

# CA Performance Handbook

for DB2 for z/OS

## About the Contributors from Yevich, Lawson and Associates Inc.

**DAN LUKSETICH** is a senior DB2 DBA. He works as a DBA, application architect, presenter, author, and teacher. Dan has over 17 years working with DB2 as a DB2 DBA, application architect, system programmer and COBOL and BAL programmer — working on major implementations on z/OS, AIX, and Linux environments.

His experience includes DB2 application design and architecture, database administration, complex SQL and SQL tuning, performance audits, replication, disaster recovery, stored procedures, UDFs, and triggers.

**SUSAN LAWSON** is an internationally recognized consultant and lecturer with a strong background in database and system administration. She works with clients to help development, implement and tune some of the world's largest and most complex DB2 databases and applications. She also performs performance audits to help reduce costs through proper performance tuning.

She is an IBM Gold Consultant for DB2 and z/Series. She has authored the IBM 'DB2 for z/OS V8 DBA Certification Guide', 'DB2 for z/OS V7 Application Programming Certification Guide' and 'DB2 for z/OS V9 DBA Certification Guide' — 2007. She also co-authored several books including 'DB2 High Performance Design and Tuning' and 'DB2 Answers' and is a frequent speaker at user group and industry events. (Visit [DB2Expert.com](http://DB2Expert.com))

## About CA

CA (NYSE: CA), one of the world's largest information technology (IT) management software companies, unifies and simplifies the management of enterprise-wide IT for greater business results. Our vision, tools and expertise help customers manage risk, improve service, manage costs and align their IT investments with their business needs. CA Database Management encompasses this vision with an integrated and comprehensive solution for database design and modeling, database performance management, database administration, and database backup and recovery across multiple database systems and platforms.

# Table of Contents

## About This Handbook

Originally known as “The DB2 Performance Tool Handbook” by PLATINUM Technologies (PLATINUM was acquired by Computer Associates in 1999), this important update provides information to consider as you approach database performance management, descriptions of common performance and tuning issues during design, development, and implementation of a DB2 for z/OS application and specific techniques for performance tuning and monitoring.

- **Chapter 1:** Provides a overview information on database performance tuning
- **Chapter 2:** Provides descriptions of SQL access paths
- **Chapter 3:** Describes SQL tuning techniques
- **Chapter 4:** Explains how to design and tune tables and indexes for performance
- **Chapter 5:** Describes data access methods
- **Chapter 6:** Describes how to properly monitor and isolate performance problem
- **Chapter 7:** Describes DB2 application performance features
- **Chapter 8:** Explains how to adjust subsystem parameters, and configure subsystem resources for performance
- **Chapter 9:** Describes tuning approaches when working with packaged applications
- **Chapter 10:** Offers tuning tips drawn from real world experience

## Who Should Read This Handbook

The primary audiences for this handbook are physical and logical database administrators. Since performance management has many stakeholders, other audiences who will benefit from this information include application developers, data modelers and data center managers. For performance initiatives to be successful, DBAs, developers, and managers must cooperate and contribute to this effort. This is because there is no right and wrong when it comes to performance tuning. There are only trade-offs as people make decisions about database design and performance. Are you designing for ease of use, minimized programming effort, flexibility, availability, or performance? As these choices are made, there are different organizational costs and benefits 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 a good working knowledge of DB2 and SQL, and is designed to help you build good performance into the application, database, and the DB2 subsystem. It provides techniques to help you monitor DB2 for performance, and to identify and tune production performance problems.



# Introducing DB2 Performance

Many people spend a lot of time thinking about DB2 performance, while others spend no time at all on the topic. Nonetheless, performance is on everyone's mind when one of two things happen: an online production system does not respond within the time specified in a service level agreement (SLA), or 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 once the problem is solved the tuning effort is abandoned.

This handbook aims to provide information that performance fire fighters and performance tuners need. It also provides the information necessary to design databases for high performance, to understand how DB2 processes SQL and transactions, and to tune those processes for performance. For database performance management, the philosophy of performance tuning begins with database design, and extends through (database and application) development and production implementation.

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, goes a long way to ensure well performing applications, databases, and subsystems.

## Using This Handbook

This handbook provides important information to consider as you approach database performance management, descriptions of common performance and tuning issues during design, development, and implementation of a DB2 for z/OS application and specific techniques for performance tuning and monitoring.

- Throughout this handbook, you'll find:
- 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 both 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

The following chapters discuss how to fix various performance problems:

- Chapter 2 provides descriptions of SQL access paths
- Chapter 3 describes SQL tuning techniques
- Chapter 4 explains how to design and tune tables and indexes for performance
- Chapter 5 describes data access methods
- Chapter 6 describes how to properly monitor and isolate performance problems
- Chapter 7 describes DB2 application performance features that you can take advantage of
- Chapter 8 explains how to adjust subsystem parameters, and configure subsystem resources for performance
- Chapter 9 describes actions that can be taken when working with packaged applications
- Chapter 10 offers tuning tips drawn from experience at a variety of implementations

## Measuring DB2 Performance

Performance is a relative term. That is, good performance is a balance between expectations and reality. Good performance is not something that is typically delivered at the push of a button. If someone tells you they want everything to run as fast as possible, then you should tell them that the solution is to buy an infinite number of machines. In reality, however, in order for effective performance tuning to take place, proper expectations and efforts should be identified and understood. Only then will the full benefits of performance tuning be realized.

Most people don't inherently care about performance. They want their data, they want it now, they want it accurate, they want it all the time, and they want it to be secured. Of course, once they get these things they are also interested in getting the data fast, and at the lowest cost possible. Enter performance design and tuning.

Getting the best possible performance from DB2 databases and applications means a number of things. First and foremost, it means that users do not need to wait for the information they require to do their jobs. Second, it means that the organization using the application is not spending 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. Is it an online transaction processing system (OLTP)? A data warehouse? A batch processing system?

Your site may use one or all of the following 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

(See Chapters 2, 3, 5 and 6.)

## Possible Causes of DB2 Performance Problems

When the measurements you use at your site warn you of a performance problem, you face the challenge of finding the cause of the problem. Some common causes of performance problems are:

- Poor database design
- Poor application design
- Catalog statistics that do not reflect the data size and/or organization
- Poorly written SQL
- Generic SQL design
- Inadequately allocated system resources, such as buffers and logs

In other situations you may be dealing with packaged software applications that cause performance problems. In these situations it may be difficult to tune the application but quite possible to improve performance through database and subsystem tuning. (See Chapter 9.)

## Designing for Performance

An application can be designed and optimized for high availability, ease of programming, flexibility and reusability of code, or performance. The most thorough application design should take into account all of these things. Major performance problems are caused when performance is entirely overlooked during design, is considered late in the design cycle or is not properly understood, tested and addressed prior to production.

Designing an application with regards to performance will help avoid many performance problems after the application has been deployed. Proactive performance engineering can help eliminate redesign, recoding, and retrofitting during implementation in order to satisfy performance expectations, or alleviate performance problems. Proactive performance engineering allows you to analyze and stress test the designs before they are implemented. Proper techniques (and tools) can turn performance engineering research and testing from a lengthy process inhibiting development to one that can be performed efficiently prior to development.

There are many options for proper index and table design for performance. These options have to do specifically with the type of application that is going to be implemented, specifically with the type of data access (random versus sequential, read versus update, etc.). In addition, the application as well as to be properly designed for performance. This involves understanding the application data access patterns, as well as the units of work (UOWs) and inputs. Taking these things into consideration, and designing the application and database accordingly is the best defense against performance problems. (See Chapters 4 and 7.)

## Finding and Fixing 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 don't 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, of course, 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. (See Chapter 6.)

### Fixing the Performance Problems

Once a performance problem has been identified, what is the best approach to address it? Is it a matter of updating catalog statistics? Tuning an SQL statement? Adjusting parameters for indexing, table space, or subsystem? With the proper monitoring in place, as described in Chapter 6, the problem should be evident. Proper action will be the result of properly understanding the problem first. (See Chapters 2-4; 7-10.)

# SQL and Access Paths

One of the most important things to understand about DB2, and DB2 performance, is that DB2 offers a level of abstraction from the data. We don't have to know where exactly the data physically exists. We don't need to know where the datasets are, we don't need to know the access method for the data, and we don't need to know what index to use. All we need to know is the name of the table, and the columns we are interested in. DB2 takes care of the rest. This is a significant business advantage in that we can quickly build applications that access data via a standardized language that is independent of any access method or platform. This enables ultra-fast development of applications, portability across platforms, and all the power and flexibility that comes with a relational or object-relational design. What we pay in return is more consumption of CPU and I/O.

Using DB2 for z/OS can be more expensive than more traditional programming and access methods on the z/OS platform. Yes, VSAM and QSAM are generally cheaper than DB2. Yes, flat file processing can be cheaper than DB2, sometimes! However, with continually shrinking people resources and skill sets on the z/OS platform, utilizing a technology such as DB2 goes a long way in saving programming time, increasing productivity, and portability. This means leverage, and leverage goes a long way in the business world.

So, with the increased cost of using DB2 over long standing traditional access methods on the z/OS platform, can we possibly get the flexibility and power of DB2, along with the best performance possible? Yes, we can. All we need is an understanding of the DB2 optimizer, the SQL language, and the trade-offs between performance, flexibility, portability, and people costs.

## The DB2 Optimizer

There is no magic to computer programming. Computers are stupid. They only understand zeros and ones. However, computers fool us in that in spite of their stupidity they are incredibly fast. This speed enables an incredible number of calculations to be performed in a fraction of a second. The DB2 optimizer is no exception. The DB2 optimizer (in a simplistic view) simply accepts a statement, checks it for syntax, checks it against the DB2 system catalog, and then builds access paths to the data. It uses catalog statistics to attach a cost to the various possible access paths, and then chooses the cheapest path. Not to insult the incredible efforts that the IBM DB2 development team has performed in making the DB2 optimizer the most accurate and efficient engine for SQL optimization on the planet! However, understanding the optimizer, the DB2 access paths, and how your data is accessed will help you understand and adjust DB2 performance.

The optimizer is responsible for interpreting your queries, and determining how to access your data in the most efficient manner. However, it can only utilize the best of the available access paths. It does not know what access paths are possible that are not available. It also can only deal with the information that it is provided. That is, it is dependent upon the information we give it. This is primarily the information stored in the DB2 system catalog, which includes the basic information about the tables, columns, indexes and statistics. It doesn't know about 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. This is why an understanding of the DB2 optimizer, statistics, and access paths is very important, but also important is the design of applications and databases for performance. These topics are covered in the remainder of this guide, and so this chapter serves only as an education into the optimizer, statistics, and access paths, and is design to build a basis upon which proper SQL, database, and application tuning can be conducted.

## The Importance of Catalog Statistics

It is possible to maintain information about the amount of data, and the organization of that data in DB2 tables. These are the DB2 catalog statistics, and are typically collected by the DB2 RUNSTATS utility.

The DB2 optimizer is cost based, and so catalog statistics are critical to proper performance. The optimizer uses a variety of statistics to determine the cost of accessing the data in tables. These statistics can be cardinality statistics, frequency distribution statistics, or histogram statistics. The type of query (static or dynamic), use of literal values, and runtime optimization options will dictate which statistics are used. The cost associated with various access paths will affect whether or not indexes are chosen, or which indexes are chosen, the access path, and table access sequence for multiple table access paths. For this reason maintaining accurate up to date statistics is critical to performance.

It should be noted that catalog statistics and database design are not the only things the DB2 optimizer uses to calculate and choose an efficient access path. Additional information, such as central processor model, number of processors, and size of the RID pool, various installation parameters, and buffer pool sizes are used to determine the access path. You should be aware of these factors as you tune your queries.

There are three flavors to DB2 catalog statistics. Each provides different details about the data in your tables, and DB2 will utilize these statistics to determine the best way to access the data dependent upon the objects and the query.

### Cardinality Statistics

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

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 of what is known as the filter factor, a percentage of the number of rows expected to be returned, for a column or a table. DB2 will use these statistics to determine such things as to whether or not access is via an index scan or a table space scan, and when joining tables which table to access first. For example, for the following statement embedded in a COBOL program:

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

DB2 has to choose what is the most efficient way to access the EMP table. This will depend upon the size of the table, as well as the cardinality of the SEX column, and any index that might be defined upon the table (especially one on the SEX column). It will use the catalog statistics to determine the filter factor for the SEX column. In this particular case, 1/COLCARDF, COLCARDF being a column in the SYSIBM.SYSCOLUMNS catalog table. The resulting fractional number represents the number of rows that are expected to be returned based upon the predicate. DB2 will use this number, the filter factor, to make decisions as to whether or not to use an index (if available), or table space scan. For example, if the column cardinality of the SEX column of the EMP table is 3 (male, female, unknown), then we know there are 3 unique occurrences of a value of SEX in the table. In this case, DB2 determines a filter factor of 1/3, or 33% of the values. If the table 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 returned from the query will be 3,333. DB2 will use this information to make decisions about which access path to choose.

If the predicate in the query above is used in a join to another table, then the filter factor 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 E.EMPNO, D.DEPTNO, D.DEPTNAME
FROM   EMP E
INNER JOIN
        DEPT D
ON     E.WORKDEPT = D.DEPTNO
WHERE  LASTNAME   = :LAST-NAME
```

In addition to normal cardinality statistics, statistics can be gathered upon groups of columns, which is otherwise known as columns correlation statistics. This is very important to know because again the DB2 optimizer can only deal with the information it is given. Say, for example you have the following query:

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

Now, imagine in a truly hypothetical example that the amount of money that someone is paid is correlated to the amount of education they have received. For example, an intern being paid minimum wage won't be 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 may result in an exaggerated filter factor for the table, and could negatively influence the choice of 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, it may be wise to gather column correlation statistics on any columns referenced together in the WHERE clause.

### Frequency Distribution Statistics

In some cases cardinality statistics are not enough. With only cardinality statistics the DB2 optimizer has to make assumptions about the distribution of data in a table. That is, it has to assume that the data is evenly distributed. Take, for example, the SEX column from the previous queries. The column cardinality is 3, however it is quite common that the values in the SEX column will be 'M' for male, or 'F' for female, with each representing approximately 50% of the values, and the value 'U' occurs only a small fraction of the time. This is an example of what we will call 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 and/or the input values in a query predicate. In the following query:

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

DB2 has no idea what the input value of the :SEX host variable is, and so it uses the default formula of 1/COLCARD. Likewise, if the following statement is issued:

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

and most of the values are 'M' or 'F', and DB2 has only cardinality statistics available, then the same formula and filter factor applies. This is where distribution statistics can pay off.

DB2 has the ability to collect distribution statistics via the RUNSTATS utility. These statistics reflect the percentage of frequently occurring values. That is, 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 have a dramatic impact on 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)

So, if we gathered frequency distribution statistics for our SEX column in the previous examples, we may find the following frequency values in the SYSIBM.SYSCOLDIST table (simplified in this example):

VALUE	FREQUENCY
'M'	.49
'F'	.49
'U'	.01

Now, if DB2 has this information, and is provided with the following query:

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

It can determine that the value 'U' represents about 1% of the values of the sex column. This additional information can have dramatic effects on access path and table access sequence decisions.

### Histogram Statistics

Histogram statistics improve on distribution statistics in that they provide value-distribution statistics that are collected over the entire range of values in a table. This goes beyond the capabilities of distribution statistics in that distribution statistics are limited to only the most or least occurring values in a table (unless they are, of course, collected all the time for every single column which can be extremely expensive and difficult). To be more specific, 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 via a specification of quantiles in a RUNSTATS execution. Thus a quantile will represent a range of values within the table. Each quantile will contain approximately the same percentage of rows. This can go beyond the distribution statistics in that all values are represented.

Now, the previous example is no good because we had only three values for the SEX column of the EMP table. So, we'll use a hypothetical example with our EMP sample table. Let's suppose, theoretically, that we needed to query on the middle initial of person's names. Those initials are most likely in the range of 'A' to 'Z'. If we had only cardinality statistics then any predicate referencing the middle initial column would receive a filter factor of  $1/\text{COLCARD}$  or  $1/26$ . If we had distribution statistics then any predicate that used a literal value could take advantage of the distribution of values to determine the best access path. That determination would be dependent upon the value being recorded as one of the most or least occurring values. If it is not, then the filter factor is determined based upon the difference in frequency of the remaining values. Histogram statistics eliminate this guesswork.

If we calculated histogram statistics for the middle initial, they may look something like this:

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%

Now, the DB2 optimizer has information about the entire span of values in the table, and any possible skew. This is especially useful for range predicates such as this:

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

## Access Paths

There are a number of access paths that DB2 can choose when accessing the data in your database. This section describes those access paths, as well as when they are effective. The access paths are exposed via use of the DB2 EXPLAIN facility. The EXPLAIN facility can be invoked via one of these techniques:

- EXPLAIN SQL statement
- EXPLAIN(YES) BIND or REBIND parameter
- Visual Explain (DB2 V7, DB2 V8, DB2 9)
- Optimization Service Center (DB2 V8, DB2 9)

### Table Space Scan

If chosen, DB2 will scan an entire table space in order to satisfy a query. What is actually scanned depends upon the type of table space. Simple table spaces (supported in DB2 9, but you can't create them) will be scanned in their entirety. Segmented and universal tablespaces will be scanned for only the table specified in the query (or for the spacemaps that qualify). Nonetheless, a table space scan will search the entire table space to satisfy a query. A table space scan is indicated by an "R" in the PLAN\_TABLE ACESSTYPE column.

A table space scan will be 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(s)
- DB2 calculates that a high percentage of rows will be retrieved, and in this case 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

A table space scan is not necessarily a bad thing. It depends upon the nature of the request and the frequency of access. Sure, running a table space scan every one second in support of online transactions is probably a bad thing. A table space scan in support of a report that spans the content of an entire table once per day is probably a good thing.

### Limited Partition Scan and Partition Elimination

DB2 can take advantage of the partitioning key values to eliminate partitions from a table space scan, or via index access. DB2 utilizes the column values that define the partitioning key to eliminate partitions from the query access path. Queries have to be coded explicitly to eliminate partitions. This partition elimination can apply to table space scans, as well as index access. Suppose, for example, that the EMP table is partitioned by the EMPNO column (which it is in the DB2 sample database). The following query would eliminate any partitions not qualified by the predicate:

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

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

DB2 can also choose a secondary index for the access path, and still eliminate partitions using the partitioning key values if those values are supplied in a predicate. For example, in the following query:

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

DB2 could choose a partitioned secondary index on the WORKDEPT column, and eliminate partitions based upon the range provided in the query for the EMPNO.

DB2 can employ partition elimination for predicates coded against the partitioning key using:

- Literal values (DB2 V7, DB2 V8, DB2 9)
- Host variables (DB2 V7 with REOPT(VARS), DB2 V8, DB2 9)
- Joined columns (DB2 9)

### Index Access

DB2 can utilize indexes on your tables to quickly access the data based upon the predicates in the query, and to avoid a sort in support of the use of the DISTINCT clause, as well as for the ORDER BY and GROUP BY clauses, and INTERSECT, and EXCEPT processing.

There are several types of index access. The index access is indicated in the PLAN\_TABLE by an ACESSTYPE value of either I, I1, N, MX, or DX.

DB2 will match the predicates in your queries to the leading key columns of the indexes defined against your tables. This is indicated by the MATCHCOLS column of the PLAN\_TABLE. If the number of columns matched is greater than zero, then the index access is considered a matching index scan. If the number of matched columns is equal to zero, then the index access is considered a non-matching index scan. In a non-matching index scan all of the key values and their record identifiers (RIDs) are read. A non-matching index scan is typically used if DB2 can use the index in order to avoid a sort when the query contains an ORDER BY, DISTINCT, or GROUP BY clause, and also possibly for an INTERSECT, or EXCEPT.

The matching predicates on the leading key columns are equal (=) or IN predicates. This would correspond to an ACESSTYPE 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, for the following query assume that the EMPPROJECT DB2 sample table has an index on the PROJNO, ACTNO, EMSTDATE, and EMPNO columns:

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

In the above example DB2 could choose a matching index scan matching on the PROJNO, ACTNO, and EMSTDATE columns. It does not match on the EMPNO column due to the fact that the previous predicate is a range predicate. However, the EMPNO column can be applied as an index screening column. That is, since it is a stage 1 predicate (predicate stages are covered in Chapter 2) DB2 can apply it to the index entries after the index is read. So, although the EMPNO column does not limit the range of entries retrieved from the index it can eliminate the entries as the index is read, and therefore reduce 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, then DB2 may choose index only access. This is indicated in the PLAN\_TABLE with the value “Y” in the INDEXONLY column.

DB2 is also able to access indexes via multi-index access. This is indicated via an ACCESTYPE of “M” in the PLAN\_TABLE. 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 (depending upon the predicates in the SQL statement), 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 (ACCESTYPE = “MX” or “DX”) for regular indexes or DOCID XML indexes, and then unions or intersections of the rids (ACCESTYPE = “MU”, “MI”, “DU”, or “DI”).

The following example demonstrates a SQL statement that can perhaps utilize multi-index access if there is an index on the LASTNAME and FIRSTNAME columns of the EMP table.

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

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

### List Prefetch

The preferred method of table access, when access is via an index, is by utilizing a clustered index. When an index is defined as clustered, then DB2 will attempt to keep the data in the table in the same sequence as the key values in the clustering index. Therefore, when the table is accessed via the clustering index, and the data in the table is well organized (as represented by the catalog statistics), then the access to the data in the table will typically be sequential and predictable. Accessing the data in a table via a non-clustering index, or via the clustering index with a low cluster ratio (the table data is disorganized), then the access to the data can be very random and unpredictable. In addition, the same data pages could be read multiple times.

When DB2 detects that access to a table will be via a non-clustering index, or via a clustering index that has a low cluster ratio, then DB2 may choose an index access method called list prefetch. List prefetch will also be used as part of the access path for multi-index access and access to the inner table of a hybrid join. List prefetch is indicated in the PLAN\_TABLE via a value of "L" in the PREFETCH column.

List prefetch will access the index via a matching index scan of one or more indexes, and collect the RIDs into the RID pool, which is a common area of memory. The RIDs are then sorted in ascending order by the page number, and the pages in the table space are then retrieved in a sequential or skip sequential fashion. List prefetch will not be used if the matching predicates include an IN-list predicate.

List prefetch will not be chosen as the access path if DB2 determines that the RIDs to be processed will take more than 50% of the RID pool when the query is executed. Also, list prefetch can terminate during execution if DB2 determines that more than 25% of the rows of the table must be accessed. In these situations (called RDS failures) the access path changes to a table space scan.

In general, a list prefetch is useful for access to a moderate number of rows when access to the table is via a non-clustering or clustering index when the data is disorganized (low cluster ratio). It is usually less useful for queries that process large quantities of data or very small amounts of data.

### Nested Loop Join

In a nested loop join DB2 will scan the outer table (the first table accessed in the join operation) via a table space scan or index access, and then for each row in that table that qualifies DB2 will then search for matching rows in the inner table (the second table accessed). It will concatenate any rows it finds in the inner table with the outer table. Nested loop join is indicated via a value of "1" in the METHOD column of the PLAN\_TABLE.

A nested loop join will repeatedly access the inner table as rows are accessed in the outer table. Therefore, the nested loop join is most efficient when a small number of rows qualify for the outer table, and is a good access path for transactions that are processing little or no data. It is important that there exist an index that supports the join on the inner table, and for the best performance that index should be a clustering index in the same order as the outer table. In this way the join operation can be a sequential and efficient process. DB2 (DB2 9) can also dynamically build a sparse index on the inner table when no index is available on the inner table. It may also sort the outer table if it determines that the join columns are not in the same sequence as the inner table, or the join columns of the outer table are not in an index or are in a clustering index where the table data is disorganized.

The nested loop join is the method selected by DB2 when you are joining two tables together, and do not specify join columns.

### Hybrid Join

The preferred method for a join is to join two tables together via a common clustering index. In these cases it is likely that DB2 will choose a nested loop join as the access method for small amounts of data. If DB2 detects that the inner table access will be via a non-clustering index or via a clustering index in which the cluster ratio is low, then DB2 may choose a hybrid join as the access method. The hybrid join is indicated via a value of “4” in the METHOD column of the PLAN\_TABLE.

In a hybrid join DB2 will scan the outer table either via a table space scan or via an index, and then join the qualifying rows of the outer table with the RIDs from the matching index entries. DB2 also creates a RID list, as it does for list prefetch, for all the qualifying RIDs of the inner table. Then DB2 will sort the outer table and RIDs, creating a sorted RID list and an intermediate table. DB2 will then access the inner table via list prefetch, and join the inner table to the intermediate table.

Hybrid join can out perform a nested loop join when the inner table access is via a non-clustering index or clustering index for a disorganized table. This is especially true if the query will be retrieving more than a trivial amount of data. That is, the hybrid join is better if the equivalent nested loop join would be accessing the inner table in a random fashion, and thus 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. Thus, hybrid join is better when the query processes more than a tiny amount of data (nested loop is better in this case), or less than a very large amount of data (merge scan is better in this case). Since list prefetch is utilized, the hybrid join can experience RID list failures. This can result in a table space scan of the inner table on initial access to the inner table, or a restart of the list prefetch processing upon subsequent accesses (all within the same query). If you are experiencing RID list failures in a hybrid join then you should attempt to encourage DB2 to use a nested loop join.

### Merge Scan Join

If DB2 detects that a large number of rows of the inner and outer table will qualify for the join, or the tables have no indexes on the matching columns of the join, then DB2 may choose a merge scan join as the access method. A merge scan join is indicated via a value of “2” in the METHOD column of the PLAN\_TABLE.

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

Merge scan join is best for large quantities of data. The general rule should be that if you are accessing small amounts of data in a transaction then nested loop join is best. Processing large amounts of data in a batch program or a large report query? Then merge scan join is generally the best.

### Star Join

A star join is another join method that DB2 will employ in special situations in which it detects that the query is joining tables that are a part of a star schema design of a data warehouse. The star join is indicated via the value of “S” in the JOIN\_TYPE column of the PLAN\_TABLE. The star join is best described in the DB2 Administration Guide (DB2 V7, DB2 V8) of the DB2 Performance Monitoring and Tuning Guide (DB2 9).

## Read Mechanisms

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

### Sequential Prefetch

When your query reads data in a sequential manner, DB2 can elect to use sequential prefetch to read the data ahead of the query requesting it. That is, DB2 can launch asynchronous read engines to read data from the index and table page sets into the DB2 buffers. This hopefully allows the data pages to already be in the buffers in expectation of the query accessing them. The maximum number of pages read by a sequential prefetch operation is determined by the size of the buffer pool used. When the buffer pool is smaller than 1000 pages, then the prefetch quantity is limited due to the risk of filling up the buffer pool. If the buffer pool has more than 1000 pages then 32 pages can be read with a single physical I/O.

Sequential prefetch is indicated via a value of “S” in the PREFETCH column of the PLAN\_TABLE. DB2 9 will only use sequential prefetch for a table space scan. Otherwise, DB2 9 will rely primarily on dynamic prefetch.

### Dynamic Prefetch and Sequential Detection

DB2 can perform sequential prefetch dynamically at execution time. This is called dynamic prefetch. It employs a technique called sequential detection in order to detect the forward reading of table pages and index leaf pages. DB2 will track the pattern of pages accessed in order 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 4 of the last eight pages are page-sequential. A page is considered page-sequential if it is within one half of the prefetch quantity for the buffer pool. Therefore, it is not five of eight sequential pages that activates dynamic prefetch, but instead five of eight sequentially available pages. As DB2 can activate dynamic prefetch, it can also deactivate dynamic prefetch if it determines that the query or application is no longer reading sequentially. DB2 will also indicate in an access path if a query is likely to get dynamic prefetch at execution time. This is indicated via a value of “D” in the PREFETCH column of the PLAN\_TABLE.

Dynamic prefetch can be activated for single cursors, or for repetitive queries within a single application thread. This could improve the performance of applications that issue multiple SQL statements that access the data as a whole sequentially. It is also important, of course, to cluster tables that are commonly accessed together to possibly take advantage of sequential detection.

It should be noted that dynamic prefetch is dependent upon the bind parameter `RELEASE(DEALLOCATE)`. This is because the area of memory used to track the last eight pages accessed is destroyed and rebuilt for `RELEASE(COMMIT)`. It should also be noted that `RELEASE(DEALLOCATE)` is not effective for remote connections, and so sequential detection and dynamic prefetch are less likely for remote applications.

### Index Lookaside

Index lookaside is another powerful performance enhancer that is active only after an initial access to an index. For repeated probes of an index DB2 will check the most current index leaf page to see if the key is found on that page. If it is then DB2 will not read from the root page of the index down to the leaf page, but simply use the values from the leaf page. This can have a dramatically positive impact on performance in that getpage operations and I/O's can be avoided.

Index lookaside depends on the query reading index data in a predictable pattern. Typically this is happening for transactions that are accessing the index via a common cluster with other tables. Index lookaside is also dependent upon the `RELEASE(DEALLOCATE)` bind parameter, much in the same way as sequential detection.

## Sorting

DB2 may have to invoke a sort in support of certain clauses in your SQL statement, or in support of certain access paths. DB2 can sort in support of an `ORDER BY` or `GROUP BY` clause, to remove duplicates (`DISTINCT`, `EXCEPT`, or `INTERSECT`), or in join or subquery processing. In general, sorting in an application will always be faster than sorting in DB2 for 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. Perhaps instead you can take advantage of the ways in which DB2 can avoid a sort.

### Avoiding a Sort for an ORDER BY

DB2 can avoid a sort for an ORDER BY by utilizing the leading columns of an index to match the ORDER BY columns if those columns are the leading columns of the index used to access a single table, or the leading columns of the index to access the first table in a join operation. DB2 can also utilize something called order by pruning to avoid a sort. Suppose for the following query we have an index on (WORKDEPT, LASTNAME, FIRSTNAME):

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

In the above query the sort will be avoided if DB2 uses the index on those three columns to access the table. DB2 can also avoid a sort if any number of the leading columns of an index match the ORDER BY clause, or if the query contains an equals predicate on the leading columns (ORDER BY pruning). So, both of the following queries will 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 utilize a unique or duplicate index, as well as a full or partial index key to avoid a sort. The grouped columns must match the leading columns of the index, and any missed leading columns must be in equals or IN predicates in the WHERE clause. Assuming again an index on (WORKDEPT, LASTNAME, FIRSTNAME), each of the following statements can avoid a sort if the index is used:

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

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

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

### Avoiding a Sort for a DISTINCT

A DISTINCT can avoid a sort by utilizing only a unique index if that index is used to access the table. You could possibly utilize 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. So, with this in mind the following two statements are identical:

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

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

When using this technique, make sure you properly document so other programmers understand why you are grouping on all columns.

### Avoiding a Sort for a UNION, EXCEPT, or INTERSECT

DB2 can possibly avoid a sort for an EXCEPT or INTERSECT. A sort could be avoided if there is a matching index with the unioned/intersected/excepted columns, or if it realizes that the inputs to the operation are already sorted by a previous operation. Sort cannot be avoided for a UNION. However, you have to ask yourself, “are duplicates possible?” If not, then change each to a UNION ALL, EXCEPT ALL, or INTERSECT ALL. If duplicates are possible, you might consider if the duplicates are produced by only one subselect of the query. If so, then perhaps a GROUP BY can be used on all columns in those particular subselects since GROUP BY has a greater potential to avoid a sort.

## Parallelism

DB2 can utilize something called parallel operations for access to your tables and/or indexes. This can have a dramatic performance impact on 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 dramatically reduced. There are two types of query parallelism; query I/O parallelism and query CP parallelism.

I/O parallelism manages concurrent I/O requests for a single query. It 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 can break a large query up into smaller queries, and then run the smaller queries in parallel on multiple processors. This can also significantly reduce the elapsed time for large queries.

You can enable query parallelism by utilizing the DEGREE(ANY) parameter of a BIND command, or by setting the value of the CURRENT DEGREE special register to the value of 'ANY'. Be sure, however, that you utilize query parallelism for larger queries, as there is overhead to the start of the parallel tasks. For small queries this can actually be a performance detriment.

DB2 can also utilize something called Sysplex query parallelism. In this situation DB2 can split a query across multiple members of a data sharing group to utilize all of the processors on the members for extreme query processing of very large queries.



# Predicates and SQL Tuning

This chapter will address the fundamentals of SQL performance and SQL processing in DB2. In order to write efficient SQL statements, and to tune SQL statements, we need a basic understanding of how DB2 optimizes SQL statements, and how it accesses your data.

When writing and tuning SQL statements, we should maintain a simple and important philosophy. That being “filter as much as possible as early as possible”. We have to keep in mind that the most expensive thing we can do is to travel from our application to DB2. Once we are processing inside DB2 we need to do our filtering as close to the indexes and data as possible.

We also need to understand how to reduce repeat processing. Keep in mind that the most efficient SQL statement is the one that never executes. That is, we should only do absolutely what is necessary to get the job done.

## Types of DB2 Predicates

Predicates in SQL statements fall under classifications. These classifications dictate how DB2 processes the predicates, and how much data is filtered when during the process. These classifications are:

- Stage 1 Indexable
- Stage 1
- Stage 2
- Stage 3

You can apply a very simple philosophy in order to understand which of your predicates might fall within one of these classifications. That is, the DB2 stage 1 engine understands your indexes and tables, and can utilize an index for efficient access to your data. Only a stage 1 predicate can limit the range of data accessed on a disk. The stage 2 engine processes functions and expressions, but is not able to directly access data in indexes and tables. Data from stage 1 is passed to stage 2 for further processing, and so stage 1 predicates are generally more efficient than stage 2 predicates. Stage 2 predicates cannot utilize an index, and thus cannot limit the range of data retrieved from disk. Finally, a stage 3 predicate is a predicate that is processed in the application layer. That is, filtering performed once the data is retrieved from DB2 and processed in the application. Stage 3 predicates are the least efficient.

There is a table in the DB2 Administration Guide (DB2 V7 and DB2 V8) or the DB2 Performance Monitoring and Tuning Guide (DB2 9) that lists the various predicate forms, and whether they are stage 1 indexable, stage 1, or stage 2. Only examples are given in this guide.

### Stage 1 Indexable

The best performance for filtering will occur when stage 1 indexable predicates are used. But just because a predicate is listed in the chart as stage 1 indexable does not mean it will be used for either an index or processed in stage 1 as there are many other factors that must also be in place. The first thing that determines if a predicate is stage 1 is the syntax of the predicate and then secondly, it is the type and length of constants used in the predicate. This is one area which has improved dramatically with version 8 in that more data types can be promoted inside stage 1 thus improving stage 1 matching. If the predicate, even though it would classify as a stage 1 predicate, is evaluated after a join operation then it is a stage 2 predicate. All indexable predicates are stage 1 but not all stage 1 predicates are indexable.

Stage 1 indexable predicates are predicates that can be used to match on the columns of an index. The most simple example of a stage 1 predicate would be of the form <col op value>, where col is a column of a table, op is an operator (=, >, <, >=, <=) and value represents a non-column expression (an expression that does not contain a column from the table). Predicates containing BETWEEN, IN (for a list of values), and LIKE (without a leading search character) can also be stage 1 indexable. The stage 1 indexable predicate, when an index is utilized, provides the best filtering in that it can actually limit the range of data accessed from the index. You should try to utilize stage 1 matching predicates whenever possible. Assuming that an index exists on the EMPNO column of the EMP table, then 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 utilize 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 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, FIRSTNAME
FROM EMP
WHERE EMPNO <> '000010'
```

### Stage 2

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

Stage 2 predicates are generally those that contain column expressions, correlated subqueries, and CASE expressions, among others. A predicate can also appear to be stage 1, but processed as stage 2. For example, any predicate processed after a join operation is stage 2. Also, although DB2 does a good job of promoting mismatched data types to stage 1 via casting (as of version 8) 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, FIRSTNAME
FROM EMP
WHERE EMPNO > '00000010'
```

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

### Stage 3

A stage 3 predicate is a fictitious predicate we've made up to describe filtering that occurs after the data has been retrieved from DB2. That is, filtering that is done in the application program. You should have no stage 3 predicates. Performance is best served when all filtering is done in the data server, and not in the application code. Imagine for a moment that 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.
```

This is an example of a stage 3 predicate. The data retrieved from the table isn't used unless the condition is false. Why did this happen? Because the programmer was told that no stage 2 predicates are allowed in SQL statements as part of a shop standard. The truth is, however, that the following stage 2 predicate would always outperform a stage 3 predicate:

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

### Combining Predicates

It should be known that when simple predicates are connected together by an OR condition, the resulting compound predicate will be evaluated at the higher stage of the two simple predicates. The following example contains two simple predicates 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

Simply put, 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 can be considered for single index access. Non-Boolean term predicates can at best be considered for multi-index access. For example, the following predicate contains Boolean term predicates:

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

If the predicate on the LASTNAME column evaluates as false, then the entire WHERE clause is false. The same is true for the predicate on the MIDINIT column. DB2 can take advantage of this in that it could utilize an index (if available) on either 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, then the rest of the WHERE clause must be evaluated. These predicates are non-Boolean term.

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

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

A redundant predicate can be added that, while making the WHERE clause functionally equivalent, contains a Boolean term predicate, and thus could take better advantage of an index on the LASTNAME column:

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

### Predicate Transitive Closure

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

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

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

Predicates of the form  $\langle \text{COL op value} \rangle$ , where the operation is an equals or range predicate, are available for predicate transitive closure. So are BETWEEN predicates. 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 feel they can provide a benefit.

## Coding Efficient SQL

There's a saying that states "the most efficient SQL statement is the SQL statement that is never executed". Recently, someone was overheard to say that the most efficient SQL statement is the one that is never written. Funny stuff. However, there is truth in those sayings. We need to do two things when we place SQL statements into our applications. The first is that we need to minimize the number of times we go to the data server. The second is that if we need to go to the data server we should do so in the most efficient manner. Remember, there is no right or wrong answer, there are only trade-offs to performance.

### Avoid Unnecessary Processing

The very first thing you need to ask yourself is if a SQL statement is necessary. The number one performance issue in recent times has to be the quantity of SQL issued. The main reason behind 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 you pay, however, is performance.

You need to look at your SQL statements and ask some questions. Is the statement needed? Do I really 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

There are some basic guidelines for coding SQL statements for the best performance:

- **Retrieve the fewest number of rows** The only rows of data returned to the application should be those actually needed for the process and nothing else. We should never find a data filtering statement in a program, as DB2 should be handling all the filtering of the rows. Whenever there is a process that needs to be performed on the data, and there is a decision as to where the process is done, program or in DB2, then leave it in DB2. This is increasingly the case as DB2 evolves into the "ultimate" server, and more and more applications are distributed across the network and using DB2 as the database. There will always be that rare situation when all the data needs to be brought to the application, filtered, transformed and processed. However, this is generally 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 the columns needed by the program** Retrieving extra columns 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 program's work area. That is an unnecessary expenditure of CPU. You should code your SQL statements to return only the columns required. Can these mean possibly coding multiple SQL statements against the same table for different sets of columns? Yes, it does. However, that is the effort 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 SQL statements are issued that don't need to be. This is especially true for programmatic joins. If separate application programs are retrieving data from related tables, then this can result in extra SQL statements issued. Code SQL joins instead of programmatic joins. In one simple test of a programmatic join of two tables in a COBOL program compared to the equivalent SQL join, the SQL join consumed 30% less CPU.
- **Use stage 1 predicates** You should be aware of the stage 1 indexable, stage 1, and stage 2 predicates, and try to use stage 1 predicates for filtering whenever possible. See this section in this chapter on promoting predicates to help 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 only has logic that will retrieve some specific data or data relationship, 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 we've seen. Generic SQL and generic I/O layers do not belong in high performing systems.
- **Avoid unnecessary sorts** Avoid unnecessary sorting is a requirement in any application but more so in any high performance environment especially with a database involved. Generally, if ordering is always necessary then there should probably be indexes to support the ordering. Sorting can be caused by GROUP BY, ORDER BY, DISTINCT, INTERSECT, EXCEPT, and join processing. If ordering is required for a join process, it generally is a clear indication that an index is missing. If ordering is necessary after a join process, then hopefully the result set is small. The worse possible scenario is where a sort is performed as the result of the SQL in a cursor, and the application only processes a subset of the data. In this case the sort overhead needs to be removed. It is more an application of common sense. If the SQL has to sort 100 rows and only 10 are processed by the application, then it may be better to sort in the application, or perhaps an index needs to be created to help avoid the sort.
- **Only sort truly necessary columns** When DB2 sorts data, the columns used to determine the sort order actually appear twice in the sort rows. Make sure that if you specify the sort only on the 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 can make a statement easier to read, easier to convert between inner and outer join, and harder to forget to code a join predicate.
- **Avoid UNIONS (not necessarily UNION ALL)** Are duplicates possible? If not then replace the UNION with a UNION ALL. Otherwise, see if it is possible to produce the same result with an outer join, or in a situation in which the subselects of the UNION are all against the same table try using a CASE expression and only one pass through the data instead.
- **Use joins instead of subqueries** Joins can outperform subqueries for existence checking if there are good matching indexes for both tables involved, and if the join won't introduce any duplicate rows in the result. DB2 can take advantage of predicate transitive closure, and also pick the best table access sequence. These things are not possible with subqueries.

- **Code the most selective predicates first** DB2 processes the predicates in a query in a specific order:
  - Indexable predicates are applied first in the order of the columns of the index
  - Then other stage 1 predicates are applied
  - Then stage 2 predicates are applied

Within each of the stages above DB2 will process the predicates in this 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
- Then all other predicate types

Within each grouping DB2 will process the predicates in the order they have been coded in the SQL statement. Therefore, all SQL queries should be written to evaluate the most restrictive predicates first to filter unnecessary rows earlier, reducing processing cost at a later stage. This includes subqueries as well (within the grouping of correlated and non-correlated).

- **Use the proper method for existence checking** For existence checking using a subquery an EXISTS predicate generally will outperform an IN predicate. For general existence checking you could code a singleton SELECT statement that contains a FETCH FIRST 1 ROW ONLY clause.
- **Avoid unnecessary materialization** Running a transaction that processes little or no data? You might want to use correlated references to avoid materializing large intermediate result sets. Correlated references will encourage nested loop join and index access for transactions. See the advanced SQL and performance section of this chapter for more details.

### Promoting Predicates

We need to code SQL predicates as efficiently as possible, and so when you are writing predicates you should make sure to use stage 1 indexable predicates whenever possible. If you've coded a stage 1 non-indexable predicate, or a stage 2 predicate, then you should be asking yourself if you can possibly promote those predicates to a more efficient stage.

If you have a stage 1 non-indexable predicate, can it be promoted to stage 1 indexable? Take for example the following predicate:

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

If we use the “end of the DB2 world” as an indicator of data that is still active, then why not change the predicate to something that is indexable:

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

Do you have a stage 2 predicate that can be promoted to stage 1, or even stage 1 indexable? Take for example this predicate:

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

The predicate above is applying 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 we can make the predicate stage 1 indexable:

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

These examples can be applied as general rules for promoting predicates. It should be noted that as of 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's not possible to eliminate the column expression in the query.

### Functions, Expressions and Performance

DB2 has lots of scalar functions, as well as other expressions that allow you to manipulate data. This contributes to the fact that the SQL language is indeed a programming language as much as a data access language. It should be noted, however, that the processing of functions and expressions in DB2, if not for filtering of data, but instead for pure data manipulation, are generally more expensive than the equivalent application program process. Keep in mind that even the simple functions such as concatenation:

```
SELECT FIRSTNAME CONCAT LASTNAME  
FROM EMP
```

will be executed for every row processed. If you need the ultimate in ease of coding, time to delivery, portability, and flexibility, then code expressions and functions like this in your SQL statements. If you need the ultimate in performance, then do the manipulation of data in your application program.

CASE expressions can be very expensive, but CASE expressions will utilize “early out” logic when processing. Take the following CASE expression as an example:

```
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, then the following functionally equivalent CASE expression will consume significantly less CPU:

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

DB2 will take the early out from the Case expression for the first not true of an AND, or the first TRUE of an OR. So, in addition to testing for the blank value above, 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. Because the possibilities with advanced SQL are endless it would be impossible to document all of the various situations and the performance impacts. Therefore, this section will give some examples and tips for improving performance of some complex queries. Please keep in mind, however, that complex SQL will always be 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 can be very good for your transaction queries that process very little data. However, it can be bad for your report queries that process vast quantities of data. The following query is generally a very good performer 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
```

If there is one sales rep per department, or if most of the employees are sales reps, then the above query would be the most efficient way to retrieve the data. In the query above, the entire employee table will be read and materialized in the nested table expression called TAB2. It is very likely that the merge scan join method will be used to join the materialized TAB2 to the first table expression called TAB1.

Suppose now that the employee table is extremely large, but that there are very few sales reps, or perhaps all the sales reps are in one or a few departments. Then the above query may not be the most efficient due to the fact that the entire employee table still has to be read in TAB2, but most of the results of that nested table expression won't be returned in the query. In this case, the following query may be 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'
```

While this statement is functionally equivalent to the previous statement, it operates in a very different way. In this query the employee table referenced as TAB1 will be read first, and then the nested table expression will be executed repeatedly in a nested loop join for each row that qualifies. An index on the WORKDEPT column is a must.

**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 will possibly merge that nested table expression or view with the referencing statement. If DB2 cannot merge then it will materialize the view or nested table expression into a work file, and then apply the referencing statement to that intermediate result. IBM states that merge is more efficient than materialization. In general, that statement is correct. However, materialization may be 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 via application of functions, user-defined functions, or other expressions
- References to the generated columns in the outer referencing statement

In general, DB2 will materialize when some sort of aggregate processing is required inside the view or nested table expression. So, typically this means that the view or nested table expression contains aggregate functions, grouping (GROUP BY), or DISTINCT. If materialization is not required then the merge process happens. Take for example the following query:

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

DB2 will materialize TAB1 in the above example. Now, take a look at the next 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 there is no DISTINCT, GROUP BY, or aggregate functions, and so DB2 will merge in inner table expression with the outer referencing statement. Since there are two references to COL1 in the outer referencing statement, then the CASE expression in the nested table expression will be calculated twice during query execution. The merged statement would look something like this:

```
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 you may want to introduce a non-deterministic function into the view or nested table expression to force materialization. We 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 allows for some really complex SQL processing, but also enables you to create logically partitioned tables. That is, you can store data in multiple tables and then reference them all together as one table in a view. This is useful for quickly rolling through yearly tables, or to create optional table scenarios with little maintenance overhead.

While each SQL statement in a union in view (or table expression) results in an individual query block, and SQL statements written against our view are distributed to each query block, DB2 does employ a technique to prune query blocks for efficiency. DB2 can, depending upon the query, prune (eliminate) query blocks at either statement compile time or during statement execution. If we consider the account history view:

```
CREATE VIEW V_ACCOUNT_HISTORY
  (ACCOUNT_ID, AMOUNT) AS
SELECT ACCOUNT_ID, AMOUNT
FROM   HIST1
WHERE  ACCOUNT_ID BETWEEN 1 AND 10000000
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 both of the query blocks generated. However, DB2 will compare the distributed predicate against the predicates coded in the UNION inside our view, looking for redundancies. In any situations in which the distributed predicate renders a particular query block unnecessary, DB2 will prune (eliminate) that query block from the access path. So, when our distributed predicates look like this:

```

. . .
WHERE  ACCOUNT_ID BETWEEN 1 AND 100000000
AND    ACCOUNT_ID = 12000000

. . .
WHERE  ACCOUNT_ID BETWEEN 100000001 AND 200000000
AND    ACCOUNT_ID = 12000000

```

DB2 will prune the query blocks generated at statement compile time base upon the literal value supplied in the predicate. So, although in our example above two query blocks would be generated, one of them will be pruned when the statement is compiled. DB2 compares the literal predicate supplied in the query against the view with the predicates in the view. Any unnecessary query blocks are pruned. So, since one of the resulting combined predicates is impossible, DB2 eliminates that query block. Only one underlying table will then be 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. So, let's take the previous query example, and replace the literal with a host variable:

```

SELECT *
FROM   V_ACCOUNT_HISTORY
WHERE  ACCOUNT_ID = :H1

```

If this statement was embedded in a program, and bound into a plan or package, two query blocks would be generated. This is because DB2 does not know the value of the host variable in advance, and distributes the predicate amongst both generated query blocks. However, at run time DB2 will examine the supplied host variable value, and dynamically prune the query blocks appropriately. So, if the value 12000000 was supplied for the host variable value, then one of the two query blocks would be pruned at run time, and only one underlying table would be accessed. This is a complicated process that does not always work. You should test it by stopping one of the tables, and then running a query with a host variable that should prune the query block on that table. If the statement is successful then runtime query block pruning is working for you.

We can get query block pruning on literals and host variables. However, we can't get query block pruning 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, we have recommended using programmatic joins in situations in which the query can benefit from runtime query block pruning using the joining column. Please be aware of the fact that this is an extremely specific recommendation, and certainly not a general recommendation. You should also be aware of the fact that there are limits to UNION in view (or table expression), and that you should always test to see if you get bind time or run time query block pruning. In some cases it just doesn't happen, and there are APARs out there that address the problems, but are not comprehensive. So, testing is important.

You can influence proper use of runtime query block pruning by encouraging distribution of joins and predicates into the UNION in view (see section below). This is done by reducing the number of tables in the UNION, or by repeating host variables in predicates instead of, or in addition to using correlation. Take a look at the query below.

```
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 isn't. The same goes for the predicate on the HIST\_EFF\_DTE predicate in the subquery. The reason for repeating the host variable and value references was to take advantage of the runtime pruning. Correlated predicates would not have gotten the pruning.

### Recommendations for Distributed Dynamic SQL

DB2 for z/OS is indeed the ultimate data server, and there is a proliferation of dynamic distributed SQL hitting the system, either via JDBC, ODBC, or DB2 CLI. Making sure this SQL is always performing the best can sometimes be a challenge. Here are some general recommendations for improving the performance of distributed dynamic SQL:

- Turn on the dynamic statement cache.
- Use a fixed application level authorization id. This allows accounting data to be grouped by the id, and applications can be identified by their id. It also makes for a more optimal 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 only helpful when there is a skewed distribution of data, and that skewed distribution is properly reflected via frequency distribution or histogram statistics in the system catalog.
- Use the EXPLAIN STMTCACHE ALL command to expose all the statements in the dynamic statement cache. The output from this command goes into a table that contains all sorts of runtime performance information. In addition, you can EXPLAIN individual statements in the dynamic statement cache once they have been identified.
- Use connection pooling.
- Consolidate calls into stored procedures. This only works if multiple statements and program logic, representing a transaction, are consolidated together in a single stored procedure.
- Use the DB2 CLI/ODBC/JDBC Static Profiling feature. This CLI feature can be used to turn dynamic statements into static statements. This is documented in the DB2 Call Level Guide and Reference, Volume 1. Static profiling enables you to capture dynamic SQL statements on the client, and then bind the statements on the server into a package. Once 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.

Be aware of the fact that if you are moving a local batch process to a remote server, you are going to 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 very good at picking the correct access path, as long as it is given the proper information. In situations in which it does not pick the optimal access path it typically has not been given enough information. This is sometimes the case when we are using parameter markers or host variables in a statement.

### Proper Statistics

What can we say? DB2 utilizes a cost-based optimizer, and that optimizer needs accurate statistical information about your data. Collecting the proper statistics is a must to good performance. With each new version of DB2 the optimizer takes more advantage of catalog statistics. This also means that with each new version DB2 is more dependent upon catalog statistics. You should have statistics on every column referenced in every WHERE clause in your shop. If you are using parameter markers and host variables then in the least you need cardinality statistics. If you have skewed data, or are using literal values in your SQL statements, then perhaps you need frequency distribution and/or histogram statistics. If you suspect columns are correlated, you can gather column correlation statistics. To determine if columns are correlated you can run these two queries (DB2 V8, and DB2 9):

```
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 then 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 can give you a good indication as to whether or not you need frequency distribution statistics or histogram statistics, and runtime reoptimization for skewed data distributions.

### Runtime Reoptimization

If your query contains a predicate with an embedded literal value then DB2 knows something about the input to the query, and can take advantage of frequency distribution or histogram statistics if available. This can result in a much improved filter factor, and better access path decisions by the optimizer. However, what if DB2 doesn't know anything about your input value:

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

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

DB2 can employ something called runtime reoptimization to help your queries. For static SQL the option of REOPT(ALWAYS) is available. This bind option will instruct DB2 to recalculate access paths at runtime using the host variable parameters. This can result in improved execution time for large queries. However, if there are many queries in the package then they will all get reoptimized. This could negatively impact statement execution time for these queries. In situations where you use REOPT(ALWAYS) consider separating the query that can benefit into it's own package.

For dynamic SQL statements there are three options, REOPT(ALWAYS), REOPT(ONCE), and REOPT(AUTO). REOPT(AUTO) is DB2 9 only.

- **REOPT(ALWAYS)** This will reoptimize a dynamic statement with parameter markers based upon the values provided on every execution.
- **REOPT(ONCE)** Will reoptimize a dynamic statement the first time it is executed base upon the values provided for parameter markers. The access path will then be reused until the statement is removed from the dynamic statement cache, and needs to be prepared again. This reoptimization option should be used with care as the first execution should have good representative values.
- **REOPT(AUTO)** Will track how the values for the parameter markers change on every execution, and will then reoptimize the query based upon those values if it determines that the values have changed significantly

There is also a system parameter called REOPTTEXT (DB2 9) that enables the REOPT(AUTO) like behavior, subsystem wide, for any dynamic SQL queries (without NONE, ALWAYS, or ONCE already specified) that contain parameter markers when it detects changes in the values that could influence the access path.

### OPTIMIZE FOR Clause

One thing that the optimizer certainly does not know about your query is how much data you are actually going to fetch from the query. DB2 is going to use the system catalog statistics to get an estimate of the number of rows that will be 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, then you should use the OPTIMIZE FOR n ROWS clause, where n is the number of rows you intent to fetch.

The OPTIMIZE FOR clause is a way, within a SQL statement, to tell DB2 how many rows you intent to process. DB2 can then make access path decisions in order to determine the most efficient way to access the data for that quantity. The use of this clause will discourage such things as list prefetch and sequential prefetch, and multi-index access. It will encourage index usage to avoid a sort, and a join method of nested loop join. A value of 1 is the strongest influence on these factors.

You should actually put the number of rows you intend to fetch in the OPTIMIZE FOR clause. Incorrectly representing the number of rows you intent to fetch can result in a more poorly performing query.

### Encouraging Index Access and Table Join Sequence

There are certain techniques you can employ to encourage DB2 to choose a certain index or a table access sequence in a query that accesses multiple tables:

When DB2 joins tables together in an inner join it attempts to select the table that will qualify the fewest rows first in the join sequence. If, for some reason, DB2 has chosen the incorrect table first (maybe due to statistics or host variables) then you can attempt to change the table access sequence by employing one or more of these techniques:

- Enable predicates on the table you want to be first. By increasing potential matchcols on this table DB2 may select an index for more efficient access and change the table access sequence.
- Disable predicates on the table you don't want accessed first. Predicate disablers are documented in the DB2 administration Guide (DB2 V7, DB2 V8) or the DB2 Performance Monitoring and Tuning Guide (DB2 9). We do not recommend using predicate disablers.
- Force materialization of the table you want accessed first by placing it into a nested table expression with a DISTINCT or GROUP BY. This could change the join type as well as 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, then the outer query is executed, and then any correlated subqueries are executed. Of course, this is only effective if the table moved from a join to a subquery doesn't have to return data.
- Convert a joined table to a correlated nested table expression. This will force another table to be accessed first as the data for the correlated reference is required prior to the table in the correlated nested table expression being accessed.
- Convert an inner join to a left join. By coding a left join you have absolutely dictated the table join sequence to DB2, as well as the fact that the right table will filter 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 will encourage DB2 to access that table first in order 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 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. You can also try converting from implicit join syntax to explicit join syntax and vice versa.
- Try coding a predicate on the table you want accessed first as a non-transitive closure predicate, for example 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. Since the subquery is not eligible for transitive closure, then DB2 will not generate the predicate redundantly against the other table, and has less encouragement to choose that table first.

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

- Code an ORDER BY clause on the leading columns of the index you want chosen. This may encourage DB2 to chose that index to avoid a sort.
- Add columns to the index to make the access index-only.
- Increase the index matchcols either by modifying the query or the index.
- You could disable predicates that are matching other indexes. The IBM manuals document predicate disablers. We don't recommend them.
- Try using the OPTIMIZE FOR clause.

CA, one of the world's largest information technology (IT) management software companies, unifies and simplifies complex IT management across the enterprise for greater business results. With our Enterprise IT Management vision, solutions and expertise, we help customers effectively govern, manage and secure IT.

HB05ESMDBM01E MP318760707