Art and Science of SQL Performance Tuning

PartnerWorld for Developers iSeries
Rochester, MN USA

- Optimizing your queries (using any of the IBM query products - SQL, Query Manager, Open Query File, Query/400, ODBC) is one of the most challenging tasks in making your applications perform as fast as possible. This session will cover the art of query optimization in easy-to-understand terms. New query explain tools are available which will help simplify the process.

Science

- What are you asking the system to do?
  - Type of request
  - SQL coding
- How can the system do it?
  - SQL Query implementation via the Optimizer, OS and SLIC
  - DB design
- Where is the system going to do it?
  - I/O Intensive
  - CPU Intensive
  - Available Resources

- Words behind the previous framework
- A method to understand all the major aspects that affect optimization and performance
- Where to start investigating SQL request optimization and/or performance issues

Optimization

The Optimizer

Writes the best? program to fulfill your request

The Optimizer

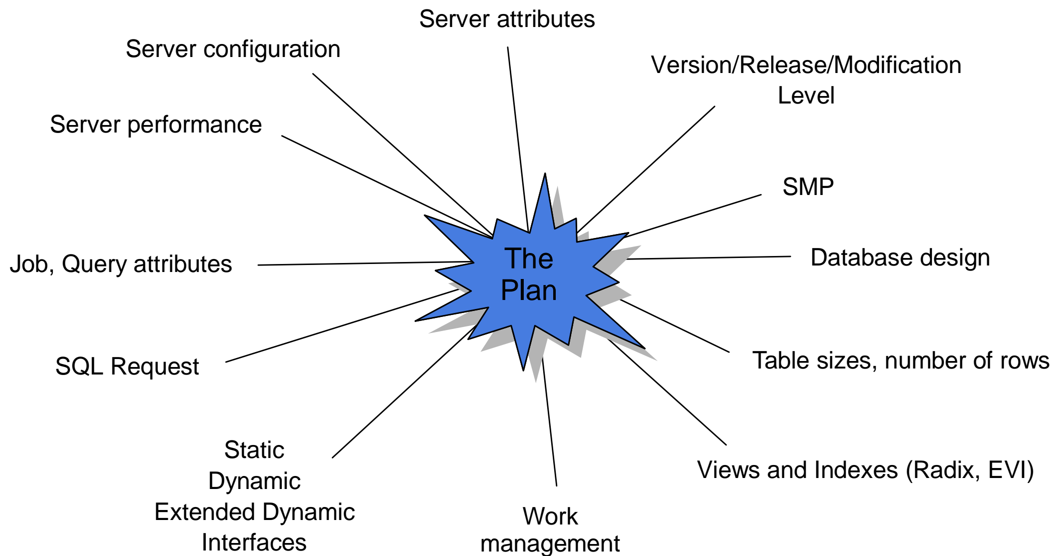
Provides the recipe

Provides the methods

Does no cooking

- The query optimizer's job is to build the access plan
- DB run time actually executes the plan
- Think of the optimizer as the program that writes the program to fulfill the user's request

Optimization... the intersection of various factors



- A given query plan can be thought of as an intersection of all the factors that affect cost based optimization on a given server with a given database design.
- To really understand a given implementation plan and it performance, one must know and understand all the various factors and settings in effect at the time of query optimization and execution.
- Change any one or more of the factors and the implementation plan and performane may change.

Common Terms

Term	Meaning	Object
Table	Data repository or dataspace. Physical File.	*File, PF
Index	Binary radix tree built over a table to order particular columns (keys) of the table, useful for quick binary searches. Encoded vector (EVI) built over a table, used to create bitmaps for query processing. Logical File.	*File, LF
Temporary Index	Radix index built "on the fly" by the optimizer.	
Temporary Result	Copy of data from an intermediate step of the query. Needed to complete the query.	
Access Plan	Plan generated by the optimizer of how to access the tables being queried.	Dynamic, *PGM, *SRVPGM, *SQLPKG
ODP (Open Data Path)	Active path through which query data is read.	
Reusable ODP	ODP kept open by the system when an SQL query is repeatedly executed (run).	

- These terms will be used throughout the course
- Used to compare / contrast AS/400 objects with SQL DB objects
- A table is the same as a physical file and can be created through CRTPF or CREATE TABLE.
- A table created through SQL may have a performance advantage since it checks data integrity at input rather then output.
- An index is the same as a keyed logical file and can be created through CRTLF or CREATE INDEX.

Access Plans

Contents

- A control structure that contains information on the actions necessary to satisfy each SQL request
- These contents include:
  - Access Method
  - Info on associated tables and indexes
  - Any applicable program and/or environment information

- High level view and explaination of an "access plan"
- Will be covered in detail later in the course

Data Access

- Write a program to find the rows that contain the color purple within a 1 million row DB table

...WHERE COLOR = 'Purple'...

- When...
  - 1 row contains the color purple
  - 1,000 rows contain the color purple
  - 100,000 rows contain the color purple
  - 1,000,000 rows contain the color purple

- Simple example to be used through out the course
- Used to understand what the optimizer's job is and what it goes through to pick the best implementation method
- Ask the audience to write a HLL program to statisfy the request
- Points out the fact that, based on the actual data values and skew, might want to use different access methods to handle the request
  - Accessing a few rows = one access method
  - Accessing many rows = a dfferent access method
  - A different color, or a change in the number of rows for a given value should result in a different (better?) access method, automatically
- Audience should start realize that an index or keyed access method might be used, or a table scan may be better
- Optimizer has to be able to change the method as the external factors change (recall that the optimizer's job is to write the best program to handle the SQL request)

Data Access

- SQL to find the rows that contain the color purple, within a 1 million row DB table, when...
  - 300,000 rows contain the color purple

```
SELECT ORDER, COLOR, QUANTITY
FROM ITEM_TABLE
WHERE COLOR = 'PURPLE'
```

  - Without index over COLOR, assume 100,000 rows (10% default from =)
  - With radix index over COLOR, estimate 291,357 rows (read keys)
  - With EVI over COLOR, actual 300,000 rows (read symbol table)

Optimizer uses number of rows, not a percentage

- Example of default filer factor, estimate from radix index and actual value(s) from EVI
- Point: index(s) provide the optimizer with important information and stats to determine the best access method(s)

Implementation Methods

Implementation Methods Overview

- Non-Keyed Data Access Methods
  - Table Scan
  - Parallel Table Scan
  - Parallel Pre-fetch
  - Parallel Table Pre-load
  - Skip Sequential with dynamic bitmap
  - Parallel Skip Sequential
- Keyed Data Access Methods
  - Key Positioning and Parallel Key Positioning
  - Dynamic Bitmaps / Index ANDing ORing
  - Key Selection and Parallel Key Selection
  - Index-From-Index
  - Index-Only Access
  - Parallel Index Pre-load
- Joining, Grouping, Ordering
  - Nested Loop Join
  - Hash Join
  - Index Grouping
  - Hash Grouping
  - Index Ordering
  - Sort

- High level list of the access methods and techniques available to fulfill the SQL request
- Science topics will cover all of these methods, in this order...
- Audience can refer to this list as a way to understand all the various ways or methods available

Implementation Methods Overview

- Non-Keyed Data Access Methods ←
  - Table Scan
  - Parallel Table Scan
  - Parallel Pre-fetch
  - Parallel Table Pre-load
  - Skip Sequential with dynamic bitmap
  - Parallel Skip Sequential
- Keyed Data Access Methods
  - Key Positioning and Parallel Key Positioning
  - Dynamic Bitmaps / Index ANDing ORing
  - Key Selection and Parallel Key Selection
  - Index-From-Index
  - Index-Only Access
  - Parallel Index Pre-load
- Joining, Grouping, Ordering
  - Nested Loop Join
  - Hash Join
  - Index Grouping
  - Hash Grouping
  - Index Ordering
  - Sort

- Covering the "non keyed" or "arrival sequence" access methods

Table Scan

Reads all rows from the table and applies the selection criteria to the data within the table.

- Advantages:
  - Minimizes page I/O operations through asynchronous pre-fetching of the data since the pages are scanned sequentially
  - Can perform selection directly on the table image in memory or on the intermediate buffer after all derived operations have been performed
- Potential disadvantages:
  - All rows in the table are examined regardless of the selectivity of the query
  - Rows marked as deleted are examined even though none will be selected
- Used when:
  - Greater than ~20% of the rows are selected
  - Table size is less than 32K

- Percentage is a "rule of thumb", the optimizer estimates actual I/Os.
- Rule of thumb is very system and data dependent (ex. 12-way 20GB system might perform a full table scan for 7% of the rows).
- Table scan = arrival sequence processing.

Table Scan Example

```
SELECT * FROM EMPLOYEE
   WHERE WORKDEPT BETWEEN 'A01' AND 'E01'
```

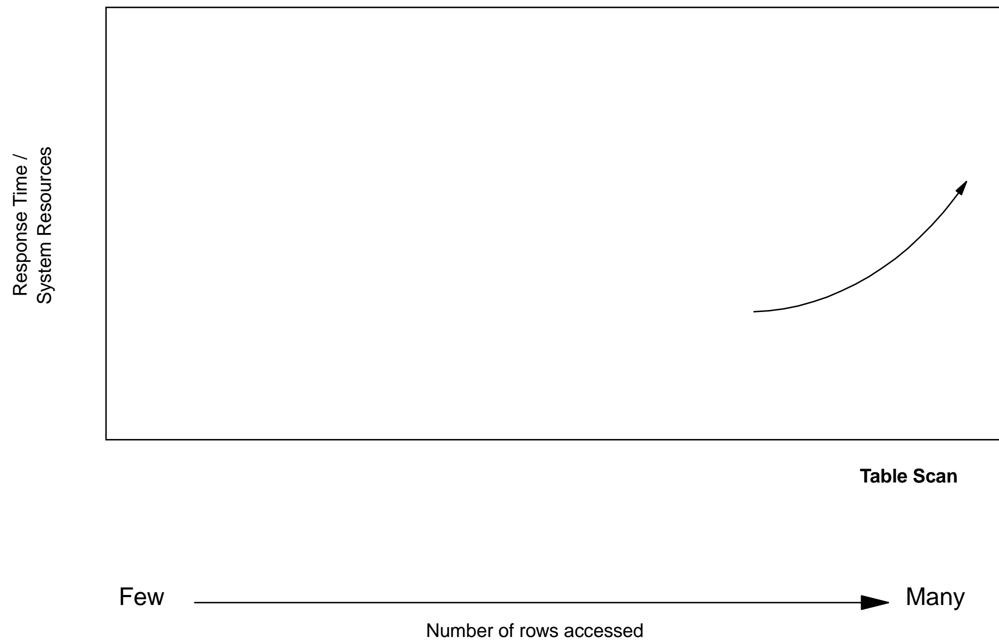
SQL4010 Arrival sequence access for file 1.

This message = Table scan

Why might a table scan be used...?

- Table scan (arrival sequence processing) is very efficient on the AS/400 due to the independent I/O subsystem, large memory and single level storage and automatic data stripping.
- SQLxxxx messages are used to illustrate feedback from the optimization.
- SQLxxxx messages are used for PRTSQLINF and allow the meaningful 1st level text to be shown in the spool file and the message text to be translatable MRI.

Data Access Methods



- The first major method of access is established (table scan)

Encoded Vector Index (EVI)

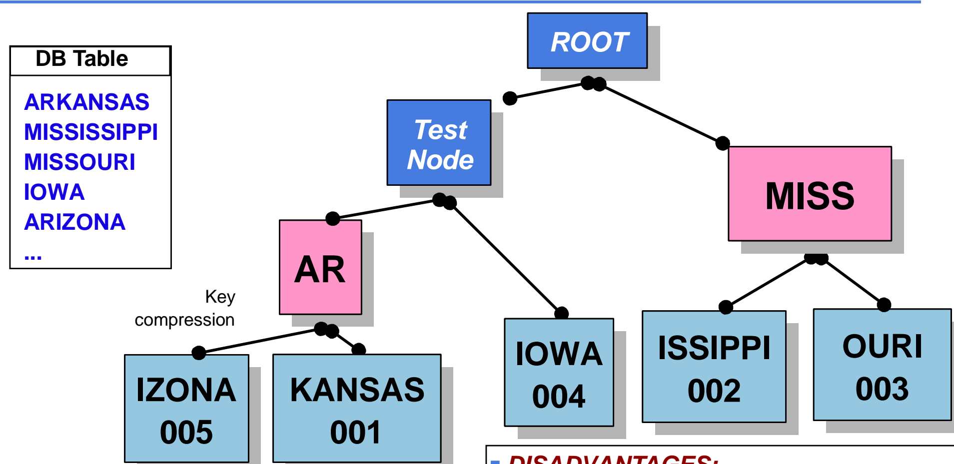
- New index object for delivering fast data access in decision support and query reporting environments
  - Complementary alternative to existing index object (binary radix tree structure - logical file or SQL index)
  - Advanced technology from IBM Research, that's a variation on bitmap indexing
  - Easy to access data statistics improve query optimizer decision making

The encoded vector index (EVI) object was architected explicitly for enhanced query performance and optimization. The design is based on IBM patented technology and DB2 for AS/400 is the first database to deliver this exciting new technology.

The current AS/400 indexing technology (binary radix tree index) is more of a general purpose index. The current technology is quite effective in boosting the performance of OLTP and basic decision support solutions on the AS/400. Encoded vector index offers a complementary technology that will allow DB2/400 to improve query performance across a wider range of queries. EVIs will have the biggest impact on the complex query workloads found in business intelligence solutions and adhoc query environments.

EVIs also offer a dramatic advancement on bitmap indexing technology. Bitmap indexing technology is used by many database vendors to improve performance in query-heavy environments. As you will see on the next page, EVIs offer precise accurate statistics for use by the DB2 for AS/400 query optimizer

Comparing - Binary Radix Indexes



- ***ADVANTAGES:***
  - Quick access to a single key value (million-entry index, on average, only 20 tests)
  - Also efficient for small, selected range of key values (low cardinality)

- ***DISADVANTAGES:***
  - Table rows retrieved in order of key values (not physical order) which equates to many ***RANDOM*** I/Os when selecting a large number of keys (high cardinality)
  - No way to predict which physical index pages are next when traversing the index for large number of key values

This sample figure contains a high-level representation of the AS/400 binary radix tree index structure. It's a multilevel structure that allows a large number of key values to be stored efficiently while minimizing access times. The lowest level of the tree contains the leaf nodes. The leaf nodes contain the row id of the rows in the base table that contain the associated key value. In this example, the key value of 'MISSOURI' can be found only in the third row of the sample table. The bit representation of the key value is used to quickly navigate to the leaf node for that key value with a few simple binary search tests. Thus, a single key value can be accessed quite fast with a small number of tests. This quick access is pretty consistent across all key values in the index since the system keeps the depth of the index shallow. The binary radix tree structure is very good for finding a small number of rows because it's able to find a given row with a minimal amount of processing. A sample query matching this description would be an OLTP request like find the outstanding orders for customer number X where fast performance is delivered by creating a binary radix tree index over the customer number field. An index created over the customer number field would be considered the perfect index for this type of query. It's worth noting that the perfect binary radix tree index will almost always be faster than an encoded vector index when a small number of rows are being selected.

When the requested key range (e.g., find customers in the Western region with outstanding orders) identifies a large number of rows in the base table, then the binary radix tree efficiency is reduced. This data access method is less efficient because the row id's are retrieved in the order of the key values, not the physical ordering of the base table. The first customer from California processed could be in row number 5000 of the base table, the second Californian customer could be in row number 2, the next customer in row number 2000, and so on. These random jumps around the base table usually result in less efficient I/O and possibly page faults because DB2/400 cannot predict which part of the table will be accessed next. The page fault situation is worsened by the fact that the binary radix tree index can be quite large since for every row in the base table, a copy of the key value and row identifier is stored in the index structure. In a data warehouse environment, you could have both a large database table and a large binary radix tree index fighting for the same main memory space. This type of situation is common in most business intelligence solutions due to the large amounts of historical data being stored in the warehouse.

Encoded Vector Index (EVI)

What is it?

- New type of index
- File object type, LF subtype

EVI composed of two parts:

SYMBOL TABLE:

Key Value	Code	First Row	Last Row	Count
Arizona	1	1	80005	5000
Arkansas	2	5	99760	7300
......				
Virginia	37	1222	30111	340
Wyoming	38	7	83000	2760

VECTOR:

Vector	Record
Code 1	1
Code 17	2
Code 18	3
Code 9	4
Code 2	5
Code 7	6
Code 38	7
Code 38	8
Code 1	9
...	...

- Symbol table contains information for each distinct key value. Each key value is assigned a unique hex code
  - Code is 1, 2, or 4 bytes - depending on number of distinct key values
- Rather than a bit array for each distinct key value, the index has one array of codes (a.k.a., the Vector)

Encoded vector indexes are an advanced form of bitmap indexing that use a single array (or vector) to address some of the issues that arise when there are more than just a few distinct key values (high cardinality). The encoded vector index structure consists of the single vector and a symbol table as shown in this figure. The symbol table contains an entry for each distinct key value and the unique code value assigned for that key value. Data statistics such as the number of occurrences are also maintained for each key value to aid in query optimization. The single vector has an entry for each row in the base table and the entry contains the unique code value corresponding to the key value found in the row of the base table. Just like the bitmap example, 'Wyoming' (code value 25) is the key value found in rows 7 & 8 of the table. The vector element is 1, 2, or 4 bytes in size depending on the number of distinct key values - if there's less than 256 distinct key values a 1 byte element will be used. If new distinct key values are added that overflow the number supported by the element size, then DB2/400 will automatically rebuild the EVI with a larger element size. In addition, the order of the elements in the vector mirrors the physical ordering of the table data which leads to more efficient I/O processing. The key value statistics are automatically maintained by DB2 for AS/400 and the simple structure makes it very fast and easy for the optimizer to analyze the data statistics.

NOTE: Symbol table is initially sorted when it's first created.

Dynamic Bitmaps

Index ANDing ORing

Courses

Location	Topic
NY	DB2/400
LA	JAVA
NY	JAVA
CHI	NOTES

Index LocEVI, created over the Location column and index TopIX, created over the Topic column in the Courses table.

EncodedVector Index

CREATE ENCODED VECTOR INDEX LocEVI on Courses (Location)

CREATE INDEX TopIX on Courses (Topic)

Binary Radix Index

Encoded vector indexes can also retain the advantages provided by bitmap indexing while removing the disadvantages. DB2/400 delivered dynamic bitmap indexing support in V4R2 for rapid data retrieval. The optimizer can build bitmaps on the fly from your existing binary radix tree indexes to identify exactly which rows the user requested. In addition, bitmaps are only built for those distinct key values being referenced on the query instead of building a bitmap for every distinct key value. Beginning in V4R3, dynamic bitmaps will can also be built from encoded vector indexes. The creation of these dynamic bitmaps will be up to 10 times faster since the EVI structure is simpler to navigate and smaller in size than the radix tree structure. Early testing performed in the lab showed that EVIs can require up to 16 times less space and have build times 1.5 times faster than binary radix tree indexes. This dramatic difference in size offers two advantages. First, DB2/400 can handle more concurrent query users since EVI processing requires less main memory. And second, more encoded vector indexes can be created on the system to improve the performance of a larger number of queries by either using a single index or combining multiple indexes together.

DB2 UDB's dynamic bitmap indexing also allows the optimizer to combine multiple indexes from a single table together with bitmaps to more effectively return the requested data. For example, when the search criteria contains Boolean AND and OR operations for different columns (e.g., Location='NY' AND Topic='Java'), the same Boolean AND and OR operations can be applied to any associated bitmaps to more efficiently identify the requested rows. Faster bitmap build times with EVIs means that DB2/400 has more time to build and combine bitmaps in its quest to implement a complex query efficiently.

Encoded vector indexes are even more important in ad hoc query environments where combining multiple indexes (via bitmaps) is a bigger requirement than decision support solutions featuring static, pre-canned reports. You can do a good job of creating the perfect index for the pre-canned queries and reports that you know about ahead of time. For example, if you are going to search for customers by state, then creating an index with the state column as the key would most likely provide the perfect index for that type of query. Tuning a database for ad hoc queries is more trying since you cannot predict and create all the optimal indexes ahead of time. As you know, the performance of ad hoc queries can range from a little slower to hours slower since the database hasn't been tuned ahead of time. Dynamic bitmap processing with EVIs allows DB2/400 to provide acceptable performance even for adhoc queries. Instead of relying on the perfect index for implementing the query, the optimizer instead can combine bitmaps from multiple existing indexes to improve query performance. In addition, bitmaps from encoded vector indexes and radix tree indexes can be combined to provide even more flexibility. The bitmap processing offered by encoded vector indexes offers a performance safety net especially for ad hoc query environments by allowing the optimizer to benefit more from existing indexes instead of just resorting to a more resource intensive table scan when the ideal index doesn't exist.

Dynamic Bitmaps

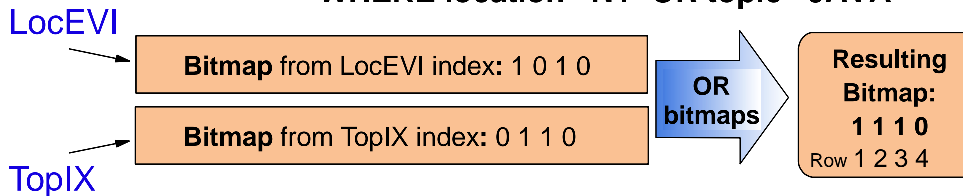
Index ANDing ORing

Courses

Location	Topic
NY	DB2/400
LA	JAVA
NY	JAVA
CHI	NOTES

Index LocEVI, has been created over the Location column and index TopIX, created over the Topic column in the Courses table.

SELECT coursenumber FROM courses WHERE location='NY' OR topic='JAVA'



- Bitmaps can be derived from binary radix or encoded vector indices
  - bit order mirrors physical ordering of table data

Encoded vector indexes can also retain the advantages provided by bitmap indexing while removing the disadvantages. DB2/400 delivered dynamic bitmap indexing support in V4R2 for rapid data retrieval. The optimizer can build bitmaps on the fly from your existing binary radix tree indexes to identify exactly which rows the user requested. In addition, bitmaps are only built for those distinct key values being referenced on the query instead of building a bitmap for every distinct key value. Beginning in V4R3, dynamic bitmaps will can also be built from encoded vector indexes. The creation of these dynamic bitmaps will be up to 10 times faster since the EVI structure is simpler to navigate and smaller in size than the radix tree structure. Early testing performed in the lab showed that EVIs can require up to 16 times less space and have build times 1.5 times faster than binary radix tree indexes. This dramatic difference in size offers two advantages. First, DB2/400 can handle more concurrent query users since EVI processing requires less main memory. And second, more encoded vector indexes can be created on the system to improve the performance of a larger number of queries by either using a single index or combining multiple indexes together.

DB2 UDB's dynamic bitmap indexing also allows the optimizer to combine multiple indexes from a single table together with bitmaps to more effectively return the requested data. For example, when the search criteria contains Boolean AND and OR operations for different columns (e.g., Location='NY' AND Topic='Java'), the same Boolean AND and OR operations can be applied to any associated bitmaps to more efficiently identify the requested rows. Faster bitmap build times with EVIs means that DB2/400 has more time to build and combine bitmaps in its quest to implement a complex query efficiently.

Encoded vector indexes are even more important in ad hoc query environments where combining multiple indexes (via bitmaps) is a bigger requirement than decision support solutions featuring static, pre-canned reports. You can do a good job of creating the perfect index for the pre-canned queries and reports that you know about ahead of time. For example, if you are going to search for customers by state, then creating an index with the state column as the key would most likely provide the perfect index for that type of query. Tuning a database for ad hoc queries is more trying since you cannot predict and create all the optimal indexes ahead of time. As you know, the performance of ad hoc queries can range from a little slower to hours slower since the database hasn't been tuned ahead of time. Dynamic bitmap processing with EVIs allows DB2/400 to provide acceptable performance even for adhoc queries. Instead of relying on the perfect index for implementing the query, the optimizer instead can combine bitmaps from multiple existing indexes to improve query performance. In addition, bitmaps from encoded vector indexes and radix tree indexes can be combined to provide even more flexibility. The bitmap processing offered by encoded vector indexes offers a performance safety net especially for ad hoc query environments by allowing the optimizer to benefit more from existing indexes instead of just resorting to a more resource intensive table scan when the ideal index doesn't exist.

Dynamic Bitmaps

Index ANDing ORing

Why is index ANDing / ORing a good technique...?

Leverages multiple indexes (radix or EVI) to select to rows
Eliminates I/O by combining selection criteria and representing it in one bitmap.

Dynamic Bitmaps

Index ANDing ORing

Bitmaps are dynamically generated from existing indexes to reduce the I/O operations against the table

- Advantages:
  - Multiple indexes can be used against a single table
  - OR'ed predicates can be implemented with a tertiary index
  - Bitmaps can be generated and analyzed (logical AND and OR operations) in parallel
  - Can help to avoid some index creations
- Potential disadvantages:
  - The entire bitmap must be generated prior to retrieving any records
  - The generated bitmaps are static for the duration of the query
- Used when:
  - The savings from eliminating I/O operations outweigh the cost to generate and analyze the bitmap(s)

CPU parallelism

The most common case is where only one index is used. The bitmap will let us avoid the random I/O to the DS normally found using an index, thereby allowing higher % selectivity indexes to be used.
The IxAO can help to avoid some index creations. Since tertiaries can be used in conjunction with primary index, a primary index that isn't that great but satisfies the ordering, grouping or join can be used. The tertiaries help the selectivity enough to avoid building the sparse temporary indexes. This also reduces the ISV to V2 non-reusable ODP stuff.
The bitmaps, once generated, are not maintained as changes occur to the table. Bitmaps are static.
Selection is reapplied to rows when access underlying table.
Even w/o SMP 2 tasks are used to scan IX and create bitmap.

Skip Sequential Access

Scans the bitmap and sequentially reads rows from the table that match the selection criteria represented in the bitmap

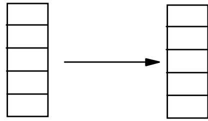
- Advantages:
  - Minimizes page I/O operations through the use of dynamically generated bitmap(s), by skipping pages that have no rows represented in the bitmap
  - Minimizes page I/O operations through asynchronous pre-fetching of the data since the pages are scanned sequentially
  - Can perform selection directly on the table image in memory or on the intermediate buffer after all derived operations have been performed
- Potential disadvantages:
  - The entire bitmap must be generated prior to retrieving any records
  - The generated bitmaps are static for the duration of the query
- Used when:
  - Greater than ~20% of the rows are selected
  - The savings from eliminating I/O operations outweigh the cost to generate and analyze the bitmap(s)

- Percentage is a "rule of thumb", the optimizer estimates actual I/Os.
- Rule of thumb is very system and data dependent (ex. 12-way 20GB system might perform a full table scan for 7% of the rows).
- Table scan = arrival sequence processing.

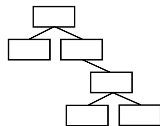
Skip Sequential Access

Step 1
Select keys
and build dynamic
bitmaps

EVI_LOCATION Bitmap



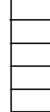
IX_TOPIC Bitmap



ANDing
ORing

Step 2
Scan final
bitmap and
select RRNs

Final
Bitmap



EMPLOYEE

Step 3
Skip
sequentially
and read row
from table

- Picture of skip sequential access to illustrate the steps involved and their relationship
- Either EVI or radix index can be used to create the bitmaps
- Note: there are no RRNs in the bitmaps (only 1s and 0s)

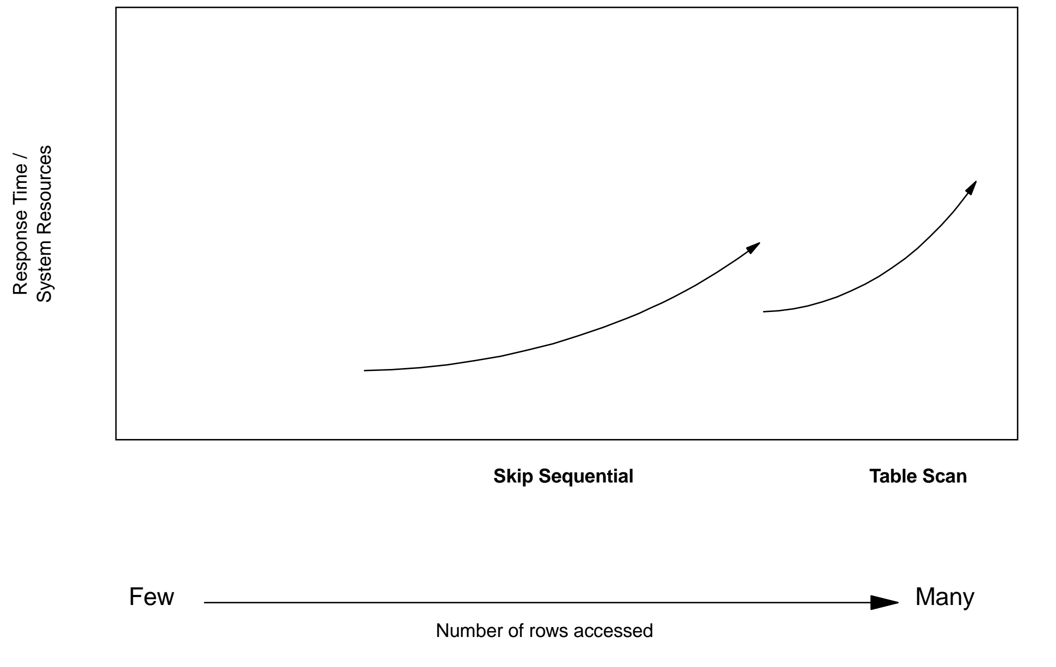
Skip Sequential Example

```
CREATE ENCODED VECTOR INDEX EVI1 ON EMPLOYEE
   (WORKDEPT)
   :
SELECT * FROM EMPLOYEE
   WHERE WORKDEPT BETWEEN 'A01' AND 'E01'
```

SQL4010 Arrival sequence access for file 1.
SQL4032 Access path EVI1 used for bitmap processing of file 1.

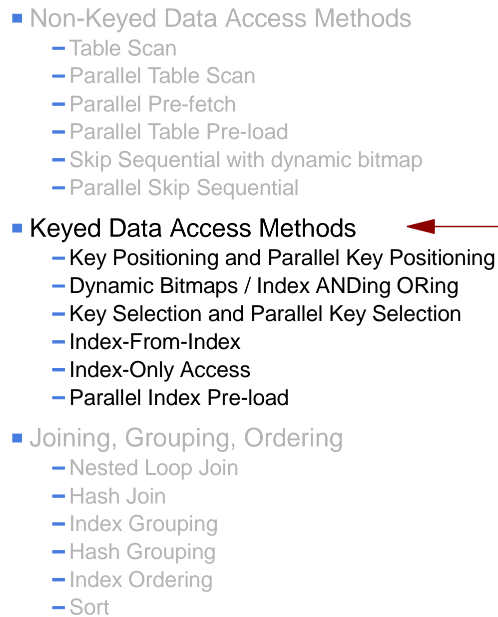
- Arrival processing with a bitmap created from an EVI (aka skip sequential)

Data Access Methods



- The second major method of access is established (skip sequential)

Implementation Methods Overview



- Non-Keyed Data Access Methods
  - Table Scan
  - Parallel Table Scan
  - Parallel Pre-fetch
  - Parallel Table Pre-load
  - Skip Sequential with dynamic bitmap
  - Parallel Skip Sequential
- Keyed Data Access Methods
  - Key Positioning and Parallel Key Positioning
  - Dynamic Bitmaps / Index ANDing ORing
  - Key Selection and Parallel Key Selection
  - Index-From-Index
  - Index-Only Access
  - Parallel Index Pre-load
- Joining, Grouping, Ordering
  - Nested Loop Join
  - Hash Join
  - Index Grouping
  - Hash Grouping
  - Index Ordering
  - Sort

- Covering the "keyed" access methods
- Topics used to discuss radix index usage

Key Positioning

Selection criteria are applied to ranges of index entries before the table is processed.

- Advantages:
  - Only those index entries that are within a selected range are processed
  - Can process both join and selection processing within a single operation if the correct index exists
- Potential disadvantages:
  - Can perform poorly when a large number of rows are selected
- Used when:
  - Less than ~20% of the keys are selected
  - Ordering, grouping, or join operation requires the use of an index
  - The selection columns match the first (n) key fields of the index
  - May be used in combination with key selection

Since there at least two I/Os for every record selected the cost for retrieving a large number of records is cheaper through a table scan since the number of I/Os will be reduced.

This is known affectionately as Multi-key Frogger (MKF) internally.
Equivalent to RPG SETLL or COBOL START.

Key Positioning Example

```
CREATE INDEX X1 ON EMPLOYEE
   (LASTNAME, WORKDEPT)
   :
SELECT * FROM EMPLOYEE
   WHERE WORKDEPT BETWEEN 'A01' AND 'E01'
      AND LASTNAME IN ('SMITH', 'JONES', 'PETERSON')
```

SQL4008 Access path X1 used for file 1.

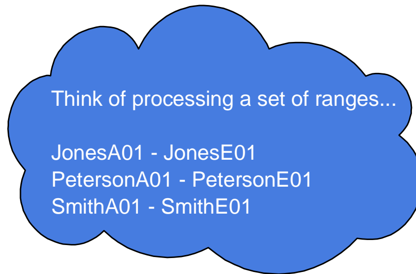
SQL4011 Key row positioning used on file 1.

Since there at least two I/Os for every record selected the cost for retrieving a large number of records is cheaper through a table scan since the number of I/Os will be reduced.

This is known affectionately as Multi-key Frogger (MKF) internally.

Key Positioning Example

CREATE INDEX X1 ON EMPLOYEE
(LASTNAME, WORKDEPT)
:
SELECT * FROM EMPLOYEE
WHERE WORKDEPT BETWEEN 'A01' AND 'E01'
AND LASTNAME IN ('SMITH', 'JONES', 'PETERSON')



LASTNAME	WORKDEPT
Adamson	B01
Anderson	B01
Anderson	G01
Cain	A01
Caine	G01
Doe	E01
Jones	A01
Jones	C01
Jones	D01
Milligan	A01
Peterson	C01
Peterson	F01
Smith	B01
Smith	C01
Smith	D01
Smith	F01
Wulf	A01

Picture to illustrate key positioning

Can think of processing a set of ranges, by "positioning" into the first "key" of the range and reading all the "keys" in that range, then "positioning" into the first "key" of the next range, so on and so forth.

This process works because we have a radix index with contiguious, primary keys that represent the local selection of SQL request.

Key Selection

Selection criteria are applied to the key(s) of the index before the table page is retrieved.

- Advantages:
  - The table is only accessed for rows that satisfy the key selection criteria
- Potential disadvantages:
  - The entire index is read and the key selection criteria is applied to each key entry
  - A random I/O is performed against the table for each key selected from the index
  - Can perform poorly when a large number of rows are selected
- Used when:
  - Less than ~20% of the keys are selected
  - Ordering, grouping, or join operation requires the use of an index
  - May be used in combination with key positioning

- Key selection is a scan of the entire radix index, testing keys for the selection criteria
- Can be effective when the index is small and can be scanned quickly

Key Selection Example

```
CREATE INDEX X1 ON EMPLOYEE
   (LASTNAME, WORKDEPT)
   :
SELECT * FROM EMPLOYEE
   WHERE WORKDEPT BETWEEN 'A01' AND 'E01'
```

SQL4008 Access path X1 used for file 1.

Key row positioning for local selection (on workdept) cannot be done with index X1 since the column workdept is not the primary key
Index X1 is scanned and workdept is tested
Once a key is found that meets the criteria, the corresponding row in the table is read

Key Selection Example

```
CREATE INDEX X1 ON EMPLOYEE
     (LASTNAME, WORKDEPT)
     :
SELECT * FROM EMPLOYEE
     WHERE WORKDEPT BETWEEN 'A01' AND 'E01'
```

Think of scanning the entire index...

testing WORKDEPT for A01 - E01

LASTNAME	WORKDEPT	
Adamson	B01	←
Anderson	B01	←
Anderson	G01	
Cain	A01	←
Caine	G01	
Doe	E01	←
Jones	A01	←
Jones	C01	←
Jones	D01	←
Milligan	A01	←
Peterson	C01	←
Peterson	F01	
Smith	B01	←
Smith	C01	←
Smith	D01	←
Smith	F01	
Wulf	A01	←

...

- Picture of key selection illustrating why key row positioning (on workdept) cannot be done with index X1 since the column workdept is not the primary key
- Index X1 is scanned and workdept is tested
- Once a key is found that meets the criteria, the corresponding row in the table is read

Temporary Index Creation

Create a temporary index as required to implement the query.

- Advantages:
  - Allows for the most efficient index to be chosen to implement the selection criteria, joining, grouping and/or ordering
  - Temporary index may only contain entries for rows that match the selection
  - Uses a larger index page size
  - Index can be created in parallel
- Potential disadvantages:
  - Time and overhead of creating the temporary index
  - Index cannot be shared for any other queries over that same table
  - If host variables are used for selection, then the index is not reusable
  - Not created when query contains only selection
- Used when:
  - The access method chosen for selection, joining, grouping or ordering requires an index and that index does not exist, or the query interface attributes prevent the use of temporary objects (hashing, sorting, etc.)
  - Optimizer thinks the query is long running and using a sparse temporary index is worth the cost of creating the temporary index (ex. nested loop join with high fanout)

CPU parallelism

- A sparse index is one that contains selection such that only a subset of the keys are in the new index.
- The only disadvantage about using a 4-byte index over 3-byte index is that it will take more storage. The advantage is that this is the only method that will be enhanced in the future at the SLIC layer of the operating system.
- The XPF layer must continue to support and enhance both methods.
- The only way to migrate a 3-byte index to a 4-byte index is through a new creation (CHGLF recreates the access path).
- May be forced to build a temp if the types of join fields are different. For example, with following query, even though we have an index on y.zoned it would be forced to build an index over y. the *tmp index would be built of integers (the conversion from zoned to int will take place on the build of the index). thus we can frog into y useing the ints from x. SELECT * FROM MURAS/T1 X INNER JOIN MURAS/T1 Y ON X.int = Y.ZONED

Temporary Index Creation Example

```
SELECT * FROM EMPLOYEE E, DEPENDENTS D
   WHERE E.EMPNO = D.EMPNO AND
      WORKDEPT BETWEEN 'A01' AND 'E01'
```

SQL4010 Arrival sequence access for file 1.
SQL4007 Query implementation for join position 2 file 2.
SQL4009 Access path created for file 2.
SQL4014 1 join field pair(s) are used for this join position.
SQL4015 From-field 1.EMPNO, to-field 2.EMPNO, join operator EQ, join predicate 1.

- A sparse index is one that contains selection such that only a subset of the keys are in the new index.
- The only disadvantage about using a 4-byte index over 3-byte index is that it will take more storage. The advantage is that this is the only method that will be enhanced in the future at the SLIC layer of the operating system.
- The XPF layer must continue to support and enhance both methods.
- The only way to migrate a 3-byte index to a 4-byte index is through a new creation (CHGLF recreates the access path).

Index-Only Access

All of the data needed to process a table can be extracted from the index key fields.

- Advantages:
  - Can be used in conjunction with any of the other index access methods
  - Allows for larger I/O operations against the index for parallel pre-load by eliminating any storage set aside for the table
- Potential disadvantages:
  - May perform poorly when a small result set of rows are probed for join secondary dials
- Used when:
  - All columns used in the query exist as key fields within the index

- This access method was previously only available when DB2 Symmetric Multiprocessing for OS/400 was installed on your system.
- Starting in V4R1 this is part of the base operating system.
- This is available through an enabling PTF for V3R7 (SF36173).
- If key fields are null capable or varchar, KOA is not allowed.

Index-Only Access Example

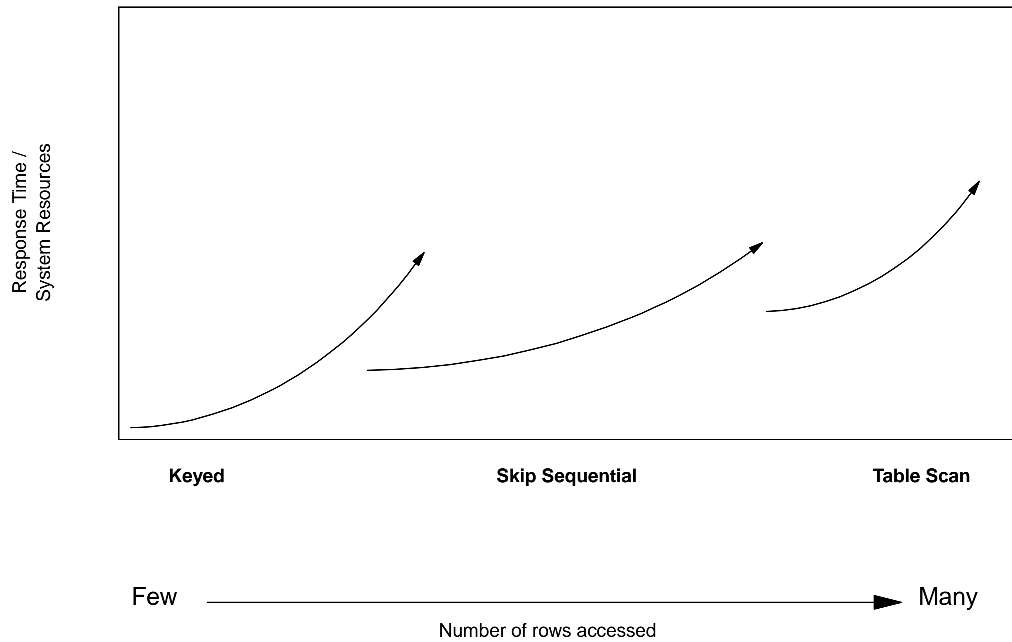
```
CREATE INDEX X1 ON EMPLOYEE
   (LASTNAME, WORKDEPT, FIRSTNAME)
   :
SELECT FIRSTNAME FROM EMPLOYEE
   WHERE WORKDEPT BETWEEN 'A01' AND 'E01'
```

SQL4008 Access path X1 used for file 1.

SQL4026 Index only access used on file 1.

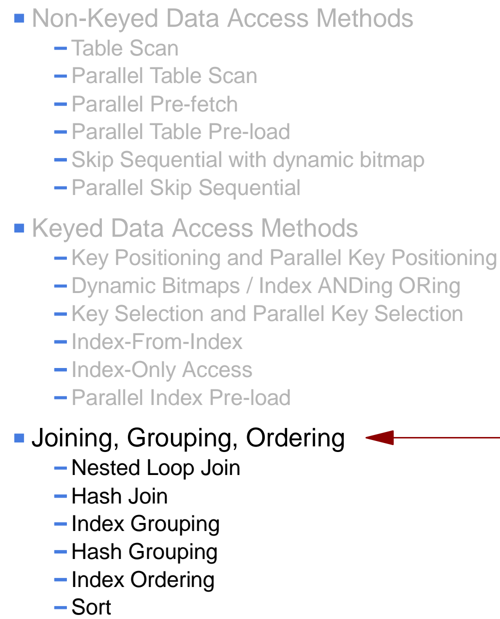
- This access method was previously only available when DB2 Symmetric Multiprocessing for OS/400 was installed on your system.
- Starting in V4R1 this is part of the base operating system.
- This is available through an enabling PTF for V3R7 (SF36173).

Data Access Methods



- The third major method of access is established (keyed or index access)

Implementation Methods Overview



- Joining, Grouping, Ordering
  - Nested Loop Join
  - Hash Join
  - Index Grouping
  - Hash Grouping
  - Index Ordering
  - Sort

- Covering the join, grouping and ordering methods

Joins

Common Terms

Term	Meaning
Join Position	Position in which this file is being joined.
Join Dial	Same as Join Position.
Join Order	The order of all of the files used to process the join. (Dial1 --> Dial2 --> Dial3 --> Dial4)
Average Duplicates	Average number of records for each distinct value. Statistic derived from an index.
Dial	Synonymous with the odometer on a vehicle. For each record of dial 1, you must spin through all of the records of dial 2.
Join Fanout	The number of join combinations that can be expected for each join value.

- Terms used when discussing and understanding join optimization
- Avg Dups are derived from an index by using the "number of unique values" information maintained in the LF.
- Keys 1-4 can be used from a radix index
- Only single key EVIs can be used

Join Support for SQL

- For inner join, optimizer not biased toward using specified join order
- For left outer and exception join, tables are joined from left to right
  - INNER JOIN tables can be reordered
- Multiple join types supported for a single query
- Join implementation methods
  - Nested Loop
  - Hash

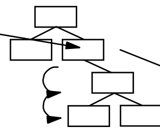
- Summary
- In V4R4 optimizer will chose join order even when join syntax is coded
  - INI setting is available to force join orde
- **Last chart for Day 1...???**
- **Or, continue thru nested loop join**

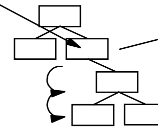
Nested Loop Joins

- Each row selected from the primary file is joined to each secondary file using a key value built over the join-to fields.
  - The join spins like an odometer on a car (from right to left).
  - After a file has been completely cycled, then it backs up to the previous dial and gets the next join value.
  - The join is performed again spinning through the next secondary file in the odometer (N-1).
- The join is not complete until all the rows of the primary file have been processed.

- Nested loop join description/explanation
- Background that aids in understanding the performance issues when using nested-loop join

Nested Loop Joins






```
SELECT * FROM TABLE_1, TABLE_2, TABLE_3
WHERE FKEY1 = PKEY3
AND FKEY2 = PKEY3
```

- Illustration of nested loop join, showing the objects and work involved
- Highlight the fact that nested loop requires a radix index and involves 2 I/Os
- More tables (or dials) the more downstream I/O that occurs
  - Lots of fanout = lots of I/Os

Nested Loop Joins

Creating a Temporary Index for the Join Criteria

- If an index over the join fields of the secondary file(s) does not exist, one is created.
- Advantages:
  - Since local selection is performed ahead of the join (during index creation) the temporary index generally is smaller so there are less index pages to be faulted in.
- Disadvantages:
  - Creating a temporary index is very CPU intensive and is not suitable for OLTP.
  - If index is built using host variable selection, then the query is not reusable.

Optimizer may create a temporary index when a permanent index exists. If the join fan out is high (or the cost of the NL join is very high) the optimizer may chose to create a sparse, temporary index, trying to make the join as efficient as possible.

Hashing Algorithm

Joining

A hashing algorithm is used to correlate data with a common value together for grouping and/or join queries.

- Advantages:
  - Allows for the exploitation of CPU parallelism for the creation of the hash table
  - Reduces the random I/O to the table generally associated with longer running queries using an index
  - Generally faster than creating a temporary index to perform the specified operation
- Potential disadvantages:
  - May perform poorly when processing a small subset of the rows that will be used as input into the hashing algorithm
  - A temporary copy of the data is required to process the hash
- Used when:
  - The value *OPTIMIZE is specified or *YES if a temporary file is required, on the ALWCPYDTA parameter
  - Grouping and/or a join processing is specified in the query

CPU parallelism

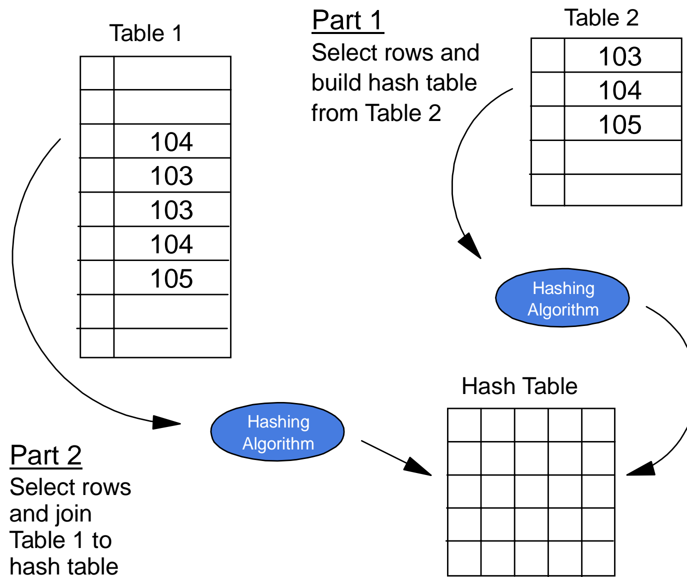
- Description of hashing algorithm or technique in preparation of explaining hash join
- Influenced by ALWCPYDTA and ALL I/O attributes
- Goal is build a (small) hash table that fits in memory so that accesses to the hash table are fast and involve no physical I/O
- Don't get caught up in explaining the hashing concept or algorithm
- Application developers cannot get access to the hash table
  - Optimizer's implementation method

Hash Joins

- The hash join uses a hash table to correlate all of the distinct join values into common buckets (hash points) and then using the buckets to find all of the join combinations.
- Query rewritten to take advantage of SMP
- This has the following improvements over a Nested Loop Join:
  - An index is no longer required to find the join matches.
  - No longer have to iterate through all of the files for each of the join possibilities (i.e. odometer processing).
- Equal join predicates only
- No specific information on hash joins in DB monitor
  - Query to select records and build hash table(s) do show up

- Hash join description and/or attributes
- SQL request is rewritten into multiple "steps" and implemented as multiple queries
  - Query(s) to select rows and build the hash table(s)
  - Query to select rows from dial 1 and join into the hash tables
- Key points
  - No radix index required for join
  - SMP can be used to build the hash table and access rows in the 1st dial
- Only the data required for the SQL request is put in the hash table (select, grouping, ordering)

Hash Join



Like join values are grouped together by their hash value and all the appropriate data is stored.
Collisions are handled by linked list.

Example:

SELECT *
FROM TABLE1 A, TABLE2 B
WHERE A.PARTKEY in ('103', '104', '105')
and A.PARTKEY = B.PARTKEY

- Illustration of hash join implementation steps
- These steps DO NOT correspond to the HASH STEPS in the joblog messages
- Optimizer estimates the size of the hash table, if hash table can fit in job's share of memory pool, then hashing used.
- Hash join tends to join large table to small table (ex. Detail to Master).
- Nested loop tends to join small table to large table (ex. Master to Detail).
- Table 1 is joined to hash table, not to Table 2. Table 2 is only used to select rows and populate hash table.
- For a given input value, the hashing algorithm will always output the same hash value.

Hash Join Example

```
SELECT * FROM EMPLOYEE, DEPARTMENT
   WHERE EMPLOYEE.WORKDEPT = DEPARTMENT.DEPTNO
      AND EMPLOYEE.HIREDATE BETWEEN '1995-01-30'  AND
         '1996-01-30'
      AND DEPARTMENT.DEPTNO IN ('A00', 'D01', 'D21', 'E11')
   OPTIMIZE FOR ALL ROWS
```

SQL402A Hashing algorithm used to process join.
SQL402B File EMPLOYEE used in hash join step 1.
SQL402B File DEPARTMENT used in hash join step 2.

- Rows from DEPARTMENT are selected and hashed using DEPTNO into the hash table first, then rows are selected from EMPLOYEE, WORKDEPT is hashed and used to probe into the hash table to perform the join

Join Optimization

- The main optimization rule of thumb for a join query is the reordering of the files.
  - This minimizes the join fanout and that in turn minimizes I/Os.
- Reordering of files is allowed only on inner joins. Left Outer or Exception joins cannot be reordered.
- The DB2 UDB for AS/400 optimizer uses a greedy join algorithm to determine the most efficient table order.

- Starting the JOIN OPTIMIZATION discussion
- Key concept: reduce or eliminate I/O
- Fanout = I/O
- Join order influences fanout
- Greedy join = check all combinations, not satisfied with the first "better" combination (better than what's coded for join order)

Greedy Join Algorithm

- Determine best access for each individual file, ignoring the join for a moment (keyed access, data space scan, etc.)
- For each join combination determine the join cost
  - For a 4 file join, the join combinations would be:

  1-2, 2-1, 1-3, 3-1, 1-4, 4-1, 2-3, 3-2, 2-4, 4-2, 3-4, 4-3
- The combination with lowest cost determines the primary and first secondary files

  2 3 x x
- For each remaining file, determine cost of joining to previous files
  - File with lowest cost becomes next secondary file

  2 3 1 x
- Repeat join cost calculation until complete join order has been determined

  2 3 1 4

- Average duplicates stats help the optimizer determine the join order.
- If no radix index exists over the join columns, then use default avg dup value
- Default avg dups = 3, resulting in a 1-3 join fanout.
- The first join pair is the most important.
- Radix indexes over the join columns is the single best way to influence the optimizer's join order calculation

Implementation Methods Overview

- Non-Keyed Data Access Methods
  - Table Scan
  - Parallel Table Scan
  - Parallel Pre-fetch
  - Parallel Table Pre-load
  - Skip Sequential with dynamic bitmap
  - Parallel Skip Sequential
- Keyed Data Access Methods
  - Key Positioning and Parallel Key Positioning
  - Dynamic Bitmaps / Index ANDing ORing
  - Key Selection and Parallel Key Selection
  - Index-From-Index
  - Index-Only Access
  - Parallel Index Pre-load
- Joining, Grouping, Ordering
  - Nested Loop Join
  - Hash Join
  - Index Grouping
  - Hash Grouping
  - Index Ordering
  - Sort

- Covering the join, grouping and ordering methods

Group-By Optimization

- The query optimizer chooses between index grouping and hash grouping
- Indexes are used for grouping statistics (number of groups)
  - Keys over grouping column(s)
  - Average duplicates statistics
  - Number of hash points (hash table size)
    - 16K
    - 64K
- Query attributes affect which method is used
  - Index Group by
    - First I/O
    - ALWCPYDTA(*NO), ALWCPYDTA(*YES)
  - Hash Group by
    - All I/O
    - ALWCPYDTA(*OPTIMIZE)

Index Group-By

The index provides inherent grouping of the data

- Advantages:
  - Record selection and grouping can be performed with the same index
- Potential disadvantages:
  - A random I/O is performed against the table for each key selected from the index
  - Can perform poorly when a large number of rows are selected
  - Key fields and key order must match the grouping fields
  - No parallelism
  - May effect join order
- Used when:
  - Selection and grouping can be performed with the same index
  - Large number of groups with few records per group
  - Grouping and ordering over same fields, in the same order
  - Grouping operation requires the use of an index
    - ALWCPYDTA *NO or *YES is specified (live data)
    - Temporary index may be created over temporary result

Index Group-by Example

```
CREATE INDEX X1 ON EMPLOYEE
   (WORKDEPT, FIRSTNAME)
   :
SELECT WORKDEPT, FIRSTNAME, FROM EMPLOYEE
   WHERE WORKDEPT BETWEEN 'A01' AND 'E01'
   GROUP BY WORKDEPT, FIRSTNAME
```

SQL4008 Access path X1 used for file 1.
SQL4026 Index only access used on file number 1.
SQL4011 Key row positioning used on file 1.

Hashing Algorithm

Grouping

A hashing algorithm is used to correlate data with a common value together for grouping and/or join queries.

- Advantages:
  - Allows for the exploitation of SMP for the creation of the hash table
  - Reduces the random I/O to the table generally associated with longer running queries using an index
  - Generally faster than creating a temporary index to perform the specified operation
- Potential disadvantages:
  - May perform poorly when processing a small subset of the rows that will be used as input into the hashing algorithm
  - A temporary copy of the data is required to process the hash
- Used when:
  - The value *OPTIMIZE is specified or *YES if a temporary file is required, on the ALWCPYDTA parameter
  - Grouping and/or a join processing is specified in the query

CPU parallelism

- Previously this was only available when DB2 Symmetric Multiprocessing was installed on your system. However, the parallel aspects of these algorithms still are.

Hashing Algorithm

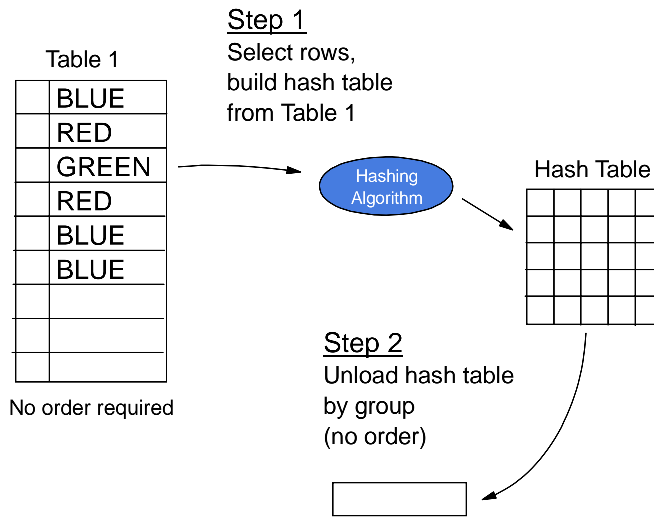
Grouping

A hashing algorithm is used to correlate data with a common value together for grouping and/or join queries.

- Two sizes
  - 16K and 64K hash points
  - Quicker startup and more efficient
  - Determined by statistics such as average duplicates

- Previously this was only available when DB2 Symmetric Multiprocessing was installed on your system. However, the parallel aspects of these algorithms still are.
- V4R4: parallel hash group by redesigned
- Much more scalable
- 50 million - 80 million groups
- Prior to V4R4: 1 million groups

Hash Group-By



Like values are grouped together by their hash value and the function applied. Collisions are handled by linked list.

Example:

SELECT COLOR, SUM(QUANTITY)
FROM TABLE1
WHERE COLOR in ('RED', 'GREEN', 'BLUE')
GROUP BY COLOR

BLUE hashes to 126, sum
RED hashes to 47, sum
GREEN hashes to 227, sum
RED hashes to 47, sum
BLUE hashes to 126, sum
BLUE hashed to 126, sum
etc.

Implementation Methods Overview

- Non-Keyed Data Access Methods
  - Table Scan
  - Parallel Table Scan
  - Parallel Pre-fetch
  - Parallel Table Pre-load
  - Skip Sequential with dynamic bitmap
  - Parallel Skip Sequential
- Keyed Data Access Methods
  - Key Positioning and Parallel Key Positioning
  - Dynamic Bitmaps / Index ANDing ORing
  - Key Selection and Parallel Key Selection
  - Index-From-Index
  - Index-Only Access
  - Parallel Index Pre-load
- Joining, Grouping, Ordering
  - Nested Loop Join
  - Hash Join
  - Index Grouping
  - Hash Grouping
  - Index Ordering
  - Sort

- Covering the join, grouping and ordering methods

Order-By Optimization

- The query optimizer chooses between index ordering and sort
- Optimizer costs the use of each method and picks the fastest
- Query attributes affect which method is used
  - Index Order by
    - First I/O
    - ALWCPYDTA(*NO), ALWCPYDTA(*YES)
  - Sort Order by
    - All I/O
    - ALWCPYDTA(*OPTIMIZE)

Index Ordering

The index provides inherent ordering of the data

- Advantages:
  - Record selection and ordering can be performed with the same index
- Potential disadvantages:
  - A random I/O is performed against the table for each key selected from the index
  - Can perform poorly when a large number of rows are selected
  - Key fields and key order must match the ordering fields
  - No parallelism
  - May effect join order
- Used when:
  - Selection and ordering can be performed with the same index
  - Grouping and ordering over same fields, in the same order
  - Ordering operation requires the use of an index
    - ALWCPYDTA *NO or *YES is specified (live data)
    - Temporary index may be created over temporary result

Index Ordering Example

```
CREATE INDEX X1 ON EMPLOYEE
   (WORKDEPT, LASTNAME)
   :
SELECT * FROM EMPLOYEE
   WHERE WORKDEPT = 'A01'
   ORDER BY LASTNAME
```

SQL4008 Access path X1 used for file 1.

SQL4011 Key row positioning used on file 1.

Sort Routine

A sort routine is used to order or perform distinct processing on the results of a query.

- Advantages:
  - Allows for the most efficient index and/or join order to be chosen to implement the query
- Potential disadvantages:
  - A temporary copy of the data will be required to process the sort
- Used when:
  - The value *OPTIMIZE is specified or *YES if a temporary file is required, on the ALWCPYDTA parameter
  - Ordering or distinct processing is specified in the query

- By specifying ALWCPYDTA *OPTIMIZE for your queries you are allowing the optimizer to make a temporary copy of the data when the query is already going to be read-only.

Sort Routine Example

```
CREATE INDEX X1 ON EMPLOYEE
   (LASTNAME, WORKDEPT)
   :
SELECT * FROM EMPLOYEE
   WHERE WORKDEPT BETWEEN 'A01' AND 'E01'
   ORDER BY FIRSTNAME
```

SQL4002 Reusable ODP sort used.

SQL4008 Access path X1 used for file 1.

- By specifying ALWCPYDTA *OPTIMIZE for your queries you are allowing the optimizer to make a temporary copy of the data when the query is already going to be read-only.

Temporary Results

A temporary result may be required to complete the query

- DISTINCT and an index is not used
- UNION or UNION ALL
- ORDER BY columns from more than one table
- GROUP BY columns from more than one table
- Grouping and ordering columns are different
- Key length > 2000 bytes
- Complex view or logical file being queried
- Sort done for performance
- UPDATEs with subSELECTs or sub queries

- By specifying ALWCPYDTA *OPTIMIZE for your queries you are allowing the optimizer to make a temporary copy of the data when the query is already going to be read-only.

Art of SQL Optimization

Indexing Strategies

- You must create some indexes
  - Statistics
  - Implementation
- Proactive
  - Create indexes over primary, foreign key columns and dependent columns
  - Create indexes for selection and joining
  - Create indexes for selection, grouping and ordering
- Reactive
  - Create indexes based on optimizer feedback
  - Create indexes based on optimization, implementation, system resources and performance

- PF created with keyed access path can be accessed by the system using sequential access or keyed access via the keyed access path
- ANZDBF should be run first, provides input to ANZDBFKEY

Indexing Strategies - Basic Approach

- In general: equal selection columns first, then join columns -or- group-by and order-by columns
- May have to play around with key order based on the queries, the data and selectivity of the columns
- Create indexes that give the optimizer the most information/stats
  - Create index as unique when the data is unique
  - Variable length and null capable columns provide *limited* statistics
- Consider dynamic bitmaps and index ANDing/ORing
  - Simple indexes can be combined together for selection
- Consider EVIs for stats and dynamic bitmaps
  - Single key, low number of unique values
  - Fact table foreign key
  - Over temporary results table to provide stats
- NEED TO ITERATE!

- EVI created over single key with few distinct values results in a smaller symbol table and smaller vector. The use of this EVI is quicker and more efficient.

Indexing Strategy - Join Tips

- At a minimum, make sure there are radix indexes built over all the join columns
- May have indexes built over both join columns and selection columns, this allows for *multi-key* joins
- Create multiple, single key indexes over selection columns to take advantage of dynamic bitmaps, then create radix index over join columns
- Create multiple, single key EVIs over foreign key columns to take advantage of dynamic bitmaps
- Indexes are used to determine the average number of duplicate values *(stats)* for the join columns

- EVIs are not used for joins in V4R3 and are never used for nested loop joins.
- Avg dup stats available for first 4 contiguous key columns only.
- Avg. dup stats from single key EVI

Indexing Strategies - The Perfect Index

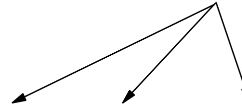
- A "perfect" index is a radix index that is permanent and can provide:
  - Good, useful statistics to the optimizer
    - Index contains appropriate selection, joining, grouping, ordering fields
    - Applicable key fields are contiguous
    - Equal predicate fields first, one non-equal predicate field last
  - Multiple implementation methods
    - Index ANDing / ORing with dynamic bitmaps
    - Single key and multi-key row positioning
    - Index scan
    - Index only access
    - Nested loop join
    - Index grouping & ordering
  - Multi-key index that provides very narrow range of values
    - Think in terms of lower and upper bounds

Art

"Perfect" radix indexes...

- Statistics
- Multi key selection
- Index only access
- Nested loop join
- Index grouping
- Index ordering

Selection, grouping, ordering



CREATE INDEX IX1 on TABLE1 (YEAR, MONTH, CUSTOMER, ORDERNO)

Joining

CREATE INDEX IX2 on TABLE2 (ORDERNO, QUANTITY, SALES_AMOUNT)

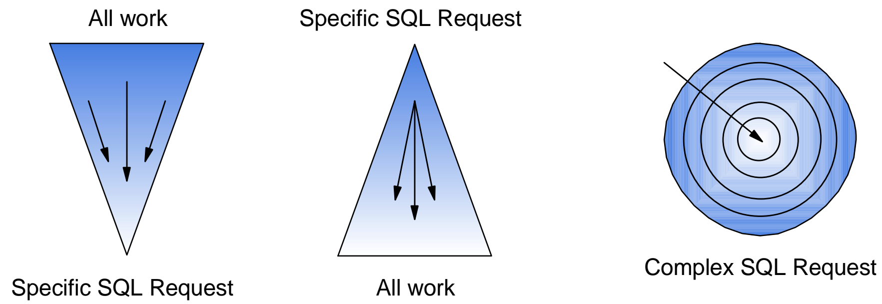
```
SELECT A.YEAR, A.MONTH, A.CUSTOMER, SUM(B.QUANTITY),
SUM(B.SALES_AMOUNT)
    FROM TABLE1 A, TABLE2 B
    WHERE A.YEAR = 2000 and A.MONTH in (10, 11, 12) and A.CUSTOMER = 'SMITH'
    and A.ORDERNO = B.ORDERNO
    GROUP BY A.YEAR, A.MONTH, A.CUSTOMER
    ORDER BY A.YEAR, A.MONTH, A.CUSTOMER
```

Tools and Methodologies

Overview

- How do I know what's going on with my queries?
- How can I tell what the optimizer is doing?
- Answer: Tools and analysis
  - Query optimizer debug messages
  - Print SQL Information (PRTSQLINF)
  - Database Monitor Statistics
    - Detailed Monitor (STRDBMON)
    - Summary (Memory-based) Monitor (Operations Navigator)
  - Visual Explain
  - Change Query Attributes (CHGQRYA)
  - QAQQINI file attributes

Discovery Methodologies



- Proactive and reactive approaches
- All methodologies are iterative in nature

Debug Messages

- Informational messages written to the joblog about the implementation of a query
- Describes query implementation method
  - Indexes
  - Join order
  - Access plans
  - ODPs (Open Data Paths)
- Messages explain what happened during query optimization
  - Why index was or was not used
  - Why a temporary index result was required
  - Index advised by the optimizer
- STRDBG UPDPROD(*YES) & STRSRVJOB and STRDBG for batch jobs
- ODBC & JDBC Driver Exit program
- MESSAGES_DEBUG = *YES in QAQQINI file

- Debug mode does not slow down the query
- Joblog size (and growth) should be considered

Print SQL Information

- OS/400 command that lists SQL information contained in a program, SQL package, or service program.

  PRTSQLINF OBJ(MY_PGM) OBJTYPE(*PGM)

  PRTSQLINF OBJ(MY_PKG) OBJTYPE(*SQLPKG)

- Creates a spooled file that contains:
  - SQL statements
  - Type of access plan used by each statement
  - Command (CRTSQLxxx) and parameters used to invoke the SQL precompiler
- iSeries version of SQL EXPLAIN utility
- Output similar to debug messages

- Whenever QQ goes through QQQPDCMP (LCP, hash join, subquery materialization, view push down, nested joins) they do not save the individual implementations for these recursive QDTs. The only thing they save is the plan that dictates how they will split up the original query into these multiple steps. This is why you see the hash join steps through PRTSQLINF and not how we accessed the dataspace.
- Each time they run one of those steps with their mini-QDTs they will in essence be running a seperate query with its access plan and optimization all over again. So each step for hash join gets re-validated each time the query is run.

Database Monitors

- Integrated tools used to gather database performance related statistics for SQL-based requests
- Monitor data dumped into table(s) where it can be queried to help identify and tune performance problem areas
  - Detailed monitor writes all of the information out to a single table as it's collected
    - Interface: STRDBMON & Operations Navigator
  - Summary monitor collects similar information at a summarized level in memory and then dumps the data into multiple tables
    - Interface: Operations Navigator & APIs

Database Monitors

- Provides all information from STRDBG or PRTSQLINF plus additional details:
  - SQL statement text
  - Start and end timestamp
  - Estimated processing time
  - Total rows in table queried
  - Number of rows selected
  - Estimated number of rows selected
  - Key fields for advised index...

- Once the data is collected, analysis is performed by running queries against the tables
  - EXAMPLES:
    Which statements are the most time consuming?
    Which statements are not having ODPs reused?
    Which statements are causing temporary index builds?

Predictive Query Governor

- Allows user to stop long-running queries before they even start. The query time limit is set on a per-job basis via CHGQRYA CL command
  - Can also be set via the system value QQRYTIMLMT or a QAQQINI option
- Query time limit is checked against estimated elapsed query time before initiating a query
  - Cost based optimization = costs and access plan are determined prior to execution
- An inquiry message is dispalyed to the end user showing the predicted runtime and asking if the query should be cancelled
- Debug messages will be written to the joblog if the query is canceled.
- Time limit of zero is used to optimize performance on queries without having to run through several iterations.

- Allows a 'Query From Hell' to be stopped prior from being started.
- The SQL return code for QFHs is -666.
- Use the governor with a QRYTIMLMT(0) to analyze query implementation without any execution.
- Will discuss in detail all of the query attributes in 102.

Query Performance Tuner - QAQQINI

- Provides central point of control for all attributes, options, and knobs that can impact query opitmization
  - Table design allows attributes to be set dynamically with just database updates or insert/delete

```
UPDATE mylib/QAQQINI SET QQVAL='600'
    WHERE QQPARM='QUERY_TIME_LIMIT';

INSERT mylib/QAQQINI
    VALUES('MESSAGES_DEBUG','*YES','Activated - 4pm');
```

  - One row per attribute/parm and 3 character columns
    - QQPARM - the attribute/option name
    - QQVAL - value of the attribute/option
    - QQTEXT - optional description of the attribute or it values

QQPARM	QQVAL	QQTEXT
MESSAGES_DEBUG	*YES	Debug Set - 11pm
QUERY_TIME_LIMIT	600	New time limit - set 7/25
PARALLEL_DEGREE	*DEFAULT	
FORCE_JOIN_ORDER	*DEFAULT	
...	...	

- From: Diane Hawkins/Rochester/IBM@IBMUS
- All of the external options are available to the customer; However, the options will only be populated to the QAQQINI file shipped with the system on a release boundary because they are MRI. Following are what is shipped in the QAQQINI file for V4R4, V4R5 and V5R1.

```
V4R4
VALUES('APPLY_REMOTE                ','*DEFAULT');
VALUES('QUERY_TIME_LIMIT            ','*DEFAULT');
VALUES('PARALLEL_DEGREE             ','*DEFAULT');
VALUES('ASYNC_JOB_USAGE             ','*DEFAULT');
VALUES('MESSAGES_DEBUG              ','*DEFAULT');
VALUES('PARAMETER_MARKER_CONVERSION','*DEFAULT');
VALUES('UDF_TIME_OUT                ','*DEFAULT');
VALUES('OPTIMIZE_STATISTIC_LIMITATION', '*DEFAULT');

V4R5
VALUES('APPLY_REMOTE                ','*DEFAULT');
VALUES('QUERY_TIME_LIMIT            ','*DEFAULT');
VALUES('PARALLEL_DEGREE             ','*DEFAULT');
VALUES('ASYNC_JOB_USAGE             ','*DEFAULT');
VALUES('MESSAGES_DEBUG              ','*DEFAULT');
VALUES('PARAMETER_MARKER_CONVERSION','*DEFAULT');
VALUES('UDF_TIME_OUT                ','*DEFAULT');
VALUES('OPTIMIZE_STATISTIC_LIMITATION', '*DEFAULT');
VALUES('FORCE_JOIN_ORDER              ', '*DEFAULT');

V5R1
VALUES('APPLY_REMOTE                ','*DEFAULT');
VALUES('QUERY TIME LIMIT            ','*DEFAULT');
```

Visual Explain

- Visualization of the query access plan
  - Details and attributes of the query plan, execution, and database objects involved
  - V5R1 includes auto-highlighting of icons
- Visual Explain can be used in one of two ways
  - Interactively with Ops Navigator SQL Script window
  - Reactively based on previously collectly database monitor data (detailed monitor)
- Requires V4R5 or higher of OS/400 and IBM Client Access Operations Navigator

Tuning Tools Comparison Table

PRTSQLINF	STRDBG/CHGQRYA QAQQINI	STRDBMON	Memory -based Monitor
Available without running query (after access plan has been created)	Only available when the query is run	Only available when the query is run	Only available when the query is run
Displayed for all queries in SQL pgm or pkg, whether executed or not	Displayed only for those queries which are executed	Displayed only for those queries which are executed	Displayed only for those queries which are executed
Information on host variable implementation	Limited information on the implementation of host variables	All information on host variables, implementation, and values	All information on host variables, implementation and values
Available only to SQL users with pgms, packages, or service pgms	Available to all query users (OPNQRYF, SQL, QUERY/400)	Available to all query users (OPNQRYF, SQL, QUERY/400)	Available only to SQL interfaces
Messages printed to spool file	Messages displayed in job log	Performance records written to database file	Performance information collected in memory and then written to database file
Easier to tie messages to query with subqueries or unions	Difficult to tie messages to query with subqueries or unions	Uniquely identifies every query	Repeated query requests are summarized

- Mention Query INI file and how other CHGQRYA options can impact performance
- Memory based monitor only keeps the 300x records/info for the most expensive execution of a statement.

Additional Resources

- DB2 UDB for iSeries home page: www.iseries.ibm.com/db2
- iSeries SQL Performance Workshop (Course #S6140)
  www.iseries.ibm.com/db2/db2educ_m.htm
- Online DB2 UDB publications www.iseries.ibm.com/db2/books.htm
  - Database Performance & Query Optimization
- SQL Interface FAQs:
  - CLI - www.iseries.ibm.com/db2/clifaq.htm
  - JDBC
    - Toolbox: www.iseries.ibm.com/toolbox/faqjdbc.htm
    - Native: www.iseries.ibm.com/developer/jdbc/index.html
- QAQQINI script builder: www.iseries.ibm.com/developer/bi/tuner.html
- DB2 UDB for iSeries Online Education
  www.iseries.ibm.com/developer/education/ibo/view.html?biz
- Third-pary performance tools:
  - Centerfield Technology (www.centerfieldtechnology.com)