN, NOT IN (for a list of values), NOT LIKE (without a leading search character), or LIKE (with a leading search character) can also be stage 1 indexable. Although non-indexable stage 1 predicates cannot limit the range of data that is read from an index, they are available as index screening predicates. The following is an example of a non-indexable stage 1 predicate:

```
SELECT LASTNAME, FIRSTNME
FROM   EMP
WHERE  EMPNO <> '000010'
```

Stage 2

DB2 manuals provide a complete description of when a predicate can be stage 2 versus stage 1. Generally, stage 2 occurs after data accesses and performs such actions as sorting and evaluation of functions and expressions. Stage 2 predicates cannot take advantage of indexes to limit the data access and are more expensive than stage 1 predicates because they are evaluated later in the processing.

Stage 2 predicates generally contain column expressions, correlated subqueries, CASE expressions, and so on. A predicate can also appear to be stage 1, but it can be processed as stage 2. For example, any predicate that is processed after a join operation is stage 2. Although DB2 promotes mismatched data types to stage 1 through casting (as of version DB2 V8), some predicates with mismatched data types are stage 2. One example is a range predicate comparing a character column to a character value that exceeds the length of the column. The following are examples of stage 2 predicates (EMPNO is a character column of fixed length 6):

```
SELECT LASTNAME, FIRSTNME
FROM   EMP
WHERE  EMPNO > '00000010'
```

```
SELECT LASTNAME, FIRSTNME
FROM   EMP
WHERE  SUBSTR(LASTNAME,1,1) = 'B'
```

Stage 3

A *stage 3 predicate* is a fictitious predicate that is made up to describe filtering that occurs in the application program after retrieving data from DB2. Avoid stage 3 predicates.

Performance is best served when all filtering is done in the data server and not in the application code. Imagine a COBOL program has read the EMP table. After reading the EMP table, the following statement is executed:

```
IF (BONUS + COMM) <  (SALARY*.10) THEN
        CONTINUE
ELSE
     PERFORM PROCESS-ROW
END-IF.
```

The previous SQL is an example of a stage 3 predicate. Data that is retrieved from the table is not used unless the condition is false. A stage 2 predicate always outperforms a stage 3 predicate. The following is an example of a stage 2 predicate that performs the same function as the previous stage 3 predicate:

```
SELECT EMPNO
FROM   EMP
WHERE  BONUS + COMM >= SALARY * 0.1
```

Combining Predicates

When simple predicates are connected by an OR condition, the resulting compound predicate is evaluated at the higher stage of the two simple predicates. The following example contains two simple predicates that are combined by an OR. The first predicate is stage 1 indexable, and the second is non-indexable stage 1. The result is that the entire compound predicate is stage 1 and not indexable:

```
SELECT  EMPNO
FROM    EMP
WHERE   WORKDEPT = 'C01' OR SEX <> 'M'
```

In the next example, the first simple predicate is stage 1 indexable, but the second (connected again by an OR) is stage 2. Thus, the entire compound predicate is stage 2:

```
SELECT  EMPNO
FROM    EMP
WHERE   WORKDEPT = 'C01' OR SUBSTR(LASTNAME,1,1) = 'L'
```

Boolean Term Predicates

A *Boolean term predicate* is a simple or compound predicate, that when evaluated false for the row, makes the entire WHERE clause evaluate to false. This is important because only Boolean term predicates are considered for single index access. At best, non-Boolean term predicates can at best be considered for multi-index access. The following example contains Boolean term predicates:

```
WHERE LASTNAME = 'HAAS' AND MIDINIT = 'T'
```

If the predicate on the LASTNAME column evaluates as false, the entire WHERE clause is false. The same is true for the predicate on the MIDINIT column. DB2 can take advantage of this because it can use an index (if available) on the LASTNAME column or the MIDINIT column. This is opposed to the following WHERE clause:

```
WHERE LASTNAME = 'HAAS' OR MIDINIT = 'T'
```

In this case, if the predicate on LASTNAME or MIDINIT evaluates as false, the rest of the WHERE clause must be evaluated. These predicates are non-Boolean term.

You can modify predicates in the WHERE clause to take advantage of this to improve index access. The following query contains non-Boolean term predicates:

```
WHERE LASTNAME > 'HAAS'
OR      (LASTNAME = 'HAAS' AND MIDINIT > 'T')
```

A redundant predicate can be added that makes the WHERE clause functionally equivalent. This Boolean term predicate can make better use of an index on the LASTNAME column:

```
WHERE LASTNAME >= 'HAAS'
AND (LASTNAME > 'HAAS'
     OR (LASTNAME = 'HAAS' AND MIDINIT > 'T'))
```

Predicate Transitive Closure

DB2 can add redundant predicates if it determines that the predicate can be applied by transitive closure. Remember the rule of transitive closure: If a=b and b=c, then a=c. DB2 can take advantage of this closure to introduce redundant predicates. For example, in a join between two tables such as the following:

```
SELECT *
FROM   T1 INNER JOIN T2
ON        T1.A = T2.B
WHERE  T1.A = 1
```

DB2 generates a redundant predicate on T2.B = 1. This predicate gives DB2 more choices for filtering and table access sequence. You can add your own redundant predicates; however, DB2 does not consider them when it applies predicate transitive closure, and so they are redundant.

Predicates of the form *col op value*, where the operation is an equals or range predicate, are available for predicate transitive closure. BETWEEN predicates are also available for this type of closure. However, predicates that contain a LIKE, an IN, or subqueries are not available for transitive closure, and so it may benefit you to code those predicates redundantly if you believe they can provide a benefit.

Coding Efficient SQL

The following views are popular in the database sector:

- The most efficient SQL statement is the SQL statement that is never executed.
- The most efficient SQL statement is the one that is never written.

Some truth exists in both views. You need to perform two tasks when you place SQL statements into your applications.

- Minimize the number of times you go to the data server.
- Go to the data server in the most efficient way.

Avoid Unnecessary Processing

Ask yourself if an SQL statement is necessary. The biggest performance issue is the amount of SQL issued. The main reason for this is typically a generic development or an object-oriented design. These types of designs run contrary to performance, but speed development and create flexibility and adaptability to business requirements. The price that you pay for this is performance.

Consider the following:

- Is the SQL statement needed?
- Do you need to run the statement again?
- Does the statement have a DISTINCT, and are duplicates possible?
- Does the statement have an ORDER BY, and is the order important?
- Are you repeatedly accessing code tables, and can the codes be cached within the program?

Coding SQL for Performance

These basic guidelines for coding SQL statements provide the best performance:

- Retrieve the fewest number of rows.

  Only rows of data that are needed for the process should be returned to the application. Do not locate data filtering statements in a program. DB2 should handle all row filtering. When a process must be performed on the data, decide where to do the process — in the program or in DB2 — then leave it in DB2. DB2 is evolving into the "ultimate" server, and a growing number of applications are distributed across the network and use DB2 as the database. A situation may occur when all data must be brought to the application, filtered, transformed, and processed. This situation is due to an inadequacy somewhere else in the design, normally a shortcoming in the physical design, missing indexes, or some other implementation problem.

- Retrieve only columns that are needed by the program.

  Retrieving extra column results in the column being moved from the page in the buffer pool to a user work area, passed to stage 2, and returned cross-memory to the work area of the program. This is an unnecessary expenditure of CPU. Code your SQL statements to return only the columns required. This code design may mean coding multiple SQL statements against the same table for different sets of columns. This effort is required for improved performance and reduction of CPU.

- Reduce the number of SQL statements.

  Each SQL statement is a programmatic call to the DB2 subsystem that incurs fixed overhead for each call. Careful evaluation of program processes can reveal situations in which unnecessary SQL statements are issued. This is especially true for programmatic joins. Separate application programs that retrieve data from related tables can result in extra SQL statements issued. Code SQL joins instead of programmatic joins. When comparing the programmatic join of two tables in a COBOL program and the equivalent SQL join, the SQL join consumed 30 percent less CPU.

- Use stage 1 predicates.

  Familiarize yourself with stage 1 indexable, stage 1, and stage 2 predicates. Try to use stage 1 predicates for filtering whenever possible. Determine if you can convert your stage 2 predicates to stage 1, or stage 1 non-indexable to indexable.

- Never use generic SQL statements.

  Generic SQL equals generic performance. Generic SQL generally has logic that only retrieves specific data or data relationships and leaves the entire business rule processing in the application. Common modules, SQL that is used to retrieve current or historical data, and modules that read all data into a copy area only to have the caller use a few fields are all examples of generic SQL usage. Generic SQL and generic I/O layers do not belong in high-performing systems.

- Avoid unnecessary sorts.

  Avoiding unnecessary sorting is a requirement in any application but more so in any high-performance environment, especially when a database is involved. Generally, if ordering is always necessary, indexes should support the ordering. Sorting can be caused by GROUP BY, ORDER BY, DISTINCT, UNION, INTERSECT, EXCEPT, and join processing. If ordering is required for a join process, the ordering generally is a clear indication that an index is missing. If ordering is necessary after a join process, hopefully the result set is small. The worst-case scenario is when a sort is performed as the result of the SQL in a cursor, and the application processes only a subset of the data. In this case, the sort overhead must be removed. If the SQL has to sort 100 rows and only 10 are processed by the application it may be better to sort in the application, or you can create an index to help avoid the sort.

- Only sort necessary columns.

  When DB2 sorts data, the columns that are used to determine the sort order actually appear twice in the sort rows. Make sure that you specify the sort only on necessary columns. For example, any column specified in an equals predicate in the query does not need to be in the ORDER BY clause.

- Use the ON clause for all join predicates.

  By using *explicit* join syntax instead of *implicit* join syntax, you make a statement easier to read, easier to convert between inner and outer joins, and harder to forget to code a join predicate.

- Avoid UNIONs (not necessarily UNION ALL).

  To avoid duplication, replace the UNION with a UNION ALL. Otherwise, see if it is possible to produce the same result with an outer join. If all the subqueries of the UNION are against the same table, try using a CASE expression and only one pass through the data.

- Use joins instead of subqueries.

  Joins can outperform subqueries for existence checking if there are good matching indexes for both tables that are involved, and if the join does not introduce any duplicate rows in the result. DB2 can take advantage of predicate transitive closure and also pick the best table access sequence. These two outcomes are not possible with subqueries.

- Code the most selective predicates first.

  DB2 processes the predicates in a query in the following order:

  - Indexable predicates are applied in the order of the columns of the index.
  - Other stage 1 predicates are applied.
  - Stage 2 predicates are applied.

  In each of the stages, DB2 processes the predicates in the following order:

  - All equals predicates (including single IN list and BETWEEN with only one value).
  - All range predicates and predicates of the form IS NOT NULL.
  - All other predicate types.

  In each grouping, DB2 processes the predicates in the order they are coded in the SQL statement. Write all SQL queries to evaluate the most restrictive predicates first to filter unnecessary rows earlier. This action reduces processing cost at a later stage. This suggestion includes subqueries (in the grouping of correlated and non-correlated).

- Use the proper method for existence checking.

  For existence checking using a subquery, an EXISTS predicate generally outperforms an IN predicate. For general existence checking, code a singleton SELECT statement that contains a FETCH FIRST 1 ROW ONLY clause.

- Avoid unnecessary materialization.

  When you run a transaction that processes little or no data, use correlated references to avoid materializing large intermediate result sets. Correlated references encourage nested loop join and index access for transactions.

Promoting Predicates

Code SQL predicates as efficiently as possible. When you are writing predicates, stage 1 indexable predicates whenever possible. If possible, promote stage 1 non-indexable predicates or stage 2 predicates to a more efficient stage.

If you have a stage 1 non-indexable predicate, promote it to stage 1 indexable. For example:

```
WHERE END_DATE_COL <> '9999-12-31'
```

If you use the *end of the DB2 world* as an indicator of data that is still active, change the predicate to something that is indexable. For example:

```
WHERE END_DATE_COL < '9999-12-31'
```

Promote stage 2 predicates to stage 1 or even stage 1 indexable. For example:

```
WHERE BIRTHDATE + 30 YEARS < CURRENT DATE
```

The previous predicate applies date arithmetic to a column. This is a column expression, which makes the predicate a stage 2 predicate. By moving the arithmetic to the right side of the inequality, you make the predicate stage 1 indexable. For example:

```
WHERE BIRTHDATE < CURRENT DATE – 30 YEARS
```

These examples can be applied as general rules for promoting predicates. Using DB2 9, it is possible to create an index on an expression. An index on an expression can be considered for improved performance of column expressions when it is not possible to eliminate the column expression in the query.

Functions, Expressions, and Performance

DB2 has many scalar functions and other expressions that let you manipulate data. This functionality contributes to the fact that the SQL language is indeed a programming language as much as a data access language. However, the processing of functions and expressions in DB2, if not for the filtering of data, but instead for pure data manipulation, are generally more expensive than the equivalent application program process. Even the simple functions such as concatenation are executed for every row processed:

```
SELECT FIRSTNME CONCAT LASTNAME
FROM   EMP
```

If you need the ultimate in ease of coding, time to delivery, portability, and flexibility, code expressions and functions using this practice in your SQL statements. If you need the ultimate in performance, perform the manipulation of data in your application program.

CASE expressions can be expensive, but CASE expressions use "early out" logic when processing. Examine the following example of a CASE expression:

```
CASE WHEN C1 = 'A'
       OR C1 = 'K'
       OR C1 = 'T'
       OR C1 = 'Z'
THEN 1 ELSE NULL END
```

If most of the time the value of the C1 column is a blank, the following functionally equivalent CASE expression consumes less CPU:

```
CASE WHEN C1 <> ' '
     AND (C1 = 'A'
       OR C1 = 'K'
       OR C1 = 'T'
       OR C1 = 'Z')
THEN 1 ELSE NULL END
```

DB2 takes the early out from the CASE expression for the first NOT TRUE of an AND, or the first TRUE of an OR. In addition to testing for the previous blank value, all the other values should be tested with the most frequently occurring values first.

Advanced SQL and Performance

Advanced SQL queries can be a performance advantage or performance disadvantage. Advanced SQL possibilities are endless, and it is impossible to document all the various situations and the performance impacts. Complex SQL is always a performance improvement over an application program process when the complex SQL is filtering or aggregating data.

Correlation

In general, correlation encourages nested loop join and index access. This functionality is good for transaction queries that process small out quantities of data but not for report queries that process large quantities of data. The following query works well when processing large quantities of data:

```
SELECT  TAB1.EMPNO, TAB1.SALARY,
                TAB2.AVGSAL,TAB2.HDCOUNT
FROM
        (SELECT EMPNO, SALARY, WORKDEPT
        FROM            EMP
        WHERE   JOB='SALESREP') AS TAB1
LEFT OUTER JOIN
        (SELECT AVG(SALARY) AS AVGSAL, COUNT(*) AS HDCOUNT,
                        WORKDEPT
        FROM            EMP
        GROUP   BY WORKDEPT) AS TAB2
ON TAB1.WORKDEPT = TAB2.WORKDEPT
```

The aforementioned query is the most efficient way to retrieve the data if there is one sales representative per department, or if most of the employees are sales representatives. The employee table is read and materialized in the nested table expression called *TAB2*. It is likely that the merge scan join method will be used to join the materialized TAB2 to the first table expression called TAB1.

The aforementioned query is not the most efficient if the employee table is extremely large with few sales representatives, or with sales representatives in one or a few departments. The entire employee table still must be read in TAB2, but most of the results of the nested table expression are returned in the query. In this case, the following query is more efficient:

```
SELECT  TAB1.EMPNO, TAB1.SALARY,
                TAB2.AVGSAL,TAB2.HDCOUNT
FROM            EMP TAB1
,TABLE(SELECT   AVG(SALARY) AS AVGSAL,
                        COUNT(*) AS HDCOUNT
          FROM    EMP
         WHERE  WORKDEPT = TAB1.WORKDEPT) AS TAB2
WHERE   TAB1.JOB = 'SALESREP'
```

This statement is functionally equivalent to the previous statement, but it operates in a different way. In this query, the employee table referenced as TAB1 is read first. The nested table expression is executed repeatedly in a nested loop join for each row that qualifies. An index on the WORKDEPT column is necessary.

Merge Versus Materialization for Views and Nested Table Expressions

When you have a reference to a nested table expression or view in your SQL statement, DB2 may merge that nested table expression or view with the referencing statement. If DB2 cannot merge the statement, then DB2 materializes the view or nested table expression into a work file. DB2 then applies the referencing statement to that intermediate result. IBM states that merge is more efficient than materialization. In general, that statement is correct. However, materialization is more efficient if your complex queries have the following combined conditions:

- Nested table expressions or view references, especially multiple levels of nesting.
- Columns generated in the nested expressions or views through application of functions, user-defined functions, or other expressions.
- References to the generated columns in the outer referencing statement.

In general, DB2 materializes when some sort of aggregate processing is required inside the view or nested table expression. This means that the view or nested table expression contains aggregate functions, grouping (GROUP BY), or DISTINCT. If materialization is not required, the merge process occurs. The following query is an example:

```
SELECT MAX(CNT)
FROM   (SELECT ACCT_RTE, COUNT(*) AS CNT
        FROM   YLA.ACCT_TABLE) AS TAB1
```

DB2 materializes TAB1 in the aforementioned example. Examine the following query:

```
SELECT SUM(CASE WHEN COL1=1 THEN 1 END) AS ACCT_CURR
      ,SUM(CASE WHEN COL1>1 THEN 1 END) AS ACCT_LATE
FROM
(SELECT CASE ACCT_RTE WHEN 'AA' THEN 1 WHEN 'BB'
                THEN 2 WHEN 'CC' THEN '2' WHEN 'DD' THEN 2
                WHEN 'EE' THEN 3 END AS COL1
FROM            YLA.ACCT_TABLE) AS TAB1
```

In this query, DISTINCT, GROUP BY, and aggregate functions do not exist. DB2 merges inner table expression with the outer referencing statement. There are two references to COL1 in the outer referencing statement. The CASE expression in the nested table expression is calculated twice during query execution. The merged statement resembles the following example:

```
SELECT SUM(CASE WHEN
          CASE ACCT_RTE WHEN 'AA' THEN 1 WHEN 'BB'
                THEN 2 WHEN 'CC' THEN '2' WHEN 'DD' THEN 2
                WHEN 'EE' THEN 3 END =1 THEN 1 END) AS ACCT_CURR
      ,SUM(CASE WHEN
          CASE ACCT_RTE WHEN 'AA' THEN 1 WHEN 'BB'
                THEN 2 WHEN 'CC' THEN '2' WHEN 'DD' THEN 2
                WHEN 'EE' THEN 3 END >1 THEN 1 END) AS ACCT_LATE
```

```
FROM            YLA.ACCT_TABLE
```

For this particular query, the merge is probably more efficient than materialization. However, if you have multiple levels of nesting and many references to generated columns, merge can be less efficient than materialization. In these specific cases, introduce a non-deterministic function into the view or nested table expression to force materialization. Use the RAND() function.

UNION in a View or Nested Table Expression

You can place a UNION or UNION ALL into a view or nested table expression. This practice allows for complex SQL processing, but also lets you create logically partitioned tables. This means that you can store data in multiple tables and reference them together as one table in a view. This practice is useful for quickly rolling through yearly tables or for creating optional table scenarios with little maintenance overhead.

Although each SQL statement in a union in view or table expression results in an individual query block, and SQL statements written against the view are distributed to each query block, DB2 uses a technique to prune (eliminate query blocks for efficiency. DB2 can, depending upon the query, prune query blocks at statement compile time or during statement execution. Consider the following account history view:

```
CREATE VIEW V_ACCOUNT_HISTORY
(ACCOUNT_ID, AMOUNT) AS
SELECT ACCOUNT_ID, AMOUNT
FROM   HIST1
WHERE  ACCOUNT_ID BETWEEN 1 AND 100000000
UNION ALL
SELECT ACCOUNT_ID, AMOUNT
FROM   HIST1
WHERE  ACCOUNT_ID BETWEEN 100000001 AND 200000000
```

Then consider the following query:

```
SELECT *
FROM   V_ACCOUNT_HISTORY
WHERE  ACCOUNT_ID = 12000000
```

The predicate of this query contains the literal value 12000000, and this predicate is distributed to the query blocks generated. However, DB2 compares the distributed predicate against the predicates coded in the UNION inside your view, looking for redundancies. In any situations where the distributed predicate renders a particular query block unnecessary, DB2 prunes that query block from the access path. Examine the following example:

```
. . .
WHERE  ACCOUNT_ID BETWEEN 1 AND 100000000
AND    ACCOUNT_ID = 12000000
. . .
WHERE  ACCOUNT_ID BETWEEN 100000001 AND 200000000
AND    ACCOUNT_ID = 12000000
```

DB2 prunes the query blocks generated at statement compile time, based upon the literal value supplied in the predicate. In the previous example, two query blocks are generated, but one of them is pruned when the statement is compiled. DB2 compares the literal predicate supplied in the query against the view that contains the predicates. Any unnecessary query blocks are pruned. Because one of the resulting combined predicates is impossible, DB2 eliminates that query block. Only one underlying table is accessed.

Query block pruning can happen at statement compile (bind) time or at run time if a host variable or parameter marker is supplied for a redundant predicate. Replace the literal with a host variable in the previous example. The new query resembles the following example:

```
SELECT *
FROM   V_ACCOUNT_HISTORY
WHERE  ACCOUNT_ID = :H1
```

If you embed this statement in a program, and it is bound into a plan or package, two query blocks are generated. This outcome occurs because DB2 does not know the value of the host variable in advance and distributes the predicate among both generated query blocks. However, at run time, DB2 examines the supplied host variable value, and dynamically prunes the query blocks appropriately. If the value 12000000 is supplied for the host variable value, one of the two query blocks is pruned at run time, and only one underlying table is accessed. This complicated process does not always succeed. Test the process by stopping one of the tables and running a query with a host variable that prunes the query block on that table. If the statement is successful, the run-time query block pruning works properly.

You can prune query blocks on literals and host variables. However, you cannot prune query blocks on joined columns. In certain situations with many query blocks (UNIONs), many rows of data, and many index levels for the inner view or table expression of a join, use programmatic joins in situations in which the query can benefit from run-time query block pruning using the joining column. This is an extremely specific recommendation. There are limits to UNION in view (or table) expressions. Test to see if you get bind time or run-time query block pruning. In some cases this outcome does not occur, and APARs are available that address the problems, but they are not comprehensive; therefore testing is important.

You can influence proper use of run-time query block pruning by encouraging distribution of joins and predicates into the UNION in view. Accomplish this task by reducing the number of tables in the UNION, or by repeating host variables in predicates instead of, or in addition to using correlation. Examine the following query:

```
SELECT
        {history data columns}
FROM V_ACCOUNT_HISTORY HIST
WHERE HIST.ACCT_ID = :acctID
AND HIST.HIST_EFF_DTE = '2005-08-01'
AND   HIST.UPD_TSP =
```

```
       (SELECT MAX(UPD_TSP)
        FROM   V_ACCOUNT_HISTORY HIST2
        WHERE  HIST2.ACCT_ID = :acctID
        AND    HIST2.HIST_EFF_DTE = '2005-08-01')
WITH UR;
```

The predicate on ACCT_ID in the subquery could be correlated to the outer query, but it is not. The same applies to the predicate on the HIST_EFF_DTE predicate in the subquery. The reason for repeating the host variable and value references is to take advantage of the runtime pruning. Correlated predicates are not pruned in this case.

Recommendations for Distributed Dynamic

DB2 for z/OS is the ultimate data server, and there is a proliferation of dynamic distributed SQL hitting the system, through JDBC, ODBC, or DB2 CLI. Ensuring this SQL is always performing the best can sometimes be a challenge. We offer the following general recommendations to improve distributed dynamic SQL performance:

- Turn on the dynamic statement cache.
- Use a fixed application level authorization ID. This practice lets accounting data be grouped by the ID, and applications can be identified by their ID. It also makes better use of the dynamic statement cache in that statements are reused by authorization ID.
- Parameter markers are almost always better than literal values. You need an exact statement match to reuse the dynamic statement cache. Literal values are helpful only when a skewed distribution of data exists, and that skewed distribution is properly reflected through frequency distribution or histogram statistics in the system catalog.
- Use the EXPLAIN STMTCACHE ALL command to expose all statements in the dynamic statement cache. The output from this command goes into a table that contains runtime performance information. In addition, you can explain individual statements in the dynamic statement cache after they have been identified.
- Use connection pooling.
- Consolidate calls into stored procedures. This practice works only if multiple statements and program logic, representing a transaction, are consolidated in a single stored procedure.
- Use the DB2 CLI/ODBC/JDBC static profiling feature. This CLI feature turns dynamic statements into static statements. Static profiling lets you capture dynamic SQL statements on the client and then bind the statements on the server into a package. After the statements are bound, the initialization file can be modified, instructing the interface software to use the static statement whenever the equivalent dynamic statement is issued from the program.

If you are moving a local batch process to a remote server, you will lose some of the efficiencies that go along with the RELEASE(DEALLOCATE) bind parameter, in particular sequential detection and index lookaside.

Influencing the Optimizer

The DB2 optimizer is good at picking the correct access path, when it is given the proper information. In situations where it does not pick the optimal access path, it typically does not have enough information. This is sometimes the case when you use parameter markers or host variables in a statement.

Proper Statistics

DB2 uses a cost-based optimizer, and that optimizer needs accurate statistical information about your data. Collecting the proper statistics is necessary for good performance. With each new version of DB2, the optimizer better utilizes catalog statistics. This point means that with each new version, DB2 is more dependent on catalog statistics. You should have statistics on every column referenced in every WHERE clause. If you are using parameter markers and host variables, you need at least cardinality statistics. If you have skewed data or you are using literal values in your SQL statements, you need frequency distribution or histogram statistics. If you suspect columns are correlated, gather column correlation statistics.

To determine if columns are correlated, run these two queries:

```
SELECT COUNT (DISTINCT CITY) AS CITYCNT *
               COUNT (DISTINCT STATE) AS STATECNT
        FROM CUSTOMER

SELECT COUNT (*) FROM
               (SELECT DISTINCT CITY, STATE
                   FROM CUSTOMER) AS FINLCNT
```

If the number from the second query is lower than the number from the first query, the columns are correlated.

You can also run GROUP BY queries against tables for columns used in predicates to count the occurrences of values in these columns. These counts indicate whether you need frequency distribution statistics or histogram statistics, and run-time reoptimization for skewed data distributions.

Run-time Reoptimization

If your query contains a predicate with an embedded literal value, DB2 knows about the input to the query. DB2 can take advantage of frequency distribution or histogram statistics, if available. The result is a much improved filter factor and better access path decisions by the optimizer. However, DB2 may not know about your input value:

```
SELECT *
FROM    EMP
WHERE MIDINIT > :H1
```

In this case, if the values for MIDINT are highly skewed, DB2 could make an inaccurate estimate of the filter factor for some input values.

DB2 can employ the run-time reoptimization to help your queries. For static SQL, the option of REOPT(ALWAYS) is available. This bind option instructs DB2 to recalculate access paths at runtime using the host variable parameters. This practice can improve execution time for large queries. However, if many queries are in the package, they are all reoptimized. This practice can affect the statement execution time for these queries. When you use REOPT(ALWAYS), consider separating the query that can benefit in its own package.

Dynamic SQL statements have three options:

REOPT(ALWAYS)

Reoptimizes a dynamic statement with parameter markers based upon the values that are provided on every execution.

REOPT(ONCE)

Reoptimizes a dynamic statement the first time it is executed based on the values that are provided for parameter markers. The access path is reused until the statement is removed from the dynamic statement cache and needs to be prepared again. Use this reoptimization option with care because the first execution should have good representative values.

REOPT(AUTO)

Tracks how the values for the parameter markers change on every execution and reoptimizes the query that is based upon those values if it determines that the values have changed significantly.

A system parameter called *REOPTEXT* (DB2 9) enables REOPT(AUTO)-like behavior, subsystem wide, for any dynamic SQL queries (without NONE, ALWAYS, or ONCE already specified) that contain parameter markers when changes are detected in the values that could influence the access path.

OPTIMIZE FOR Clause

The optimizer does not know how much data you are going to fetch from the query. DB2 uses the system catalog statistics to estimate the number of rows that are returned if the entire query is processed by the application. However, if you are not going to read all the rows of the result set, use the OPTIMIZE FOR *n* ROWS clause, where *n* is the number of rows you intend to fetch.

The OPTIMIZE FOR clause lets you tell DB2 how many rows you intend to process. DB2 can then make access path decisions to determine the most efficient way to access the data for that quantity. The use of this clause discourages such actions as the list prefetch, sequential prefetch, and multi-index access. It encourages index usage to avoid a sort, and a join method of the nested loop join. A value of 1 is the strongest influence on these factors.

Insert the number of rows you intend to fetch in the OPTIMIZE FOR clause. Incorrectly representing the number of rows you intend to fetch can result in a poorly performing query.

Encouraging Index Access and Table Join

You can use certain techniques to encourage DB2 to choose a certain index or a table access sequence in a query that accesses multiple tables.

When DB2 joins tables in an inner join, it tries to select the table that qualifies the fewest rows first in the join sequence. If DB2 chooses the incorrect table first (maybe due to statistics or host variables), change the table access sequence by using one or more of these techniques:

- Enable predicates on the table that you want to be first. By increasing potential match cols on this table, DB2 can select an index for more efficient access and can change the table access sequence.
- Disable predicates on the table that you do not want accessed first. Predicate disablers are documented in the IBM *DB2 Performance Monitoring and Tuning Guide*. We do not recommend using predicate disablers.
- Force materialization of the table that you want accessed first by placing the table in a nested table expression with a DISTINCT or GROUP BY. This technique changes the join type and the join sequence. This technique is especially useful when a nested loop join is randomly accessing the inner table.
- Convert joins to subqueries. When you code subqueries, you tell DB2 the table access sequence. Non-correlated subqueries are accessed first, the outer query is executed, and then any correlated subqueries are executed. This technique is effective only if the table moved from a join to a subquery does not have to return data.

- Convert a joined table to a correlated nested table expression. This technique forces another table to be accessed first. Data for the correlated reference is required before the table in the correlated nested table expression being accessed.
- Convert an inner join to a left join. By coding a left join, you dictate the table join sequence to DB2, and that the right table filters no data.
- Add a CAST function to the join predicate for the table you want accessed first. By placing this function on that column, you encourage DB2 to access that table first to avoid a stage 2 predicate against the second table.
- Code an ORDER BY clause on the columns of the index of the table that you want to be accessed first in the join sequence. This technique may influence DB2 to use that index to avoid the sort, and access that table first.
- Change the order of the tables in the FROM clause. Try converting the implicit join syntax to explicit join syntax and the reverse.
- Code a predicate on the table you want accessed first as a non-transitive closure predicate. An example of this is a non-correlated subquery against the SYSDUMMY1 table that returns a single value rather than an equals predicate on a host variable or literal value. The subquery is not eligible for transitive closure, and DB2 does not generate the predicate redundantly against the other table, and has less encouragement to choose that table first.

If DB2 chooses one index over another and you disagree, try one of these techniques to influence index selection:

- Code an ORDER BY clause on the leading columns of the index you want DB2 to choose. This technique may encourage DB2 to choose that index to avoid a sort.
- Add columns to the index to make the access index-only.
- Modify the query or the index to increase the index match cols.
- Disable predicates that match other indexes. We do not recommend predicate disablers.
- Use the OPTIMIZE FOR clause.

Note: For details about predicate disablers, see the IBM documentation.

Chapter 4: Designing Tables and Indexes for Performance

The best way to perform logical database modeling is to use strong guidelines that are developed by an expert in relational data modeling.

You could also use one of the many relational database modeling tools that is supplied by vendors, but it is important to remember that using a tool to migrate your logical model into a physical model does not mean that the physical model is the most optimal for performance. It is acceptable to modify the physical design to improve performance provided the logical model is not compromised.

DB2 objects must be designed for availability, ease of maintenance, and overall performance, as well as for business requirements.

Table Space Performance Recommendations

This section includes general recommendations for table space performance.

Use Segmented or Universal Table Spaces

DB2 is eliminating simple tablespaces support. Although you can create them in DB2 V7 and DB2 V8, they cannot be created in DB2 9 (although you can still use previously created simple tablespaces). Whatever version you are using, we recommend that you use a segmented table space instead of a simple tablespace.

Segmented Table Spaces

You gain several advantages by using segmented table spaces. Because the pages in a segment contain only rows from one table, no locking interference with other tables exists. In simple table spaces, rows are intermixed on pages, and if one table page is locked, it can inadvertently lock a row of another table just because it is on the same page. This is not an issue if you have only one table per table space; however, you can obtain following benefits from having a segmented table space for one table:

- If a table scan is performed, the segments belonging to the table being scanned are the only ones accessed; empty pages are not scanned.
- If a mass delete or a DROP table occurs, segment pages are available for immediate reuse, and it is not necessary to run a REORG utility.
- Mass deletes are much faster for segmented table spaces, and they produce less logging, provided the table has not been defined with DATA CAPTURE CHANGES.

- The COPY utility does not have to copy empty pages that are left by a mass delete.
- When inserting records, some read operations can be avoided by using the more comprehensive space map of the segmented table space.

Universal Table Spaces

DB2 9 introduces a new type of table space that is called a *universal table space*. A universal table space is segmented and partitioned. Two types of universal table spaces are available:

- The partition-by-growth table space
- The range-partitioned table space

A universal table space offers the following benefits:

- Better space management relative to varying-length rows. A segmented space map page provides more information about free space than a regular partitioned space map page.
- Improved mass delete performance. A mass delete in a segmented table space organization tends to be faster than in table spaces that are organized differently. In addition, you can immediately reuse all or most of the segments of a table.

Partition-by-Data-Growth Table Space

Before DB2 9, partitioned tables required key ranges to determine the target partition for row placement. Partitioned tables provide more granular locking and parallel operations by spreading the data over more data sets. In DB2 9, you can partition according to data growth. This practice enables segmented tables to be partitioned as they grow, without the need for key ranges. As a result, segmented tables benefit from increased table space limits and SQL and utility parallelism that were formerly available only to partitioned tables. Also, with a partition-by-data-growth table space, you do not need to reorganize a table space to change the limit keys.

You can implement partition-by-growth table space organization in one of the following ways:

- Use the new MAXPARTITIONS clause on the CREATE TABLESPACE statement to specify the maximum number of partitions that the partition-by-growth table space can accommodate. The value that you specify in the MAXPARTITIONS clause is used to protect against run away applications that perform an insert in an infinite loop.
- Use the MAXPARTITIONS clause on the ALTER TABLESPACE statement to alter the maximum number of partitions to which an existing partition-by-growth table space can grow. This ALTER TABLESPACE operation acts as an immediate ALTER.

Range-Partitioned Table Space

A *range-partitioned table space* is a type of universal table space that is based on partitioning ranges and that contains a single table. The new range-partitioned table space does not replace the existing partitioned table space, and operations that are supported on a regular partitioned or segmented table space are supported on a range-partitioned table space.

To create a range-partitioned table space, specify the SEGSIZE and NUMPARTS keywords on the CREATE TABLESPACE statement.

With a range-partitioned table space, you can also control the partition size, choose from a wide array of indexing options, and take advantage of partition-level operations and parallelism capabilities. Because the range-partitioned table space is also a segmented table space, you can run table scans at the segment level. As a result, you can immediately reuse all or most of the segments of a table after the table has been dropped or a mass delete has been performed.

Range-partitioned universal table spaces follow the same partitioning rules as partitioned table spaces in general. That is, you can add, rebalance, and rotate partitions. The maximum number of partitions possible for both range-partitioned and partition-by-growth universal table spaces (as for partitioned table spaces) is controlled by the DSSIZE and page size.

Clustering and Partitioning

The clustering index is not always the primary key. It is generally 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 requirements, but partitioning is an important requirement and can be the most critical requirement, especially as tables get extremely large. If you do not specify an explicit clustering index, DB2 clusters 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.

The basic rule to clustering is that if your application will have a certain sequential access pattern or a regular batch process, you should cluster the data according to that input sequence.

Clustering and partitioning can be independent, and a log of options is available for organizing your data as follows:

- In a single dimension (clustering and partitioning are based on the same key)
- Dual dimensions (clustering inside each partition by a different key)
- Multiple dimensions (combining different tables with different partitioning unioned inside a view).

You should choose a partitioning strategy that is based on a concept of application-controlled parallelism, separating old and new data, grouping data by time, or grouping data by some meaningful business entity (for example, sales region or office location). Within those partitions, you can cluster the data by your most common sequential access sequence.

Note: For more information about dismissing clustering for inserts, see Append Processing for High Volume Inserts (see page 74).

For large tables, partitioning is the only way to store large amounts of data, but partitioning also has advantages for smaller tables. Consider the following:

- DB2 lets you define up to 4096 partitions of up to 64 GB each; however, total table size is limited depending on the DSSIZE specified. Non-partitioned table spaces are limited to 64 GB of data.
- You can take advantage of the ability to execute utilities on separate partitions in parallel. This practice also lets you access data in certain partitions while utilities are executing on others.
- In a data-sharing environment, you can spread partitions among several members to split workloads.
- You can also spread your data over multiple volumes and need not use the same storage group for each data set belonging to the table space. This practice also lets you place frequently accessed partitions on faster devices.

Free Space

The FREEPAGE and PCTFREE clauses can help improve the performance of updates and inserts by allowing free space to exist on table spaces. Performance improvements include improved access to the data through the better clustering of data, faster inserts, fewer row overflows, and a reduction in the number of REORGs required.

Some tradeoffs include an increase in the number of pages, fewer rows per I/O, less efficient use of buffer pools, and more pages to scan. Therefore, it is important to achieve a good balance for each individual table space and index space when deciding on free space. The balance depends on the processing requirements of each table space or index space. When inserts and updates are performed, DB2 uses the available free space, and by doing so, it can keep records in clustering sequence as much as possible. When the free space is used up, the records must be located elsewhere; when this happens, performance can begin to suffer.

Read-only tables do not require any free space, and tables with a pure insert-at-end strategy (append processing) generally do not require free space. Exceptions to this guideline would be tables with VARCHAR columns and tables using compression that are subject to updates. When DB2 attempts to maintain clustering during inserting and updating, it searches nearby for free space or free pages for the row. If DB2 does not find this space, it searches the table space for a free place to put the row before extending a segment or a dataset. Indications of this activity are as follows:

- A gradual increase in insert CPU times in your application; you can see this when you examine the accounting records.
- Increasing getpage counts
- Relocated row counts

When this activity happens, perform a REORG and reevaluate your free space allocations.

Allocations

The PRIQTY and SECQTY clauses of the CREATE TABLESPACE and ALTER TABLESPACE SQL statements specify the space that is to be allocated for the table space if the table space is managed by DB2. These settings influence the allocation by the operating system of the underlying VSAM datasets in which table space and index space data is stored.

PRIQTY

Specifies the minimum primary space allocation for a DB2-managed data set of the table space or partition. The primary space allocation is in kilobytes, and the maximum that can be specified is 64 GB. DB2 requests a data set allocation corresponding to the primary space allocation, and the operating system attempts to allocate the initial extent for the data set in one contiguous piece.

SECQTY

Specifies the minimum secondary space allocation for a DB2-managed data set of the table space or partition. DB2 requests secondary extents in a size according to the secondary allocation. However, the actual primary and secondary data set sizes depend on various settings and installation parameters.

You can specify the primary and secondary space allocations for table spaces and indexes or can let DB2 choose them. Having DB2 choose the values, especially for the secondary space quantity, increases the possibility of reaching the maximum data set size before running out of extents. In addition, the MGEXTSZ subsystem parameter influences the SECQTY allocations, and when set to YES (NO is the default), it changes the space calculation formulas to help use all of the potential space allowed in the table space before running out of extents.

You can alter the primary and secondary space allocations for a table space. The secondary space allocation takes effect immediately. However, because the primary allocation happens when the data set is created, that allocation does not take effect until a data set is added, depending on the type of table space, or until the data set is recreated through utility execution such as a REORG or LOAD REPLACE.

Column Ordering

When defining a table, you can order your columns in specific ways to achieve the following results:

- Reduce CPU consumption when reading and writing columns with variable-length data
- Reduce the amount of logging performed when updating rows

The version of DB2 you are running influences the way you or DB2 organizes your columns.

Table Space Compression

Using the COMPRESS clause of the CREATE TABLESPACE and ALTER TABLESPACE SQL statements allows for the compression of data in a table space or in a partition of a partitioned table space. In many cases, using the COMPRESS clause can significantly reduce the amount of DASD space that is required to store data. But the compression ratio that is achieved depends on the characteristics of the data.

Compression lets you place more rows on a page resulting in the following performance benefits, depending on the SQL workload and the amount of compression:

- Higher buffer pool hit ratios
- Fewer I/Os
- Fewer getpage operations
- Reduced CPU time for image copies

The processing cost when using compression is relatively low, but consider the following:

- The processor cost to decode a row using the COMPRESS clause is less than the cost to encode the same row.
- The data access path that DB2 uses affects the processor cost for data compression. In general, the relative overhead of compression is higher for table space scans and less costly for index access.

Some data does not compress well, so you should query the PAGESAVE column in SYSIBM.SYSTABLEPART to be sure that you are getting a savings (at least 50 percent is average). Data that does not compress well includes binary data, encrypted data, and repeating strings. Additionally, do not compress small tables or rows if you are concerned about concurrency issues because this configuration places more rows on a page.

Note: When you compress, the row is treated as varying length and the length may change when updates occur, resulting in potential row relocation causing high numbers in NEARINDREF and FARINDREF. This outcome indicates that you are now doing more I/O to get to your data because it has been relocated, and you will have to REORG to get it back to its original position.

Referential Constraints

Referential integrity lets you define required relationships between and within tables. The database manager maintains these relationships, which are expressed as referential constraints. These relationships require that all values of a given attribute or table column also exist in some other table column.

In general, DB2-enforced referential integrity is much more efficient than coding the equivalent logic in your application program. In addition, having the relationships enforced in a central location in the database is much more powerful than making it dependent upon application logic. You will need indexes to support the relationships that are enforced by DB2.

Referential integrity checking has a cost that is associated with it. Referential integrity is meant for checking parent/child relationships, not code checking. Better options for code checking include check constraints, or even better, to put codes in memory and check them there.

Table check constraints enforce data integrity at the table level. After a table-check constraint has been defined for a table, every UPDATE and INSERT statement involves checking the restriction or constraint. If the constraint is violated, the data record is not inserted or updated, and an SQL error is returned.

A table-check constraint can be defined when you create a table or by using the ALTER TABLE statement. The table-check constraints can help implement specific rules for the data values contained in the table by specifying the values allowed in one or more columns in every row of a table. This practice can save time for the application developer because the validation of each data value can be performed by the database and not by each of the applications accessing the database. However, check constraints should, in general, not be used for data edits in support of data entry. For this scenario, it is best to cache code values locally within the application and perform the edits local to the application. This practice avoids numerous trips to the database to enforce the constraints.

Indexing

Depending upon your application and the type of access, indexing can enhance or impair performance. If your application is a heavy reader or even a read-only application, numerous indexes can enhance performance. If your application is constantly inserting, updating, and deleting from your table, numerous indexes might prove detrimental.

If you are adding a secondary index to a table for inserts and deletes, and even updates, you are adding another random read to these statements. Consider whether your application can afford that in support of queries that may use the index.

Efficient Database Design

When designing your database, for tables that will contain a significant amount of data and considerable DML activity, aim to have only one index per table. This practice provides for an efficient database design. Each index would need to support the following:

- Insert strategy
- Primary key
- Foreign key (if a child table)
- SQL access path
- Clustering

Note the following design objectives:

- Avoid surrogate keys. Use meaningful business keys instead.
- Let the child table inherit the parent key as part of the child's primary key.
- Cluster all tables in a common sequence.
- Determine the common access paths, respect them, and try not to change them during design.
- Never entertain a *just in case* type of design mentality.

Index Design Recommendations

After you have determined the indexes that you require, design them properly for performance.

Index Compression

In DB2 9, an index can be defined with the COMPRESS YES option; COMPRESS NO is the default. Index compression can be used when you have to reduce the amount of disk space that an index consumes. We recommend index compression for applications that do sequential insert operations with few, or no, delete operations. Random inserts and deletes can adversely affect compression. We also recommend index compression for applications where the indexes are created primarily for scan operations.

A buffer pool that is used to create the index must be 8 KB, 16 KB, or 32 KB. The physical page size for the index on the disk will be 4 KB. The buffer pool size is larger than the page size because index compression saves space only on the disk. The data in the index page is expanded when read into the pool. So, index compression can possibly save you read time for sequential, and random operations (but far less likely).

Index compression can have a significant impact on the REORGs and index rebuilds, resulting in significant savings in this area; however, if you use the copy utility to back up an index, that image copy is uncompressed.

Index Free Space

Setting the PCTFREE and FREEPAGE for your indexes depends upon how much insert and delete activity is going to occur against those indexes. For indexes that have little or no inserts and deletes, you could use a small PCTFREE with no free pages.

Note: Updates that change key columns are actually inserts and deletes.

For indexes with heavy changes, consider larger amounts of free space. Adding free space may increase the number of index levels, and then increase the amount of I/O for random reads. If you do not have enough free space, you might experience an increased frequency of index page splits. When DB2 splits a page, it looks for a free page in which to place one of the split pages. If it does not find a page nearby, it searches the index for a free page. For large indexes, this search could lead to CPU and locking problems. We recommend that you set a predictive PCTFREE that anticipates growth over a period such that you do not split pages. You should then monitor the frequency of page splits to determine when to REORG the index or establish a regular REORG policy for that index.

Secondary Indexes

Two types of secondary indexes exist; non-partitioning secondary indexes and data-partitioned secondary indexes as follow:

Non-Partitioning Secondary Indexes (NPSIs)

Are used on partitioned tables. They are not the same as the clustered partitioning key, which is used to order and partition the data, but rather they are for access to the data. NPSIs can be unique or non unique. You can have only one clustered partitioning index, but you can have several NPSIs on a table, if necessary. NPSIs can be broken into multiple pieces (data sets) by using the PIECESIZE clause on the CREATE INDEX statement. Pieces can vary in size from 254 KB to 64 GB—the optimum size depends on the amount of data you have and how many pieces you want to manage. If you have several pieces, you can achieve more parallelism on processes, such as heavy INSERT batch jobs, by alleviating the bottlenecks that are caused by contention on a single data set. In DB2 V8 and later, the NPSI can be the clustering index.

Note: NPSIs are good for fast read access because there is a single index b-tree structure. NPSIs can become extremely large and can cause maintenance and availability issues.

Data-Partitioned Secondary Indexes (DPSIs)

Provides many advantages over the traditional NPSIs for secondary indexes on a partitioned table space in terms of availability and performance.

The partitioning scheme of the DPSI is the same as the table space partitions, and the index keys in 'x' index partition matches those in 'x' partition of the table space. Some of the benefits that this provides include the following:

- Clustering by a secondary index
- Ability to easily rotate partitions
- Efficient utility processing on secondary indexes (no BUILD-2 phase)
- Allow for reducing overhead in data sharing (affinity routing)

Drawbacks of DPSIs

While DPSIs provide gains in furthering partition independence, some queries may not perform as well. If the query has predicates that reference columns in a single partition and are therefore restricted to a single partition of the DPSI, the query benefits from this new organization. To accomplish this task, design the queries to allow for partition pruning through the predicates. This practice means that the leading column of the partitioning key has to be supplied in the query for DB2 to prune partitions from the query access path. However, if the predicate references only columns in the DPSI, it may not perform well because it may need to probe several partitions of the index. Other limitations of DPSIs include that they cannot be unique (some exceptions in DB2 9), and they may not be the best candidates for ORDER BYs.

Rebuild or Recover?

In DB2 V8 and later, you can define an index as COPY YES. This definition means, as with a table space, you can use the COPY and RECOVER utilities to back up and recover these indexes. This practice may be especially useful for large indexes; however, large NPSIs cannot be copied in pieces. You will need large data sets to hold the backup. This requirement could mean large quantities of tapes or reaching the 59 volume limit for a dataset on DASD.

REBUILD requires large quantities of temporary DASD to support sorts, as well as more CPU than a RECOVER. Carefully consider whether your strategy for an index should be backup and recover or should be rebuild.

Special Tables Used for Performance

Some table designs can dramatically help boost application performance.

Materialized Query Tables

Decision support queries are often difficult and expensive. They typically operate over a large amount of data and may have to scan or process terabytes of data and possibly perform multiple joins and complex aggregations. With these types of queries, traditional optimization and performance is not always optimal.

In DB2 V8 and later, one solution is the use of Materialized Query Tables (MQTs). This practice lets you precompute whole or parts of each query and then use computed results to answer future queries. MQTs provide the means to save the results of previous queries and then reuse the common query results in subsequent queries. This practice helps avoid redundant scanning, aggregating, and joins. MQTs are also useful for data warehouse type applications.

MQTs do not completely eliminate optimization problems, but rather move optimization issues to other areas. Some challenges include finding the best MQT for an expected workload, maintaining the MQTs when underlying tables are updated, ability to recognize usefulness of MQT for a query, and the ability to determine when DB2 will actually use the MQT for a query. Most of these types of problems are addressed by OLAP tools, but MQTs are the first step.

The main advantage of the MQT is that DB2 can recognize a summary query against the source tables for the MQT, and can rewrite the query to use the MQT instead. It is, however, your responsibility to move data into the MQT by using a REFRESH TABLE command, or by manually moving the data yourself.

Volatile Tables

In DB2 V8 and later, volatile tables provide a way to prefer index access over table space scans or non-matching index scans for tables that have statistics that make them appear to be small. They are good for tables that shrink and grow, allowing matching index scans on tables that have grown larger without new RUNSTATS.

They also improve cluster table support. Cluster tables are tables that have groups or clusters of data that logically belong together. Within each group, rows should be accessed in the same sequence to avoid lock contention during concurrent access. The sequence of access is determined by the primary key, and if DB2 changes the access path lock, contention can occur. To best support cluster tables (volatile tables), DB2 uses index-only access when possible. This practice minimizes application contention on cluster tables by preserving the access sequence by primary key. You should ensure that indexes are available for single table access and joins.

You can specify the keyword VOLATILE on the CREATE TABLE or the ALTER TABLE statements. If specified, you are forcing an access path of index accessing and no list prefetch.

Clone Tables

In DB2 9, you can create a clone table on an existing base table at the current server by using the ALTER TABLE statement. Although ALTER TABLE syntax is used to create a clone table, the authorization that is granted as part of the clone creation process is the same as you would get during regular CREATE TABLE processing. The schema (creator) for the clone table is the same as the base table. You can create a clone table only if the base table is in a universal table space.

To create a clone table, issue an ALTER TABLE statement with the ADD CLONE option. For example:

```
ALTER TABLE base-table-name ADD CLONE clone-table-name
```

The creation or drop of a clone table does not affect the applications that are accessing base table data. No base object quiesce is necessary and this process does not invalidate plans, packages, or the dynamic statement cache.

You can exchange the base and clone data by using the EXCHANGE statement. To exchange table and index data between the base table and clone table, issue an EXCHANGE statement with the DATA BETWEEN TABLE *table-name*1 AND *table-name*2 syntax.

Note: This is a method of performing an online load replace.

After a data exchange, the base and clone table names remain the same as they were before the data exchange. No data movement actually occurs. The instance numbers in the underlying VSAM data sets for the objects (tables and indexes) change, which changes the data that appears in the base and clone tables and their indexes.

Example

A base table that exists with the data set name *I0001.*. The table is cloned and the clone's data set is initially named *.I0002.*. After an exchange, the base objects are named *.I0002.* and the clones are named *I0001.*. Each time that an exchange happens, the instance numbers that represent the base and the clone objects change, which immediately changes the data contained in the base and clone tables and indexes. When the clone is dropped and an uneven number of EXCHANGE statements have been executed, the base table will have an *I0002.* data set name. This could be confusing.

Table Designs for Special Situations

When you are storing large amounts of data or maximizing transaction performance, you can do many creative actions.

UNION in View for Large Table Design

The amount of data being stored in DB2 is increasing dramatically. Availability of databases is also an increasingly important issue. As you build these giant tables in your system, you must help ensure that they are built in a way that they are available 24-hours a day, 7-days per week. Large tables must be easy to access, hold significant quantities of data, and easy to manage.

Traditionally, larger database tables have been placed into partitioned table spaces. Partitioning helps with database management because it is easier to manage several small objects versus one very large object. Some limits to partitioning exist. For example, each partition is limited to a maximum size of 64 GB, a partitioning index is required (DB2 V7 only), and if efficient alternate access paths to the data are desired, then non-partitioning secondary indexes (NPSIs) are required. These NPSIs are not partitioned, and exist as single large database indexes. Thus, NPSIs can present themselves as an obstacle to availability (that is, a utility operation against a single partition may potentially make the entire NPSI unavailable), and as an impairment to database management because it is more difficult to manage large database objects.

You can use a UNION in a view as an alternative to table partitioning in support of very large database tables. In this type of design, several database tables can be created to hold different subsets of the data that would have otherwise been held in a single table. Key values, similar to what may be used in partitioning, can be used to determine which data goes into which of the various tables. For example, the following view definition:

```
CREATE VIEW V_ACCOUNT_HISTORY
   (ACCOUNT_ID, PAYMENT_DATE, PAYMENT_AMOUNT,
    PAYMENT_TYPE, INVOICE_NUMBER)
AS
SELECT ACCOUNT_ID, PAYMENT_DATE, PAYMENT_AMOUNT,
       PAYMENT_TYPE, INVOICE_NUMBER
FROM   ACCOUNT_HISTORY1
WHERE  ACCOUNT_ID BETWEEN 1 AND 100000000
UNION ALL
SELECT ACCOUNT_ID, PAYMENT_DATE, PAYMENT_AMOUNT,
       PAYMENT_TYPE, INVOICE_NUMBER
FROM   ACCOUNT_HISTORY2
WHERE  ACCOUNT_ID BETWEEN 100000001 AND 200000000
UNION ALL
SELECT ACCOUNT_ID, PAYMENT_DATE, PAYMENT_AMOUNT,
       PAYMENT_TYPE, INVOICE_NUMBER
FROM   ACCOUNT_HISTORY3
WHERE  ACCOUNT_ID BETWEEN 200000001 AND 300000000
UNION ALL
SELECT ACCOUNT_ID, PAYMENT_DATE, PAYMENT_AMOUNT,
       PAYMENT_TYPE, INVOICE_NUMBER
FROM   ACCOUNT_HISTORY4
WHERE  ACCOUNT_ID BETWEEN 300000001 AND 999999999;
```

Advantages of UNION in a View

By separating the data into different tables and creating the view over the tables, you can create a logical account history table with these distinct advantages over a single physical table:

- You can add or remove tables with small outages, usually just the time it takes to drop and recreate the view.
- You can partition each of the underlying tables, creating still smaller physical database objects.
- NPSIs on each of the underlying tables can be much smaller and easier to manage than they would be under a single table design.
- Utility operations can execute against an individual underlying table, or just a partition of that underlying table. This practice greatly decreases utility times against these individual pieces, and improves concurrency, as well as providing full partition independence.

- The view can be referenced in any SELECT statement in the same way as a physical table would be.
- Each underlying table could be as large as 16 TB, logically setting the size limit of the table that is represented by the view at 64 TB.
- Each underlying table could be clustered differently, or could be a segmented or partitioned tablespace.

DB2 distributes predicates against the view to every query block within the view and then compares the predicates. Any impossible predicates will result in the query block being pruned (not executed).

Limitations of UNION in a View

Consider the following limitations to UNION in a view:

- The view is read-only, which means you would have to use special program logic and possibly even dynamic SQL to perform inserts, updates, and deletes against the base tables. If you are using DB2 9, however, you could use INSTEAD OF triggers to provide this functionality.
- Predicate transitive closure happens after the join distribution. So, if you are joining from a table to a UNION in a view and predicate transitive closure is possible from the table to the view, you have to code the redundant predicate.
- DB2 can apply query block pruning for literal values and host variables, but not joined columns. For that reason, if you expect query block pruning on a joined column, code a programmatic join (this is the only case). Also, in some cases, the pruning does not always work for host variables, so you have to test.
- When using UNION in a view, keep the number of tables in a query to a reasonable number. This recommendation is especially true for joining because DB2 distributes the join to each query block. This practice multiplies the number of tables in the query, which can increase bind time and execution time. Also, you could exceed the 225 table limit in which case DB2 will materialize the view.
- In general, we recommend that you limit the number of tables UNIONed in the view to 15 or under.

Append Processing for High Volume Inserts

In situations in which you have very high insert rates, you may want to use append processing. When the append processing is turned on, DB2 uses an alternate insert algorithm that simplifies add data to the end of a partition or a table space. Append processing can be turned on for DB2 V7 or DB2 V8 by setting the table space options PCTFREE 0 FREEPAGE 0 MEMBER cluster. Make sure that APARS PQ86037 and PQ87381 are applied.

Note: In DB2 9, turn on append processing by using the APPEND YES option of the CREATE TABLE statement. When the append processing is turned on, DB2 does not respect the cluster during inserts and simply puts all new data at the end of the table space.

If you have a single index for read access, append processing may mean more random reads. This activity may require more frequent REORGs to keep the data organized for read access. Also, if you are partitioning, and the partitioning key is not the read index, then you still have random reads during the insert to the non-partitioned index. Ensure that you have adequate free space to avoid index page splits.

You can also use append processing to store historical or seldom read audit information. In these cases, perform your partitioning based upon an ascending value (for example, a date) and have all new data go to the end of the last partition. In this situation, all table space maintenance, such as copies and REORGs, are against the last partition. All other data is static and does not require maintenance. You may need to create a secondary index, or each read query will have to be for a range of values within the ascending key domain.

Building an Index on the Fly

In situations in which you are storing data through a key that is designed for a high speed insert strategy with minimal table maintenance, avoid secondary indexes. Scanning billions of rows is not an efficient use of resources.

A solution may be to build a lookup table that acts as a sparse index. This lookup table will contain nothing more than your ascending key values. One example would be dates (one date per month for every month possible in your database). If the historical data is organized and partitioned by the date, and you have only one date per month (to sub categorize the data), you can use the new sparse index to access the data you need. Using user-supplied dates as starting and ending points, the look-up table can be used to fill the gap with the dates in between. This practice provides the initial path to the history data. Read access is performed by constructing a key during SELECT processing.

In the following examples, we access an account history table (ACCT_HIST) that has a key on HIST_EFF_DTE, ACCT_ID, and the date lookup table that is called ACCT_HIST_DATES, which contains one column and one row for each legitimate date value corresponding to the HIST_EFF_DTE column of the ACCT_HIST table.

Current Data Access

Current data access is simple. You can retrieve the account history data directly from the account history table.

```
SELECT {columns}
FROM ACCT_HIST
WHERE ACCT_ID = :ACCT-ID
AND HIST_EFF_DTE = :HIST_EFF-DTE;
```

Range of Data Access

Accessing a range of data is a little more complicated than simply retrieving the most recent account history data. Use the sparse index history date table to build the key on demand. Apply the range of dates to the date range table, and then join the date range table to the history table.

```
SELECT {columns}
FROM ACCT_HIST HIST
INNER JOIN
     ACCT_HIST_DATES DTE
ON HIST.HIST_EFF_DTE = DTE.EFF_DTE
WHERE HIST.ACCT_ID = :ACCT-ID
AND HIST.HIST_EFF_DTE BETWEEN :BEGIN-DTE AND :END-DTE;
```

Full Data Access

To access all of the data for an account, you need a version of the previous query without the date range predicate.

```
SELECT {columns}
FROM ACCT_HIST HIST
INNER JOIN
     ACCT_HIST_DATES DTE
ON HIST.HIST_EFF_DTE = DTE.EFF_DTE
WHERE HIST.ACCT_ID = :ACCT-ID;
```

Denormalization "Light"

In many situations, especially those in which there is a conversion from a legacy flat file-based system to a relational database, there is a performance concern (or more importantly a performance problem in that an SLA is not being met) for reading the multiple DB2 tables. These are situations in which the application is expecting to read all of the data that was once represented by a single record, but it is now in many DB2 tables.

In these situations, a typical response is to begin denormalizing the tables.

Denormalizing tables counteracts all the advantages of moving your data into DB2; that is, efficiency, portability, flexibility, and faster time to delivery for your new applications.

In some situations, however, the performance issues resulting from reading multiple tables that are compared to the equivalent single record read are unacceptable. In these instances, you can consider implementing denormalization "light" instead. This type of denormalization can be applied to parent and child tables, when the child table data is in an optional relationship to the parent. Instead of denormalizing the optional child table data into the parent table, add a column to the parent table. This configuration indicates whether the child table has any data for that parent key.

Note: If you apply this method, ensure that you factor in maintenance of that indicator column. However, DB2 can use a during join predicate to avoid probing the child table when there is no data for the parent key.

For example, assume that you have an account table and an account history table. The account may or may not have account history, and so the following query would join the two tables together to list the basic account information (balance) along with the history information if present:

```
SELECT A.CURR_BAL, B.DTE, B.AMOUNT
FROM   ACCOUNT A
LEFT OUTER JOIN
       ACCT_HIST B
ON     A.ACCT_ID = B.ACCT_ID
ORDER BY B.DTE DESC
```

In this example, the query will always probe the account history table in support of the join, whether or not the account history table has data. You can employ denormalization "light" by adding an indicator column to the account table. You can then use a during join predicate. DB2 will perform the join operation only when the join condition is true. In this case, the access to the account history table is avoided when the indicator column has a value not equal to Y:

```
SELECT A.CURR_BAL, B.DTE, B.AMOUNT
FROM   ACCOUNT A
LEFT OUTER JOIN
       ACCT_HIST B
ON     A.ACCT_ID = B.ACCT_ID
AND    A.HIST_IND = 'Y'
ORDER BY B.DTE DESC
```

DB2 is going to test that indicator column first before performing the join operation, and supply nulls for the account history table when data is not present as indicated.

This type of design provides significant benefits when you are doing a legacy migration from a single record system to around 40 relational tables with lots of optional relationships. This form of denormalizing can improve performance in support of legacy system access, while maintaining the relation design for efficient future applications.

Chapter 5: EXPLAIN Facility and Predictive Analysis

EXPLAIN Facility

The DB2 EXPLAIN facility is used to expose query access path information. This result enables application developers and DBAs to see what access path DB2 is going to take for a query and decide if the query tuning is needed. DB2 can gather basic access path information in a special table called the PLAN_TABLE (DB2 V7, DB2 V8, DB2 9), as well as detailed information about predicate stages, filter factor, predicate matching, and dynamic statements that are cached (DB2 V8, DB2 9).

You can invoke EXPLAIN by doing one of the following:

- Executing the EXPLAIN SQL statement for a single statement
- Specifying the BIND option EXPLAIN(YES) for a plan or package bind
- Executing the EXPLAIN through the Optimization Service Center or through Visual EXPLAIN

EXPLAIN can populate many tables when it is executed. The target set of EXPLAIN tables depends on the authorization ID associated with the process. The creator (schema) of the EXPLAIN tables is determined by the CURRENT SQLID of the person running the EXPLAIN statement, or the owner of the plan or package at bind time. The EXPLAIN tables are optional, and DB2 only populates the tables that it finds under the SQLID or owner of the process invoking the EXPLAIN.

The following tables can be defined manually, and the DDL can be found in the DB2 sample library member DSNTESC:

PLAN_TABLE

Contains basic access path information for each query block of your statement. This information includes, details about index usage and the number of matching index columns, which join method is used, which access method is used, and whether a sort will be performed. The PLAN_TABLE forms the basis for access path determination.

DSN_STATEMNT_TABLE

Contains estimated cost information for the cost of a statement. If the statement table is present when you run EXPLAIN on a query, the table is populated with the cost information that corresponds to the access path information for the query stored in the PLAN_TABLE. For a given statement, this table contains the estimated processor cost in milliseconds and service units and it places the cost values into the following categories:

Category A - **DB2 has enough information to make a cost estimation without using any defaults.**

Category B - **DB2 has to use default values to make some cost calculations.**

The statement table can be used to compare estimated costs when you are attempting to modify statements for performance. This is a cost estimate, and is not truly reflective of how your statement will be used in an application (given input values, transaction patterns, and so on). You should always test your statements for performance in addition to using the statement table and EXPLAIN.

DSN_FUNCTION_TABLE

Contains information about user-defined functions that are a part of the SQL statement. Information from this table can be compared to the cost information (if populated) in the DB2 System Catalog table, SYSIBM.SYSROUTINES, for the user-defined functions.

DSN_STATEMENT_CACHE_TABLE

Is not populated by a normal invocation of EXPLAIN, but instead by the EXPLAIN STMTCACHE ALL statement. Issuing this statement results in DB2 reading the contents of the dynamic statement cache, and putting runtime execution information into the table. This includes information about the frequency of execution of these statements, the statement text, the number of rows that are processed by the statement, lock and latch requests, I/O operations, number of index scans, number of sorts, and so on. This is valuable information about the dynamic queries executing in a subsystem. This table is available only for DB2 V8 and DB2 9.

Note: Many EXPLAIN tables exist. Additional EXPLAIN tables are typically populated by the optimization tools that use them. You can make some tables without using the various optimization tools. For information about these tables, see the *DB2 Performance Monitoring and Tuning Guide* (DB2 9).

What EXPLAIN Tells You

The PLAN_TABLE is the key table for determining the access path for a query. The PLAN_TABLE provides the following critical information:

METHOD

Indicates the joint method, or whether an additional sort step is required.

ACCESSTYPE

Indicates whether the access to is through a table space scan or through index access. If it is by index access, the specific type of index access is indicated.

MATCHCOLS

Indicates the number of columns that are matched against the index if an index access is used.

INDEXONLY

Indicates that the access required can be served by accessing the index only, and avoiding any table access when a value of Y is in this column.

SORTN####, SORTC####

Indicates any sorts that may happen in support of a UNION, grouping, joint operation, and so on.

PREFETCH

Indicates whether the prefetch can play a role in the access.

By manually running EXPLAIN and examining the PLAN-TABLE, you can retrieve the following information:

- Access path
- Indexes that are used
- Join operations
- Any sorting that occurs as part of your query

If you have additional EXPLAIN tables (for example, those created by Visual Explain or the Optimization Service Center), those tables are populated automatically. This is done by using those tools or by manually running EXPLAIN. If you do not have the remote access that is required from a PC to use the tools, you can also query those tables manually.

The tables provide detailed information about such entities as predicates stages, filter factors, eliminated partitions, parallel operations, detailed cost information, and so on.

Note: For more information about tables, see the *DB2 Performance Monitoring and Tuning Guide* (DB2 9).

What EXPLAIN Does Not Tell You

EXPLAIN does not tell you everything about your queries. You have to be aware of this limitation to effectively performance tune and predictively analyze. EXPLAIN does not tell you about the following:

- INSERT indexes

  EXPLAIN does not tell you the index that DB2 will use for an INSERT statement. It is important that you understand your clustering indexes, and whether DB2 will be using APPEND processing for your inserts. This understanding is important for INSERT performance, and the proper organization of your data. For more information, see Designing Tables and Indexes for Performance (see page 59).

- Access path information for enforced referential constraints

  If you have INSERTS, UPDATES, and DELETES in the program to which you have applied EXPLAIN, the database enforced RI relationships and associated access paths are not exposed in the EXPLAIN tables. Ensure, therefore, that proper indexes in support of the RI constraints are established and in use.

- Predicate evaluation sequence

  The EXPLAIN tables do not show you the order in which predicates of the query are evaluated. For more information on predicate evaluation sequence, see Predicates and SQL Tuning.

- Statistics used

  The optimizer used catalog statistics to help determine the access path at the time EXPLAIN was applied to the statement. Unless you have historical statistics that correspond to the time that EXPLAIN was executed, you do not know that how the statistics appeared when the EXPLAIN was executed. Additionally, you do not know if they are different now and therefore you provide a different access path.

- Input values

  If you are using host variables in your programs, EXPLAIN does not know about the potential input values to those variables. Therefore, it is important for you to understand these values, what the most common values are, and if data is skewed relative to the input values.

- SQL statement

  The SQL statement is not captured in the EXPLAIN tables, although some of the predicates are. If you dynamically applied EXPLAIN to a statement, or you applied it through one of the tools, you know what the statement looks like. However, if you applied EXPLAIN to a package or plan, you have to see the program source code.

- Order of input to your transactions

  The SQL statement may appear to be correct when you view the access path, but you cannot tell the order of the input data in relation to the statement, what ranges are supplied, or how many transactions are issued. EXPLAIN output does not let you see whether it is possible to order the input or the data in the tables more efficiently.

- Program source code

  To understand the impact of the access path that a statement used, you have to see how that statement is used in the application program. We recommend that you regularly review the program source code of the program in which the statement is embedded in. By reviewing the program source code, you can answer the following questions:

  - How many times will the statement be executed?
  - Is the statement in a loop?
  - Can you avoid executing the statement?
  - Is the program issuing 100 statements on individual key values in a range predicate when one statement would suffice?
  - Is the programming performing programmatic joins?

  Although the EXPLAIN output may show an efficient access path, the statement might be unnecessary. You can also use a monitor or performance trace, which lets you see exactly which statements are executed.

Predictive Analysis Tools

You can use several predictive analysis tools to help you plan your implementation, including the Optimization Service Center (OCS), Visual EXPLAIN, and DB2 Estimator.

Optimization Service Center and Visual EXPLAIN

As the complexity of managing and tuning various workloads continues to escalate, many database administrators (DBAs) are struggling to maintain quality of service (QoS) while also managing costs. IBM Optimization Service Center for DB2 for z/OS (OSC) is a Windows workstation tool that is designed to ease the workload of DBAs through a set of automation tools that help optimize query performance and workloads. You can use OSC to identify and analyze problem SQL statements and to receive expert advice about statistics that are gathered to improve the performance of problematic and poorly performing SQL statements on a DB2 subsystem. It provides the following:

- The ability to snap the statement cache
- Collect statistics information

- Analyze indexes
- Group statements into workloads
- Monitor workloads
- An easy-to-understand display of a selected access path
- Suggestions for changing a SQL statement
- An ability to invoke EXPLAIN for dynamic SQL statements
- An ability to provide DB2 catalog statistics for referenced objects of an access path, or for a group of statements
- A subsystem parameter browser with the keyword find capabilities
- The ability to graphically create optimization hints (a feature not found in Visual EXPLAIN V8)

You can use OSC to analyze previously generated EXPLAIN data, or to gather EXPLAIN data and clarify dynamic SQL statements.

OSC is available for DB2 V8 and DB2 9.

DB2 Estimator

DB2 Estimator is another useful predictive analysis tool. This product runs on a PC and provides a graphical user interface for entering information about DB2 tables, indexes, and queries. Table and index definitions and statistics can be entered directly or imported to view DDL files. SQL statements can be imported from text files.

The table definitions and statistics can be used to accurately predict database sizes. SQL statements can be organized into transactions, and then information about DASD models, CPU size, access paths, and transaction frequencies can be set. After all of this information is put into DB2 Estimator, capacity reports can be produced. These reports contain estimates of the DASD required, as well as the amount of CPU required for an application. These reports are helpful during the initial stage of capacity planning, before any real programs or test data is available. DB2 Estimator is not available with DB2 9.

Predicting Database Performance

An approach for large systems design is to spend little time considering performance during the development and set aside project time for performance testing and tuning. This strategy frees the developers from having to consider performance in every aspect of their programming, and gives them the incentive to code more logic in their queries. This strategy makes for faster development time, and more logic in the queries means more opportunities for tuning.

If you choose to make performance decisions during the design phase, each decision should be backed by solid evidence, and not assumptions. You can create test tables, generate some test data, and write some small programs to simulate a performance situation and test your assumptions about how the database behaves. The feedback from these tests provides data to help you design your system for optimum performance. Reports can then be generated and reviewed by stakeholders.

Tools that you can use for testing statements, design ideas, or program processes include, but are not limited to, the following:

- REXX DB2 programs
- COBOL test programs
- Recursive SQL to generate data
- Recursive SQL to generate statements
- Data in tables to generate more data
- Data in tables to generate statements

Methods to Generate Data

To simulate program access, you need data in tables. You can simply type some data into INSERT statements, insert them into a table, and then use data from that table to generate more data. For example, you might have to test the various program processes against a PERSON_TABLE table and a PERSON_ORDER table. No actual data has been created yet, but you have to test the access patterns of incoming files. You can key some INSERT statements for the parent table, and then use the parent table to propagate data to the child table. For example, if the parent table, PERSON_TABLE, contains this data:

```
PERSON_ID    NAME
        1    JOHN SMITH
        2    BOB RADY
```

The following statement can be used to populate the child table, PERSON_ORDER, with some test data:

```
INSERT INTO PERSON_ORDER
(PERSON_ID, ORDER_NUM, PRDT_CODE, QTY, PRICE)
SELECT PERSON_ID, 1, 'B100', 10, 14.95
FROM    YLA.PERSON_TABLE
UNION ALL
SELECT PERSON_ID, 2, 'B120', 3, 1.95
FROM    YLA.PERSON_TABLE
```

The resulting PERSON_ORDER data would look like this:

```
PERSON_ID     ORDER_NUM    PRDT_CDE    QTY    PRICE
        1             1        B100     10    14.95
        1             2        B120      3     1.95
        2             1        B100     10    14.95
        2             2        B120      3     1.95
```

You can repeatedly use the statements to add more data, or additional statements can be executed against the PERSON_TABLE to generate more PERSON_TABLE data.

Recursive SQL (DB2 V8 and DB2 9) is a useful way to generate test data. The following is a simple recursive SQL statement:

```
WITH TEMP(N) AS
     (SELECT 1
       FROM SYSIBM.SYSDUMMY1
      UNION ALL
      SELECT N+1
      FROM TEMP
      WHERE N < 10)
 SELECT N
 FROM TEMP
```

This statement generates the numbers 1 through 10, one row each. You can use the power of recursive SQL to generate mass quantities of data that can be inserted into DB2 tables, ready for testing.

Example

The following piece of an SQL statement was used to insert 300,000 rows of data into a large test lookup table. The table was quickly populated with data, and a test was conducted to determine the performance. It was determined that the performance of this large lookup table would not be adequate, but that could not have been confirmed without testing:

```
WITH LASTPOS (KEYVAL) AS
   (VALUES (0)
    UNION ALL
    SELECT KEYVAL + 1
    FROM   LASTPOS
    WHERE  KEYVAL < 9)
,STALETBL (STALE_IND) AS
   (VALUES 'S', 'F')
SELECT STALE_IND, KEYVAL
 ,CASE STALE_IND WHEN 'S' THEN
   CASE KEYVAL WHEN 0 THEN 1
    WHEN 1 THEN 2 WHEN 2 THEN 3
    WHEN 3 THEN 4 WHEN 4 THEN 4
    WHEN 5 THEN 6 WHEN 6 THEN 7
    WHEN 7 THEN 8 WHEN 8 THEN 9
    WHEN 9 THEN 10 END
                  WHEN 'F' THEN
   CASE KEYVAL WHEN 0 THEN 11
    WHEN 1 THEN 12 WHEN 2 THEN 13
    WHEN 3 THEN 14 WHEN 4 THEN 15
    WHEN 5 THEN 16 WHEN 6 THEN 17
    WHEN 7 THEN 18 WHEN 8 THEN 19
    WHEN 9 THEN 20 END
       END AS PART_NUM
FROM  LASTPOS INNER JOIN
      STALETBL ON 1=1;
```

Methods to Generate Statements

Just as data can be generated, so can statements. You can write SQL statements that generate statements. For example, you might need to generate single select statements against the EMP table to test a possible application process or scenario. You can write a statement such as the following to generate those statements:

```
SELECT 'SELECT LASTNAME, FIRSTNME ' CONCAT
       'FROM EMP WHERE EMPNO = ''' CONCAT
       EMPNO CONCAT ''';'
FROM   SUSAN.EMP
WHERE  WORKDEPT IN ('C01', 'E11')
AND    RAND() < .33
```

The previously noted query generates SELECT statements for approximately 33 percent of the employees in departments C01 and E01. The output appears similar to the following:

```
1
-----------------------------------------------------------
SELECT LASTNAME, FIRSTNME FROM EMP WHERE EMPNO = '000030';
SELECT LASTNAME, FIRSTNME FROM EMP WHERE EMPNO = '000130';
SELECT LASTNAME, FIRSTNME FROM EMP WHERE EMPNO = '200310';
```

You can also use recursive SQL statements to generate statements. The following statement was used during the testing of high performance INSERTs to an account history table. The following statement generated 50,000 random insert statements:

```
WITH GENDATA (ACCT_ID, HIST_EFF_DTE, ORDERVAL) AS
(VALUES (CAST(2 AS DEC(11,0)), CAST('2003-02-01' AS DATE), CAST(1 AS FLOAT))
UNION ALL
SELECT ACCT_ID + 5, HIST_EFF_DTE, RAND()
FROM   GENDATA
WHERE  ACCT_ID < 249997)
SELECT 'INSERT INTO YLA.ACCT_HIST (ACCT_ID, HIST_EFF_DTE)' CONCAT
' VALUES(' CONCAT CHAR(ACCT_ID) CONCAT ',' CONCAT ''''
CONCAT CHAR(HIST_EFF_DTE,ISO) CONCAT '''' CONCAT ');'
 FROM GENDATA ORDER BY ORDERVAL;
```

Determine Your Access Patterns

If you have access to input data or transactions, build small sample test programs to determine the effect of these inputs on the database design. This action involves simple REXX or COBOL programs that contain little or no business logic, just simple queries that simulate the anticipated database access. Running these programs can give you an idea about the impact of random versus sequential processing, or the impact of sequential processing versus skip sequential processing. It can also give you an idea of how buffer settings can affect performance and whether or not performance enhancers, such as sequential detection or index lookaside, will be effective.

Simulating Access with Tests

Consider writing a simple COBOL program to access a table to understand the following:

- What is the proper clustering?
- How effective compression will be for performance?
- Whether or not one index performs better than another?
- If adding an index adversely affects INSERTs and DELETEs?

Example:

A database team debated whether they would gain the greatest flexibility for their database by having an entire application interface use large joins between parent and child tables, or whether all access should be through individual table access (programmatic joins). Coding for both types of access meant extra programming effort, but they had to understand the cost of the programmatic joins for the application.

They wrote two simple COBOL programs against a test database—one with a two-table programmatic join and the other with the equivalent SQL join. As a result, the team determined that the SQL join consumed 30 percent less CPU than the programmatic join.

Chapter 6: Monitoring

It is critical to design for performance when building applications, databases, and SQL statements. You have designed the correct SQL, avoided programmatic joins, clustered commonly accessed tables in the same sequence, and have avoided inefficient repeat processing. Your application may be in production and running without problems, but there is still more you can do to increase its performance.

Which statement is the most expensive? Is it the tablespace scan that runs once per day, or the matching index scan running millions of times per day? Are all your SQL statements sub-second responders that do not need tuning? Which statement is the number one consumer of CPU resources? All of these questions can be answered by monitoring your DB2 subsystems and the applications that access them.

DB2 provides facilities for monitoring the behavior of the subsystem, as well as the applications that are connected to it. This feature is achieved primarily though the DB2 trace facility. DB2 has several different types of traces.

DB2 Traces

DB2 provides a trace facility to help track and record events within a DB2 subsystem. There are six types of traces:

- Statistics
- Accounting
- Audit
- Performance
- Monitor
- Global

These traces should play an integral part in your performance monitoring process.

Statistics Trace

The data collected in the statistics trace lets you conduct DB2 capacity planning and tune the entire set of DB2 programs. The statistics trace reports information about how much the DB2 system services and database services are used. It is a system-wide trace and should not be used for charge-back accounting. Statistics trace classes 1, 3, 4, 5, and 6 are the default classes for the statistics trace if statistics are specified as 'yes' in the installation panel DSNTIPN. If the statistics trace is started using the START TRACE command, class 1 is the default class.

The statistics trace can collect information about the number of threads that are connected, the number of SQL statements that are executed, and the amount of storage that is consumed within the database manager. The statistics trace also collects information about address space, deadlocks, timeouts, logging, and buffer pool utilization. This information is collected at regular intervals for an entire DB2 subsystem. The interval is typically 10 minutes or 15 minutes per record.

Accounting Trace

The accounting trace provides data that lets you assign DB2 costs to individual authorization IDs and to tune individual programs. The DB2 accounting trace provides information that is related to application programs, such as:

- Start and stop times
- Number of commits and aborts
- The number of times certain SQL statements are issued
- Number of buffer pool requests
- Counts of certain locking events
- Processor resources that are consumed
- Thread wait times for various events
- RID pool processing
- Distributed processing
- Resource limit facility statistics

Accounting times are usually the prime indicator of performance problems, and most often should be the starting point for analysis. DB2 times are classified as follows:

Class 1: Specifies the time that the application spent since connecting to DB2, including the time that is spent outside DB2.

Class 2: Specifies the elapsed time spent in DB2. It is divided into CPU time and waiting time.

Class 3: Specifies elapsed time is divided into various waits, such as the duration of suspensions due to waits for locks and latches or waits for I/O.

DB2 trace begins collecting this data at successful thread allocations to DB2 and writes a completed record when the thread terminates or when the authorization ID changes. Having the accounting trace active is critical for proper performance monitoring, analysis, and tuning. When an application connects to DB2, it is executing across address spaces, and the DB2 address spaces are shared by perhaps thousands of users across many address spaces. The accounting trace provides information about the time that is spent within DB2, as well as the overall application time. Class 2 time is a component of class 1 time, and class 3 time a component of class 2 time.

Accounting data for class 1 (the default) is accumulated by several DB2 components during normal execution. This data is collected at the end of the accounting period and does not involve as much overhead as the individual event tracing. Conversely, when you start class 2, 3, 7, or 8, many additional trace points are activated. Every occurrence of these events is traced internally by DB2 trace, however these traces are not written to any external destination. The accounting facility rather uses these traces to compute the additional total statistics that appear in the accounting record when class 2 or class 3 is activated. Accounting class 1 must be active to externalize the information.

We recommend that you set accounting classes 1,2,3,7,8. This can add between 4 percent and 5 percent of your overall system CPU consumption. However, if you are already writing accounting classes 1, 2, 3, adding 7 and 8 typically should not add much overhead. Also, if you are using an online performance monitor, it could already have these classes started. If that is the case, adding SMF as a destination for these classes should not add any CPU overhead.

Performance Trace

The performance trace provides information about various DB2 events, including those related to distributed data processing. You can use this information to identify a suspected problem or to tune DB2 programs and resources for individual users or for DB2 as a whole. To start a performance trace, you must use the –START TRACE(PERFM) command. Performance traces cannot be automatically started. Performance traces are expensive to run, and consume much CPU. They also collect a large volume of information. Performance traces are usually run using an online monitor tool, or the output from the performance trace can be sent to SMF and then analyzed using a monitor reporting tool, or sent to IBM for analysis.

Because performance traces can consume numerous resources and generate numerous data, you have many options when starting the trace to balance the information that is desired with the resources consumed. These options include limiting the trace data that is collected by plan, package, trace class, and even IFCID.

Performance traces are typically used by online monitor tools to track a specific problem for a given plan or package. Reports can then be produced by the monitor software, and can detail SQL performance, locking, and many other detailed activities. Performance trace data can also be written to SMF records, and batch reporting tools can read those records to produce detailed information about the execution of SQL statements in the application.

Accounting and Statistics Reports

After DB2 trace information has been collected, you can create reports by reading the SMF records. These reports are typically produced by running reporting software that is provided by a performance reporting product. These reports are a critical part of performance analysis and tuning. You should familiarize yourself with the statistics and accounting reports, as they are the best way to gauge the health of a DB2 subsystem and the applications using it. These reports are also the best way to monitor performance trends and proactively detect potential problems before they become critical.

Statistics Report

Because statistics records are collected typically at 10-minute or 15-minute intervals, many records can be collected on a daily basis. Your reporting software should be able to produce summary reports, which can gather and summarize the data for a period, or detail reports that can report on every statistics interval. Start with a daily summary report, and look for specific problems within the DB2 subsystem. After you detect a problem, you can produce a detailed report to determine the specific period that the problem occurred, and also coordinate the investigation with detailed accounting reports for the same time period to attribute the problem to a specific application or process.

We recommend that you use statistics reports on a regular basis, and use monitoring software documentation, along with the *DB2 Administration Guide* (DB2 V8) or *DB2 Performance Monitoring and Tuning Guide* (DB2 9).

Note the following in a statistics report:

RID Pool Failures

The statistics report should include a reason to detail the usage of the RID pool for activities such as list prefetch, multiple index access, and hybrid joins. The report will also indicate RID failures. There can be RDS failures, DM failures, and failures due to insufficient size.

If you are getting failures due to insufficient storage you can increase the RID pool size. However, if you are getting RDS or DM failures in the RID pool, the access path that is selected may be reverting to a table space scan. In these situations, determine which applications are getting these RID failures. Therefore, you have to produce a detailed statistics report that can identify the time of the failures, and also produce detailed accounting reports that show which threads are getting the failures. You to determine the packages within the plan. DB2 EXPLAIN can be used to determine which statements are using list prefetch, hybrid join, or multi-index access. You may have to test the queries to determine if they are the one's seeing the failures, and if they are, try to influence the optimizer to change the access path.

Bufferpool Issues

One of the most valuable piece of information in a statistics report is the section covering buffer utilization and performance. For each buffer pool in use, the report includes the size of the pool, sequential and random getpages, prefetch operations, pages that are written, and number of sequential I/Os, buffer thresholds that are reached, random I/Os, and write I/Os, and much more.

Watch for the number of synchronous reads for sequential access, which may be an indication that the number of pages is too small and pages for a sequential prefetch are stolen before they are used. Additionally watch whether any critical thresholds are reached, if there are write engines not available, and whether deferred write thresholds are triggering. It is also important to monitor the number of getpages per synchronous I/O, as well as the buffer hit ratio.

Logging Problems

The statistics report provides important information about logging. This information includes the number of system checkpoints, number of reads from the log buffer, active log data sets, or archived log datasets, number of unavailable output buffers, and total log writes. This information could give you an indication whether you have to increase log buffer sizes or you have to investigate frequent application rollbacks or other activities that could cause excessive log reads.

EDM Pool Hit Ratio

The statistics report details how often database objects such as DBDs, cursor tables, and package tables are requested as well as how often those requests have to be satisfied using a disk read to one of the directory tables. You can use this information to determine if you should increase the EDM pool size. You also receive statistics about the use of dynamic statement cache and the number of times statement access paths were reused in the dynamic statement cache. This information could let you see the size of your cache and its effectiveness, but it could also highlight potential reusability of the statements in the cache.

Deadlocks and Timeouts

The statistics report provides subsystem-wide perspective on the number of deadlocks and timeouts your applications have experienced. You can use this information as an overall method of detecting deadlocks and timeouts across all applications. If the statistics summary report shows a positive count, you can use the detailed report to determine what time the problems are occurring. You can also use accounting reports to determine which applications are experiencing the problem.

Accounting Report

An accounting report reads the SMF accounting records to produce the thread or application level information from the accounting trace. These reports typically can summarize information at the level of a plan, package, correlation ID, authorization ID, and so on. In addition, you can also produce one report per thread. This accounting detail report can provide detailed performance information for the execution of a specific application process. If you have accounting classes 1, 2, 3, 7, and 8 turned on, the information will be reported at the plan and the package level.

You can use the accounting report to find specific problems within certain applications, programs, or threads.

Class 1 and Class 2 Timings

Indicates the entire application time, including the time that is spent within DB2. Class 2 is a component of class 1, and represents the amount of time the application spent in DB2. The first question to ask when an application is experiencing a performance problem is, "Where is the time being spent?" The first indication that the performance issue is with DB2 will be a high class 2 time relative to the class 1 time. Within class 2, you could be having a CPU issue (CPU time represents most of the class 2 time), or a wait issue (CPU represents little of the overall class 2 time, but class 3 wait represents most of the time), or maybe your entire system is CPU bound (Class 2 overall elapsed is not reflected in class 2 CPU and class 3 wait time combined).

Buffer Usage

Displays information about buffer usage at the thread level. This information can be used to determine how a specific application or process is using the buffer pools. If you have situations in which certain buffers have high random getpage counts, look at which applications are causing the high number of random getpages. You can use this thread level information to determine which applications are accessing buffers randomly versus sequentially. Perhaps then you can see which objects the application uses, and can use this information to separate sequentially accessed objects from randomly accessed objects into different buffer pools. The buffer pool information in the accounting report also indicates how well the application is using the buffers. The report can be used during buffer pool tuning to determine the impact of buffer changes on an application.

Package Execution Times

Displays information about the DB2 processing on a package level, if accounting classes 7 and 8 are turned on. This information is important for performance tuning because it lets you determine which programs in a poorly performing application should be reviewed first.

Deadlocks, Timeouts, and Lock Waits

Indicates the number of deadlocks and timeouts that occurred on a thread level. It also reports the time that the thread spent waiting on locks. This information lets you know whether you have to do additional investigation into applications that are having locking issues.

Excessive Synchronous I/Os

Indicates if there are many excessive random synchronous I/Os being issued, and how much time the application spends waiting on I/O, if you have a slow running job or online process. The report also indicates exactly what is slow about that job or process. The information in the report can also be used to approximately determine the efficiency of your DASD by simply dividing the number of synchronous I/O's into the total synchronous I/O wait time.

RID Failures

Displays thread level RID pool failure information. This information provides important details to help you determine if you have access path problems in a specific application.

High Getpage Counts and High CPU Time

Indicates if your performance problem is related to an inefficient repeat process. If the report shows a high getpage count, or that most of the elapsed time is actually class 2 CPU time, you may have an inefficient repeat process in one of the programs for the plan. Often it is hard to determine if an application is doing repeat processing when there are not many I/Os being issued, so you can use the package level information to determine which program uses the most CPU and you can try to identify any inefficiencies in that program.

Online Performance Monitors

DB2 can write accounting, statistics, monitor, and performance trace data to line buffers. This ability allows for online performance reporting software to read those buffers, and report on DB2 subsystem and application performance in real time. These online monitors can be used to review subsystem activity and statistics, thread level performance information, deadlocks and timeouts, I/O activity, and dynamically execute and report on performance traces.

Overall Application Performance Monitoring

You will not find one method for improving the overall performance of large systems. We can tune the DB2 subsystem parameters, logging and storage, but often we are only making a bad situation worse. In these circumstances, the "80/20" rule applies, and you will eventually have to address application performance.

You can quickly identify the easy to recognize performance issues in a report to your boss, and change the application or database to support a more efficient path to the data. Management support is required and an effective manner of communicating performance tuning opportunities and results is crucial.

Setting the Appropriate Accounting Traces

DB2 accounting traces play a valuable role in reporting on application performance tuning opportunities. DB2 accounting traces 1, 2, 3, 7, and 8 must be set to monitor performance at the package level. After you do that, you can further examine the most expensive programs (identified by package) to look for tuning changes. This reporting process can serve as a quick solution to identifying an application performance problem, but you can incorporate it into a long-term solution that identifies problems and tracks changes.

Some have expressed concern about the performance impact of this level of DB2 accounting. The *IBM DB2 Administration Guide* (DB2 V8) or the *DB2 Performance Monitoring and Tuning Guide* (DB2 9) states that the performance impact of setting these traces is minimal and the benefits can be substantial. Tests that are performed at a customer site demonstrated an overall system impact of 4.3 percent for all DB2 activity when accounting classes 1, 2, 3, 7, and 8 are started. In addition, adding accounting classes 7 and 8 when 1, 2, and 3 are already started has the nominal impact, as does the addition of most other performance monitor equivalent traces (that is, your online monitor software).

Summarizing Accounting Data

To communicate application performance information to management, the accounting data must be organized and summarized up to the application level. You need a reporting tool that formats DB2 accounting traces from System Management Facilities (SMF) files to produce the type of report you are interested in. Most reporting tools can produce DB2 accounting reports at a package summary level. Some can even produce customized reports that can filter only the information you want out of the wealth of information in trace records.

You can process whatever types of reports you produce so that a concentrated amount of information about DB2 application performance can be extracted. This information is reduced to the amount of elapsed time and CPU time the application consumes daily and the number of SQL statements each package issues daily. This highly specific information is your first indication as to which packages provide the best DB2 tuning opportunity. The following example is from a package-level report with the areas of interest highlighted (Package Name, Total DB2 Elapsed, Total SQL Count, Total DB2 TCB):

```
                                                - AVERAGE ---- -- TOTAL ----
PACKAGE EXECUTIONS                               HHH:MM:SS.TTT HHH:MM:SS.TTT
------------------                 DB2 TCB......         0.008     25:18.567
COLL ID CPG2SU01                   I/O..........         0.205 010:28:50.335
PROGRAM FHPRDA2                    LOCK/LATCH...         0.000        43.652
                                   OTHER RD I/O.         0.000      1:29.015
                    HHH:MM:SS.TTT  OTHER WR I/O.         0.000         0.000
AVG DB2 ELAPSED             0.214  DB2 SERVICES.         0.000         0.001
TOTAL DB2 ELAPSED  010:55:40.211   LOG QUIESCE..         0.000         0.000
                                   DRAIN LOCK...         0.000         0.000
AVG SQL COUNT               86.9   CLAIM RELEASE         0.000         0.000
TOTAL SQL COUNT         15948446   ARCH LOG READ         0.000         0.000
AVG STORPROC EXECUTED        0.0   PG LATCH CONT         0.000         0.000
TOT STORPROC EXECUTED          0   WT DS MSGS            0.000         0.000
AVG UDFS SCHEDULED           0.0   WT GLBL CONT          0.000         0.000
TOT UDFS SCHEDULED             0   GLB CHLD L-LK         0.000         0.000
                                   GLB OTHR L-LK         0.000         0.000
                                   GLB PSET P-LK         0.000         0.000
                                   GLB PAGE P-LK         0.000         0.000
                                   GLB OTHR P-LK         0.000         0.000
                                   OTHER DB2....         0.000        58.196
```

If you do not have access to a reporting tool that can filter pieces of information, you can write a simple program in any language to read the standard accounting reports and pull out the information you need. REXX is an excellent programming language that is well-suited to this type of "report scraping"; you can write a REXX program to do such work in a few hours. If you want to avoid dependency on reporting software, you could write a slightly more sophisticated program to read the SMF data directly to produce similar summary information. After the standard reports are processed and summarized, the information for a specific interval (say one day) can appear in a simple spreadsheet. You can sort the spreadsheet by descending CPU. With high consumers at the top of the report, the obvious issues are then easy to spot. The following spreadsheet can be derived by extracting the fields of interest from a package-level summary report:

Package	Executions	Total Elapsed	Total CPU	Total SQL	ElapsedExecution	CPUExecution	ElapsedSQL	CPUSQL
ACCT001	246745	75694,2992	**5187.4262**	1881908	0.3067	0.021	0.0402	0.0027
ACCT002	613316	26277.2022	**4381.7928**	1310374	0.0428	0.0071	0.02	0.0033
ACCTB01	8833	4654.4292	2723.1485	531455	0.5269	0.3082	0.0087	0.0051
RPTS001	93	6998.7605	**2491.9989**	5762	74.2554	26.7956	**1.2146**	**0.4324**
ACCT003	169236	33439.2804	2198.0959	1124463	0.1975	0.0129	0.0297	0.0019
RPTS002	2686	2648.3583	**2130.2409**	2686	0.9859	0.793	**0.9859**	**0.793**
HRPK001	281	4603.1262	2017.7179	59048	16.3812	7.1804	0.0779	0.0341
HRPKB01	21846	3633.5143	200636083	316746	0.1663	0.0918	0.0114	0.0053
HRBKB01	505	2079.5351	1653.5773	5776	4.1178	3.2744	0.36	0.2862
CUSTB01	1	4653.9935	1416.6254	759111	4653.9935	1416.6254	0.0006	0.00001
CUSTB02	1	3862.1498	1399.9468	7971317	3862.1498	1399.9468	0.0004	0.0001
CUST001	246670	12636.0232	1249.7678	635911	0.0512	0.005	0.0198	0.0019
CUSTB03	280	24171.1267	1191.0164	765906	86.3254	4.2536	0.0315	0.0015
RPTS003	1	5163.3568	884.0148	1456541	5163.3568	884.0148	0.0035	0.0006
CUST002	47923	10796.5509	875.252	489288	0.2252	0.0182	0.022	0.0017
CUST003	68628	3428.4162	739.4523	558436	0.0499	0.0107	0.0061	0.0013
CUSTB04	2	1183.2068	716.2694	3916502	591.6034	358.1347	0.0003	0.0001
CUSTB05	563	713.9306	713.9306	1001	2.1886	1.268	1.2309	0.7132

Identify basic issues when choosing the first programs to address. For example, package ACCT001 consumes the most CPU each day, and issues nearly 2 million SQL statements. Although the CPU consumed per statement on average is low, the quantity of statements that are issued indicates an opportunity to save significant resources. If just a small amount of CPU can be saved, it will quickly add up. The same recommendation applies to package ACCT002 and packages RPTS001 and RPTS002. These are some of the highest consumers of CPU, and they also have a relatively high average CPU per SQL statement. This information indicates that there may be some inefficient SQL statements involved. Because the programs consume significant CPU per day, tuning these inefficient statements could yield significant savings.

ACCT001, ACCT002, RPTS001, and RPTS002 represent the best opportunities for saving CPU, so examine those first. Without this type of summarized reporting, it is difficult to do any sort of truly productive tuning.

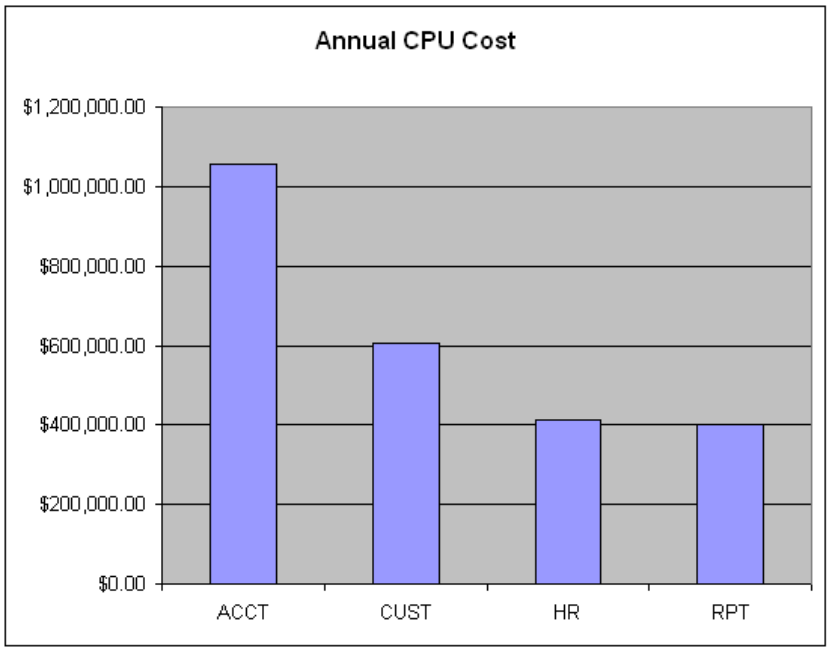
Reporting to Management

To do this type of tuning, you need approval from management and application developers. These approvals can sometimes be the most difficult part because, unfortunately, most application tuning involves costly changes to programs. To demonstrate the potential Return On Investment (ROI) for programming time, report the cost of application performance problems in terms of dollars.

The summarized reports can report on information on the application level. An in-house naming standard can be used to combine all the performance information from various packages into application-level summaries. This practice lets you classify applications and address the ones that use the most resources.

For example, if the in-house accounting application has a program naming standard where all program names begin with "ACCT," the corresponding DB2 package accounting information can be grouped by this header. Thus, the DB2 accounting report data for programs ACCT001, ACCT002, and ACCT003 can be grouped together, and their accounting information summarized to represent the "ACCT" application.

Most capacity planners have formulas for converting CPU time into dollars. If you get this formula from the capacity planner, and you categorize the package information by application, you can easily turn your daily package summary report into an annual CPU cost per application. The following shows a simple chart that is developed using an in-house naming standard and a CPU-to-dollars formula.



Make sure that you produce a "cost reduction" report, in dollars, after a phase of the tuning has completed. This report makes it clear to management what the tuning efforts have accomplished and gives them an incentive for further tuning efforts. Consider providing a visual representation of your data. A bar chart with before and after results can be highly effective in conveying performance tuning impact.

Finding the Statements of Interest

After you find the highest consuming packages and obtain management approval to tune them, you need additional analysis of the programs that present the best opportunity. Ask questions such as:

- Which DB2 statements are executing and how often?
- How much CPU time is required?
- Is there logic in the program that is resulting in an excessive number of unnecessary database calls?
- Are there SQL statements with relatively poor access paths?

Involve managers and developers in your investigation. It is much easier to tune with a team approach where different team members can be responsible for different analysis.

The following list details a few ways to gather statement-level information:

- Get program source listings and plan table information.
- Watch an online monitor.
- Run a short performance trace against the application of interest.

Performance traces are expensive, sometimes adding as much as 20 percent to the overall CPU costs. However, a short-term performance trace may be an effective tool for gathering information on frequent SQL statements and their true costs.

If plan table information is not available for the targeted package, you can rebind that package with EXPLAIN(YES). If it is hard to get the outage to rebind EXPLAIN(YES) or a plan table is available for a different owner ID, you could also copy the package with EXPLAIN(YES) (for example, execute a BIND into a special/dummy collection-ID) rather than rebinding it.

The following example shows PLAN_TABLE data for two of the most expensive programs in our example.

```
Q_QB_PL       PNAME   MT  TNAME         T_NO AT  MC ACC_NM    IX  NJ_CUJOG  PF
000640-01-01  ACCT001  0  PERSON_ACCT   1    I   1  AAIXPAC0    N   NNNNN     S
000640-01-02  ACCT001  3                0        0              N   NNNYN
001029-01-01  RPTS001  0  PERSON        1    I   1  AAIXPRC0    N   NNNNN     S
001029-01-02  RPTS001  4  ACCOUNT       2    I   1  CCIXACC0    N   NNNNN     L
001029-01-03  RPTS001  1  ACCT_ORDER    3    I   2  CCIXAOC0    N   NNNNN
```

In this example, the most expensive program issues a simple SQL statement with matching index access to the PERSON_ACCT table, which results in a sort (Method=3). The programmer, when consulted, advised that the query rarely returns more than a single row of data. In this case, a bubble sort in the application program replaced the DB2 sort. The bubble sort algorithm was almost never used because the query rarely returned more than one row, and the CPU associated with DB2 sort initialization was avoided. Because this query was executing many thousands of times per day, the CPU savings were substantial.

Our high-consuming reporting program is performing a hybrid join (Method=4). While hybrid joins are acceptable, our system statistics were showing us that there were Relational Data Server (RDS) subcomponent failures in the subsystem. We considered whether this query was causing an RDS failure, and we reverted to a tablespace scan. This notion was proven true. We tuned the statement to use the nested loop over hybrid join, the RDS failures and subsequent tablespace scan were avoided, and CPU and elapsed time improved dramatically.

While these statements may not have caused concern when looking at EXPLAIN results, when combined with the accounting data, they raised an alert for further investigation.

The Tools You Need

The application performance summary report identifies applications of interest and provides the initial tool for reporting to management. You can also use the other tools and reports that are listed here to identify and tune SQL statements in your most expensive packages:

Package report

> Lists all packages that are sorted by the most expensive first (usually by CPU) in your system. This information is the starting point for your application tuning exercise. Use this report to identify your most expensive applications or programs. You can also use this report to propose tuning to management and to drill down to more detailed levels in the other reports.

Trace or monitor the report

> Lists output from the online monitor for the packages identified as high consumers in your package report. This type of monitoring will help to drill down to the high-consuming SQL statements within these packages.

Plan table report

> Lists extractions of plan table information for the high-consuming programs that are identified in your package report. This report helps you identify quickly bad access paths that can be tuned. Consider the frequency of execution as indicated in the package report. Even a simple action such as a small sort can be expensive if executed often.

Index report

Lists all indexes on tables in the database of interest. This report should include the index name, table name, columns names, columns sequence, cluster ratio, clustering, first key cardinality, and full key cardinality. Use this report when tuning SQL statements identified in the plan table or trace/monitor report. There may be indexes that you can add, change, drop. Unused indexes create an overhead for Insert, Delete, and Update processing as well as utilities.

DDL or ERD

Lists relationships between tables, column data types, and data locations. An Entity Relationship Diagram (ERD) is the best tool for this information. If one is not available, print the Data Definition Language (DDL) SQL statements that are used to create the tables and indexes. If the DDL is not available, use a tool such as DB2LOOK to generate the DDL.

Do not overlook the importance of examining the application logic. This recommendation has to do primarily with the quantity of SQL statements being issued. The best performing SQL statement is the one that is never issued, and it is surprising how often application programs go to the database when they do not have to. The program may be executing the world's best-performing SQL statements, but if the data is not used, it means that they are poor-performing statements.

Chapter 7: Application Design and Tuning for Performance

Typically, you realize most performance improvements using proper application design or by tuning the application; however, nowhere else does the phrase *it depends* matter more than when dealing with applications. The performance design and tuning techniques that you apply vary depending upon the nature, needs, and characteristics of your application.

Avoiding Redundant Statements

The best performing statement is the statement that you never execute. Before you create and issue a statement, ask yourself the following questions:

- Is this statement necessary?
- Have I already read that data before?

One of the major issues with applications today is the quantity of SQL statements issued per transaction. The most expensive task you can perform is to leave your allied address space (DB2 distributed address space for remote queries) to go to the DB2 address spaces for an SQL request. Avoid this practice whenever possible.

Caching Data

If you are accessing code tables for validating data, editing data, translating values, or populating drop-down boxes, locally cache the code tables for better performance. This practice is relevant for any code tables that rarely change.

For example, a batch COBOL job reads your code tables into in-core tables. For larger code tables, employ a binary or quick search algorithm to quickly look up values. For CICS applications, set up the code in VSAM files, and use them as CICS data tables. This practice is much faster than using DB2 to look up the value for every CICS transaction. The VSAM files can be refreshed on regular intervals using a batch process or on an as-needed basis. If you are using a remote application or Windows-based clients, read the code tables once when the application starts and cache the values locally to populate various on-screen fields or to validate the data input on a screen.

For object- or service-oriented applications, check if an object has already been read in before reading it again. This practice helps you avoid constantly rereading the data every time a method is invoked to retrieve the data. If you are concerned that an object is not *fresh* and that you may be updating old data, employ a concept called *optimistic locking*. With optimistic locking, you do not have to constantly reread the object (the table or query to support the object). Instead, you read it once, and when you go to update the object, you can check the update timestamp to see if someone else has updated the object before you.

More Information:

Optimistic Locking (see page 121)

System-Generated Keys

You can reduce the quantity of SQL issued, as well as a major source of locking contention in your application, by using DB2 system-generated key values.

Traditionally, people have used a *next-key table* to generate keys. This method is where a table contains one row of one column with a numeric data type. This method typically involves reading the column, incrementing and updating the column, and then using the new value as a key to another table. These next-key tables often result in a significant bottleneck.

Several ways exist to generate key values inside DB2, two of which are identity columns (DB2 V8, DB2 9) and sequence objects (DB2 V8, DB2 9). Identity columns are attached to tables, and sequence objects are independent of tables.

The high performance solution for key generation in DB2 is the sequence object. Confirm that when you use sequence objects or identity columns you use the CACHE and ORDER settings according to your high performance needs. These settings affect the number of values that are cached in advance of a request and whether the order the values are returned is important. For example, the settings for a high level of performance in a data sharing group would be CACHE 50 NO ORDER.

When using sequence objects and identity columns, you can reduce the number of SQL statements your application issues by using a SELECT from a result table (DB2 V8 and DB2 9). In this case, the result table will be the result of an INSERT.

System-Generated Key Example

The following example shows the sequence object to generate a unique key and then returns that unique key to the application:

```
SELECT ACCT_ID
FROM FINAL TABLE
 (INSERT INTO UID1.ACCOUNT (ACCT_ID,NAME, TYPE, BALANCE)
  VALUES
  (NEXT VALUE FOR ACCT_SEQ,'Master Card','Credit',50000))
```

Avoiding Programmatic Joins

If you are using a modern generic application design with an access module for each DB2 table or if you are using an object-oriented or service-oriented design, you are using a form of programmatic joins. In almost every situation in which an SQL join can be coded instead of a programmatic join, the SQL join outperforms the programmatic join.

Consider the following trade-offs when you use SQL join versus a programmatic join:

- Flexible program and table access design versus improved performance for SQL joins
- Extra programming time for multiple SQL statements to perform single table access when required or joins when required versus the diminished performance of programmatic joins
- Extra programming that is required to add control breaks in the fetch loop versus the diminished performance of programmatic joins

DB2 provides the INSTEAD OF trigger (DB2 9) feature that allows mapping between the object world and multiple tables in a database. You can create a view to join two tables that are commonly accessed together. The object-based application then uses the view as a table. Because the view is on a join, it is a read-only view. However, you can define an INSTEAD OF trigger on that view to allow INSERTs, UPDATEs, and DELETEs against the view. You can code the INSTEAD OF trigger to perform the necessary changes to the tables of the join using the transition variables from the view.

In our tests of a simple two-table SQL join versus the equivalent programmatic join (FETCH loop within a FETCH loop in a COBOL program), the two-table SQL join exhibited 30 percent less CPU than the programmatic join.

Multi-Row Operations

To reduce the quantity of SQL statements the application issues, DB2 provides the following multi-row operations:

- Multi-row FETCH (DB2 V8, DB2 9)
- Multi-row INSERT (DB2 V8, DB2 9)
- MERGE statement (DB2 9)

In addition, you can perform a mass INSERT, UPDATE, or DELETE operation.

Multi-Row FETCH

Multi-row fetching lets you return up to 32,767 rows in a single API call with a potential CPU performance improvement near 50 percent. Multi-row fetching works for static or dynamic SQL, scrollable or non-scrollable cursors, and UPDATEs and DELETEs. The sample programs DSNTEP4, which is DSNTEP2 with multi-row FETCH, and DSNTIAUL also can exploit multi-row FETCH.

The following list highlights two key reasons to implement multi-row FETCH capability:

- Reduces the number of statements that are issued between your program address space and DB2
- Reduces the number of statements that are issued between DB2 and the DDF address space

The first way to take advantage of multi-row FETCH is to program for it in your application code. The second way to take advantage of multi-row FETCH is in your distributed applications that are using block fetching. After you are in compatibility mode in DB2 V8, the blocks that are used for block fetching are built using the multi-row capability without any code change on your part, which results in significant savings for your distributed SQLJ applications. In one situation, when using this feature for a remote SQLJ application migration from DB2 V7 to DB2 V8, the CPU *did not* increase.

The results of implementing this feature can affect performance. In our tests of a sequential batch program, the use of 50-row FETCH (the point of diminishing return for our test) of 39 million rows of a table reduced CPU consumption by 60 percent over single-row FETCH. In a random test where we expected on average 20 rows per random key, our 20-row FETCH used 25 percent less CPU than the single-row FETCH.

Note: Multi-row FETCH is a CPU saver, but it is not necessarily an elapsed time saver.

When using multi-row FETCH, the GET DIAGNOSTICS statement is not initially necessary. We recommend that you limit its use due to the high CPU overhead. Instead, use the SQLCODE field of the SQLCA to determine the status of your FETCH:

- Successful (SQLCODE 000)
- Failed (negative SQLCODE)
- Hit the end of file (SQLCODE 100)

If you receive an SQLCODE 100, check the SQLERRD3 field of the SQLCA to determine the number of rows to process.

How to Code for Multi-Row FETCH

Use the following process to code for multi-row FETCH:

1. Add the phrase WITH ROWSET POSITIONING to a cursor declaration.
2. Add the phrases NEXT ROWSET and FOR n ROWS to the FETCH statement.
3. Change the host variables to host variable arrays. For COBOL, add an OCCURS clause.
4. Place a loop within your FETCH loop to process the rows.

Multi-Row Insert

As with multi-row FETCH reading multiple rows per FETCH statement, a multi-row INSERT can insert multiple rows into a table in a single INSERT statement. The INSERT statement must contain the FOR *n* ROWS clause, and the host variables referenced in the VALUES clause must be host variable arrays. IBM states that multi-row INSERTs can result in as much as a 25 percent CPU savings over single row INSERTs. In addition, multi-row INSERTs can dramatically impact the performance of remote applications because the number of statements that are issued across a network can be reduced.

You can code the multi-row INSERT two ways:

ATOMIC

Specifies that if one INSERT fails, the entire statement fails

NOT ATOMIC

Specifies that any failure of any of the INSERTs impacts only that one INSERT of the set.

As with the multi-row FETCH, the GET DIAGNOSTICS statement is not initially necessary. We recommend that you limit its use due to the high CPU overhead. In the case of a failed NOT ATOMIC multi-row INSERT, you receive an SQLCODE of -253 if one or more of the INSERTs failed. Only then should you use GET DIAGNOSTICS to determine which one failed. If you receive an SQLCODE of zero, all INSERTs succeeded, and you do not need to perform further analysis.

MERGE Statement

Applications often interface with other applications. In these situations, an application may receive a large quantity of data that applies to multiple rows of a table. Typically, the application performs a blind update. That is, the application attempts to update the rows of data in the table, and if any update fails because a row was not found, the application inserts the data. In other situations, the application may read all of the existing data, compare the data to the new incoming data, and then programmatically insert or update the table with the new data.

DB2 9 supports this type of processing using the MERGE statement. The MERGE statement updates a target (table, view, or the underlying tables of a full select) using the specified input data. Rows in the target that match the input data are updated as specified, and rows that do not exist in the target are inserted. MERGE can use a table or an array of variables as input.

Because the MERGE operates against multiple rows, you can code it as ATOMIC or NOT ATOMIC. The NOT ATOMIC option allows rows that have been successfully updated or inserted to remain if others have failed. Use the GET DIAGNOSTICS statement with NOT ATOMIC to determine the updates or inserts that failed. As with the multi-row INSERT operation, limit the use of GET DIAGNOSTICS due to the high CPU overhead.

Employee Sample Table

The following example shows a merge of rows on the employee sample table:

```
MERGE INTO EMP AS EXISING_TBL
USING (VALUES (:EMPNO, :SALARY, :COMM, :BONUS)
       FOR :ROW-CNT ROWS) AS INPUT_TBL(EMPNO, SALARY, COMM, BONUS)
ON INPUT_TBL.EMPNO = EXISTING_TBL.EMPNO
WHEN MATCHED THEN
   UPDATE SET SALARY =  INPUT_TBL.SALARY
             ,COMM   =  INPUT_TBL.COMM
             ,BONUS  =  INPUT_TBL.BONUS
WHEN NOT MATCHED THEN
   INSERT (EMPNO, SALARY, COMM, BONUS)
   VALUES (INPUT_TBL.EMPNO, INPUT_TBL.SALARY, INPUT_TBL.COMM,
           INPUT_TBL.BONUS)
```

Placing Data-Intensive Business Logic in the Database

DB2 for z/OS offers many advanced features that let you take advantage of the power of the mainframe server. The following list details some key advanced features:

- Advanced and Complex SQL Statements
- User-Defined Functions (UDFs)
- Stored Procedures
- Triggers and Constraints

These advanced features let you quickly write applications by pushing some of the logic of the application into the database server. Most of the time advanced functionality can be incorporated into the database using these features at a much lower development cost than coding the feature into the application itself. A feature such as database enforced referential integrity (RI) is a good example of something that is relatively simple to implement in the database, but it would take longer time to code in a program.

These advanced database features also allow application logic to be placed as part of the database engine itself, making this logic more easily reusable enterprise wide. Reusing existing logic means faster time to market for new applications that need that logic, and having the logic that is centrally located makes it easier to manage than client code. Also, in many cases, having data-intensive logic on the database server results in improved performance because that logic can process the data at the server, and can return a result only to the client.

User-Defined Functions

Functions are a useful way of extending the programming power of the database engine. Functions let you push additional logic into your SQL statements. User-defined scalar functions work on individual values of a parameter list and return a single value result. A table function can return an actual table to an SQL statement for further processing, just like any other table. User-defined functions (UDFs) provide a major breakthrough in database programming technology. UDFs let developers and DBAs extend the capabilities of the database, which allows for more processing to be pushed into the database engine, which then allows these types of processes to become more centralized and controllable. Virtually any type of processing can be placed in a UDF, including legacy application programs.

When your processing is inside SQL statements, you can put the SQL statements anywhere, which means that anywhere you run your SQL statements, you can run your programs. Much like complex SQL statements, UDFs place more logic into the highly portable SQL statements.

Special Considerations for UDFs

UDFs can be a performance advantage or disadvantage. If the UDFs process large amounts of data, and return a result to the SQL statement, a performance advantage may be realized over the equivalent client application code. However, if a UDF is used to process only data, it can be a performance disadvantage, especially if the UDF is invoked many times or is embedded in a table expression, as data type casting (for SQL scalar UDFs compared to the equivalent expression coded directly in the SQL statement) and task switch overhead (external UDFs run in a stored procedure address space) are expensive (DB2 V8 relieves some of this overhead for table functions).

Converting a legacy program into a UDF in approximately a day, invoking that program from an SQL statement, and then placing that SQL statement where it can be accessed using a client process may be a financially prudent option. If the UDF results in the application program issuing fewer SQL statements or getting access to a legacy process, the UDF is the right decision.

Stored Procedures

A *stored procedure* is a procedure that is stored on the database server. Stored procedures are becoming more prevalent on the mainframe and can be part of a valuable implementation strategy. Stored procedures can be a performance benefit for distributed applications or a performance problem. In every good implementation, you must manage trade-offs. Most trade-offs involve sacrificing performance for attributes such as flexibility, reusability, security, and time to delivery. It is possible to minimize the impact of distributed application performance with the proper use of stored procedures.

Because stored procedures can encapsulate business logic in a central location on the mainframe, they offer a great advantage as a source of secured, reusable code. By using a stored procedure, you must have authority only to execute the stored procedures and do not need authority to the DB2 tables that are accessed from the stored procedures.

A properly implemented stored procedure can help improve availability. Stored procedures can be stopped, queuing all requestors. A change can be implemented while access is prevented, and the procedures can restart after the change occurs. If business logic and SQL access is encapsulated within stored procedures, you are less dependent on client or application server code for business processes. That is, you manage aspects such as display logic and edits, and the stored procedure contains the business logic. This practice simplifies the change process and makes the code more reusable. In addition, like user-defined functions (UDFs), stored procedures can access legacy data stores and quickly web-enable your legacy processes.

Special Considerations for Stored Procedures

You realize the major advantage of stored procedures when you implement them in a client/server application that must issue several remote SQL statements. The network overhead involved in sending multiple SQL commands and receiving result sets is significant; therefore, proper use of stored procedures to accept a request, process the request with encapsulated SQL statements and business logic, and return a result lessens traffic across the network and reduces the application overhead. If a stored procedure is coded in this manner, you realize a significant performance improvement. Conversely, if the stored procedures contain only a few SQL statements, you can realize the advantages of security, availability, and reusability, but performance will be worse than the equivalent single statement executions from the client due to the task switch overhead.

DB2 9 Enhancements

DB2 9 offers a significant performance improvement for stored procedures with the introduction of native SQL procedures. These unfenced SQL procedures execute as run-time structures rather than being converted into external C program procedures, as with DB2 V7 and DB2 V8. Running these native SQL procedures eliminates the task switch overhead of executing in the stored procedure address space. This practice represents a significant performance improvement for SQL procedures that contain little program logic and few SQL statements.

Triggers and Constraints

You can use triggers and constraints to move application logic into the database. The greatest advantage to triggers and constraints is that they are generally data intensive operations, and these types of operations are better performers when placed close to the data. These features consist of the following entities:

- Triggers
- Database Enforced Referential Integrity (RI)
- Table Check Constraints

A *trigger* is a database object that contains some application logic in the form of SQL statements that are invoked when data in a DB2 table is changed. These triggers are installed into the database, and then depend upon the table on which they are defined. SQL DELETE, UPDATE, and INSERT statements can activate triggers and replicate data, enforce certain business rules, and fabricate data. Database enforced RI can help ensure that relationships from tables are maintained automatically. Child table data cannot be created unless a parent row exists, and rules can be implemented to tell DB2 to restrict or cascade deletes to a parent when child data exists. Table check constraints help ensure values of specific table columns, and are invoked during LOAD, INSERT, and UPDATE operations.

Triggers and Constraints Advantages

Triggers and constraints ease the programming burden because the logic, in the form of SQL, is much easier to code than the equivalent application programming logic, which helps make the application programs smaller and easier to manage. In addition, because the triggers and constraints are connected to DB2 tables, centrally located rules are universally enforced, which helps to ensure data integrity across many application processes. You can also use triggers to automatically invoke user-defined functions (UDFs) and stored procedures, which can introduce automatic and centrally controlled intense application logic.

Triggers and constraints offer many benefits. They can provide better data integrity, faster application delivery time, and centrally located reusable code. Because the logic in triggers and constraints is data-intensive, their use typically outperforms the equivalent application logic. This is because no data has to be returned to the application when these automated processes fire. One trade-off for performance exists. When triggers, RI, or check constraints are used in place of application edits, performance may suffer. This trade-off is especially true if several edits on a data entry screen are verified at the server. The performance could be as bad as one trip to the server and back per edit. This action would seriously increase message traffic between the client and the server. For this reason, data edits are best performed at the client when possible.

When you are working with triggers, you must respect the triggers in relation to performance schema migrations or changes to the triggering tables. In some situations, the triggers have to be recreated in the same sequence that they were originally created. In certain situations, trigger execution sequence may be important, and if multiple triggers of the same type are against a table, they will be executed in the order that they were defined.

Organizing Your Input

You can help ensure the fastest level of performance for large-scale batch operations by confirming that the input data to the process is organized in a meaningful manner. That is, the data is sorted according to the cluster of the primary table. Better yet, all of the tables that are accessed by the large batch process should have the same cluster as the input. This could mean pre-sorting data into the proper order before processing. If you code a generic transaction processor that handles online and batch processing, you could be creating problems. If online is truly random, you can organize the tables and batch input files for the highest level of batch processing, and it should have little or no impact on the online transactions.

A locally executing batch process, which is processing the input data in the same sequence as the cluster or your tables, which are bound with RELEASE(DEALLOCATE) uses several performance enhancers, especially dynamic prefetch and index lookaside, to improve the performance of these batch processes.

Search Strategies

Search queries and driving cursors, which are large queries that provide the input data to a batch process, can be expensive. How you use queries presents a trade-off between the amount of program code you are willing to write and the performance of your application.

If you code a generic search query, you get generic performance. In the following example, the SELECT statement basically supports a direct read, a range read, and a restart read in one statement. To enable this type of generic access, you have to code a generic predicate. In most cases, this means that for every SQL statement issued more data will be read than is needed because DB2 has a limited ability to match these types of predicates.

In the following statement, the predicate supports a direct read, sequential read, and restart read for at least one part of a three-part compound key:

```
WHERE COL1 = WS-COL1-MIN
AND   ((    COL2 >= WS-COL2-MIN
        AND COL2 <=  WS-COL2-MAX
        AND COL3 >= WS-COL3-MIN
        AND COL3 <= WS-COL3-MAX)
       OR
        (      COL2 > WS-COL2-MIN
         AND   COL2 <= WS-COL2-MAX ))
OR    (    COL1 > WS-COL1-MIN
       AND COL1 <= WS-COL1-MAX )
```

These predicates are flexible; however, they are not the best performing. The predicate in this example most likely results in a non-matching index scan even though an index on COL1, COL2, COL3 is available, which means that the entire index will have to be searched each time that the query is executed. This is not a bad access path for a batch cursor that is reading an entire table in a particular order. For any other query, however, it is problematic. This is especially true for online queries that are actually providing three columns of data (all min and max values are equal). For larger tables, the CPU and elapsed time that is consumed can be significant.

Separate Predicates

We recommend that you use separate predicates for various numbers of key columns provided. This practice lets DB2 have matching index access for each combination of key parts provided.

First key access:

```
WHERE COL1 = WS-COL1
```

Two keys provided:

```
WHERE COL1 = WS-COL1
AND   COL2 = WS-COL2
```

Three keys provided:

```
WHERE COL1 = WS-COL1
AND   COL2 = WS-COL2
```

This practice increases the number of SQL statements that are coded within the program, but it also increases statement performance.

Boolean Term Predicate

By adding a redundant Boolean term predicate, you can enable DB2 to match on one column of the index. Therefore, for this WHERE clause, you can add a redundant predicate:

Before:

```
WHERE COL1 = WS-COL1-MIN
AND   ((    COL2 >= WS-COL2-MIN
        AND COL2 <=  WS-COL2-MAX
        AND COL3 >= WS-COL3-MIN
        AND COL3 <= WS-COL3-MAX)
       OR
        (      COL2 > WS-COL2-MIN
         AND   COL2 <= WS-COL2-MAX ))
OR    (    COL1 > WS-COL1-MIN
       AND COL1 <= WS-COL1-MAX )
```

After:

```
WHERE (COL1 = WS-COL1-MIN
AND    ((    COL2 >= WS-COL2-MIN
          AND COL2 <=  WS-COL2-MAX
          AND COL3 >= WS-COL3-MIN
          AND COL3 <= WS-COL3-MAX)
         OR
          (      COL2 > WS-COL2-MIN
           AND   COL2 <= WS-COL2-MAX ))
     OR (     COL1 > WS-COL1-MIN
          AND COL1 <= WS-COL1-MAX ))
AND   (    COL1 => WS-COL1-MIN
       AND COL1 <= WS-COL1-MAX )
```

The redundant predicate does not affect the result of the query, but it lets DB2 match on the COL1 column.

Name Searching

Name searching can be a challenge. You may be faced with multiple queries to solve multiple conditions or one large generic query to solve any request. In many cases, we recommend that you study the common input fields for a search, and then code specific queries that match those columns only and are supported by an index. The generic query can support the less frequently searched on fields.

Name Change

To search for two variations of a name to try and find someone in your database, code a name reversal. This query accomplishes your goal, but it uses multi-index access at its best.

```
SELECT PERSON_ID
FROM   PERSON_TBL
WHERE  (LASTNAME = 'RADY' AND
        FIRST_NAME = 'BOB') OR
       (LASTNAME = 'BOB' AND
        FIRST_NAME = 'RADY');
```

Probing the Table Twice

The following example gets better index access, but it will probe the table twice.

```
SELECT PERSON_ID
FROM   PERSON_TBL
WHERE  LASTNAME = 'RADY'
AND    FIRST_NAME = 'BOB'
UNION ALL
SELECT PERSON_ID
FROM   PERSON_TBL
WHERE  LASTNAME = 'BOB'
AND    FIRST_NAME = 'RADY'
```

Common Table Expression

The following example shows a common table expression (DB2 V8, DB2 9) to build a search list, and then divides that table into the person table. This query offers good index matching and reduced probes.

```
WITH PRSN_SEARCH(LASTNAME, FIRST_NAME) AS
(SELECT 'RADY', 'BOB' FROM SYSIBM.SYSDUMMY1
UNION ALL
SELECT 'BOB', 'RADY' FROM SYSIBM.SYSDUMMY1)
SELECT PERSON_ID
FROM   PERSON_TBL A, PRSN_SEARCH B
WHERE  A.LASTNAME = B.LASTNAME
AND    A.FIRST_NAME = B.FIRST_NAME
```

During Join Predicate

The following query uses a during join predicate to probe on the first condition and to apply the second condition only if the first does not find anything. That is, it will execute the second search only if the first finds nothing and completely avoid the second probe into the table. This query may not produce the same results as other queries:

```
SELECT COALESCE(A.PERSON_ID, B.PERSON_ID)
FROM      SYSIBM.SYSDUMMY1
LEFT OUTER JOIN
       PERSON_TBL A
ON IBMREQD = 'Y'
AND  (A.LASTNAME = 'RADY' OR A.LASTNAME IS NULL)
AND  (A.FIRST_NAME = 'BOB' OR A.FIRST_NAME IS NULL)
LEFT OUTER JOIN
(SELECT PERSON_ID
 FROM   PERSON_TBL
 WHERE  LASTNAME = 'BOB'  AND FIRSTNME = 'RADY') AS B
ON A.EMPNO IS NULL
```

Existence Checking

The best option for existence checking within a query depends on your situation, but these types of existence checks are always better resolved in an SQL statement than with separate queries in your program.

This section details recommendations in the following areas:

- Non-correlated subqueries (see page 119)
- Correlated subqueries (see page 119)
- Joins (see page 120)

Non-Correlated Subqueries

Non-correlated subqueries are generally a good option in the following scenarios:

- When an index for the inner select is not available, but an outer table column (indexable) does exist
- When an index does not exist on either inner or outer columns
- When a small amount of data is provided by the subquery

Note: DB2 can transform a non-correlated subquery to a join.

The following example shows a non-correlated subquery:

```
SELECT SNAME
FROM   S
WHERE  S# IN
(SELECT S# FROM SP
 WHERE   P# = 'P2')
```

Correlated Subqueries

Correlated subqueries are generally a good option in the following scenarios:

- When a supporting index is available on the inner select and cost benefit is realized in reducing repeated executions to the inner table and distinct sort for join.
- When the inner query could return a large amount of data if coded as a non-correlated subquery, provided a supporting index exists for the inner query.

The correlated subquery can outperform the equivalent non-correlated subquery if an index on the outer table is not used (DB2 chose a different index that is based upon other predicates) and one exists in support of the inner table.

The following example shows a correlated subquery:

```
SELECT SNAME
FROM   S
WHERE EXISTS
(SELECT * FROM SP
 WHERE SP.P# = 'P2'
 AND   SP.S# = S.S#)
```

Joins

Joins are generally a good option in the following scenarios:

- If supporting indexes are available and most rows hook up in the join
- If the join results in no extra rows returned then the DISTINCT can also be avoided

Joins can let DB2 pick the best table access sequence, and apply predicate transitive closure.

ORDER BY and FETCH

As of DB2 9, it is possible to code and ORDER BY and FETCH first in a subquery, which can provide even more options for existence checking in subqueries.

```
SELECT DISTINCT SNAME
FROM   S, SP
WHERE  S.S# = SP.S#
AND    SP.P# = 'P2'
```

For singleton existence checks, you can code FETCH FIRST and ORDER BY clauses in a singleton SELECT. This practice could provide the best existence checking performance in a stand-alone query:

```
SELECT 1 INTO :hv-check
FROM   TABLE
WHERE  COL1 = :hv1
FETCH  FIRST 1 ROW ONLY
```

Lock Avoidance Strategies

DB2 includes lock avoidance to reduce the overhead of always locking everything. DB2 checks to confirm that a lock is most likely necessary for data integrity before acquiring a lock. Lock avoidance is critical for performance. Application commits control its effectiveness. Lock avoidance is also used with isolation levels of cursor stability (CS) and repeatable read (RR) for referential integrity constraint checks. For a plan or package that is bound with RR, a page lock is required for the dependent page if a dependent row is found. This lock will be held to guarantee repeatability of the error on checks for updating primary keys when deleting is restricted.

You may need to bind your programs with CURRENTDATA(NO) and ISOLATION(CS) in DB2 V7 to allow for lock avoidance. In DB2 V8 and DB2 9, CURRENTDATA(YES) can also avoid locks. Although DB2 considers ambiguous cursors as read-only, it is always best to code your read-only cursors with the FOR FETCH ONLY or FOR READ ONLY clause.

Lock avoidance also needs frequent commits so that other processes do not have to acquire locks on updated pages. This practice also allows for page reorganization to occur to clear the *possibly uncommitted* (PUNC) bit flags in a page. Frequent commits allow the commit log sequence number (CLSN) on a page to be updated more often because it depends on the begin unit of recovery in the log, which is the oldest begin unit of recovery being required.

The best way to avoid taking locks in your read-only cursors is to read uncommitted. Use the WITH UR clause in your statements to avoid taking or waiting on locks; however, using WITH UR can result in the reading of uncommitted, or dirty, data that may eventually be rolled back. If you are using WITH UR in an application that will update the data, an optimistic locking strategy is your best performing option.

Optimistic Locking

With the high demands for full database availability, as well as high transaction rates and levels of concurrency, reducing database locks is always vitally important. To address this need, many applications employ a technique called *optimistic locking* to achieve these higher levels of availability and concurrency. This technique traditionally involves reading data with an uncommitted read or with cursor stability. Updated timestamps are maintained in all of the data tables. This updated timestamp is read along with all the other data in a row.

When a direct update is subsequently performed on the row that was selected, the timestamp is used to verify that no other application or user has changed the data between the point of the read and the update. This practice places additional responsibility on the application to use the timestamp on all updates, but the result is a higher level of DB2 performance and concurrency.

How Optimistic Locking Works

This section shows a hypothetical example to implement optimistic locking:

1. The application reads the data from a table to update it later.

```
SELECT UPDATE_TS, DATA1
FROM TABLE1
WHERE KEY1 = :WS-KEY1
WITH UR
```

2. The data is changed and the update occurs.

```
UPDATE TABLE1
SET DATA1 = :WS-DATA1, UPDATE_TS = :WS-NEW-UPDATE-TS
WHERE KEY1 = :WS-KEY1
AND UPDATE_TS = :WS-UPDATE-TS
```

   If the data has changed, the update receives an SQLCODE of 100, and restart logic has to occur for the update, which requires that all applications respect the optimistic locking strategy and they update the timestamp when updating the same table.

How to Use the ROWCHANGE Option

As of DB2 9, IBM provides built-in support for optimistic locking using the ROW CHANGE TIMESTAMP. When you create or alter a table, you can create a special column as a row change timestamp. DB2 automatically updates these timestamp columns whenever a row of a table is updated. This built-in support for optimistic locking relieves various applications of some of the responsibility of updating the timestamp.

ROWCHANGE Option Examples

The following code shows optimistic locking with the ROW CHANGE TIMESTAMP option:

```
SELECT ROW CHANGE TIMESTAMP FOR TABLE1, DATA1
FROM TABLE1
WHERE KEY1 = :WS-KEY1
WITH UR
```

The following code shows that the data has changed and update occurred:

```
UPDATE TABLE1
SET DATA1 = :WS-DATA1
WHERE KEY1 = :WS-KEY1
AND ROW CHANGE TIMSTAMP FOR TABLE1 = :WS-UPDATE-TS
```

Heuristic Control Tables and Commit Strategies

Implement heuristic control tables to allow flexibility in controlling concurrency and restartability. Processing has become more complex, tables have become larger, and requirements are now five nines availability (99.999 percent). To manage environments and their objects, we need dynamic control of everything. The number of different control tables and the indicator columns in them varies depending on the objects and the processing requirements.

Heuristic control and restart tables have rows unique to each application process to assist in controlling the commit scope using the number of database updates or time between commits as their primary focus. The table may also include an indicator to tell an application when to stop at the next commit point. These tables are accessed every time an application starts a unit-of-recovery (unit-of-work), which would be at process initiation or at a commit point.

The normal process is for an application to read the control table at the beginning of the process to get the dynamic parameters and commit time to be used. The table is then used to store information about the frequency of commits as well as any other dynamic information that is pertinent to the application process, such as the unavailability of some particular resource for a period. After the program is running, it updates and reads from the control table at commit time. Information about the status of all processing at the time of the commit is stored in the table so that a restart can occur at that point if necessary.

To be able to dynamically account for the differences in processes through time, the values in these tables can be changed through SQL in a program or by a production control specialist. For example, you would probably want to change the commit scope of a job that is running during the online day versus when it is running during the evening. You can also set indicators to tell an application to gracefully shut down, run different queries with different access paths due to a resource being taken down, or hibernate for a period.

Chapter 8: Tuning Subsystems

You can examine several areas in the DB2 subsystem for performance improvements. These areas are components of DB2 that aid in application processing. The components for tuning the subsystem are as follows:

- Buffer pools
- RID pool
- SORT pool
- Dynamic SQL caching
- Logging
- Locking and contention
- Thread management

Buffer Pools

Buffer pools are areas of virtual storage that temporarily store pages of tablespaces or indexes. Buffer pools are maintained by subsystem and, in some cases, also by application.

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 writes the data in the buffer back to disk at a DB2 system checkpoint or a write threshold. The write thresholds are a vertical threshold at the page set level or a horizontal threshold at the buffer pool level.

The proper tuning of the buffer pool operations increases application performance. The data manager issues GETPAGE requests to the buffer manager. The buffer manager uses the data from the buffer pool instead of having to retrieve the page from the disk. CPU is often traded for I/O to manage buffer pools efficiently.

DB2 buffer pool management lets the subsystem alter and display buffer pool information dynamically without requiring a bounce of the DB2 subsystem. This ability improves availability by letting you dynamically create new buffer pools when necessary and dynamically modify or delete buffer pools. You have to perform an ALTER of buffer pools several times a day because of varying workload characteristics.

Initial buffer pool definitions are set at installation and migration, but they are hard to configure during installation and migration because the application process against the objects is not detailed at that time. Use ALTER to add and delete new buffer pools, resize the buffer pools, or change any of the thresholds. The buffer pool definitions are stored in BSDS Boot Strap Dataset and you can move objects between buffer pools using an ALTER INDEX/TABLESPACE and a subsequent START/STOP command of the object.

Page Types in Virtual Pools

Three types of pages in virtual pools are available:

Available pages

Pages on an available queue LRU, FIFO, MRU that are available for stealing.

In-Use pages

In-use counts do not indicate the size of the buffer pool, but this count can help determine residency for initial sizing.

Updated pages

Pages that are not in-use, not available for stealing, and are considered dirty pages in the buffer pool that are waiting to be externalized.

Four page sizes and several buffer pools support each size:

```
BP0 - BP49                4K pages
BP8K0 - BP8K9          8K pages
BP16K0 - BP16K9        16K pages
BP32K0 – BP32K9        32K pages
```

A DSNZPARM called DSVCI lets the control interval match to the actual page size.

Asynchronous page writes per I/O changes with each page size accordingly:

```
  4K Pages             32 Writes per I/O
  8K Pages             16 Writes per I/O
16K Pages                8 Writes per I/O
32K Pages                4 Writes per I/O
```

With these new page sizes, you can achieve better hit ratios and have less I/O because you can fit more rows on a page. For example, if you have a 2200 byte row for a data warehouse, a 4 KB page would be able to hold only one row. If you used an 8 KB page, three rows could fit on a page.

Virtual Buffer Pools

You can have up to 80 virtual buffer pools. This feature allows up to any one of the following buffer pool sizes:

- Fifty 4-KB page buffer pools BP0 - BP49
- Ten 32-KB page buffer pools BP32K - BP32K9
- Ten 8-KB page buffer pools
- Ten 16-KB page buffer pools

The physical memory available on your system limits the size of the buffer pools, with a maximum size for all buffer pools of 1 TB. It does not use additional resources to search a large pool versus a small pool. If you exceed the available memory in the system, the system begins swapping pages from physical memory to disk. This swapping can severely affect performance.

Buffer Pool Queue Management

Pages used in the buffer pools are processed in two categories: Random, pages read one at a time, or Sequential, pages read by prefetch. These pages are queued separately as the Random Least Recently Used queue (LRU) or Sequential Least Recently Used queue (SLRU). The percentage of each queue in a buffer pool is controlled by the VPSEQT parameter.

This threshold is hard to adjust and often requires two settings -- one setting for batch processing and a different setting for online processing. The way that you process data between the batch and on the line processes often differs. Batch is more sequentially processed and online is processed more randomly.

DB2 breaks up these queues into multiple LRU chains. As a result, there is less overhead for queue management because the latch that is taken at the head of the queue will be latched less because the queues are smaller. Multiple subpools are created for a large virtual buffer pool and the threshold is controlled by DB2, not to exceed 4000 VBP buffers in each subpool. The LRU queue is managed within each of the subpools to reduce the buffer pool latch contention when the degree of concurrency is high. Stealing these buffers occurs in a round-robin through the subpools.

First-in first-out (FIFO) can also be used instead of the default of LRU. FIFO always moves the oldest pages out of the queue. Using FIFO decreases the cost of doing a GETPAGE operation and reduces internal latch contention for high concurrency. However, you should use FIFO only where there is little or no I/O and where table space or index is resident in the buffer pool.

You will have separate buffer pools with LRU and FIFO objects. You can set separate buffer pools using the ALTER BUFFERPOOL command and specifying the PGSTEAL option for FIFO. LRU is the PGSTEAL option default.

I/O Requests and Externalization

Synchronous reads are physical pages that are read in one page per I/O. Synchronous writes are pages written one page per I/O. Minimize synchronous read and writes to only what is necessary. If you do not, you will begin to see buffer pool stress. DB2 will begin to use synchronous writes if the IWTH threshold is reached or if two system checkpoints pass without a page being written that has been updated and not yet committed.

Asynchronous reads are several pages that are read per I/O for prefetch operations such as the sequential prefetch, dynamic prefetch, or list prefetch. Asynchronous writes are several pages per I/O for such operations as deferred writes.

Pages are externalized to disk when one of the following occurs:

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

You can control page externalization by using your DWQT and VDWQT thresholds for best performance and can avoid surges in I/O. You 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 work files.

Checkpoints and Page Externalization

DB2 checkpoints are controlled using DSNZPARM and CHKFREQ. The CHKFREQ parameter is the number of minutes between checkpoints for a value of 1 to 60 or the number of log records written between DB2 checkpoints for a value of 200 to 16,000,000. The default value is 500,000. You may need different settings for this parameter depending on your workload. For example, you may want it higher during batch processing.

CHKFREQ is a hard parameter to set because it requires a bounce of the DB2 subsystem to take effect. Use the SET LOG CHKTIME command to dynamically set the CHKFREQ parameter.

Use the SET LOG command option to SUSPEND and RESUME logging for a DB2 subsystem. SUSPEND causes a system checkpoint to be taken in a non-data sharing environment. By obtaining the log-write latch, any further log records are prevented from being created and any unwritten log buffers will be written to disk. The BSDS will be updated with the high-written RBA. All further database updates are prevented until update activity is resumed by issuing a SET LOG command to RESUME logging or until a STOP DB2 command is issued. These are single-subsystem only commands, so they will have to be entered for each member when running in a data sharing environment.

During online processing, DB2 should checkpoint about every 5 through 10 minutes or some other value that is based on investigative analysis of the impact on restart time after a failure. Two concerns for how often you take checkpoints are as follows:

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

The costs and disruption of DB2 checkpoints are often overstated. Although a DB2 checkpoint is a small task, it does not prevent processing from proceeding. Having a CHKFREQ setting that is too high along with large buffer pools and high thresholds such as the defaults can cause enough I/O to make the checkpoint disruptive. In trying to control checkpoints, some users increased the CHKFREQ value and made the checkpoints less frequent but made them much more disruptive. The situation is corrected by reducing the amount that is written and increasing the checkpoint frequency which yields much better performance and availability. For some installations, a checkpoint every minute did not affect performance or availability.

The write efficiency at DB2 checkpoints is the key factor that you have to observe to see if CHKFREQ can be reduced. If the write thresholds (DWQT/VDQWT) are doing their job, less work is performed at each checkpoint. Using the write thresholds to cause I/O to be performed in a level non-disruptive way is helpful for the non-volatile storage in storage controllers.

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

Buffer Pool Sizing

The VPSIZE parameter determines buffer pool sizes. This parameter determines the number of pages to be used for the virtual pool. DB2 can handle large buffer pools as long as enough memory is available. If insufficient storage exists to back the buffer pool storage that is requested, paging can occur. Paging can occur when the buffer pool size exceeds the available memory on the z/OS image. DB2 limits the total amount of storage that is allocated for buffer pools to approximately twice the amount of storage. There is a maximum of 1-TB total for all buffer pools, provided the storage is available.

To size buffer pools, it is helpful to know the residency rate of the pages for the object in the buffer pool.

Sequential Versus Random Processing

VPSEQT is the percentage of the virtual buffer pool that can be used for sequentially accessed pages. This percentage prevents sequential data from using all the buffer pool and to keep some space available for random processing. You can set this value from 0 to 100 percent with the default being 80 percent, which indicates that 80 percent of the buffer pool is to be set aside for sequential processing and 20 percent for random processing. Set this parameter according to how your objects in that buffer pool are processed.

One tuning option that is used is altering the VPSEQT to 0 percent to set up the pool for random use only. When VPSEQT is set at 0 percent, the SLRU will no longer be valid and the buffer pool is now random. Because only the LRU used, all pages on the SLRU have to be freed. This setting also disables prefetch operations in this buffer pool, which is beneficial for certain strategies.

Write Thresholds

The deferred write threshold is the percentage threshold that determines when DB2 starts turning on write engines to begin deferred writes. The value can be from 0 to 90 percent. When the threshold is reached, write engines begin writing pages out to disk.

Running out of write engines can occur if the write thresholds are not set to keep a constant flow of updated pages being written to disk. DB2 turns on these write engines one vertical pageset queue at a time until a 10 percent reverse threshold is met. Check the WRITE ENGINES NOT AVAILABLE indicator on Statistics report to find out when DB2 ran out of write engines.

When setting the DWQT threshold, a high value is useful to help improve the hit ratio for updated pages, but it increases I/O time when deferred write engines begin. Using a low value to reduce I/O length for deferred write engines increases the number of deferred writes. Set this threshold based on the referencing of the data by the applications.

If you choose to set the DWQT threshold to 0 so that all objects defined to the buffer pool are scheduled to be written immediately to disk, DB2 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 written per I/O, but it takes 40 updated pages to trigger the threshold so that the highly re-referenced updated pages remain in the buffer pool.

When implementing LOBs (Large Objects), a separate buffer pool should be used and this buffer pool should not be shared. The DWQT threshold should be set to 0 so that LOBs with LOG NO force-at-commit processing occurs and the updates continually flow to disk instead of surges of writes. For LOBs defined with LOG YES, DB2 could use deferred writes and could avoid massive surges at the checkpoint.

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, control writes at the pageset partition level by using the VDWQT, which is the percentage threshold that determines when DB2 starts turning on write engines and begins the deferred writes for a data set. This practice helps to keep a particular pageset partition from monopolizing the entire buffer pool with its updated pages. The value is 0 to 90 percent with a default of 10 percent. The VDWQT should always be less than the DWQT.

When setting the buffer pool writes, if fewer than ten pages are written per I/O, set the deferred write threshold to 0. You may also want to set it to 0 to trickle write the data out to disk. It is best to keep this value low to prevent heavily updated pagesets from dominating the section of the deferred write area. A percentage of pages or actual number of pages, from 0 through 9999, can be specified for VDWQT. You must set the percentage to 0 to use the number that is specified for VDWQT. Set to 0,0 and system uses MIN(32,1%) for good trickle I/O.

Set VDWQT using a number rather than a percentage because if someone increases the buffer pool, more pages for a particular pageset can occupy the buffer pool, and this result may not always be optimal.

When looking at any performance report showing the amount of activity for VDWQT and DWQT, you want to see VDWQT being triggered most of the time and DWQT less. There can be no general ratios because that would depend on the activity and the number of objects in the buffer pools. You want to be controlling I/O by VDWQT with DWQT watching for and controlling activity across the entire pool and writing out rapidly queuing up pages. This practice also assists in limiting the amount of I/O that checkpoints would have to perform.

Buffer Pool Parallelism

Virtual Pool Parallel Sequential Threshold (VPPSEQT) is the percentage of VPSEQT setting that can be used for parallel operations. The value is 0 through 100 percent with a default of 50 percent. If this is set to 0, parallelism is disabled for objects in that particular buffer pool. This setting can be useful in buffer pools that cannot support parallel operations.

Virtual Pool Sysplex Parallel Sequential Threshold (VPXPSEQT) is a percentage of the VPPSEQT to use for inbound queries. VPXPSEQT also defaults to 50 percent and if it is set to 0, Sysplex Query Parallelism is disabled when originating from the member the pool is allocated to. In affinity data sharing environments, this is set to 0 percent to prevent inbound resource consumption of work files and buffer pools.

Page Fixing

You can use the page fixing keyword, PGFIX, with the ALTER BUFFERPOOL command to fix a buffer pool in storage for an extended period. The PGFIX keyword has the following options:

PGFIX(YES)

Specifies that the buffer pool is fixed in real storage for the long term. Page buffers are fixed when they are first used and remain fixed.

PGFIX(NO)

Specifies that the buffer pool is not fixed in real storage for the long term. Page buffers are fixed and unfixed in real storage, allowing for paging to disk. PGFIX(NO) is the default option.

We recommend that you use PGFIX(YES) for buffer pools with a high I/O rate (a high number of pages that are read or written). For buffer pools with zero I/O, such as some read-only data or some indexes with a nearly 100 percent hit ratio, we do not recommend PGFIX(YES) because it does not provide a performance advantage.

Internal Thresholds

The following thresholds are a percent of unavailable pages to total pages, where unavailable means either updated or in use by a process.

SPTH

The Sequential Prefetch Threshold (SPTH) is checked before a prefetch operation is scheduled and during buffer allocation for a previously scheduled prefetch. If the SPTH threshold is exceeded, prefetch will not be scheduled or will be canceled.

PREFETCH DISABLED – NO BUFFER will be incremented every time a virtual buffer pool reaches 90 percent of active unavailable pages, disabling sequential prefetch. This value should always be 0. If this value is not 0, you are probably experiencing degradation in performance due to all prefetch being disabled. To eliminate this degradation, increase the size of the buffer pool or have more frequent commits in the application programs to free pages in the buffer pool because this puts the pages on the write queues.

DMTH

The Data Manager Threshold (DMTH) occurs when 95 percent of all buffer pages are in use. The Buffer Manager requests all threads to release any possible pages immediately. This action occurs by setting GETPAGE/RELPAGE processing by row instead of page. After a GETPAGE and single row is processed, a RELPAGE is issued. This sequence causes CPU to become high for objects in that buffer pool and I/O sensitive transaction can suffer. This result can occur if the buffer pool is too small. You can observe when this occurs by seeing a non-zero value in the DM THRESHOLD REACHED indicator on a statistics report. This is checked every time that a page is read or updated. If this threshold is not reached, DB2 accesses the page in the virtual pool once for each page. If this threshold has been reached, DB2 accesses the page in the virtual pool once for every ROW on the page that is retrieved or updated.

IWTH

The Immediate Write Threshold (IWTH) is reached when 97.5 percent of the buffers are in use. Reaching this threshold causes synchronous writes to begin, which presents a performance problem. For example, if there are 100 rows in page and there are 100 updates, 100 synchronous writes occur, one by one, for each row. Synchronous writes are not concurrent with SQL, but serial, so the application will be waiting while the write occurs including 100 log writes that must occur first. This activity causes large increases in I/O time. It is not recorded explicitly in a statistic report, but DB2 will appear to be hung and you will see synchronous writes begin to occur when this threshold is reached. Some monitors will send exception messages to the console when synchronous writes occur and refers to them as IWTH reached; however, not all synchronous writes are caused by this threshold being reached. This is simply being reported incorrectly.

Note: Some performance reports, the IWTH counter can also be incremented when dirty pages are on the write queue have been re-referenced, which has caused a synchronous I/O before the page could be used by the new process. This threshold counter can also be incremented if more than two checkpoints occur before an updated page is written because this action causes a synchronous I/O to write out the page.

Virtual Pool Design Strategies

Use separate buffer pools that are based on their type of usage by the applications. Each one of these buffer pools has its own unique settings and the type of processing may even differ between batch and on line.

Detailed Example of Buffer Pool Object Breakouts

```
BP0             Catalog and directory - DB2 only use
BP1             Work files (Sort)
BP2             Code and reference tables - heavily accessed
BP3             Small tables, heavily updated - trans tables, work tables
BP4             Basic tables
BP5             Basic indexes
BP6             Special for large clustered, range scanned table
BP7             Special for Master Table full index (Random searched table)
BP8             Special for an entire database for a special application
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            Vendor tool/utility object
```

Buffer Pool Tuning

Buffer pools can be tuned effectively using the DISPLAY BUFFER POOL command. When a tool is not available for tuning, the following items can be used to help tune buffer pools.

- Use command and view statistics.
- Make changes (for example, thresholds, size, object placement).
- Use command again during processing and view statistics.
- Measure statistics.

The output from the following items contains valuable information about the tuning information:

- Sequential, List, Dynamic Requests
- Pages Read
- Prefetch I/O and Disablement

The incremental detail display shifts the time frame every time a new display is performed.

RID Pool

The 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 RID pool is checked by the DB2 optimizer for the prefetch and RID use. The full use of RID POOL is possible for any single user at the run time. Run time can result in a tablespace scan being performed if not enough space is available in the RID. For example, if you want to retrieve 10,000 rows from a 100,000,000 row table and there is no RID pool available, a scan of 100,000,000 rows would occur at any time and without external notification. The DB2 optimizer assumes physical I/O will be less with a large pool.

RID Pool Size

The MAXRBLK installation parameter controls the size of the RID pool. The default size of the RID pool is 8 MB with a maximum size of 10000 MB. Setting the RID pool to 0 disables the types of operations that use the RID pool, and DB2 would not choose access paths that the RID pool supports.

The RID pool is created at startup time, but no space is allocated until RID storage is needed. Space is then allocated in 32-KB blocks as needed until the maximum size you specified during installation is reached.

Use the following guidelines when setting the RID pool size:

- Set the RID pool as large as required to benefit processing.
- Setting the RID pool size too small can lead to performance degradation.

Use this formula for a more specific RID size setting:

- Number of concurrent RID processing activities times average number of RIDs times 2 times 5 bytes per RID.

RID Pool Statistics to Monitor

The three statistics to monitor for RID pool problems are as follows:

RIDs Over the RDS Limit

Indicates the number of times list prefetch is turned off because the RID list that is built for a single set of index entries is greater than 25 percent of the number of rows in the table. If this is the case, DB2 determines that, instead of using the 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 does not help in this case. This is an application issue for access paths and has to be evaluated for queries using list prefetch.

There is one very critical issue regarding this type of failure. The 25 percent threshold is stored in the package/plan at bind time; therefore, it may no longer match the real 25 percent 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, rebinding the packages/plans that are dependent on them should be rebound after a RUNSTATS utility has updated the statistics. Key correlation statistics and better information about skewed distribution of data can also help to gather better statistics for access path selection and may help avoid this problem.

RIDs Over the DM Limit

RIDs over the DM limit occur when over 28 million RIDs are required to satisfy a query. DB2 has a 28 million RID limit. The consequences of hitting this limit can be fallback to a table space scan. To control this issue, you have several options:

- Fix the index by doing something creative.
- Add an index better that is suited for filtering.
- Force list prefetch off and use another index.
- Rewrite the query.
- Use a table space scan.

Insufficient Pool Size

Indicates that the RID pool is too small.

The SORT Pool

Sorts are performed in phases:

- Initialization
- DB2 builds ordered sets of runs from the given input
- Merge
- DB2 merges the runs together

At startup, DB2 allocates a sort pool in the private area of the DBM1 address space. DB2 uses a special sorting technique that is 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. If the sort cannot complete in the sort pool, the runs are moved into the work file database. These runs are later merged to complete the sort.

Note: When the work file database is 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.

Sort Pool Size

Larger sort pools produce fewer sort runs. If the sort pool is large enough, the buffer pools and sort work files may not be used. You want to set a large size for the sort pool and work file database because you do not want sorts to have pages being written to disk.

Note: If buffer pools and work file database are not used, the better performance will be due to less I/O.

The sort pool size defaults to 2 MB unless a different size is specified. The sort pool can range in size from 240 KB to 128 MB. The size of the sort pool is set during installation using DSNZPARM.

Small Sort Pool Size

The main goal for the EDM pool is to limit the I/O against the directory and catalog. If the sort pool is too small, you see increased I/O activity in the following DB2 table spaces that support the DB2 directory:

- DSNDB01.DBD01
- DSNDB01.SPT01
- DSNDB01.SCT02

If the sort pool is too small, you also see increased response times due to the loading of the SKCTs, SKPTs, and DBDs, and re-preparing the dynamic SQL statements because they could not remain cached. By correctly sizing the EDM pools, you can avoid unnecessary I/Os from accumulating for a transaction. Additional I/O is incurred if a SKCT, SKPT, or DBD has to be reloaded into the EDM pool. This action can happen if the sort pool pages are stolen because the EDM pool is too small. Pages in the sort pool are maintained on an LRU queue, and the least recently used pages get stolen if necessary. You can use a DB2 performance monitor statistics report to track the statistics about EDM pool use.

Note: If a new application is migrating to the environment, it may be helpful to look in SYSIBM.SYSPACKAGES to give you an idea of the number of packages that may have to exist in the EDM pool, which can help determine the size of the sort pool.

Environmental Descriptor Manager Pool Efficiency

You can measure the following ratios to help us determine if your EDM pool is efficient.

The EDM pool hit ratios are as follows:

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

You want a value of 5 for each of them and 80 percent hit ratio.

Dynamic SQL Caching

When you use dynamic SQL caching, note your EDM statement cache pool size. Cached statements are not backed by disk and if its pages are stolen and the statement is reused, it will have to be prepared again. Static plans and packages can be flushed from EDM by LRU, but they are backed by disk and can be retrieved when used again.

There are statistics to help monitor cache use and trace fields show effectiveness of the cache and can be seen on the Statistics Long Report. In addition, the dynamic statement cache can be snapped with the EXPLAIN STMTCACHE ALL statement. The results of this statement are placed in the DSN_STATEMENT_CACHE_TABLE.

In addition to the global dynamic statement caching in a subsystem, an application can also cache statements at the thread level with the KEEPDYNAMIC(YES) bind parameter in combination with not re-preparing the statements. In these situations, the statements are cached at the thread level in thread storage and at the global level. If no shortage of virtual storage exists, the local application thread level cache is the most efficient storage for the prepared statements.

Note: You can use the MAXKEEPD subsystem parameter to limit the amount of thread storage that is consumed by applications caching dynamic SQL at the thread level.

Logging

Every system has some component that eventually becomes the final bottleneck. Do not overlook logging when trying to get transactions through the systems in a high performance environment. Logging can be tuned and refined, but the synchronous I/O associated with logging and commits will always be there.

Log Reads

When DB2 has to read from the log, it is important that the reads perform well because reads are typically performed during recovery, restarts, and rollbacks. An input buffer must be dedicated for every process requesting a log read. DB2 first looks for the record in the log output buffer. If DB2 finds the record, it can apply it directly from the output buffer. If the record is not in the output buffer, DB2 looks for it in the active log data set and then in the archive log data set. When the record is found, it is moved to the input buffer so it can be read by the requesting process.

You can monitor the successes of reads from the output buffers and active logs in the statistics report. These reads are the better performers. If the record has to be read in from the archive log, the processing time is extended. For this reason, it is important to have large output buffers and active logs.

Log Writes

Applications move log records to the log output buffer using two methods:

No wait

Moves the log record to the output buffer and returns control to the application. If there are no output buffers available, the application will wait and the statistics report UNAVAILABLE ACTIVE LOG BUFF will have a non-zero value. These results mean that DB2 had to wait to externalize log records because no output log buffers were available. Successful moves without a wait are recorded in the statistics report under NO WAIT requests.

Force

Occurs at commit time and the application waits during this process, which is considered a synchronous write.

Log records are then written from the output buffers to the active log data sets on the disk either synchronously or asynchronously. The WRITE OUTPUT LOG BUFFERS in the statistics report contains information about the write operation.

To improve the performance of the log writes, you can increase the number of output buffers available for writing active log data sets that are performed by changing an installation parameter OUTBUFF.

Locking and Contention

DB2 uses locking to ensure consistent data based on user requirements and to avoid losing data updates. You must balance the need for concurrency with the need for performance. You want to minimize the following items:

Suspension

An application process is suspended when it requests a lock that is already held by another application process and cannot be shared. The suspended process temporarily stops running.

Time out

An application process times out when it is terminated because the process has been suspended for longer than a preset interval. DB2 terminates the process and returns a -911 or -913 SQL code.

Deadlock

A deadlock occurs when two or more application processes hold locks on resources that the others need and without which they cannot proceed.

Use the following methods to monitor DB2 locking problems:

- Use the -DISPLAY DATABASE command to find out what locks are held or waiting at any moment.
- Use EXPLAIN to monitor the locks that are required by a particular SQL statement or all the SQL in a particular plan or package.
- Activate a DB2 statistics class 3 trace with IFCID 172. This report outlines all the resources and agents involved in a deadlock and the significant locking parameters, such as lock state and duration, which is related to their requests.
- Activate a DB2 statistics class 3 trace with IFCID 196. This report shows detailed information on timed-out requests, including the holders of the unavailable resources.

The way DB2 issues locks is complex. Lock issuance depends on the type of processing being done, the LOCKSIZE parameter that is specified when the table was created, the isolation level of the plan or package being executed, and the method of data access.

Thread Management

The thread management parameters that are set during installation controls how many threads can be connected to DB2 and it also determines the main storage size needed. Improperly allocating these parameters during installation directly affects main storage usage. If the allocation is too high, storage is wasted. If the allocation is too low, performance degradation occurs because users are waiting for available threads.

You can use a DB2 performance trace record with IFCID 0073 to retrieve information about how often thread create requests wait for available threads. Starting a performance trace can involve the substantial overhead, so be sure to qualify the trace with specific IFCIDS, and other qualifiers, to limit the data collected.

The MAX USERS field during installation specifies the maximum number of allied threads that can be allocated concurrently. These threads include TSO users, batch jobs, IMS, CICS, and tasks using the call attachment facility. The maximum number of threads that can be accessing data concurrently is the sum of this value and the MAX REMOTE ACTIVE specification. When the number of users trying to access DB2 exceeds your maximum, plan allocation requests are queued.

The DB2 Catalog and Directory

The DB2 catalog and directory should be on separate volumes, and the volumes must be dedicated to the catalog and directory. If you have indexes on the DB2 catalog, place them on a separate volume and the DB2 catalog and directory into their own buffer pool.

You can reorganize the catalog and directory. You should periodically run RUNSTATs on the catalog and analyze the appropriate statistics that let you know when you have to reorganize.

Chapter 9: Understand and Tune Your Packaged Applications

Today, many organizations do not retain the manpower resources to build and maintain custom applications. In many situations, these organizations rely on pre packaged software solutions to help run their applications and automate business activities such as accounting, billing, customer management, inventory management, and human resources. These packaged applications are referred to as enterprise resource planning (ERP) applications.

Database Utilization and Packaged Applications

ERP applications are generic by design. They are also typically database agnostic, meaning that they are not written to a specific database vendor or platform. They are written with a generic set of standardized SQL statements that can work on various database management systems and platforms. Assume that your ERP application is taking advantage of few, if any, of the DB2 SQL performance features, or specific database performance features, tables, or index designs.

These packaged applications are typically written using an object-oriented (OO) design and are created in such a way that the organization implementing the software can customize the application. Packaging and creating the applications like this increases flexibility because many of the tables in the ERP applications' database can be used in different ways and for various purposes.

With great flexibility comes the potential for reduced database performance. OO design may lead to an increase in SQL statements issued, and the flexibility of the implementation could affect table access sequence, random data access, and mismatching of predicates to indexes.

When you purchase an application, you typically have no access to the source code or the SQL statements issued. Sometimes, a vendor may change an SQL statement that is based on information you provided about a performance problem, but that is unusual.

Your focus should be on getting the database organized in a manner that best accommodates the SQL statements before tuning the subsystem to provide the highest level of throughput. Even though you cannot change the SQL statements, you should first look at those statements for potential performance problems that can be improved by changing the database or subsystem.

Find and Fix SQL Performance Problems

The first thing to do when working with a packaged application is to determine where the performance problems are. Most likely, you will be unable to change the SQL statements because doing so requires the vendor to make a change to improve a query for you, and not negatively impact performance for other customers. Even though you cannot change the SQL statements, you can change subsystem parameters, memory, and can change the tables and indexes for performance. Before you begin your tuning effort, decide if you are tuning to improve the response time for end users or to save CPU dollars.

The best way to find the programs and SQL statements that have the potential for elapsed and CPU time savings is to use the technique for packaged applications that use static embedded SQL statements in the Overall Application Performance Monitoring section. If your packaged application uses dynamic SQL, this technique is less effective, and you should use one or more of the techniques in the Recommendations for Distributed Dynamic SQL section.

When the application uses static or dynamic SQL, it is best to capture all the SQL statements being issued, and catalog them in a document or a DB2 table. To do so, query SYSIBM.SYSSTMT table by plan for statements in a plan, or the SYSIBM.SYSPACKSTMT table by package for statements in a package. You can query these tables using the plan or package names corresponding to the application.

If the application uses dynamic SQL, you can capture the SQL statements by using EXPLAIN STMTCACHE ALL (DB2 V8 or DB2 9), or by running a trace (DB2 V7, DB2 V8, DB2 9). When you run a trace, expect to parse through a significant amount of data. By capturing these dynamic statements, you see only the statements executing during the time you monitored. For example, using EXPLAIN STMTCACHE ALL statement captures only the statements that reside in the dynamic statement cache when the statement executes.

After determining the packages and statements that consume the most resources or have the highest elapsed time, you can begin your tuning effort by tuning the subsystem and database. Subsystem tuning has no impact on the application and database design. However, if you change the database, consider that future product upgrades or releases can undo the changes.

Subsystem Tuning for Packaged Applications

The easiest way to tune packaged applications is to tune the subsystem by changing some subsystem parameters or increasing memory. The impact of the changes can be immediate and when the subsystem is poorly configured the impact can be dramatic. Refer to the Tuning Subsystems section for additional information about tuning the following components:

Dynamic Statement Cache

When the application uses dynamic SQL, make sure that the dynamic statement cache is enabled with enough memory to avoid binding on PREPARE. You can monitor the statement cache hit ratio by looking at a statistics report. Statement prepare time can be a significant contributor to overall response time, and you want to minimize response time. If the statement prepare time is a significant factor in query performance, consider adjusting some DB2 installation parameters that control the bind time. These parameters, MAX_OPT_STOR, MAX_OPT_CPU, MAX_OPT_ELAP, and MXQBCE (DB2 V8, DB2 9) are known as hidden installation parameters and help control statement prepare costs.

Note: Adjust these parameters only under the advice of IBM.

RID Pool

Look at your statistics report to see how much the RID pool is used, and if RID failures occur due to lack of storage. When RID failures occur due to lack of storage, increase the size of the RID pool.

DB2 Catalog

The DB2 System Catalog should be in its own buffer pool. If the application uses dynamic SQL, this pool should be large enough to avoid I/O for statement binds.

Work File Database and Sort Pool

When your packaged application does numerous sorting, make sure that your sort pool is properly sized to deal with lots of smaller sorts. You should also have several work file table spaces that are created to avoid resource contention between concurrent processes.

Memory and DSMAX

Packaged applications typically have many objects. You can monitor to see which objects are most commonly accessed, and make those objects CLOSE NO. Make sure that your DSMAX installation parameter is large enough to avoid any closing of data sets. If you have enough storage make all the objects CLOSE NO. You can also make sure your output log buffer is large enough to avoid shortages if the application changes data often. Consider using the KEEPDYNAMIC bind option for the packages, but keep in mind that this can increase thread storage, and you have to balance the number of threads with the amount of virtual storage that is consumed by the DBM1 address space. Make sure that your buffer pools are page fixed.

Note: For additional information, see the IBM redbook, *DB2 UDB for z/OS V8: Through the Looking Glass and What SAP Found There*.

Table and Index Design Options for High Performance

Review the Designing Tables and Indexes for Performance section for the general table and index design for performance.

You can focus on certain areas for your packaged applications, such as finding the portions of the application that have the biggest impact on performance. The techniques outlined in the Predicates and SQL Tuning and Monitoring section can help. Finding the packages and statements that present themselves as performance problems can then lead you to the most commonly accessed tables. Examining how these tables are accessed by the programs such as the DISPLAY BUFFERPOOL command, the accounting report, the statistics report, performance trace, and EXPLAIN, can help identify potential changes for performance.

During your examination, ask these questions:

- Are many random pages accessed?
- Are matching columns of an index scan not matching on all available columns?
- Are more columns in a WHERE clause than in the index?
- Is there repetitive table access?
- Are the transaction queries using list prefetch?

If you find issues like these, make some of the following changes to the indexes and clustering:

- **Indexes**
  - Customize the generic indexes to fit your needs. The easiest thing to do is to add indexes to support poor performing queries. Adding indexes impacts inserts, deletes, possibly updates, and some utilities, which is why having a catalog of all the statements is important.
  - Change the order of index columns to match your most common queries to increase the value of match cols.
  - Add columns to indexes to increase the value of match cols.
  - Change indexes that contain variable length columns to use the NOT PADDED (DB2 V8, DB2 9) option to improve performance of those indexes.
- **Clustering**

  Because the packaged application can anticipate your most common access paths, you could have a problem with clustering. Understanding how the application accesses the tables is important.

  You can look at joins that occur commonly and can consider changing the clustering of those tables so that they match. Although chaining clustering can have a dramatic impact on the performance of these joins, make sure that no other processes are negatively impacted.

  If tables are partitioned for a certain reason, consider clustering within each partition (DB2 V8, DB2 9) to keep the separation of data by partition, while improving the access within each partition. Look for page sets that have a high percentage of randomly accessed pages in the buffer pool as a potential opportunity to change clustering.

Any change that you make to the database can be undone on the next release of the software. Therefore, document every change completely.

Monitoring and Maintaining Your Objects

The continual health of your objects is important. Keep a list of commonly accessed objects, especially those that are changed often, and begin a policy of frequently monitoring these objects.

Frequently monitoring commonly accessed objects includes regularly running the RUNSTATS utility on these frequently changing indexes and table spaces. Performance can degrade quickly as these objects become disorganized. Set up a regular strategy of RUNSTATS, REORGs, and REBINDs (for static SQL) to maintain performance.

Understanding the quantity of inserts to a table space can also guide you to adjusting the free space. Proper PCTFREE for table space can help avoid expensive exhaustive searches for inserts. Proper PCTFREE is important for indexes where inserts are common. Because these applications should not split pages, set the PCTFREE high, especially if little or no sequential index scanning exists.

Real-Time Statistics

You can use DB2's ability to collect statistics in real time to help monitor the activity against your packaged application objects. Real-time statistics lets DB2 collect statistics on table spaces and index spaces and periodically write this information to two user-defined tables. Beginning with DB2 9, these tables are an integral part of the system catalog. User-written queries and programs, or a DB2-supplied stored procedure, or Control Center, can use the statistics to make decisions for object maintenance.

DB2 constantly collects statistics for database objects and keeps the statistics in virtual storage. The statistics are calculated and updated asynchronously during externalization. To externalize the statistics, the environment must be properly set up. A new set of DB2 objects must be created to let DB2 write the statistics. SDSNSAMP(DSNTESS) contains the information necessary to set up these objects.

You must create two tables, with appropriate indexes, to hold the statistics:

- SYSIBM.TABLESPACESTATS
- SYSIBM.INDEXSPACESTATS

These tables are kept in a database named DSNRTSDB, which must be started to externalize the statistics being held in virtual storage. DB2 populates the tables with one row per table space or index space, or one row per partition. For tables that are shared in a data-sharing environment, each member writes its own statistics to the RTS tables.

Some of the important statistics that are collected for table spaces include:

- Total number of rows
- Number of active pages
- Time of last COPY, REORG, or RUNSTATS execution

Some statistics 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
- Number of overflow records that are created since the last REORG

The number of inserts, updates, and single or mass deletes since the last RUNSTATS execution can help determine when to execute RUNSTATS.

Statistics that are collected to help with COPY determination include distinct updated pages and changes since the last COPY execution and the RBA/LRSN of first update since the last COPY.

DB2 gathers the following statistics on indexes:

- Total number of unique or duplicate entries
- Number or levels
- Number of active pages
- Space allocated and extents

The time when the last REBUILD, REORG, or LOAD REPLACE occurred can help determine when a REORG is needed.

DB2 gathers statistics since the last REORG or REBUILD to determine how our data physically looks after certain processes occur, for example, batch inserts, so you can take appropriate actions if necessary. These statistics include:

- The number of updates
- The number of deletes, real or pseudo, single or mass
- The number of inserts, random inserts and inserts after the highest key

Following are the processes that have an effect on the real-time statistics:

- SQL
- Utilities
- Deleting objects
- Creating objects

After the tables are externalized, you can write queries against the tables.

Example: Query to identify page changes since the last image copy

This example shows how to identify when a tablespace has to be copied because more than 30 percent of the pages changed since the last image copy was taken.

```
SELECT NAME
  FROM SYSIBM.SYSTABLESPACESTATS
    WHERE DBNAME = 'DB1' and
    ((COPYUPDATEDPAGES*100)/NACTIVE)>30
```

Example: Determine when RUNSTATS is needed

This example compares the last RUNSTATS timestamp to the timestamp of the last REORG on the same object. If the date of the last REORG is more recent than the last RUNSTATS, it may be time to execute RUNSTATS.

```
SELECT NAME
FROM SYSIBM.SYSTABLESPACESTATS
WHERE DBNAME = 'DB1' and
   (JULIAN_DAY(REORGLASTTIME)>JULIAN_DAY(STATSLASTTIME))
```

Example: Determine if you have to run REORG after a series of inserts

This example may be useful to monitor the number of records that were inserted since the last REORG or LOAD REPLACE that are not well-clustered with respect to the clustering index. Well-clustered means that 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 if you have to run REORG after a series of inserts.

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

DB2 includes a stored procedure, DSNACCOR, to help with this process, and possibly even work toward automating the whole determination/utility execution process. DSNACCOR is a sample procedure that queries the RTS tables to determine which objects:

- Need to be reorganized, image copied, and updated with current statistics
- Have taken too many extents
- May be in a restricted status

DSNACCOR creates and uses its own declared temporary tables and must run in a WLM address space. The output of the stored procedure provides recommendations by using a predetermined set of criteria in formulas that use the RTS and user input for their calculations. DSNACCOR can make recommendations for COPY, REORG, RUNSTATS, EXTENTS, RESTRICT, or for one or more of your choices and for specific object types, such as table spaces and indexes.

Chapter 10: Tuning Tips

Code DISTINCT Only When Needed

Using DISTINCT can cause excessive sorting, which can cause queries to become expensive. Using DISTINCT in a table expression forces materialization. Only a unique index can help to avoid sorting for a DISTINCT.

Programmers often use DISTINCT unnecessarily, sometimes because of a failure to understand the data, the process, or the data and the process. Some code generators create a DISTINCT after every SELECT clause, regardless of where the SELECT is in a statement. Some programmers use a DISTINCT as a safeguard.

When considering using DISTINCT, if duplicates are not possible, remove the DISTINCT and avoid the potential sort.

When duplicates are an undesired possibility, use DISTINCT wisely by applying these practices:

- Use a unique index to avoid a sort.
- Consider using DISTINCT as early as possible in complex queries to eliminate the duplicates as early as possible.
- Avoid using DISTINCT more than once in a query.
- A GROUP BY all columns can use non-unique indexes to avoid a sort. Coding a GROUP BY all columns can be more efficient than using DISTINCT, but should be carefully documented in your program or statement.

Prevent Java in WebSphere From Defaulting the Isolation Level

ISOLATION(RS) is the default for WebSphere applications. RS (Read Stability) is RR (Repeatable Read) with inserts allowed. RS tells DB2 that, when you read a page, you intend to read it or update it again later and you want to prevent others from updating that page until you commit. RR causes DB2 to take share locks on pages and hold those locks.

Using RR is rarely necessary and can cause increased contention.

Using Read Stability can lead to the following other known problems:

- Inability to get lock avoidance
- Potential share lock escalations if DB2 escalates from an S lock on the page to an S lock on the table space
- Searched updates can deadlock under isolation RS

When using an isolation level of RS, the search operation for a searched update reads the page with an S lock. When it finds data to update, it changes the S lock to an X lock. Changing to an X lock could be exaggerated by a self-referencing correlated subquery in the update. For example, when updating the most recent history or audit row, DB2 reads the data with an S lock, puts the results in a work file with the RIDs, and returns to do the update in RID sequence.

Deadlocks are more likely to occur as the transaction volume increases. Because WebSphere defaults to RS, these problems are more likely with applications running under WebSphere when you let WebSphere determine the isolation level.

Some possible solutions to control this situation include the following:

- Set a JDBC database connection property to change the application server connect to use an isolation level of CS. This option is the best and often the most difficult to get implemented.
- Rebind the isolation RS package being used with an isolation level of CS. This is a less than optimal solution for the following reasons:
  - Some applications may require RS.
  - When the DB2 client software is upgraded, it may result in a new RS package or as a rebind of the package, requiring a special rebind with every upgrade on every client.
- Have the application change the statement to add WITH CS.
- To avoid deadlocks for some update statements, use a DSNZPARM RRULOCK=YES option. This DSNZPARM acquires U, instead of S locks, for the update and delete with ISO(RS) or ISO(RR).

Our research indicates no concrete reason why WebSphere defaults to RS. Defaulting to RS is wasteful and should be changed for DB2 applications.

Locking a Table in Exclusive Mode Will Not Always Prevent Applications from Reading It

DB2 V7 and V8 testing indicated that issuing the LOCK TABLE <*table name*> IN EXCLUSIVE MODE fails to prevent other applications from reading the table with UR or CS.

In V7, this was the case when the tablespace was defined with LOCKPART(YES). In V8, LOCKPART(YES) is the default for the tablespace.

After issuing an update, the table was not readable.

This is working as designed. This tip is to confirm that applications are aware that after issuing this statement, the table is still unreadable.

Put Data in Your Empty Control Table

Control tables are most often used to control availability or special processing application behavior. Many times, these control tables are empty and contain data only to force the application to behave in an unusual way. Testing with these tables indicates that there is no index lookaside on an empty index, which can add a significant number of getpage operations for a busy application. When your application performs unnecessary getpage operations, determine if the application can tolerate fake data in the table to force index lookaside and save CPU cycles.

Use Recursive SQL to Assign Multiple Sequence Numbers in Batch

In some situations, a process that is used by online and batch applications may add data to a database. Using sequence objects to assign system generated key values works well for online applications, but can get expensive for batch applications. Instead of creating an object with a large increment value, keep an increment of 1 for online, and reserve a batch of sequence numbers in batch using recursive SQL.

To use recursive SQL, write a recursive common table expression that generates a number of rows equivalent to the number of sequence values you want in a batch. As you select from the common table expression, you can get the sequence values. You can significantly reduce the cost of getting these sequence values for a batch process by using the appropriate settings for block fetching in distributed remote, or by using multi-row fetch in the local batch.

Example: SQL statement for 1000 sequence values

```
WITH GET_THOUSAND (C) AS
   (SELECT 1
     FROM SYSIBM.SYSDUMMY1
    UNION ALL
    SELECT C+1
    FROM   GET_THOUSAND
    WHERE  C < 1000)
SELECT NEXT VALUE FOR SEQ1
FROM   GET_THOUSAND;
```

Code Correlated Nested Table Expression versus Left Joins to Views

DB2 may materialize a view when you are left joining to that view. Materializing a view can affect performance, especially in transaction environments when the transactions process little or no data.

Consider the following view and query:

```
CREATE VIEW HIST_VIEW (ACCT_ID, HIST_DTE, ACCT_BAL) AS
(SELECT ACCT_ID, HIST_DTE, MAX(ACCT_BAL)
 FROM   ACCT_HIST
 GROUP BY ACCT_ID, HIST_DTE);

SELECT A.ACCT_ID, A.CUST_NME, B.HIST_DTE, B.ACCT_BAL
FROM   ACCOUNT A
LEFT OUTER JOIN
       HIST_VIEW B
ON A.ACCT_ID = B.ACCT_ID
WHERE A.ACCT_ID = 1110087;
```

The following query strongly encourages DB2 to use the index on the ACCT_ID column of the ACCT_HIST table:

```
SELECT A.ACCT_ID, A.CUST_NME, B.HIST_DTE, B.ACCT_BAL
FROM   ACCOUNT A
LEFT OUTER JOIN
TABLE(SELECT ACCT_ID, HIST_DTE, MAX(ACCT_BAL)
      FROM   ACCT_HIST X
      WHERE  X.ACCT_ID = A.ACCT_ID
      GROUP BY ACCT_ID, HIST_DTE) AS B
ON A.ACCT_ID = B.ACCT_ID
WHERE A.ACCT_ID = 1110087;
```

This recommendation can apply in many situations, not just aggregate queries in views.

Use GET DIAGNOSTICS Only When Necessary for Multi-Row Operations

Multi-row operations can be a huge CPU saver, and an elapsed time saver for sequential applications. However, at about three times the cost of the other statements, the GET DIAGNOSTICS statement is one of the most expensive statements an application can use. It is not advisable to issue a GET DIAGNOSTICS statement after every multi-row statement.

Try using the SQLCA before the GET DIAGNOSTICS statement. If you get a non-negative SQLCODE from a multi-row operation, you can use the GET DIAGNOSTICS statement to identify the details of the failure. To eliminate the need to use a GET DIAGNOSTICS for multi-row fetch, use the SQLERRD3 field to determine how many rows were retrieved when you get an SQLCODE 100.

Database Design Tips for High Concurrency

This section lists some tips to design high concurrency in a database.

Note: You must consider most of these recommendations during design before physically implementing the tables and table spaces, because it would be difficult to make these changes after the data is in use.

- Use segmented or universal table spaces, not simple table spaces, to keep rows of different tables on different pages, so page locks lock rows only for a single table.
- Use LOCKSIZE parameters appropriately to minimize the amount of data that is locked, except when the application requires exclusive access.

- Consider using MAXROWS=1 to space out rows for small tables with heavy concurrent access. Although row-level lock could also help, the overhead is greater, especially in a data sharing environment.
- Use partitioning where possible:
  - Reduces contention and increases parallel activity for the batch processes.
  - Reduces overhead in data sharing by allowing the use of affinity routing to different partitions.
  - Allows locks on individual partitions.
- Use Data Partitioned Secondary Indexes (DPSI) to promote partition independence and reduce contention for utilities.
- Consider using LOCKMAX 0 to turn off lock escalation. Increase NUMLKTS to force the applications to commit frequently.
- Use volatile tables to reduce contention. Applications that always access the data in the same order use an index to access the data.
- Have enough databases to reduce DBD locking when DDL, DCL, and utility execution is high for objects in the same database.
- Use sequence objects for better number generation without the overhead of using a single control table.

Database Design Tips for Data Sharing

In DB2 data sharing, the key to achieving high performance and throughput is to minimize the amount of actual sharing across members and to design databases and applications to take advantage of DB2 and data sharing. In data sharing, properly designed databases are the best performing databases.

The following tips will help you create properly designed databases to use for data sharing:

- Proper Partitioning Strategies
  - Processing by key range
  - Reducing or avoiding physical contention
  - Reducing or avoiding logical contention
  - Parallel processes are more effective
- Appropriate clustering and partitioning allows for the better cloning of applications across members and has the following benefits:
  - Access is predetermined and sequential
  - Key ranges can be divided among members

- Improves workload distribution
- Reduces contention and I/O
- Can run utilities in parallel across members

Tips for Avoiding Locks

Lock avoidance can be a key component to high performance because it requires about 400 CPU instructions and 540 bytes of memory to take a lock. To avoid a lock, the buffer manager takes a latch at a reduced cost.

The process of lock avoidance is as follows:

- Compare the page log RBA - that is the last time that a change was made on the page - to the CLSN (Commit Log Sequence Number) on the log, that is the last time all changes to the page set were committed
- Check the PUNC (Possibly Uncommitted) bit on the page
- When the process determines the page has no outstanding changes, the process takes a latch, instead of a lock

The following are prerequisites to achieving lock avoidance:

- Before V8, use CURRENTDATA(NO) on the bind. As of V8/9, CURRENTDATA(YES) allows the detection process to begin
- Be bound ISOLATION(CS)
- Use a read-only cursor
- Committing often is the most important point, because all processes must commit changes for the CLSN to get set. If the CLSN never gets set, this process can never work and locks must always be taken. Use the URCHKTH DSNZPARM to find applications that are not committing.

Index

S

T

U

V