Indexing and Statistics Strategies for DB2 UDB for iSeries

Version 3.0

by

Michael W. Cain

iSeries Teraplex Integration Center
IBM eServer Solutions

Updated November 2003

© Copyright IBM Corporation, 2003. All Rights Reserved.

All trademarks or registered trademarks mentioned herein are the property of their respective holders.

Table of Contents

Introduction ... 1
Part 1: The Basics ... 2
Database Indexes Defined ... 2
Primary and Secondary Indexes — One and the Same ... 2
Binary Radix Tree Indexes ... 3
Bitmap Indexes — an Industry Solution for Ad-Hoc Queries ... 4
Encoded Vector Indexes — an IBM Solution for Bitmap Indexes ... 5
How the Database Uses Indexes ... 6
Indexes and the Optimizer ... 7
Selection ... 8
Nested Loop Join via Index ... 10
Grouping and Ordering ... 12
Indexes and Optimization: A Summary ... 15
Part 2: Indexing Strategies for Performance Tuning ... 16
A General Approach ... 16
A Proactive Approach ... 17
Perfect Radix Index Guidelines ... 18
Encoded Vector Index Guidelines ... 21
Proactively Tuning Many Queries ... 22
Reactive Query Tuning ... 22
Creating Indexes for Multi-table Queries (Joins) ... 25
Tuning for One Query versus Tuning for Many ... 26
Other Indexing Tips ... 26
Part 3: Indexing Considerations ... 27
SQL Indexes versus Keyed Logical Files ... 27
Estimating the Number of Indexes to Create ... 28
Index Creation ... 28
Index Maintenance ... 30
EVI Maintenance ... 31
EVI Maintenance Overview ... 33
General Index Maintenance Recommendations ... 34
Recommendations for EVI Use ... 34
Part 4: Statistics Manager (V5R2) ... 35
Part 5: Statistics Strategies for Performance Tuning ... 37
Statistics on Demand! ... 37
Proactive Strategy ... 38
Reactive Strategy ... 38
Identifying Columns with Statistics ... 39

What Columns Should Have Stats Available? 39
Maintaining Statistics 41
Summary 44
Appendix A — Examples Queries and Possible Indexing Strategies 45
Appendix B — Tools for Analysis and Tuning 55
Optimizer Feedback via “Debug Mode” 55
Database Monitor 55
iSeries Navigator 55
PRTSQLINF 56
Visual Explain 56
References 57
Books 57
Web sites 57
Related Education 57
Other Notices 58
About the Author 58
Acknowledgments 58
Production 58
For Additional Information 58
Trademarks and Disclaimers 59

Introduction

On any platform, good database performance depends on good design. And good design includes a solid understanding of indexes and column statistics: how many to build, their structure and complexity, and their maintenance requirements.

This is especially true for DB2® UDB for iSeries™, which provides a robust set of choices for indexing and allows indexes to play a key role in several aspects of query optimization. On the IBM® eServer™ iSeries platform, the use of indexes is a powerful tool, but also requires some knowledge on their application.

This paper starts with basic information about indexes in DB2 UDB for iSeries, the data structures underlying them, and how the server uses them. In the second part of the paper, index strategies are presented. Part three discusses additional indexing considerations related to maintenance, tools, and methods. Parts four and five cover column statistics and statistics collection strategies. And finally, the appendices provide examples and references.

This paper provides an initial look at indexing and statistics strategies and their effects on query performance. It is ***strongly recommended*** that database administrators, analysts, and developers who are new to the iSeries server or SQL, attend the "DB2 UDB for iSeries SQL and Query Performance Monitoring and Tuning" workshop. This course will teach the developer the proper way to architect and implement a high-performing DB2 UDB for iSeries solution. More information about this workshop can be found at:
ibm.com/eserver/iseries/service/igs/db2performance.html

Because the AS/400® system (the predecessor to iSeries servers) was designed before SQL was widely used, a proprietary language and set of APIs were made available for relational database creation and data access — Data Definition Specification (DDS) and OS/400® file commands. Also known as the native database interface, DDS and OS/400 file commands can still be used for creating DB2 objects on iSeries servers.

Because of this native, non-SQL interface, some iSeries developers and consultants will use terminology not familiar to those coming from a pure SQL background. Here is a mapping of that terminology:

SQL Term	***iSeries Term***
TABLE	PHYSICAL FILE
ROW	RECORD
COLUMN	FIELD
INDEX	KEYED LOGICAL FILE, ACCESS PATH
VIEW	NON-KEYED LOGICAL FILE
SCHEMA	LIBRARY, COLLECTION
LOG	JOURNAL
ISOLATION LEVEL	COMMITMENT CONTROL LEVEL

Due to the integrated nature of the iSeries database, both the native and SQL interfaces are almost completely interchangeable. Objects created with DDS can be accessed with SQL statements; and objects created with SQL can be accessed with the native record level access APIs. The DB2 UDB SQL interface is compliant with the SQL-92 entry level standard and has implemented over 90% of the updated standard, SQL-99.

Part 1: The Basics

Before describing indexing strategies and options for index creation, it is important to understand: what indexes are, what purposes they serve in DB2 UDB for iSeries, and their relationship to the query optimizer.

Database Indexes Defined

All relational database management systems (RDBMSs) have data structures called indexes. An index in a book allows you to quickly locate information on a specific topic without sequentially paging through the book. Database indexes provide similar benefits by providing a relatively quick method of locating data of interest. Without indexes, the database will probably be forced to perform a full sequential search or scan, accessing every row in the database table. Depending on the size of the tables and the complexity of the query, a full table scan can be a lengthy process, while also consuming a large amount of system resources.

Indexed scans are more efficient than full table scans since the index key values are usually shorter than the length of the database table row. Shorter entries means that more index entries can be stored in a single page. Indexing results in a considerable reduction in the total number of pages that must be processed (I/O requests) in order to locate the requested data. While indexed scans can improve performance, the complexity of the query and the data will determine how effectively the data access can be implemented. Different queries stress the database in unique ways and that is why different index types are needed to cope with ever-changing workloads of users. In addition to simply retrieving data more efficiently, indexes also assist in the ordering, grouping, and joining of data from different tables.

With DB2 UDB for iSeries, there are two kinds of persistent indexes: binary radix tree indexes, which have been available since AS/400 systems began shipping in 1988, and encoded vector indexes (EVIs), which became available in 1998 with OS/400 Version 4 Release 3. Both types of indexes are useful in improving performance for certain kinds of queries. This paper will explain the differences between the indexes and provide advice regarding when and how to use them.

Primary and Secondary Indexes — One and the Same

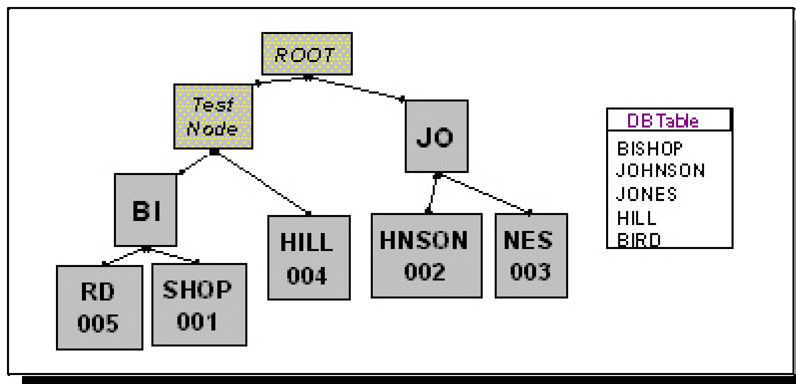
On platforms that rely on partitioning schemes, the database must distinguish between primary and secondary indexes. Primary indexes are those created with the partitioning key as the primary key. Secondary indexes are built over columns other than the partitioning key. On these platforms, primary indexes provide the majority of data retrieval.

Because of the integrated storage management and single level storage architecture on iSeries servers, there is no data partitioning on a single server; the data is automatically spread across all available disk units. One result is that all indexes are effectively primary indexes. In fact, there is no distinction between primary and secondary indexes. There is also no concept of a clustered index — where the table data is kept in the same physical order as the primary index key(s).

The net result of the integrated and automatic storage management system is that there is no need to consider the physical placement of tables or indexes on an iSeries server.

Binary Radix Tree Indexes

All commercially available RDBMSs use some form of binary tree index. A radix index is a multilevel, hybrid tree structure that allows a large number of key values to be stored efficiently while minimizing access times. A key compression algorithm assists in this process. The lowest level of the tree contains the leaf nodes, which house the address of the rows in the base table that are associated with the key value. The key value is used to quickly navigate to the leaf node with a few simple binary search tests.



Thus, a single key value can be accessed quickly with a small number of tests. This quick access is pretty consistent across all key values in the index since the server keeps the depth of the index shallow and the index pages spread across multiple disk units.

The binary radix tree structure is very good for finding a small number of rows because it is able to find a given row with a minimal amount of processing. For example, using a binary radix index over a customer number column for a typical OLTP request, such as "find the outstanding orders for a single customer," will result in fast performance. An index created over the customer number field would be considered the perfect index for this type of query because it allows the database to zero in on the rows it needs and perform a minimal number of I/Os.

In business intelligence environments, database analysts do not always have the same level of predictability. Increasingly, users want ad-hoc access to the detail data underlying their data marts. They might, for example, run a report every week to look at sales data, then "drill down" for more information related to a particular problem area they found in the report. In this scenario, the database analysts

cannot write all the queries in advance on behalf of the end users. Without knowing what queries will be run, it is impossible to build the perfect index.

Traditionally, the solution to this dilemma has been to either restrict ad-hoc query capability or define a set of indexes that cover most columns for most queries. With DB2 UDB for iSeries, the optimizer can intelligently use less-than-perfect indexes for many types of queries. But as the size of data warehouses grows into the terabyte range, less than perfect becomes less palatable.

Experts throughout the industry have recognized this less-than-perfect solution, and have developed new types of indexes that can be combined dynamically at run-time to cover a broader range of ad-hoc queries.

Bitmap Indexes — an Industry Solution for Ad-Hoc Queries

The need for newer index technologies has spawned the generation of a variety of similar solutions that can be collectively referred to as "bitmap indexes." The concept of a bitmap index is an array of distinct key values. For each value, the index stores a bitmap, where each bit represents a row in the table. If the bit is set on, then that row contains the specific key value.

Bitmap Index over STATE column

Arizona	100000001000...
California	000010000000...
....	
Vermont	000101000000...
Virginia	011000110000...

With this indexing scheme, bitmaps can be combined dynamically using Boolean arithmetic (ANDing / ORing) to identify only those rows that are required by the query. Unfortunately, this improved access comes with a price. In a Very Large Database (VLDB) environment, bitmap indexes can grow to ungainly sizes. In a one billion row table, for example, there will be one billion bits for each distinct value. If the table contains many distinct values, the bitmap index quickly becomes enormous. Usually, RDBMSs rely on some sort of compression algorithm to help alleviate this growth problem.

In addition, maintenance of very large bitmap indexes can be problematic. Every time the database is updated, the system must update each bitmap — a potentially tedious process if there are, say, thousands of unique values in a one billion row table. When adding a new distinct key value, an entire bitmap must be generated. These issues typically result in the database being used as "read only."

Encoded Vector Indexes — an IBM Solution for Bitmap Indexes

Realizing the limitations of bitmap indexes, IBM Research set out to find a better solution. The result is Encoded Vector Indexes (EVIs) — a new, patented indexing technology from IBM. DB2 UDB for iSeries is the first member of the IBM DB2 family to provide EVIs.

An EVI is a data structure that is stored as basically two components: the symbol table and the vector. The symbol table contains a distinct key list, along with statistical and descriptive information about each distinct key value in the index. The symbol table maps each distinct value to a unique code. The mapping of any distinct key value to a 1-, 2-, or 4-byte code provides a type of key compression. Any key value, of any length, can be represented by a small bytecode.

The other component, the vector, contains a bytecode value for each row in the table. This bytecode represents the actual key value found in the symbol table and the respective row in the database table. The bytecodes are in the same ordinal position in the vector, as the row it represents in the table. The vector does not contain any pointer or explicit references to the data in the table.

The optimizer can use the symbol table to obtain statistical information about the data and key values represented in the EVI. If the optimizer decides to use an EVI to process the local selection of the query, the database engine uses the vector to build a dynamic bitmap, which contains one bit for each row in the table. The bits in the bitmap are in the same ordinal position as the row it represents in the table. If the row satisfies the query, the bit is set on. If the row does not satisfy the query, the bit is set off. The database engine can also derive a list of relative row numbers (RRNs) from the EVI. These RRNs represent the rows that match the selection criteria, without the need for a bitmap.

EVI composed of two sub-objects:

SYMBOL TABLE:

Key Valuec	Code	First Row	Last Row	Count
Arizona	1	1	80005	5000
California	2	5	99760	7300
.....				
Vermont	49	4	30111	340
Virginia	50	2	83000	2760

VECTOR:

Code	1
Code	50
Code	50
Code	49
Code	2
Code	49
Code	50
Code	50
Code	1
...	

As with traditional bitmap indexes, the DB2 UDB dynamic bitmaps or RRN lists can be ANDed and ORed together to satisfy an ad-hoc query. For example, if a user wants to see sales data for a certain region during a specific time period, the database analyst can define an EVI over the Region column and the Quarter column of the database. When the query runs, the database engine will build dynamic bitmaps using the two EVIs and then AND the bitmaps together to produce a bitmap that represents all the local selection (bits turned on for only the relevant rows). This ANDing capability effectively utilizes more than one index to drastically reduce the number of rows that the database engine must retrieve and process.

Since EVIs were created primarily to support business intelligence and ad-hoc query environments, there are EVI creation and maintenance considerations, as well as recommendations — both of which will be covered in later sections.

How the Database Uses Indexes

OS/400 is an object-based operating system. Tables and indexes are objects. Like all objects, information about the object's structure, size, and attributes are contained within the table and index objects. In addition, tables and indexes contain statistical information about the number of distinct values in a column and the distribution of those values in the table. The DB2 UDB for iSeries optimizer uses this information to determine how to best access the requested data for a given query request.

For database designers and analysts coming from other platforms, this means that most of the information about tables and indexes is maintained in the object itself or derived in real time from the database objects. Unless specified explicitly by an analyst, indexes are maintained immediately, so the optimizer always has current information about the database.

Because the information that the optimizer needs is gathered automatically by the database engine, and maintained in the database objects themselves, administrators should never have to manually compile statistics for the optimizer. When the optimizer prepares an access plan for a query, it consults the objects themselves and gathers information about: the size of the object, the relevant indexes built over that object, and the columns that are required to run the query. Like other RDBMSs, the system saves its access plans for reuse whenever possible, but also has the ability to automatically change the access plan if the environment changes. This is sometimes referred to as "late binding."

Most importantly, the database engine can use indexes to identify and retrieve rows from a database table. As in any RDBMS, the optimizer can choose whether or not to use an index to identify and retrieve rows from a table. In general, the optimizer chooses the index that will efficiently narrow the number of rows matching the query selection, as well as for joining, grouping, and ordering operations. Put another way, the index is used to reduce the number of I/Os required to retrieve the data from disk, and to logically group and order the data.

Within Version 5 Release 2 of OS/400, DB2 UDB for iSeries incorporated enhancements to gather and maintain statistical information in new ways. Additional details about these enhancements can be found later in this paper (in Part 4: Statistics Manager, and in Part 5: Statistics Strategies for Performance Tuning).

Indexes and the Optimizer

In order to process a query, the database must build an access plan. Think of the access plan as a recipe, with a list of ingredients and methods for cooking. The optimizer is the component of the database that builds the recipe. The ingredients are the tables and indexes required for the query. The optimizer looks at the ingredients it has available for a given query, estimates which ones are the most cost-effective, and builds a set of instructions on how to use the ingredients. On an iSeries server, lower level database engine components do the cooking.

Since the iSeries optimizer uses cost-based optimization, the more information given about the rows and columns in the database, the better able the optimizer is at creating the best possible (least costly / fastest) access plan for the query. With the information from the indexes, the optimizer can make better choices about how to process the request (local selection, joins, grouping, and ordering).

The primary goal of the optimizer is to choose an implementation that quickly and efficiently eliminates the rows *that are not interesting* or required to satisfy the request. Normally, query optimization is concerned with trying to find the rows of interest. A proper indexing strategy will assist the optimizer and database engine with this task.

To understand indexing strategy, it is important to understand the science of query optimization and the possible implementation methods. With the limited scope of this paper, the implementation methods will only be covered at a high level, tying the use of indexes to the respective methods.

Here is an overview of the available implementation methods:

- Selection
  - Table Scan
  - Table Probe*
  - Index Scan* (also known as index selection)
  - Index Probe* (also known as key row positioning)
- Joining
  - Nested Loop Join via Index *
  - Nested Loop Join via Hashing (also known as hash join)
- Grouping
  - Grouping via Index*
  - Grouping via Hashing
- Ordering
  - Ordering via Index*
  - Ordering via Sort

* The method directly or indirectly relies on an index for implementation.

DB2 UDB for iSeries also supports parallelism when the optional OS/400 feature “DB2 Symmetric Multiprocessing” is installed. Parallelism is achieved via multiple tasks or threads that work on part, or the entirety, of the query request. Most, but not all, of the implementation methods are parallel-enabled.

Selection

The main job of an index is to reduce the number of I/Os that the database must perform. This is the first and most important aspect of query optimization. The sooner a row can be eliminated, the faster the request will be. In other words, the fewer number of rows the database engine has to process, the better, since I/O operations are usually the slowest element in the implementation.

For example, look at the following query that is asking for some customer information on orders (where the order was shipped on the first day of the month and the order amount is greater than 1,000):

```
SELECT CUSTOMER_NAME, ORDERNUM, ORDERDATE, SHIPDATE, AMOUNT
       FROM ORDER_TABLE
       WHERE SHIPDATE IN ('2000-06-01', '2000-07-01', '2000-08-01')
       AND AMOUNT > 1000
```

If the table is relatively small, it will not make much difference how the optimizer decides to process the query. The result will return quickly. However, if the table is large, choosing the appropriate access method becomes very important. If the number of rows that satisfy the query is small, it would be best to choose an access method that logically eliminates the rows that *do not match*, as quickly and efficiently as possible. This is where indexes are valuable.

To decide whether or not an index would help, it is important to have an estimate of the number of rows that satisfy this query. For example, if 90% of the rows satisfy this query, then the best way to access the rows is to perform a full table scan. But if only 1% of the rows satisfy the query, then a full table scan might be very inefficient, resource intensive, and ultimately slower. In this case, an index-based, keyed-access method would be most effective.

On an iSeries server, the optimizer estimates the number of rows that satisfy this query by looking at the query request, table attributes, and information in the indexes. Although the header information in each table contains information about the number of rows in the table, the header information will not tell the optimizer about the number of distinct values in an index or the number of rows that may contain a particular value.

Instead, indexes are used to derive this type of information. Both radix and encoded vector indexes contain information about the number of distinct values in a column and the distribution of values. DB2 UDB for iSeries is one of the few databases that can recognize data skew during optimization.

With radix indexes, the optimizer obtains cost information from the leftmost, contiguous keys in the index. In addition to knowing how many distinct values are in each column, indexes provide cross-column cardinality information. That is, the optimizer can look at the leftmost columns in an index and determine how many distinct permutations of the column values exist in table. To obtain this statistical information, the optimizer requests the database engine to run a “key estimate” of the number of rows (keys) that match the selection criteria. This estimate process is part of the query optimization and uses the index(es) to “count” a

subset of the keys — thus providing the optimizer with a good idea of the selectivity of the query.

Since most databases do not have a way to accurately represent the cardinality of columns, the other optimizers may assume that the distinct values are equally distributed throughout the table. For example, if a table contains a column for STATE and the data reflects all 50 states in the United States, most optimizers assume that each state appears equally (the data distribution for any state is1/50th of the total). But as anyone familiar with the population distribution of the United States knows, it is very unlikely that the table will contain as many entries for North Dakota as it does for California.

However, in DB2 UDB for iSeries, an index built over STATE will give the optimizer information about how many rows satisfy 'North Dakota' and how many satisfy 'California.' And if the distributions vary widely, the optimizer may build different access plans based on the actual value of STATE specified in the query request.

The number of distinct values in a key column or composite key can be looked at by using the OS/400 command Display File Description (DSPFD) or using iSeries Navigator to view the properties of the index.

In some instances, the optimizer uses the index for optimization, but chooses to implement the query using a non-indexed method. For example, if the query is going to retrieve a high number of the rows, it may be faster to perform a full table scan. Remember that on an iSeries server, full table scans are highly efficient because of the independent I/O subsystems, parallel I/O technology, and very large memory system. The full table scan would be the fastest (least costly) method to access the requested rows.

Keep in mind, however, that even when a full table scan is the best access method for a given query, the optimizer makes that decision based in part, on what it learns from the indexes. Therefore, it is important to have a set of indexes defined for each table, regardless of whether the indexes are used for data retrieval.

Nested Loop Join via Index

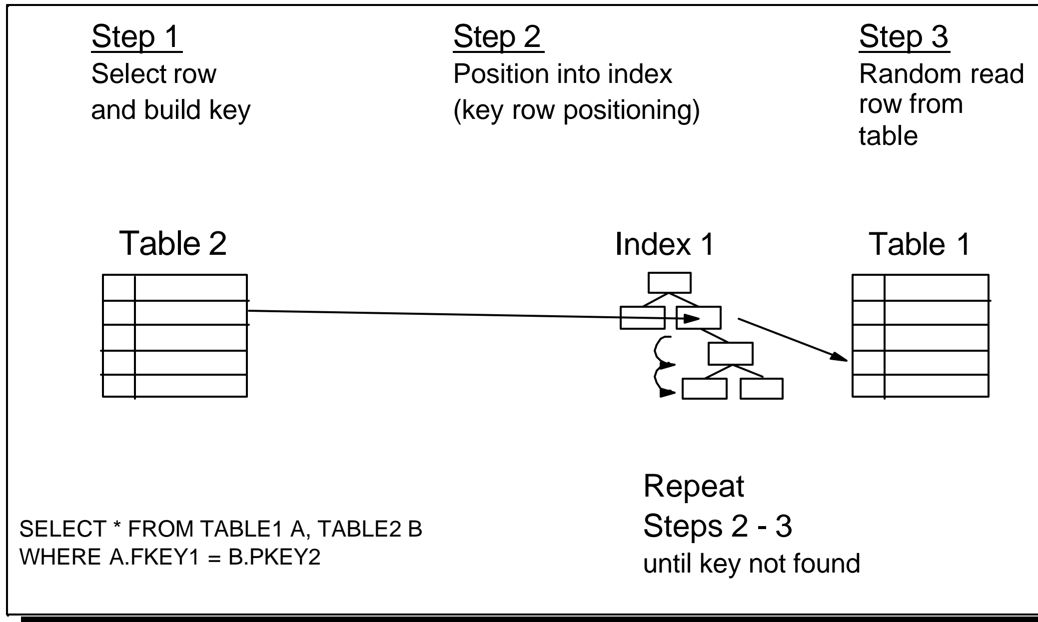
When DB2 UDB for iSeries builds an access plan for a query that uses *inner join* to access rows from more than one table, it first gathers an estimate for how many rows will be retrieved from each individual table. It then chooses the best access method for each individual table, based on the cost of the various methods available. Based on those estimates and costs, the optimizer then runs through possible variations of the join order, or the sequence in which the tables in the query will be accessed. An access plan is then built that puts the tables in the most cost-effective join order. Unlike some databases, which can only process a join in the order the query is written, DB2 UDB for iSeries will evaluate the possible options and rewrite the query to ensure the best (least costly) join order is used. For other join types, such as left outer join and exception join, the join must be implemented in the order specified in the SQL request.

As expected, the optimizer needs information from the indexes in order to evaluate the join order. Join order is dependent on which tables will retrieve the most rows and the “fan-out” of the join. Join fan-out can be simply defined as the number of expected rows that match a given join value. The optimizer estimates the number of rows retrieved by looking at the indexes. Therefore, it is very important to build indexes over the join columns.

The most common method of join processing is called a nested loop join via index. It applies to queries where there are at least two tables to join together, as in the following example:

```
SELECT A.COL1, B.COL2, B.COL3
      FROM TABLE1 A, TABLE2 B
      WHERE A.FKEY1 = B.PKEY2
```

With nested loop join via index, a row is read from the first table or dial in the join using any access method (i.e., table scan, index probe, etc.). Then, a join key value is built up to probe into the index of the second table or dial. If a key value is found, then the row is read from the table and returned. The next matching key value is read from the index and the corresponding row is read from the table. This process continues until no matching keys are found in the index. The database engine then reads the next row from the first table or dial and starts the join process for the next key value. The nested loop join is not complete until all of the rows matching the local selection are processed from the first dial. It is important to understand the nested loop process, so that it can be as efficient as possible. Nested loop join via indexes can produce a lot of I/O if there are many matching values in the secondary dials, or high join fan-out.



Nested loop join via index requires a radix index over the tables that are being joined (to the secondary dials). **If an index does not exist for the join columns in the tables following the first table in the join order, the database might choose to build temporary indexes over these columns to complete a nested loop join.** A common performance problem is that the index does not exist over the join columns, so the database must build temporary index(es) to process the query, lengthening query processing time, and requiring more system resources.

Starting in Version 4 Release 5, all nested loop joins via index are processed like index probes for local selection. In fact, when both the join and local selection predicates are present for a given join dial, the optimizer can use all of the columns to probe the radix index. This makes the join much more efficient since it narrows down the rows matching the selection and join criteria with a minimum of I/Os. This technique is called a “multi-key row positioning join.” It is very important to have a radix index available that contains both the local selection column(s) and the join column(s) for a given table. If only the join column is present in the index, the database engine must probe the index for the join key value, then read the table and test the local selection values. If the data does not match the local selection, then the probe of the index and random read of the table is wasted.

Indexes are critical to this process because the database engine can use the index instead of reading the base table. In this way, the optimizer can position the portion of the index that contains the relevant keys. For example, if the previous query is modified to:

```
SELECT A.COL1, B.COL2, B.COL3
FROM TABLE1 A, TABLE2 B
WHERE A.KEY = B.KEY
AND A.COL1 = 'BLUE'
AND B.COL2 = 123
```

Assume that the optimizer chooses to process TABLE2 first and TABLE1 second when implementing the join. If there is an index over TABLE1 with key columns COL1 and KEY, then the optimizer can use the index to locate only those values that contain both "BLUE" and the join key value. This improves performance considerably since it eliminates random reads of TABLE1 to process the local selection. An index over TABLE2 with key columns COL2 and KEY could provide the same advantage.

With V5R1 and earlier releases of OS/400, nested loop joins via index did not use parallelism during the processing of the join. Starting with V5R2, the SQL query engine does allow parallelism for nested loop joins.Symmetrical Multiprocessing (SMP) may be used to create a temporary index if required for a nested loop join. Nested loop joins via hashing can take advantage of parallelism, and do not require an index to perform the join. Joining via a hash table is another join implementation method that uses a hashing algorithm technique to "consolidate" join values together and to locate the data to be joined.

Grouping and Ordering

Other common functions within an SQL request are grouping and ordering. Using the SQL GROUP BY clause, queries will summarize or aggregate a set of rows together. In DB2 UDB for iSeries, the optimizer can use either an index or a hashing algorithm to perform grouping. The method that the optimizer picks is query and system dependent; the optimizer will make its selection based on the nature of the query, the data, and the system resources available.

When a query includes an ORDER BY clause, the database engine will order the result set based on the columns in the ORDER BY clause. In DB2 UDB for iSeries, the optimizer can use either an index or a sort. Therefore, indexes can be used for this function as well. Sometimes, the ORDER BY clause includes columns already used in the selection and grouping clauses, so the optimizer may take advantage of the "by key" processing used for other parts of the query request. The data is processed in order, so to speak.

For both grouping and ordering, the optimizer will cost the various methods available, based on the expected number of rows identified in the local selections and join. The optimizer estimates the number of unique groups based on the information that it finds in the indexes. In the absence of indexes, the optimizer uses a default number of groups and a default number of rows per group. As can be imagined, this estimate might be close, but if it is grossly inaccurate, the optimizer will choose an inefficient access plan. In working with large business intelligence applications where grouping to build aggregates is common, there may be millions of groups or millions of rows within a group. As the size of the database scales upward, it becomes even more important for the optimizer to be

able to accurately estimate how many rows are involved in a given query operation. Indexes make that possible.

In general, grouping a large number of rows per group favors hash grouping. Grouping a small number of rows per group favors index grouping.

Another factor is the fact that the index grouping does not use SMP or parallelism to process the rows.

Using an index for grouping or ordering can affect the join order of the query. This is true when the grouping and/or ordering columns are from one table. That table tends to go first in the join order, allowing the database engine to read the row from the first dial in the join by key, thus allowing the grouping and/or ordering to occur naturally. This may not be the best plan for optimal performance for the entire query. One way to help the optimizer is to create radix indexes for the selection and join statistics, and another index for the selection and grouping or ordering statistics. While the database engine cannot use both indexes for implementation, the optimizer would then get a good idea of the selectivity of the query, the join fan out, and the grouping attributes.

Example:

```
SELECT A.COL1, A.COL2, SUM(B.COL4)
FROM TABLE1 A, TABLE2 B
WHERE A.KEY1 = B.KEY2
AND A.COL3 = 'XYZ'
GROUP BY A.COL1, A.COL2
```

Indexes:

```
CREATE INDEX TABLE1_JOIN_INDEX1 ON TABLE1
      (COL3, KEY1)
CREATE INDEX TABLE1_GROUPING_INDEX1 ON TABLE1
      (COL3, COL1, COL2)
CREATE INDEX TABLE2_JOIN_INDEX1 ON TABLE2
      (KEY2)
```

Another index grouping technique the optimizer can choose is “early exit” on MIN and MAX functions. This is really just another way to employ an index probe with multiple keys, and take advantage of the data ordering via the index. The requirement is to have all of the local selection keys represented in the primary portion of the index followed by the column that is used with the MIN or MAX function. For MAX, the key column should be descending order. By specifying the local selection column(s) as key(s), followed by the column with the MIN or MAX function, the database engine can effectively read the first composite key value that matches the local selection and the MIN or MAX condition. The database engine then moves on, or positions down, to the next value that matches the local selection. This “early exit” routine saves the database from reading and processing all of the rows that match the local selection, searching for the MIN or MAX value.

```
CREATE INDEX X1 ON SALES
   (STATE, SALES) <---- ascending
   :
SELECT STATE, MIN(SALES)
   FROM EMPLOYEE
   WHERE STATE IN ( 'ARIZONA', 'CALIFORNIA')
   GROUP BY STATE
```

Key row position to ARIZONA and the *minimum* or first value for SALES by traversing the radix tree in *ascending* order, then on to California

STATE	SALES	CUSTOMER
Alabama	110.00	Jones
Alabama	150.00	Smith
Alabama	375.00	Doe
Alaska	10.00	Johnson
Alaska	55.00	Smith
Alaska	120.00	Alexander
Alaska	400.00	Lee
Arizona	50.00	White
Arizona	80.00	Doe
Arizona	210.00	Brown
Arizona	360.00	Jacobson
Arizona	540.00	Milligan
Arkansas	5.00	Weatherby
Arkansas	25.00	Smith
Arkansas	90.00	Pippen
California	30.00	Lee
California	75.00	Wayne

```
CREATE INDEX X2 ON SALES
   (STATE, SALES DESC) <---- descending
   :
SELECT STATE, MAX(SALES)
   FROM EMPLOYEE
   WHERE STATE IN ( 'ARIZONA', 'CALIFORNIA')
   GROUP BY STATE
```

Key row position to ARIZONA and the *maximum* or first value for SALES by traversing the radix tree in *descending* order, then on to California

STATE	SALES	CUSTOMER
Alabama	375.00	Jones
Alabama	150.00	Smith
Alabama	110.00	Doe
Alaska	400.00	Johnson
Alaska	120.00	Smith
Alaska	55.00	Alexander
Alaska	10.00	Lee
Arizona	540.00	White
Arizona	360.00	Doe
Arizona	210.00	Brown
Arizona	80.00	Jacobson
Arizona	50.00	Milligan
Arkansas	90.00	Weatherby
Arkansas	25.00	Smith
Arkansas	5.00	Pippen
California	75.00	Lee
California	30.00	Wayne

Indexes and Optimization: A Summary

DB2 UDB for iSeries makes indexes a powerful tool. The following table summarizes some of the concepts discussed in this section.

	Binary Radix Indexes	**Encoded Vector Indexes**
Basic data structure	A wide, flat tree	A Symbol Table and a vector
Interface for creating	Command, SQL, iSeries Navigator	SQL, iSeries Navigator
Can be created in parallel	Yes	Yes
Can be maintained in parallel	Yes	Yes
Used for statistics	Yes	Yes
Used for selection	Yes	Yes, via dynamic bitmaps or RRN list
Used for joining	Yes	No
Used for grouping	Yes	No
Used for ordering	Yes	No
Used to enforce unique Referential Integrity constraints	Yes	No

Part 2: Indexing Strategies for Performance Tuning

Now that indexes and their functions have been discussed, let's talk about how to use them most effectively.

There are two approaches to index creation: proactive and reactive. As the name implies, proactive index creation involves anticipating which columns will be most often used for selection, joining, grouping, and ordering; and then building indexes over those columns. In the reactive approach, indexes are created based on optimizer feedback, query implementation plan, and system performance measurements.

In practice, both methods will be used iteratively. As the numbers of users increase, more indexes are useful. Also, as the users become more adept at using the application, they might start using additional columns that will require more indexes.

The following section provides a tested proactive approach for index creation. Use these techniques as a starting point, and then add or delete indexes reactively after user and system behavior have been monitored.

A General Approach

It is useful to initially build indexes based on the database model and the application(s), in lieu of creating the indexes because of the needs of any particular query. As a starting point, consider designing basic indexes founded on the following criteria:

- Primary and foreign key columns based on the database model
- Commonly used local selection columns, including columns that are dependent, such as the make and model of an automobile
- Commonly used join columns not considered to be primary or foreign key columns
- Commonly used grouping columns

After analyzing the database model, consider the database requests of the application and the actual SQL. A developer can add to the basic index design and consider building some perfect indexes that incorporate the selection, join, grouping, and ordering criteria.

The "perfect index" is defined as a binary radix index that provides the optimizer with useful and adequate statistics, and multiple implementation methods — taking into account the entire query request.

A Proactive Approach

Returning to the analogy of the recipe, the goals of indexing are to give the optimizer:

- Information about ingredients or the data contained within the tables, such as the number of distinct values, the distribution of data values, and the average number of duplicate values.
- Choices about which cooking instructions to assemble, or which methods to use to process the query. In many recipes, the cooking method could be steaming, frying, or broiling. The choice depends on the desired result. In the same way, the optimizer has different methods available and will pick the appropriate method based on what it knows about the available ingredients and the desired result.

Before beginning the proactive process, the database model and a set of sample queries that will run against the database are needed. These queries will generally have the following format:

```
select b.col1, b.col2, a.col1
from table1 a, table2 b
where b.col1='some_value'        \
        b.col2=some_number,      /  selection predicates
        a.join_col=b.join_col    ---- join predicate
group by b.col1, b.col2, a.col1
order by b.col1
```

With a query like this, the proactive index creation process can begin. The basic rules are:

- Custom-build a radix index for the largest or most commonly used queries.
  - Example using the query above:
    - radix index over join column(s) - a.join_col and b.join_col
    - radix index over most commonly used local selection column(s) - b.col2
- For ad-hoc on-line analytical processing (OLAP) environments or less frequently invoked queries, build single-key EVIs over the local selection column(s) used in the queries.
  - Example using the query above:
    - EVI over non-unique local selection columns - b.col1 and b.col2

Clearly, these are general rules whose specific details depend on the environment. For example, the “most commonly used” queries can consist of

three or 300 queries. How many indexes that are built depends on user expectations, available disk storage, and the maintenance overhead.

Perfect Radix Index Guidelines

In a perfect radix index, the order of the columns is important — even making a difference as to whether the optimizer uses the index for data retrieval at all. As a general rule, order the columns in an index in the following way:

- Equal predicates first. That is, any predicate that uses the "=" operator may narrow down the range of rows the fastest and should therefore be first in the index.
- If all predicates have an equal operator, then order the columns as follows:
  - Selection predicates + join predicates
  - Join predicates + selection predicates
  - Selection predicates + group by columns
  - Selection predicates + order by columns

In addition to the guidelines above, in general, the most selective key columns should be placed first in the index.

A binary radix index can be used for selection, joins, ordering, grouping, temporary tables, and statistics. When evaluating data access methods for queries, create binary radix indexes with keys that match the query's local selection and join predicates in the query WHERE clause. A binary radix index is the fastest data access method for a query that is highly selective and returns a small number of rows.

As stated earlier, when creating a binary radix index with composite keys, the order of the keys is important. The order of the keys can provide faster access to the rows. The order of the keys should normally be local selection and join predicates, or, local selection and grouping columns (equal operators first and then inequality operators). Binary radix indexes should be created for predetermined queries or for queries that produce a standard report. A binary radix index uses disk resources; therefore, the number of binary radix indexes to create is dependent upon the system resources, size of the table, and query optimization.

The following examples illustrate these guidelines:

Example 1: A one-table query
This query uses the table ITEMS and finds all the customers who returned orders at year end 2000 that were shipped via air. It is assumed that longstanding customers have the lowest customer numbers.

```
SELECT CUSTOMER, CUSTOMER_NUMBER, ITEM_NUMBER
      FROM ITEMS
      WHERE YEAR = 2000
      AND QUARTER = 4
      AND RETURNFLAG = 'R'
      AND SHIPMODE = 'AIR'
      ORDER BY CUSTOMER_NUMBER, ITEM_NUMBER
```

The query has four local selection predicates and two ORDER BY columns. Following the guidelines, the perfect index would put the key columns covering the equal predicates first (YEAR, QUARTER, RETURNFLAG, SHIPMODE), followed by the ORDER BY columns CUSTOMER_NUMBER, ITEM_NUMBER.

To determine how to order the key columns covering equal local selection predicates, evaluate the other queries that will be running. Place the most commonly used columns first and/or the most selective columns first, based on the data distribution.

Example 2: A three-table query
Star schema join queries use joins to the dimension tables to narrow down the number of rows in the fact table to produce the result set in the report. This query finds the total first quarter revenue and profit for two years for each customer in a given sales territory.

```
SELECT T3.YEAR, T1.CUSTOMER_NAME,
SUM(T2.REVENUE_WO_TAX), SUM(T2.PROFIT_WO_TAX)
      FROM CUST_DIM T1, SALES_FACT T2, TIME_DIM T3
      WHERE T2.CUSTKEY=T1.CUSTKEY
      AND T2.TIMEKEY = T3.TIMEKEY
      AND T3.YEAR IN (2001, 2000)
      AND T3.QUARTER = 1
      AND T1.CONTINENT='AMERICA'
      AND T1.COUNTRY='UNITED STATES'
      AND T1.REGION='CENTRAL'
      AND T1.TERRITORY='FIVE'
      GROUP BY T3.YEAR, T1.CUSTOMER_NAME
      ORDER BY  T1.CUSTOMER_NAME, T3.YEAR
```

This query has two join predicates and six selection predicates. The first task is to focus on the selection predicates for each table in the query.

For the time dimension table TIME_DIM, the query specifies two local selection predicates. The index over the time dimension table should contain YEAR and QUARTER first, followed by the join predicate column TIMEKEY.

For the customer dimension table, the query specifies four local selection predicates. These predicates are related to each other along a geographical hierarchy (territory-region-country-continent). Since all the predicates are equal predicates, the order of the index keys for these predicates should follow the hierarchy of the database schema. The index over the customer dimension table should contain CONTINENT, COUNTRY, REGION, TERRITORY, followed by the join predicate column CUSTKEY.

For the fact table SALES_FACT, the two columns in the WHERE clause are TIMEKEY and CUSTKEY. Since both TIMEKEY and CUSTKEY are used as join predicates, the guidelines recommend two indexes, each with the respective join column: TIMEKEY and CUSTKEY. This will allow the optimizer to obtain statistics and to cost all the possible join orders.

According to the guidelines, an index should cover the columns from the group by clause and the order by clause. Because the group by and order by clauses use columns from two different tables, the query may be implemented in two steps (i.e., the selection and join must be completed prior to the grouping and ordering). The optimizer and database engine cannot take advantage of an existing index for grouping or order, so creating an index over GROUP BY and ORDER BY columns is required.

More information on star schema join optimization can be found in the paper, *Star Schema Join Support within DB2 UDB for iSeries*, at:
ibm.com/servers/enable/site/education/ibo/record.html?star

Example 3: Non-equal predicates in a query

Predicates with inequalities tend to return more rows than predicates with equality operators. For example, if a user requests all the rows where the date is between a start and an end point, such as the beginning and end of a quarter, then the query may return more rows than if the user asked for a specific day or week. Because an inequality predicate implies a range of values instead of a specific value, the optimizer makes different decisions about how to build the access plan. The following query asks for the same report as the query in the previous example, but the YEAR local selection predicate is much less specific:

```
SELECT T3.YEAR, T1.CUSTOMER_NAME,
SUM(T2.REVENUE_WO_TAX), SUM(T2.PROFIT_WO_TAX)
     FROM CUST_DIM T1, SALES_FACT T2, TIME_DIM T3
     WHERE T2.CUSTKEY=T1.CUSTKEY
     AND T2.TIMEKEY = T3.TIMEKEY
     AND T3.YEAR < 2001
     AND T3.QUARTER = 1
     AND T1.CONTINENT='AMERICA'
     AND T1.COUNTRY='UNITED STATES'
     AND T1.REGION='CENTRAL'
```

```
      AND T1.TERRITORY='FIVE'
      GROUP BY T3.YEAR, T1.CUSTOMER_NAME
      ORDER BY T1.CUSTOMER_NAME, T3.YEAR
```

In the previous example, the best index over the time dimension table was one built over the two local selection predicates first, then the join predicate. Here, one of the selection predicates is using a 'less than' operator, while the join predicate is an 'equal' predicate. Because 'equal' predicates provide the most direct path to the key values, an index over time dimension for this query would be QUARTER, TIMEKEY, YEAR. This produces a logical range of key values that the database engine can position to and process contiguously.

Encoded Vector Index Guidelines

EVIs are primarily used for local selection on a table; they can also provide the query optimizer with accurate statistics regarding the selectivity of a given predicate value. EVIs cannot be used for grouping or ordering and have very limited use in joins. When executing queries that contain joins, grouping, and ordering; a combination of binary radix indexes and EVIs may be used to implement the query. When the selected row set is relatively small, a binary radix index will usually perform faster access. When the selected row set is roughly between 20% and 70% of the table being queried, table probe access using a bitmap, created from an EVI or binary radix index will be the best choice. Also, the optimizer and database engine have the ability to use more than one index to help with selecting the data. This technique may be used when: the local selection contains AND or OR conditions, a single index does not contain all the proper key columns, or a single index cannot meet all of the conditions. Single key EVIs can help in this scenario since the bitmaps or RRN lists created from the EVIs can be combined to narrow down the selection process.

Example 1: A one-table query

Recall that this query uses the table ITEMS and finds all of the customers who returned orders at year end 2000 that were shipped via air. It is assumed that longstanding customers have the lowest customer numbers.

```
SELECT CUSTOMER, CUSTOMER_NUMBER, ITEM_NUMBER
      FROM ITEMS
      WHERE YEAR = 2000
      AND QUARTER = 4
      AND RETURNFLAG = 'R'
      AND SHIPMODE = 'AIR'
      ORDER BY CUSTOMER_NUMBER, ITEM_NUMBER
```

The query has four local selection predicates and two ORDER BY columns. Following the EVI guidelines, single key indexes would be created with key columns covering the equal predicates EVI1 — YEAR, EVI2 — QUARTER, EVI3 — RETURNFLAG, EVI4 — SHIPMODE. The optimizer will determine which of the indexes will be used to generate dynamic bitmaps. Based on this query, the bitmaps will be ANDed together, and table probe access will be used to locate and retrieve the rows from the ITEMS table.

If another similar query was requested, the same EVIs could be used.

```
SELECT CUSTOMER, CUSTOMER_NUMBER, ITEM_NUMBER
     FROM ITEMS
     WHERE YEAR = 2000
     AND MONTH IN (1, 2, 3)
     AND RETURNFLAG = 'R'
     AND SHIPMODE = 'RAIL'
```

In this case, EVI1 — YEAR, EVI3 — RETURNFLAG, EVI4 — SHIPMODE could be used. The local selection on MONTH would be satisfied by reading and testing the data in the row identified by the other selection criteria.

Proactively Tuning Many Queries

The previous examples assume that an index is being built to satisfy a particular query. In many environments, there are hundreds of different query requests. In business intelligence and data warehousing environments, users have the ability to modify existing queries or even create new ad-hoc queries. For these environments, it is not possible to build the perfect index for every query.

By applying the indexing concepts previously discussed, it is possible to create an adequate number of binary radix indexes to cover the majority of problem areas, such as common local selection and join predicates. For ad-hoc query environments, it is also possible to create a set of radix and EVI indexes that can be combined using the index ANDing / ORing technique to achieve acceptable response times. The best approach will be to create an initial set of indexes based on the database model, the application and the user's behavior, and then monitor the database activity and implementation methods.

Reactive Query Tuning

The reactive approach is very similar to the Wright Brothers' initial airplane flight experiences. Basically, the query is put together, pushed off a cliff, and watched to see if it flies. In other words, build a prototype of the proposed application without any indexes and start running some queries. Or, build an initial set of indexes and start running the application to see what gets used and what does not. Even with a smaller database, the slow running queries will become obvious very quickly.

The reactive tuning method is also used when trying to understand and tune an existing application that is not performing up to expectations.

Using the appropriate debugging and monitoring tools, which are described in the next section, the database feedback messages that will tell basically three things can be viewed:

- Any indexes the optimizer recommends for local selection
- Any temporary indexes used for a query
- The implementation method(s) that the optimizer has chosen to run the queries

DB2 UDB for iSeries includes an index advisor, which is a built-in tool that recommends permanent indexes. The index advisor messages in the joblog can

be viewed through iSeries Navigator or OS/400 command Display Job Log (DSPJOBLOG), by querying the Database Monitor data, or by using Visual Explain.

If the database engine is building temporary indexes to process joins or to perform grouping and selection over permanent tables, permanent indexes should be built over the same columns, and try and eliminate the temporary index creation. In some cases, a temporary index is built over a temporary table, so a permanent index will not be able to be built for those tables. The same tools can be used to note the creation of the temporary index, the reason the temporary index was created, and the key columns in the temporary index.

Understanding the queries implementation method(s) will also allow a focus on other areas that affect database performance, such as: system resources, application logic, and user behavior.

The following table outlines a few problem scenarios and offers suggestions for interpreting the recommendations of the optimizer and improving performance.

Situation	**Optimizer recommends**	**The developer should:**
There are no indexes built over the query's tables.	Build an index for local selection.	Build an index over the selection, join, and/or grouping fields.
You have built an index over some local selection columns or join columns.	Build an index for local selection.	Build an index over *all* the local selection columns, include the join columns.
You have built an index over all the selection fields and performance is a little better but still not acceptable.	Nothing	Reorder the columns in the index to place the most selective, equal predicates first, include the join columns.
You have built the perfect index and the optimizer will not use it.	Nothing	Use a database evaluation tool to determine which access method the optimizer selects. The optimizer might have determined that a full table scan is more efficient.
You have built an index that contains all of the relevant columns but the optimizer does not use it.	Build an index that contains the same columns but lists them in a different order.	Build an index with the columns ordered according to the optimizer recommendations.
You have built all the recommended indexes, yet debug messages still indicate that the optimizer's query estimate is not at all close to actual query run times.	Nothing	Add the following clause to your SQL statement: `OPTIMIZE FOR ALL ROWS`[1]
You have a query that contains several inequalities and/or the selection predicates are separated by OR conditions.	Build a radix index over the first few columns	Build single column indexes over each column, which will encourage the database to use dynamic bitmaps and index ANDing/ORing.

The main idea behind all of these recommendations is to give the optimizer as much information as possible about the tables and columns with which you are working. Remember, with DB2 UDB for iSeries, statistical information can be provided to the optimizer by creating indexes.

---

[1] The optimizer may generate a different access plan based on the user's information regarding optimizing for all rows or a subset of rows. Refer to the product documentation on how to use this clause.

Creating Indexes for Multi-table Queries (Joins)

When running queries that join tables, the need for the right indexes becomes even more critical. It is very important to analyze optimizer feedback messages to understand the query implementation and whether or not the query response time reflects building temporary indexes.

If the system is building temporary indexes over permanent tables, it is probably because the indexes are required to process a nested loop join. Nested loop join via index requires indexes over the join keys. The good news is that the optimizer will request an index be created to complete the query. The bad news is that the user must wait while the index is created, and the index is deleted when the query completes. If the same query is executed again, the temporary index will be recreated. If 20 users are running the same query, each user will have a temporary index created.

If a temporary index is being created over a permanent table, at a minimum, you should build a permanent index over the same key columns as the temporary index.

The Index Advisor might also recommend indexes that are different from the temporary indexes being created. The advisor looks only at the local selection predicates in the WHERE clause. For example, in a two-table join query, the first table might be accessed with a full table scan, and the second table might be accessed via a temporary index. The optimizer will recommend an index for the local selection of the table and provide feedback on the building of the temporary index for the nested loop join. Consider building an index over any columns recommended by the Index Advisor, and build an index like the temporary index for the nested loop join.

These recommendations may yield several indexes, all designed for the same query. As part of the analysis, run the query again with all of the recommended indexes present. If the desired results are still not being achieved, consider creating a radix index that combines all of the columns in the following priority: selection predicates with equalities, join predicates, then one selection predicate defined with inequalities.

If the WHERE clause contains only join predicates, ensure that a radix index exists over each table in the join. The join column(s) must be in the primary or left-most position of the key.

Tuning for One Query versus Tuning for Many

As you proceed through this iterative process, you will begin to see how indexes can be built that will tune many queries at one time. Start with one query and tune it. Then look at two or three. Find the columns that are used in all of the queries and build indexes over those fields. Picking the right columns and getting them in the right order will become more intuitive and productive.

Other Indexing Tips

- Avoid null-capable key columns if expecting to use index-only access. The index-only access method is not available in the Classic Query Engine (CQE) when any key in the index is null-capable.
- Avoid using derived expressions in a local selection or join condition. Access via an index may not be used for predicates that have derived values. Or, a temporary index will be created to provide key values and attributes that match the derivative. For example, if a query includes one of the following predicates:

  ```
  WHERE SHIPDATE > (current_date - 10) or
  UPPER(customer_name) = 'SMITH'
  ```

  ...the optimizer considers that predicate to be a derived value and may not use an index for local selection.
- Index access is not used for predicates where both operands are from the same table. For example, if a query includes the following statement:

  ```
  WHERE SHIPDATE > ORDERDATE
  ```

  ...the optimizer will not use an index to retrieve the data since it must access the same row for both operands.
- Consider Index Only Access (IOA). If all of the columns used in the query are represented in the index as key columns, the optimizer can request index only access. With IOA, DB2 UDB for iSeries does not have to retrieve any data from the actual table. All of the information required to implement the query is available in the index. This may eliminate the random access to the table and may drastically improve query performance.
- Use the most selective columns as keys in the index, adding one key column used with inequality comparisons. Because of how the optimizer processes inequalities as ranges of values, there is little benefit in putting more than one inequality predicate value into an index.
- For key columns that are unique, specify UNIQUE when creating the index. A primary key constraint will produce an index with unique keys.

Part 3: Indexing Considerations

Once you understand what can be done with indexes which are useful for the application, you also need to consider the more pragmatic aspects of indexes, such as: creation techniques, maintenance strategies, and capacity planning issues.

SQL Indexes versus Keyed Logical Files

As mentioned in the beginning of this paper, there are two interfaces available for creating database objects. Both the SQL CREATE INDEX statement and the OS/400 command CRTLF can be used to create a radix index. Both interfaces create the same index object, but with different attributes. These attributes can have an effect on query optimization and performance. Regardless of how the index was created, the optimizer will recognize and consider the indexes during optimization.

SQL indexes are created with a 64K logical page size. Keyed logical files are primarily created with a logical page size of 8K. This attribute cannot be specified during object creation or subsequently changed. The logical page size is determined by the interface used to create the index. Regardless of the logical page size, the overall object sizes of a SQL index and a keyed logical file tend to be equivalent. The larger logical page size can result in more efficient index scans and index maintenance. These are the key benefits in a query environment. Indexes with larger logical page sizes can have an impact on the I/O performance within environments that have smaller, less than optimal memory pools.

SQL indexes that are journaled (index logging) will cause the journal receivers to fill up more rapidly. To avoid filling up and regenerating new journal receivers more often, consider increasing the journal receiver size to at least 6.5GB, and set the receiver size attributes using CRTJRN or CHGJRN... RCVSIZOPT(*RMVINTENT *MAXOPT2)

Indexes will be created with the 64K logical page size when:

- Creating SQL indexes on SQL created tables
- Creating SQL indexes on CRTPF / DDS created physical files
- Creating SQL constraints on SQL created tables
- Creating constraints with ADDPFCST on SQL created tables
- Creating temporary indexes during query execution

In some cases, the optimizer will choose to create a temporary index with a logical page size of 64K instead of using a permanent keyed logical file with a logical page size of 8K. This occurs because the I/O cost of scanning the 8K index is higher than the cost of using the 64K index, even considering the time to create the temporary index. To overcome this behavior, the developer should consider creating a SQL index.

Based on the application development philosophy and database serving environment, use the interface that suits you. For example, if a developer is using SQL for object creation and database access, then use SQL to create indexes.

Estimating the Number of Indexes to Create

It would be nice to provide a rule of thumb for an appropriate number of indexes to build for different kinds of schemas and databases. But like many things in the application development world, there is not a simple formula for the appropriate number of indexes. It depends on the size of the tables and the relative size of the updates. It depends on the amount of system resources available for the load and update process. It depends on the maintenance window available.

Keep in mind that business intelligence environments are typically read-only, so index maintenance is only an issue during the load and update processes. In addition, business intelligence applications are more likely to allow ad-hoc queries which, especially when radix indexes are the only option, may require more than 10 or 15 indexes to satisfy all the possible query combinations.

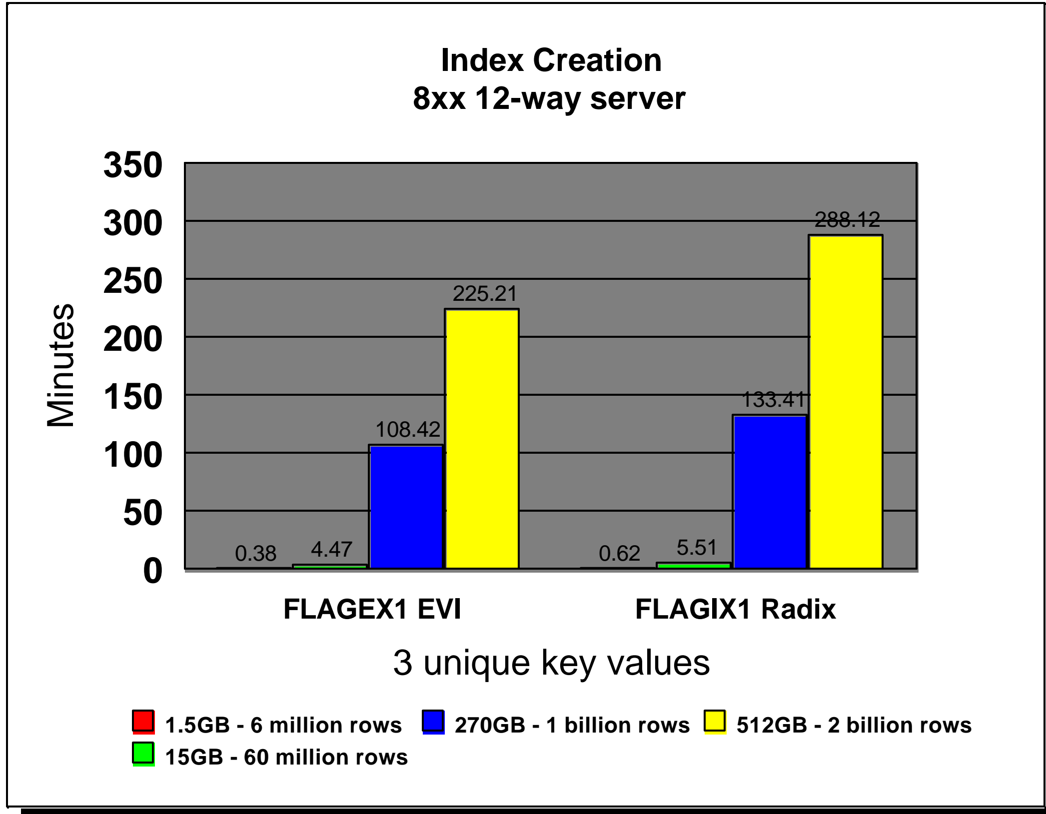
On-line transaction processing (OLTP) environments must support adds, changes, and deletes to the data — usually at high velocities. Index maintenance must be considered when tuning queries with an extensive indexing strategy.

Index Creation

With DB2 UDB for iSeries, indexes can be created with SQL or iSeries Navigator. It is recommended that some sort of documentation process be used to keep track of the index creation source. Some ideas for maintaining the source are:

- Place the SQL in an OS/400 source file and use the Run SQL Statements (RUNSQLSTM) command to execute the SQL.
- Place the SQL in a PC client text file and use iSeries Navigator — run SQL Scripts to execute.

Creating indexes can be a very time-consuming process, especially if the underlying tables are large. On iSeries servers with multiple processors, consider using SMP to create the indexes in parallel. iSeries servers enjoys linear scalability when creating large indexes in parallel. In other words, with two processors, the index will be created in half the time. With four processors, the index will be created in one fourth the time, and on a 24-processor server, the index will be created approximately 24 times faster than on a single processor system. Both binary radix and EVI indexes are eligible to be created this way. The optional SMP feature is required to be installed and enabled to create indexes in parallel.



DB2 UDB for iSeries is the only database system that can create indexes in parallel with:

- Database "on-line"
- Non-partitioned data sets
- Bottoms-up process (to keep the tree balanced and flat)
- High degree of key compression
- Linear scalability

Another technique when creating multiple indexes on a server with multiple processors is to create indexes simultaneously, one per processor. In other words, submit index creations, one to a processor without using SMP. In this way, each index will consume one processor and multiple indexes can be created at the same time. This works particularly well when the indexes are relatively small.

Index Maintenance

Index maintenance may occur anytime data is added, changed, or deleted. If a new row is added to a table, any indexes over that table will have to be updated to reflect the new row. This is also true if the row is deleted, or a value of a key column is changed.

DB2 UDB for iSeries supports three maintenance options for binary radix indexes: immediate, delay, and rebuild. Encoded vector indexes are always maintained immediately. The immediate maintenance option is the default when creating an index, and is typically the only option that should be used in a query environment. The optimizer cannot use an index that has a maintenance option of rebuild. In other words, the index with a maintenance option of rebuild is not of any value to the query and will not help with statistics or implementation. The optimizer does consider indexes with a maintenance option of delay, but must do extra optimization for these indexes. That is, it looks at how many changes are waiting to be performed on the index by looking at the delayed maintenance log and predicts how these changes will affect the index. Besides making the optimizer guess at how the pending changes will affect the index, using an index with a maintenance option of delay causes additional query execution time — simply because the pending changes must be performed on the index when it is opened. This index maintenance will increase the query response time.

The simplest and most straightforward recommendation is to balance the number of indexes required for acceptable query performance with the total number of indexes that must be maintained during I/O operations.

Another recommendation is to take advantage of parallel index maintenance using the optional SMP feature of OS/400. Parallel index maintenance has been available since Version 4 Release 3, and supports greater I/O velocities by using multiple database tasks to maintain indexes in parallel during blocked insert operations. For example, if there are eight indexes over a given table and applications are inserting data into the table, the database tasks can maintain each index in parallel. Otherwise, the application would wait for each of the eight indexes to be maintained serially. The overhead for this parallel maintenance is the use of more CPU resources within a given unit of time.

Depending on the number of rows being inserted, the system configuration, and the number of indexes that are over the database table; the recommendation may be to drop all indexes, perform the updates, and then rebuild the indexes upon completion of the process. This is due to the fact that maintenance of indexes serially will slow down the bulk insertion or update process. With SMP and parallel index maintenance, it may not be necessary to drop the indexes before beginning the update process.

In general, if the percentage of rows being added in an update process is more than 20% of the total size of a table, it is probably better to drop the indexes and rebuild them after the update completes. The threshold for when to use parallel index maintenance versus dropping indexes will vary widely based on the size, number, and complexity of indexes in your database. Even if the update delta is

not relatively large, both methods should be tested to determine which one is better.

EVI Maintenance

Although EVIs may drastically simplify the indexing strategy, it is important to understand how the system maintains EVls so that maintenance costs can be minimized while maximizing the benefits.

In general, you need to be aware of the issues that affect EVI maintenance:

1. **The maximum number of distinct values**
   When an EVI is defined, you can optionally include a WITH n DISTINCT VALUES clause. If this clause is not included, the database engine will determine the bytecode size (currently one byte, two bytes, four bytes) based on the actual number of distinct key values at index creation time. This may not reflect how many distinct key values will actually be represented in the data, once the table is fully populated. Since the EVI symbol table "compresses" the key values into one, two, or four bytes; the maximum number of distinct values for each bytecode size is as follows:

If FOR n DISTINCT VALUES is between:	Then the width of the vector table is:
1 and 255	1 byte
256 and 65,535	2 bytes
65,536 and 4.2 billion	4 bytes

   If an EVI is defined for 255 distinct values and then 256 distinct values are inserted, DB2 UDB for iSeries will automatically rebuild the index with a 2-byte vector table. However, *a performance degradation will be experienced while the index is rebuilt,* since the EVI will not be available to the optimizer or database engine. Therefore, it is important to have a good idea of how many distinct key values will be represented before creating the EVI.

   The consequence of defining more distinct values than are needed is simply that the index will take up additional disk space. For example, if an index with 300 distinct values is defined and only 200 distinct values are ever inserted, the vector will have one extra byte for every row in the database table. Except for organizations with tightly constrained disk availability, the extra byte should be insignificant.

2. **Insertion order**
   When new values are inserted into a table with indexes present, DB2 UDB for iSeries will automatically maintain the currency of the indexes. When a new row is inserted into the table, DB2 UDB for iSeries scans the EVI symbol, finds the matching value, and updates the statistics in the symbol table. If a new distinct value is introduced to an existing EVI, one of two things happens:
   - If the new value is logically ordered after the last distinct value in the symbol table, the value is added to the end of the table. For example, if there is an index over the column MONTHNUM in the table TIME_DIM and TIME_DIM currently contains values for the first quarter only, when values are inserted such as MONTHNUM = 4, the value 4 and its associated statistics will be added to the end of the EVI symbol table.

- ✍ If the distinct value is out of sequence from the indexes order, then the value is placed in an overflow area of the symbol table, where it remains until the index is rebuilt or refreshed. For example, if the table TIME_DIM contains values for months 1, 2, and 4 and MONTHNUM = 3 is inserted into rows, the distinct value 3 and its associated statistics are placed in an overflow area of the symbol table until the index is dropped and rebuilt, or the EVI is refreshed.

Prior to V5R2, the efficiency of the EVI can decrease as more and more distinct key values are placed out of order. Because of this, there is a limited number of distinct key values that will be placed in the overflow area of the symbol table. If the threshold is reached during insertion of a new key value, the EVI will automatically be refreshed.

For V5R1 and earlier OS/400 releases, the threshold limits are:

100 values for 1 byte code
1,000 values for 2 byte code
10,000 values for 4 byte code

In V5R2, the access to the EVI's overflow area was enhanced with some additional indexing technology. This improvement significantly increases the efficiency of the maintenance process and the use of the EVI when distinct key values are present in the overflow area of the symbol table. In V5R2, the threshold limit is increased to 500,000 values, regardless of bytecode size. This should allow greater latitude in regard to where EVIs can be used and maintained.

During the refresh process, the EVI is placed in delayed maintenance mode and is not available for use by the optimizer or database engine. *A significant performance degradation may be experienced during this process.* For this reason, EVIs are usually not advised for OLTP environments. Also, it is recommended that whenever several new distinct values are being inserted, you should consider dropping the EVI and recreating the index(es) after the inserts. For example, when loading data that represents many new distinct keys into data warehouse tables, it is a good idea to drop the EVIs prior to the loading process, then recreate the EVIs after the loading process.

Remember, too, that the values in the overflow area can be checked by issuing a Display Field Description (DSPFD) command or using iSeries Navigator, and looking at the overflow area parameter.

There are two ways to "refresh" the EVI and incorporate the overflow values back into the symbol table:

1. Drop all the indexes and recreate them using standard SQL statements or iSeries Navigator.
2. Use the Change Logical File (CHGLF) command with the attribute Rebuild Access Plan set to *Yes (FRCRBDAP(*YES)). This command will accomplish the same thing as dropping and recreating the index, but it does not require any knowledge about how the index was built. This command is especially effective for applications where the original index definitions are not available.

EVI Maintenance Overview

When using EVIs, there are unique challenges to index maintenance. The following table shows a progression of how EVIs are maintained and the conditions under which EVIs are most and least effective, based on the EVI maintenance idiosyncrasies.

	Condition	Characteristics
Most effective	When inserting an *existing* distinct key value	✍ Minimum overhead ✍ Symbol table key value looked up and statistics updated ✍ Vector element added for new row, with existing bytecode
	When inserting a *new* distinct key value — in order, within bytecode range	✍ Minimum overhead ✍ Symbol table key value added, bytecode assigned, statistics assigned ✍ Vector element added for new row, with new bytecode
	When inserting a *new* distinct key value — out of order, within bytecode range	✍ Minimum overhead if contained within overflow area threshold ✍ Symbol table key value added to overflow area, bytecode assigned, statistics assigned ✍ Vector element added for new row, with new bytecode ✍ Considerable overhead if overflow area threshold reached ✍ Access path invalidated — not available ✍ EVI refreshed, overflow area keys incorporated, new bytecodes assigned (symbol table and vector elements updated)
Least effective	When inserting a *new* distinct key value — out of bytecode range	✍ Considerable overhead ✍ Access path invalidated — not available ✍ EVI refreshed, next bytecode size used, new bytecodes assigned (symbol table and vector elements updated).

General Index Maintenance Recommendations

Remember, whenever indexes are created and used, there is a potential for a decrease in I/O velocity due to maintenance. Therefore, it is essential that the maintenance cost of creating and using additional indexes is considered. For radix indexes with MAINT(*IMMED) and EVIs, maintenance occurs when inserting, updating, or deleting rows.

To reduce the maintenance of the indexes, consider:

- Minimizing the number of indexes over a given table
- Dropping indexes during batch inserts, updates, and deletes
- Creating indexes, one at a time, in parallel using SMP
- Creating multiple indexes simultaneously with multiple batch jobs using multiple CPUs
- Maintaining indexes in parallel using SMP

The goal of creating indexes for performance is to balance the maximum number of indexes for statistics and implementation while minimizing the number of indexes to maintain.

Recommendations for EVI Use

Encoded vector indexes are a powerful tool for providing fast data access in decision support and query reporting environments. However, to ensure the effective use of EVIs, they should be implemented with the following guidelines:

<u>Create EVIs on</u>:

- Read-only tables or tables with a minimum of INSERT, UPDATE, and DELETE activity
- Key columns that are used in the WHERE clause — local selection predicates of SQL requests, and fact table join columns when using star schema join support
- Single-key columns that have a relatively small set of distinct values
- Multiple-key columns that result in a relatively small set of distinct values
- Key columns that have a static or relatively static set of distinct values
- Non-unique key columns, with many duplicates

<u>Create EVIs with the maximum bytecode size expected</u>:

- Use the WITH n DISTINCT VALUES clause on the CREATE ENCODED VECTOR INDEX statement
- If unsure, consider using a number greater than 65,535 to create a 4-byte code, thus avoiding the EVI maintenance overhead of switching bytecode sizes as additional new distinct key values are inserted

<u>When loading data</u>:

- Drop EVIs, load data, create EVIs
- EVI bytecode size will be assigned automatically based on the number of actual distinct key values found in the table
- Symbol table will contain all key values, in order, no keys in overflow area

Part 4: Statistics Manager (V5R2)

Within Version 5 Release 2 of OS/400, DB2 UDB for iSeries has a new, SQL Query Engine (SQE). As part of this new SQL query engine, a statistics manager component is responsible for generating, maintaining and providing statistics to the SQE optimizer. As mentioned earlier, sources for statistics within DB2 UDB for iSeries come from default values and/or indexes. With SQE, the optimizer has another source, namely column statistics stored within the table object.

During query optimization, the SQE optimizer will ask the SQE statistics manager a question, such as “how many rows contain CUSTOMER_NUMBER = 2358.” The statistics manager will answer the question using the best source of information available at the time. The source of the answer can be default values, indexes, or columns statistics. If an index (radix or EVI) or column stat is not available to satisfy the SQE optimizer’s request, the statistics manager will provide an answer based on default values. In addition, a column statistic collection is automatically requested. The column statistics will be generated by a low priority, background job; with the goal of having the column stat available for future execution(s) of this query. This automatic collection of statistics allows the SQE optimizer to benefit from columns stats, without requiring an administrator to be responsible for the collection and management of statistics, as is true for other RDBMS products. Even though it is not required, statistics can also be manually requested for iSeries users who want to take on the task of statistics collection and management, without waiting for the statistics manager to recognize the need for a column statistic. The column statistics that are generated are only used by the new, SQL query engine. The original, Classic Query Engine (CQE) continues to use only default values and indexes for statistics.

Comparing an index (used for statistics) to a column stat, you will find the storage required for column stat is much smaller. On average, a column stat takes up approximately 8K - 12K per column. This additional space will be reflected in the table’s size. As the data changes in the table, a column stat is not maintained, and thus does not affect the I/O performance on the table. On the other hand, if the data in the table is changing a lot, then the stats can become stale or outdated. Indexes are maintained immediately as the data in the table changes. This allows the indexes to always provide up-to-date statistics. The cost of this maintenance might be reflected in slightly longer insert, update, and delete times.

If you want to avoid a possible first time degradation in performance after upgrading to V5R2, then you should proactively collect statistics by manually requesting statistics on columns that are frequently searched and that do not have indexes created over them. The searched column(s) must be the leading key columns in the index for that index to provide good statistics to the optimizer. If only statistics from indexes are available, then it is possible that SQE will generate a different access plan than CQE using the same index statistics.

With the statistics enhancements provided by SQE, there may be cases where indexes created primarily for giving the optimizer statistics, can now be dropped in favor of column statistics. This situation only applies to queries that exclusively

use SQE. If the query uses CQE for optimization and execution, then the indexes will still be required to provide the statistics, and should not be dropped.

The introduction of automatic column statistics does not replace the need for a good indexing strategy. Remember, the goal of the optimizer is to provide the fastest access plan for a given request. While statistics help the optimizer reach this goal, the fastest access method may require the presence and use of an index.

More information on the V5R2 DB2 UDB for iSeries SQL Query Engine (SQE) enhancements can be found at: **ibm.com**/eserver/iseries/db2/sqe.html and **ibm.com**/redbooks (search for SG246598).

Part 5: Statistics Strategies for Performance Tuning

Let's review why a good statistics strategy is important.

If the correct statistics are collected and available, the SQE optimizer will more accurately estimate the number of rows to process. Better estimates will allow for better query optimization and the selection of the best (performing) query plan for your environment.

Statistics strategies can be separated into two main approaches, namely proactive and reactive. Both approaches can and should employ indexes along with column stats. Given that indexes are used for both statistics and implementation, you can say that the column stats complement a good indexing strategy.

A "proactive" strategy can be used to identify and create column statistics based on the data model, actual data, and the SQL requests expected. Proactively identifying the column stats to collect, and collecting the information is usually a manual process, but a developer can also employ some automatic techniques.

A "reactive" strategy can be used to identify and create column statistics based on the actual data, specific SQL requests and/or the lack of indexes. Reactively identifying the column stats to collect, and collecting the information is usually an automatic process, but a developer can also employ some manual techniques to control when and where this collection occurs.

Statistics on Demand!

As mentioned earlier, the enhancements to DB2 UDB for iSeries include the ability to autonomously recognize the need for a column statistic, generate a request for the statistical data, and then collect the information in the background. This feature can be used in both proactive and reactive statistics strategies.

The automatic statistic collection is controlled by the system value **QDBFSTCCOL**. This system value is set to ***ALL** by default. This value allows all requests for background stats collections, whether initiated by the system or initiated by the user.

The statistic collections are handled by the system job **QDBFSTCCOL**. This job is responsible for processing the requests at a very low run priority with the goal of collecting the column stat(s) as fast as possible without impacting other, higher priority work on the system. The collection process is basically a full table scan of the table that contains the column or columns in which the developer may be interested. Given no other competing work, the time it takes to gather a column stat is equal to or less than the time it takes to perform a query requiring a full table scan. The SQE stats engine has the ability to gather stats in parallel, even if the OS/400 SMP feature is not installed or not enabled. If the opportunity to collect stats presents itself, the engine will automatically take advantage of the available computing resources to collect as much information in the time allowed.

When the SQE optimizer builds a query plan, the plan and associated information is stored in a central location called the SQE plan cache. Information about the statistical questions, answers, and sources are also stored in the plan cache. This information is used to initiate the automatic stats collections. Since the plan cache is cleared during an IPL of the system or logical partition, any requests for column stat collections that have not been satisfied will be lost. Be sure to consider this behavior as the statistic collection strategy is constructed.

Proactive Strategy

When tuning a SQL-based application or queries proactively, the developer can use the automatic stats collection to their advantage. Before going "live" with the application or queries, ensure a good indexing strategy is in place, then exercise the application or SQL requests with the system value QDBFSTCCOL set to *NONE. This will allow the SQE optimizer to get a good look at the SQL requests, and more importantly the statistical sources available. If some statistics are unavailable, SQE will queue up the appropriate stats collection requests. Once all the application logic and/or queries have been executed, change the system value QDBFSTCCOL to *ALL and allow the system to go off and collect the column stats. This technique has the advantage of allowing different columns from the same table to be processed with one scan of that table object. Another value of this particular proactive approach is it allows the developer to "control" when the auto stats collection occurs — instead of just having it collecting in background with no one aware it is running.

Another proactive technique is to manually request the column stats based on analysis of the data model, data, and SQL within the application. Once a good indexing strategy is in place, consider creating column stats to complement the indexes and/or cover gaps in the indexing strategy (columns that are good candidates for stats collection will be covered later in this paper). Use iSeries Navigator - Database to select the table and column(s) on which the developer want statistics collected. The collection(s) can occur immediately (running within your OS/400 job, at your job's priority) or can be submitted to the background and run at a lower priority. When the statistics are collected, the developer are ready to go live with the application and queries.

A proactive approach to identifying and gathering column statistics is crucial if:

- The developer requires the application and queries to run their very best "right out-of-the-box."
- The developer is unsure of the indexing strategy or have a sub-optimal indexing strategy.
- The developer expects little or no computing resources to be available for the background stats collection(s) during normal operations.

Reactive Strategy

When tuning SQL-based applications or queries reactively, the developer can also use the automatic stats collection to their advantage. As a matter of fact, this is the main reason stats collection is in place. By default, the system will identify and gather stats as the application and queries are executed. Over time, the optimizer will have more, and better, information about the data.

Once again, before going "live" with the application or queries, ensure a good indexing strategy is in place. Set the system value QDBFSTCCOL to *ALL or *SYSTEM. As the SQL requests are optimized, the stats engine will interrogate the statistical sources available. If there are some statistics that are unavailable, SQE will queue up the appropriate stats collection requests.

Another reactive technique is to manually request the column stats based on analysis of the data and the SQL requests that have been optimized. Using one of the query analysis and feedback tools is very helpful. Visual Explain and the database monitor have both been enhanced to provide feedback on which columns should have statistics collected. In addition to an "index advisor," Visual Explain also has a "statistics advisor." These advisors are based on the optimizer feedback for a given SQL request. The database monitor can produce a new row (column QQRID = 3015) that contains feedback on which column(s) should have a stat collected. Once the query has been analyzed, use the feedback to determine whether an index will be best, or a column stat will be adequate. Request the column stat manually using iSeries Navigator - Database, or the Visual Explain statistics advisor. Rerun the query to see what changed based on the presence of the additional information.

Identifying Columns with Statistics

After proactive or reactive stats collections, the developer can identify and document which tables have column stats by using iSeries Navigator to display each table's statistical data. Better yet, the stats engine APIs can be used to programmatically identify and list the tables and column stats. An example utility can be found at: **ibm.com**/servers/eserver/iseries/db2/statsscriptprocedure.html

The APIs and this utility can also be used to build a script that creates a batch of column statistics requests. This will be helpful if the developer has tuned an application on one system, and then wants to migrate or install the application on another system. A script of DDL SQL statements can be used to create the physical data model (tables and indexes), and the utility can be used to construct and run a set of column stat collection requests.

What Columns Should Have Stats Available?

Now that you know when and how to collect stats, the next question is, what columns should have stats available?

The answer depends on the data model, the data, and the SQL requests. It also depends on the indexing strategy employed.

In general, statistics should be available for columns used in local selection, joining, grouping, ordering, and distinct processing. Keep in mind that providing the optimizer with a new column stat does not necessarily mean that the optimizer will re-optimize the query and build a new access plan.

Let's take a look at local selection (find all the rows where a column equals something in particular). If the data in the column has high cardinality (lots of different values) and the query is requesting one value or a very small set of values, a radix index is most likely the best object to use to identify and select the

associated rows. As it turns out, this same radix index is also well suited to give the optimizer the statistics it needs for optimization. In this case, a column stat is redundant, and not as beneficial. If, on the other hand, the query is selecting a large set of values, using a radix index may not be the best method to find the statistical information since this process involves reading some number of keys from the index. The more values requested by the query, the more keys to read. In this case, a column stat would be beneficial since the statistical information can be obtained much faster.

In the following example the query is using four columns for local selection...

```
SELECT CUSTOMER, CUSTOMER_NUMBER, ITEM_NUMBER
      FROM ITEMS
      WHERE YEAR = 2000
      AND QUARTER = 4
      AND RETURNFLAG = 'R'
      AND SHIPMODE = 'AIR'
```

Let's assume the developer has a radix index over two of the columns, YEAR and QUARTER. The optimizer can use this index to obtain statistics on YEAR = 2000 and QUARTER = 4, but what about RETURNFLAG = 'R' and SHIPMODE = 'AIR'? Since there are no stats sources available for these two columns, the optimizer will use default values. Given that the index is sub-optimal (only containing two of the four local selection columns), a set of column stats will be beneficial, and will complement the index. Creating a column stat for RETURNFLAG and SHIPMODE will allow the optimizer to understand the selectivity of all the columns and make the appropriate choice of using the two-key column index or using a table scan. Without the additional column stats, the optimize might incorrectly conclude that the selectivity is high. Thus, using the index will yield the best performance.

Another major use of column stats is to accurately predict the size of temporary data structures used when joining, grouping, and determining distinct sets of values. The optimizer can choose to use an index for joining, grouping, and distinct processing; or, it can use various data structures. Let's assume the optimizer chooses to use a hash table for grouping. To accurately cost the use of this method, and to determine the optimal size of the hash table, statistics about the associated grouping column(s) must be available. If the grouping columns are from more than one table, a permanent index cannot be created to satisfy the query. Once again, a set of column stats can be used to provide this information — without the overhead of carrying additional indexes just to provide stats. The following example illustrates this:

```
SELECT C.CUSTOMER, I.RETURNFLAG, SUM(I.QUANTITY)
      FROM       ITEMS I,
                 CUSTOMERS C
      WHERE I.YEAR = 2000
      AND        I.QUARTER = 4
      AND        I.CUSTKEY = C.CUSTKEY
      GROUP BY   C.CUSTOMER,
```

```
I.RETURNFLAG
```

To assist the optimizer, create column stats on table CUSTOMERS - column CUSTOMER, and table ITEMS - column RETURNFLAG.

Since column stats are derived and stored on a per-column basis, correlation between columns is not well represented. If you know that there is a strong correlation between two or more columns, it would be best to create an index using those columns. This will provide the optimizer with a more complete picture of the data. An example of this concept is the relationship between automobile make and model. While there are many different makes, and many different models, there exists a strong correlation between the model of the automobile and the make of that model. Only Ford produces the Mustang, and only Chevrolet produces the Camaro.

Maintaining Statistics

Obviously as the data changes, the statistics about the data must also change. Otherwise the query plans can get outdated and become sub optimal. One big difference between indexes and column stats is the fact that indexes are maintained immediately — as rows are inserted, updated, or deleted. Column stats, on the other hand, are not maintained immediately, and this can factor into the statistics strategy.

To keep the column stats from getting too far out-of-date, the statistics manager keeps track of the point at which the column stat was generated, and the subsequent changes to the table. Once a significant delta has been reached, the stats are marked as "stale." Today, this delta is approximately 15%. When a column stat is marked stale, it is still used during optimization, but will cause a stats refresh operation to be requested. The scenario might go something like this:

- No stats exist for a particular column used by SQL query "X."
- SQL query "X" is issued and the optimizer uses default stats, and requests column stats be generated.
- Column stats are generated in the background.
- SQL query "X" is issued and the optimizer uses the new column stats.
- Data processing occurs that inserts, updates, and/or deletes rows.
- SQL query "X" is issued and the optimizer marks the stats stale, uses the stale stats, and requests the column stats to be refreshed.
- Column stats are refreshed in the background.
- SQL query "X" is issued and the optimizer uses the refreshed column stats.

Based on this behavior, here are some recommendations for different environments.

<u>On-line Transaction processing (OLTP)</u>

In this environment, the data is changing constantly, and many of the SQL requests will access a narrow range of rows. A good indexing strategy (employing radix indexes) is best for statistics and implementation. Use the automatic stats collection to supplement the indexes. If the developer wants to take a more

hands-on approach and control or limit the stats collection(s) during the transaction processing period(s), consider setting the system value QDBFSTCCOL to *NONE or *USER. If there is a window of idle computing resources at the end of the day or week, set the system value back to *ALL or *SYSTEM during this period and allow the statistics manager to catch up the column stats. Set the system value back to *NONE or *USER after the stats have been generated or the window of time runs out. Analyze the column stats collected and consider using this information to modify or enhance the indexes.

Business Intelligence (data warehousing)
In this application scenario, the data is normally only changed at regular intervals of time, such as end-of-day, end-of-week, and / or end-of-month. The SQL requests will be ad-hoc and varied, accessing both narrow and wide ranges of rows. A good indexing strategy (employing radix and EVI indexes) is required for implementation, but a good stats strategy will be important for those cases where the optimizer chooses alternate methods of access and processing (i.e., table scans, joining and grouping via hashing, etc.). Use the automatic stats collection, to supplement the indexes, but plan on refreshing the column statistics as part of your daily, weekly, or monthly load process. This can be done programmatically using the stats APIs. After the data has been loaded, and the column stats refreshed, continue to use the automatic stats collection to pick up any column stats that might have been overlooked. Analyze the column stats collected and consider using this information to modify or enhance your indexes. It is plausible that some columns will be covered by both indexes and column stats.

Operational Data Store (ODS) and Transaction Reporting
In this situation, some of the data may be changing constantly as it trickles in from the transactional systems, while other data may be changed periodically. The SQL requests will be varied, accessing both narrow and wide ranges of rows. A good indexing strategy (employing radix and EVI indexes) is required for stats and implementation. Column stats will be important for those cases where the optimizer chooses alternate methods of access and processing (i.e., table scans, joining and grouping via hashing, etc.). Use the automatic stats collection to supplement the indexes. If the developer wants to take a more hands-on approach and control or limit the stats collection(s) during the transaction reporting period(s), consider setting the system value QDBFSTCCOL to *NONE or *USER. If there is a window of idle computing resources at the end of the day or week, set the system value back to *ALL or *SYSTEM during this period and allow the statistics manager to “catch up” the column stats. Set the system value back to *NONE or *USER after the stats have been generated or the window of time runs out. Analyze the column stats collected and consider using this information to modify or enhance the indexes. It is plausible that some columns will be covered by both indexes and column stats.

Temporary Tables
If the environment creates and uses temporary tables, and these tables come and go frequently, consider identifying and collecting column stats on the *permanent* tables prior to using the application(s). Turn off the automatic stats collection and avoid generating requests against the transient temporary tables. At the end of the day or week, consider turning on the automatic stats collection

for some period of time. This will allow the statistics manager to handle any legitimate requests for collection of stats.

One way to do this is to use the job scheduling function built into OS/400. Here is a simple example that allows the automatic stats collection to occur everyday between 3:30 AM and 5:30 AM:

```
ADDJOBSCDE JOB(STARTSTATS)
CMD(CHGSYSVAL SYSVAL(QDBFSTCCOL) VALUE(*ALL))
FRQ(*WEEKLY) SCDDATE(*NONE) SCDDAY(*ALL) SCDTIME('3:30')

ADDJOBSCDE JOB(STARTSTATS)
CMD(CHGSYSVAL SYSVAL(QDBFSTCCOL) VALUE(*USER))
FRQ(*WEEKLY) SCDDATE(*NONE) SCDDAY(*ALL) SCDTIME('5:30')
```

More information about job scheduling and the CL commands can be found on the Web at the iSeries Information Center:
http://publib.boulder.ibm.com/html/as400/infocenter.html

Summary

As with all databases, indexes and statistics play an important role in improving query performance on an iSeries server. In addition to assisting in data retrieval, DB2 UDB for iSeries indexes also provide valuable information to the cost-based optimization of query requests. This ability to provide the optimizer with information about data skew and column cardinality becomes critically important as databases scale into the hundreds of gigabytes and eventually terabytes of data. The Statistics Manager within DB2 UDB for iSeries greatly enhances productivity and allows the SQE optimizer to learn about and react to changes autonomously. This is very important in an “on demand” environment.

Appendix A — Examples Queries and Possible Indexing Strategies

The following are some examples of SQL query requests with a recommended set of indexes to create. The purpose of these examples is to demonstrate the concept of proactive index creation based on the actual SQL request and knowledge of the query optimizer and database engine. These examples are only listed to illustrate one proactive methodology. The actual implementation and performance of these SQL requests will be dependent upon several factors, including, but not limited to: database table and index sizes, version of OS/400 and DB2 UDB for iSeries, query interface attributes, job and system attributes, and environment. The query plans and performance results will vary.

Example 1

```
SELECT *
      FROM TABLE1 A
      WHERE A.COLOR IN (‘BLUE’, ‘GREEN’, ‘RED’)
```

```
CREATE INDEX TABLE1_INDEX1 ON TABLE1 (COLOR)
```

Or

```
CREATE ENCODED VECTOR INDEX TABLE1_EVI1 ON TABLE1 (COLOR)
```

Anticipating index probe or table probe with dynamic bitmap

Example 2

```
SELECT *
      FROM TABLE1 A
      WHERE A.COLOR IN (‘BLUE’, ‘GREEN’, ‘RED’)
      AND A.SIZE IN (‘LARGE’, ‘X-LARGE’)
```

```
CREATE INDEX TABLE1_INDEX1 ON TABLE1 (COLOR, SIZE)
```

(keys can be in any order, most selective column first)

Or

```
CREATE ENCODED VECTOR INDEX TABLE1_EVI1 ON TABLE1 (COLOR)
CREATE ENCODED VECTOR INDEX TABLE1_EVI2 ON TABLE1 (SIZE)
```

(anticipating index probe or table probe with dynamic bitmaps)

Example 3

```
SELECT *
      FROM TABLE1 A
      WHERE A.COLOR IN ('BLUE', 'GREEN', 'RED')
      AND A.SIZE IN ('LARGE', 'X-LARGE')
      AND A.STYLE = 'ADULT MENS T-SHIRT'

CREATE INDEX TABLE1_INDEX1 ON TABLE1 (COLOR, SIZE, STYLE)
```

(keys can be in any order, most selective columns first)

Or

```
CREATE ENCODED VECTOR INDEX TABLE1_EVI1 ON TABLE1 (COLOR)
CREATE ENCODED VECTOR INDEX TABLE1_EVI2 ON TABLE1 (SIZE)
CREATE ENCODED VECTOR INDEX TABLE1_EVI3 ON TABLE1 (STYLE)
```

(anticipating index probe or table probe with dynamic bitmaps)

Example 4

```
SELECT * FROM TABLE1 A
      WHERE A.COLOR IN ('BLUE', 'GREEN', 'RED')
      AND A.SIZE IN ('LARGE', 'X-LARGE')
      AND A.STYLE = 'ADULT MENS T-SHIRT'
      AND A.INVENTORY > 100

CREATE INDEX TABLE1_INDEX1 ON TABLE1 (COLOR, SIZE, STYLE,
INVENTORY)
```

(keys can be in any order, most selective columns first, non-equal predicate last)

Or

```
CREATE ENCODED VECTOR INDEX TABLE1_EVI1 ON TABLE1 (COLOR)
CREATE ENCODED VECTOR INDEX TABLE1_EVI2 ON TABLE1 (SIZE)
CREATE ENCODED VECTOR INDEX TABLE1_EVI3 ON TABLE1 (STYLE)
CREATE ENCODED VECTOR INDEX TABLE1_EVI4 ON TABLE1 (INVENTORY)
```

(anticipating index probe or table probe with dynamic bitmaps)

Example 5

```
SELECT * FROM TABLE1 A, TABLE2 B
      WHERE A.KEY = B.KEY
      AND A.COLOR IN ('BLUE', 'GREEN', 'RED')
      AND A.SIZE IN ('LARGE', 'X-LARGE')
      AND A.STYLE = 'ADULT MENS T-SHIRT'
      AND A.INVENTORY > 100
```

```
CREATE INDEX TABLE1_INDEX1 ON TABLE1
      (COLOR, SIZE, STYLE, KEY, INVENTORY)
```

(keys can be in any order, most selective local selection columns first, non-equal predicate last)

```
CREATE INDEX TABLE2_INDEX1 ON TABLE2 (KEY)
```

And / or

```
CREATE ENCODED VECTOR INDEX TABLE1_EVI1 ON TABLE1 (COLOR)
CREATE ENCODED VECTOR INDEX TABLE1_EVI2 ON TABLE1 (SIZE)
CREATE ENCODED VECTOR INDEX TABLE1_EVI3 ON TABLE1 (STYLE)
CREATE ENCODED VECTOR INDEX TABLE1_EVI4 ON TABLE1 (INVENTORY)
```

(anticipating index probe, or table probe with dynamic bitmaps, and nested loop join via index)

Example 6

```
SELECT A.STORE, A.STYLE, A.SIZE, A.COLOR SUM(A.QUANTITY_SOLD)
      FROM TABLE1 A, TABLE2 B
      WHERE A.KEY = B.KEY
      AND A.COLOR IN ('BLUE', 'GREEN', 'RED')
      AND A.SIZE IN ('LARGE', 'X-LARGE')
      AND A.STYLE = 'ADULT MENS T-SHIRT'
      GROUP BY A.STORE, A.STYLE, A.SIZE, A.COLOR

CREATE INDEX TABLE1_INDEX1 ON TABLE1
      (COLOR, SIZE, STYLE, KEY)
```

(keys can be in any order, most selective local selection columns first)

```
CREATE INDEX TABLE1_INDEX2 ON TABLE1
      (STORE, STYLE, SIZE, COLOR)
```

(keys must be in this order for grouping stats)

Generate column statistics on STORE, STYLE, SIZE, COLOR (V5R2 only)

```
CREATE INDEX TABLE2_INDEX1 ON TABLE2 (KEY)
```

And / or

```
CREATE ENCODED VECTOR INDEX TABLE1_EVI1 ON TABLE1 (COLOR)
CREATE ENCODED VECTOR INDEX TABLE1_EVI2 ON TABLE1 (SIZE)
CREATE ENCODED VECTOR INDEX TABLE1_EVI3 ON TABLE1 (STYLE)
```

(anticipating index probe, or table probe with dynamic bitmaps, and nested loop join via index)

Example 7

```
SELECT *
      FROM TABLE1 A,
           TABLE2 B
      WHERE A.KEY_COL1 = B.KEY_COL2
      AND A.COLOR IN ('BLUE', 'GREEN', 'RED')

CREATE INDEX TABLE1_INDEX1 ON TABLE1 (COLOR, KEY_COL1)
CREATE INDEX TABLE2_INDEX1 ON TABLE2 (KEY_COL2)
```

(anticipating index probe and nested loop join via index)

Example 8

```
SELECT A.COL3, A.COL4, B.COL2, C.COL6, C.COL7
     FROM TABLE1 A,
          TABLE2 B,
          TABLE3 C
     WHERE A.KEY_COL1 = B.KEY_COL1
     AND A.KEY_COL2 = C.KEY_COL2
     AND A.COLOR IN ('BLUE', 'GREEN', 'RED')

CREATE INDEX TABLE1_INDEX1 ON TABLE1 (COLOR, KEY_COL1)
CREATE INDEX TABLE1_INDEX2 ON TABLE1 (COLOR, KEY_COL2)
CREATE INDEX TABLE2_INDEX1 ON TABLE2 (KEY_COL1)
CREATE INDEX TABLE3_INDEX1 ON TABLE3 (KEY_COL2)
```

(anticipating index probe and nested loop join via index)

Example 9

```
SELECT A.COLOR, A.SIZE, SUM(A.SALES), SUM(A.QUANTITY)
     FROM TABLE1 A
     WHERE A.COLOR IN ('BLUE', 'GREEN', 'RED')
     GROUP BY A.COLOR, A.SIZE

CREATE INDEX TABLE1_INDEX1 ON TABLE1 (COLOR, SIZE)
```

(anticipating index probe and grouping via index)

Example 10

```
SELECT A.COLOR, A.SIZE, SUM(A.SALES), SUM(A.QUANTITY)
     FROM TABLE1 A
     WHERE A.COLOR IN ('BLUE', 'GREEN', 'RED')
     GROUP BY A.COLOR, A.SIZE
     ORDER BY A.COLOR, A.SIZE

CREATE INDEX TABLE1_INDEX1 ON TABLE1 (COLOR, SIZE)
```

(anticipating index probe, grouping via index and ordering via index)

Example 11

```
SELECT A.COLOR, A.SIZE, MIN(A.QUANTITY)
     FROM TABLE1 A
     WHERE A.COLOR IN ('BLUE', 'GREEN', 'RED')
     GROUP BY A.COLOR, A.SIZE

CREATE INDEX TABLE1_INDEX1 ON TABLE1 (COLOR, SIZE, QUANTITY)
```

(anticipating index probe and grouping via index with "early exit" for MIN)

Example 12

```
SELECT B.COLOR, B.SIZE, SUM(A.SALES), SUM(A.QUANTITY)
     FROM TABLE1 A,
          TABLE2 B
     WHERE A.KEY_COL1 = B.KEY_COL1
     AND B.COLOR IN ('BLUE', 'GREEN', 'RED')
     AND B.SIZE IN ('LARGE', 'X-LARGE')
     GROUP BY B.COLOR, B.SIZE
     ORDER BY B.COLOR, B.SIZE

CREATE INDEX TABLE1_INDEX1 ON TABLE1 (KEY_COL1)
CREATE INDEX TABLE2_INDEX1 ON TABLE2 (COLOR, SIZE, KEY_COL1)
```

(anticipating index probe, nested loop join via index, and grouping and ordering via index)

Example 13

```
SELECT A.COLOR, A.SIZE, A.SALES
     FROM TABLE1 A
     WHERE A.SALES < (SELECT AVG(B.SALES)
                      FROM TABLE1 B
                      WHERE B.SIZE IN ('LARGE', 'X-LARGE'))
     AND A.COLOR IN ('BLUE', 'GREEN', 'RED')
     AND A.SIZE IN ('LARGE', 'X-LARGE')

CREATE INDEX TABLE1_INDEX1 ON TABLE1 (SIZE, COLOR)
```

(anticipating index probe and grouping via index)

Since SIZE is the first key column in the index, the index can be used for both the inner and outer queries.)

Example 14

```
SELECT A.COLOR, A.SIZE, SUM(A.SALES), SUM(A.QUANTITY)
     FROM TABLE1 A
     GROUP BY B.COLOR, B.SIZE

CREATE INDEX TABLE1_INDEX1 ON TABLE1 (COLOR, SIZE)
```

(anticipating grouping via index)

Or

```
CREATE INDEX TABLE1_INDEX1 ON TABLE1 (COLOR, SIZE, SALES,
QUANTITY)
```

[anticipating grouping via index and index only access (all columns in the index)]

Example 15

```
SELECT A.CUSTOMER, A.CUSTOMER_NUMBER, A.YEAR, A.MONTH,
A.SALES
     FROM TABLE1 A
     WHERE A.SALES > (SELECT AVG(B.SALES)
                      FROM TABLE1 B
                      WHERE B.CUSTOMER = A.CUSTOMER
                      AND B.YEAR = A.YEAR)
     AND A.YEAR = 2001

CREATE INDEX TABLE1_INDEX1 ON TABLE1 (YEAR, CUSTOMER, SALES)
```

[anticipating index probe, index only access (subquery) and grouping via index (subquery) — index used for both queries]

Example 16

```
SELECT A.CUSTOMER, A.CUSTOMER_NUMBER, A.YEAR, A.MONTH,
A.SALES
     FROM TABLE1 A
     WHERE A.CUSTOMER_NUMBER IN
          (SELECT B.CUSTOMER_NUMBER
                         FROM TABLE2 B
                         WHERE NUMBER_OF_RETURNS > 10
                         AND B.YEAR = 2000
                         AND B.MONTH IN (10, 11, 12))

     AND A.YEAR = 2001
     AND A.MONTH IN (1, 2, 3)

CREATE INDEX TABLE1_INDEX1 ON TABLE1
     (YEAR, MONTH, CUSTOMER_NUMBER)

CREATE INDEX TABLE2_INDEX1 ON TABLE2
     (YEAR, MONTH, CUSTOMER_NUMBER, NUMBER_OF_RETURNS)
```

(anticipating index probe, subquery join composite via nested loop join via index)

Example 17

```
UPDATE TABLE1
     SET COL1 = 125
     SET COL2 = 'ABC'
     SET COL3 = 'This is updated'
     WHERE CUSTOMER_NUMBER IN (4537, 7824, 2907)
     AND YEAR = 2001

CREATE INDEX TABLE1_INDEX1 ON TABLE1 (YEAR,
CUSTOMER_NUMBER)
```

Anticipating index probe

Example 18

```
UPDATE TABLE1 A
     SET LAST_YEARS_MAX_SALES =
          (SELECT MAX(SALES)
                         FROM TABLE2 B
                         WHERE B.YEAR = 2000
                         AND B.MONTH = A.MONTH
                         AND B.CUSTOMER = A.CUSTOMER)
     WHERE A.YEAR = 2001

CREATE INDEX TABLE1_INDEX1 ON TABLE1 (YEAR)

CREATE INDEX TABLE2_INDEX1 ON  TABLE2
     (YEAR, MONTH, CUSTOMER, SALES DESC)
```

(anticipating index probe and grouping via index with "early exit" for MAX)

Example 19

```
DELETE FROM TABLE1
     WHERE ITEM_NUMBER IN ('23-462', '45-7124', '21-2007')
     AND QUANTITY = 0

CREATE INDEX TABLE1_INDEX1 ON TABLE1 (ITEM_NUMBER,
QUANTITY)
```

(anticipating index probe)

Example 20

```
SELECT     T.CHAR_DATE,
           C.COUNTRY,
           C.CUSTOMER_NAME,
           P.PART_NAME,
           S.SUPPLIER_NAME,
           SUM(F.QUANTITY),
           SUM(F.REVENUE_WO_TAX)
      FROM STARLIB/SALES_FACTS F,
           STARLIB/PART_DIM P,
           STARLIB/TIME_DIM T,
           STARLIB/CUST_DIM C,
           STARLIB/SUPP_DIM S
      WHERE F.PARTKEY = P.PARTKEY
      AND       F.TIMEKEY = T.TIMEKEY
      AND       F.CUSTKEY = C.CUSTKEY
      AND       F.SUPPKEY = S.SUPPKEY
      AND       T.YEAR = 2000
      AND       T.MONTH = 06
      AND       T.DAY = 30
      AND       C.COUNTRY = 'JAPAN'
      AND       P.MFGR = 'Manufacturer#3'
      GROUP BY  T.CHAR_DATE,
                C.COUNTRY
                C.CUSTOMER_NAME,
                P.PART_NAME,
                S.SUPPLIER_NAME
      ORDER BY  T.CHAR_DATE,
                C.COUNTRY,
                C.CUSTOMER_NAME,
                P.PART_NAME

CREATE INDEX SALES_FACTS_INDEX1 ON SALES_FACTS (PARTKEY)
CREATE INDEX SALES_FACTS_INDEX2 ON SALES_FACTS (TIMEKEY)
CREATE INDEX SALES_FACTS_INDEX3 ON SALES_FACTS (CUSTKEY)
CREATE INDEX SALES_FACTS_INDEX4 ON SALES_FACTS (SUPPKEY)

CREATE INDEX PART_DIM_INDEX1 ON PART_DIM (MFGR, PARTKEY,
PART_NAME)
CREATE INDEX TIME_DIM_INDEX1 ON TIME_DIM (YEAR, MONTH, DAY,
TIMEKEY)
CREATE INDEX CUST_DIM_INDEX1 ON CUST_DIM (COUNTRY, CUSTKEY)
CREATE INDEX SUPP_DIM_INDEX1 ON SUPP_DIM (SUPPKEY,
SUPPLIER_NAME)

Generate column statistics on T.CHAR_DATE, C.COUNTRY,
C.CUSTOMER_NAME, P.PART_NAME, S.SUPPLIER_NAME (V5R2 only)
```

(anticipating index probe, index only access and nested loop join via index)

Appendix B — Tools for Analysis and Tuning

With a cost-based optimizer and a broad set of indexing choices, the right analysis tools are a necessity. On an iSeries server, a variety of tools are available for evaluating what the optimizer is choosing for implementation methods and how to influence its choices. Depending how much is known at the beginning of the analysis process, different tools may be selected, or a combination of tools and methodologies may be used. While the scope of this document does not include the utilization of the various tools, here is a list and brief description of five of the tools available:

Optimizer Feedback via "Debug Mode"

The DB2 UDB for iSeries optimizer is generous in providing feedback in the form of OS/400 messages. The messages will be written to the query's joblog. Users can initiate these messages by using the OS/400 command Start Debug (STRDBG), or the QAQQINI parameter "MESSAGES_DEBUG" with a value of "*YES".

Database Monitor

As part of DB2 UDB for iSeries, users have access to the Database Monitor. The monitor has the ability to collect summary data, as well as detail data on the query implementation plans and runtime attributes. Once set up, the monitor can track query plans and performance, either across the system or for a particular job. The analyst turns the monitor on and lets the system run normally. The monitor collects information about the system and the queries being run on it. When the monitor is turned off, the information is sent to a table that the analyst can then query. The monitor can provide information, such as: which queries are the longest running, which queries are building temporary indexes, and index (access path) suggestions from the optimizer. Once these queries are identified, the analyst can address any performance problems that these queries are causing. More information on a set of analysis queries can be found at:
ibm.com/eserver/iseries/db2/dbmonqrys.htm

iSeries Navigator

iSeries Navigator is the graphical interface to an iSeries server and DB2 UDB for iSeries. iSeries Navigator is part of OS/400 and is delivered via Client Access Express. iSeries Navigator has been significantly enhanced over the last few releases. Through its GUI front end, experienced, as well as new, iSeries administrators can manage their systems and database effectively and efficiently.

Most important for application administrators and database analysts, however, is the improved capabilities to manage databases, run SQL statements, and monitor query performance. With iSeries Navigator, it is easy to display a list of schemas (libraries) and tables, authorize users, and even look at the contents of a table — all with a double click of the mouse.

An SQL script window is available from the File menu for running interactive SQL statements. While running SQL statements in the Script window, you can put your server job into "debug mode." This tells the optimizer to provide feedback in

the joblog. From the SQL Script window, you can display the messages in the joblog and analyze the optimization plan of the query.

The Database Monitor interface lets you monitor query performance just by specifying a job and library name. Through iSeries Navigator, a summary or detailed monitor can be started and stopped, and the output analyzed with a standard set of reports. Database monitors started via the OS/400 command interface can be imported into iSeries Navigator and analyzed, as well. More information on iSeries Navigator can be found at:
ibm.com/eserver/iseries/navigator/

PRTSQLINF

Print SQL Information is an OS/400 command that will print the query implementation plan contents from a high-level language (HLL) program or service program with embedded static SQL, or a SQL package object used with extended dynamic SQL requests. The output of the PRTSQLINF command is a spooled file. Browsing the spooled file output will show the query optimizer feedback for the SQL statements contained in the program or package.

SQL Explain information can also be requested for an object using iSeries Navigator if your operating system is at V5R2.

Given that a query plan can be regenerated dynamically, and the fact that the SQL Query Engine can store and use up to three different plans for the same SQL request, the PRTSQLINF information may not reflect the query plan that is being executed. You may have to make use of one of the other tools — such as the database monitors, debug messages, or Visual Explain — to see the plan you are interested in analyzing.

Visual Explain

Visual Explain (VE) is the latest tool used to understand and analyze query implementation plans and optimizer feedback. Visual Explain is part of iSeries Navigator and is used to render the data in a database monitor table into visual and textual information. The output is a combination of diagrams and text that "explains" the access plans of the query, optimizer recommendations, and database and system environments.

Visual Explain can render the plans of a query, with or without actually running the query to completion. The two options are: EXPLAIN and RUN AND EXPLAIN.

From the information provided by VE, you can verify indexes used, optimizer index and statistics recommendations, and look for temporary index creation. Even create the temporary index permanently. More information on Visual Explain can be found at: **ibm.com**/eserver/iseries/navigator/

References

Books

Description	Details
Information about DB2 UDB for iSeries, SQL and the query optimizer	publib.boulder.ibm.com/iseries/v5r2/ic2924/index.htm Database Performance and Query Optimization SQL Programming Concepts SQL Programming with Host Languages SQL Reference
Overview of the AS/400 and DB2 UDB for iSeries	Soltis, Frank, *Inside the AS/400*, 1997, Duke Press.

Web sites

Description	URL
DB2 UDB for iSeries	**ibm.com**/eserver/iseries/db2
IBM Business Intelligence Home page	**ibm.com**/solutions/businessintelligence/
EVI Patent information	delphion.com/
Centerfield Technology	centerfieldtechnology.com/

Related Education

Description	URL for more information
DB2 UDB for iSeries SQL and Query Performance Tuning and Monitoring workshop (4 days)	**ibm.com**/eserver/iseries/service/igs/db2performance.html
IBM Technical Studio (on-line tutorials delivered over the Web)	**ibm.com**/eserver/iseries/tstudio/
DB Monitor and Debug Information	**ibm.com**/eserver/iseries/developer/client/performance/csperdg5
Internet-based Offering courses for DB2 UDB for iSeries	**ibm.com**/eserver/iseries/developer/education/
Various different types of iSeries education	**ibm.com**/eserver/iseries/education/

Other Notices

About the Author

Mike Cain IBM DB2 Technology Team Leader
IBM eServer Solutions Enablement

Mike Cain is an IBM senior technical staff member and member of the DB2 UDB for iSeries Technology team in the IBM eServer Solutions organization and the iSeries Teraplex Integration Center in Rochester, Minnesota. Prior to his current position, he worked as an IBM AS/400 systems engineer and technical consultant.

Acknowledgments

Thanks to Amy Anderson, Rob Bestgen, Randy Egan, Robert Driesch, Mike Faunce, Kent Milligan, and Jarek Miszczyk for their input and reviews.

Production

Teresa Powell IBM eServer Solutions Enablement

For Additional Information

To get more details on becoming a member of PartnerWorld for Developers, visit:
ibm.com/eserver/iseries/developer/membership/reg_info.html

Check out the various IBM eServer Solutions offerings at:
ibm.com/ servers/enable/i/index.html

Additional information regarding the Teraplex Center can be found at:
ibm.com/servers/enable/site/bi/teraplex/index.html

Trademarks and Disclaimers

IBM, eServer, iSeries, OS/400, AS/400, DB2, and are trademarks or registered trademarks of International Business Machines Corporation.

IBM makes no commitment to make available any products referred to herein.

All other registered trademarks and trademarks are properties of their respective owners.

References in this publication to IBM products or services do not imply that IBM intends to make them available in every country in which IBM operates.