

Db2 Top Reasons for Poor SQL, Program, and Application Performance

Presented by Tony Andrews, Themis Inc.
tandrews@themisinc.com

Twitter:



Objectives

By the end of this presentation, developers should be able to:

- Understand some of the more common areas that may cause poor performance in a query, program, or application.
- Understand how data statistics can affect our program coding
- Understand data clustering and its affect on performance
- Understand Table Scans and what may cause them

My Experience Shows

- That the majority of performance problems among applications are caused by poorly coded programs or improperly coded SQL.
- That poor performing SQL and programs are responsible for the highest percentage of response-time issues.
- That most developers need educated in how to read and analyze the Db2 Explain output.
- That most developers do not understand the performance issues involved with SQL programming, where to start, or how to fix them.

My Experience Shows

- The largest performance gains are often obtained from tuning application code. The next performance gains are tweaking/adding indexes and Runstat statistics.
- That performance, no matter how good IT shops think it is, can always be better.
- That a little bit of training, shop coding standards, and program walkthroughs can totally change the culture of IT development departments, and minimize performance issues and incident reporting in production.

Developers Should Always Program with Performance in Mind

Programmers should always have two goals in mind when developing programs and applications for user groups.

- To get the correct results for requested data
- To get the results back as quickly as possible

So many times programmers today lose sight of the second goal. They either:

- Do not know what to do to get programs to run faster
- Blame other pieces of their environment (database, network, TCP/IP, workstation, operating system, etc.), or
- Think that the time spent processing the data was pretty good based on the amount of data. Set expectations in test and production!



Experience Shows

That standards, guidelines, and walkthroughs saves CPU costs
and incident reporting in IT shops.

*“There is always time for a 10 minute
walkthrough”*

What Do We Look Into for Performance Issues?

- **SQL / Program Code**
- **Statistics on Tables**
- **Physical Design**
- **System Tuning**



Improving SQL performance can be done in one of at least 4 ways. System tuning may be done to adjust the parameters under which the Db2 subsystem operates to effectively match the workload. Altering system parameters, tuning temporary space, and adjusting bufferpool sizes and thresholds are all examples of this type of tuning. An appropriately tuned system can affect an improvement in performance. Most of the time, however, other factors dominate a tuning scenario. The SQL itself must be written in a way that may be processed efficiently by the database. An appropriate level of statistics about the data must be gathered to tell the optimizer about the nature of the data being accessed. Lastly, the way the physical objects are defined must be aligned with the types of queries that are to be performed.

What Causes CPU in Db2?

- **Buffer Manager**
 - Retrieval of pages into the buffer
 - Managing Getpage requests
- **Data Manager (Stage 1)**
 - Break up raw data into columns and rows and filtering data
- **Relational Data Services (Stage 2)**
 - Reformatting, transforming, calculating values used in filtering. Typically 10% more CPU to evaluate than Stage 1.



CPU is consumed by many layers of software within Db2. The code responsible for storing, retrieving and filtering data for queries consumes the bulk of CPU related to application performance.

The Buffer Manager is responsible for managing the pages of data for various tablespaces and indexes inside Db2's bufferpools. It determines if the requested pages are already in the bufferpool and if not issues appropriate requests to retrieve the data from the underlying linear VSAM files that store data. Buffer Manager is not aware of columns and rows of data, only the pages and page numbers. The more getpage requests that are necessary to run an SQL statement, the more CPU will be consumed by Buffer Manager.

Data Manager (sometimes called "Stage 1") is responsible for translating the raw data stored on pages into columns and rows for a given query. As it is doing this, it can throw away some data that does not meet the criteria of the WHERE clause. Predicates that are written in a fairly straightforward way can usually be evaluated by Data Manager with relatively little expense.

Relational Data Services (RDS) is also referred to as "Stage 2". Complex predicates that require data transformation or computations will be evaluated by RDS rather than Data Manager. These "Stage 2" predicates are much more expensive to resolve than "Stage 1" predicates. Additionally, RDS cannot make effective use of indexes, so there is a risk of retaining data much longer that does not qualify for the query. It is obviously desirable to write queries in a way that may be evaluated at Stage 1 rather than by RDS.

These layers of code will be discussed in detail in a later chapter.

What Causes CPU in Db2?

- **Built-in Function Usage: Transformation of data for retrieval**
- **CASE statements (very expensive)**
- **Summarization for Column Functions**
- **Sorts**
- **User Defined Functions (UDFs)**
- **Stored Procedures**
- **Triggers**
- **Locking of data**

- **DEVELOPERS: Know what your programs are doing!**
Know how much data is typically processed!
Have counts for all actions!



Use of Built-in functions (BIFs) in a query may also involve substantial CPU to reformat or summarize data. In the case where BIFs are used in the WHERE clause, this will almost always result in Stage 2 processing for that predicate. Among the most powerful BIFs is the CASE statement which allows conditional logic to be done inside an SQL statement. The CASE statement may also be one of the most CPU intensive functions used in SQL. Care should be taken when using CASE to be sure that it is being used appropriately.

User Defined Functions (UDFs) are custom functions written by programmers to accomplish reformatting or summarization of data that cannot be done with Db2's BIFs. UDFs run in Workload Manager (WLM) address spaces and may involve significant overhead to call and manage. Stored procedures can incur similar overhead.

Any sorts use CPU. Stored procedures called during processing will use a percentage of the overall CPU.

Are there any triggers being executed on table processing?

How much locking is taking place? Can it be minimized? Can the program execute optimistic locking?

Reducing I/O in Db2

- **Appropriate use of indexes**
- **Appropriate physical clustering of table data**
- **Appropriate partitioning**
- **Table compression. More rows per page.
More data in buffer pool memory**
- **Bufferpool tuning**
- **Early elimination of data from consideration (i.e. get a good access path and no Stage 2 predicates)**



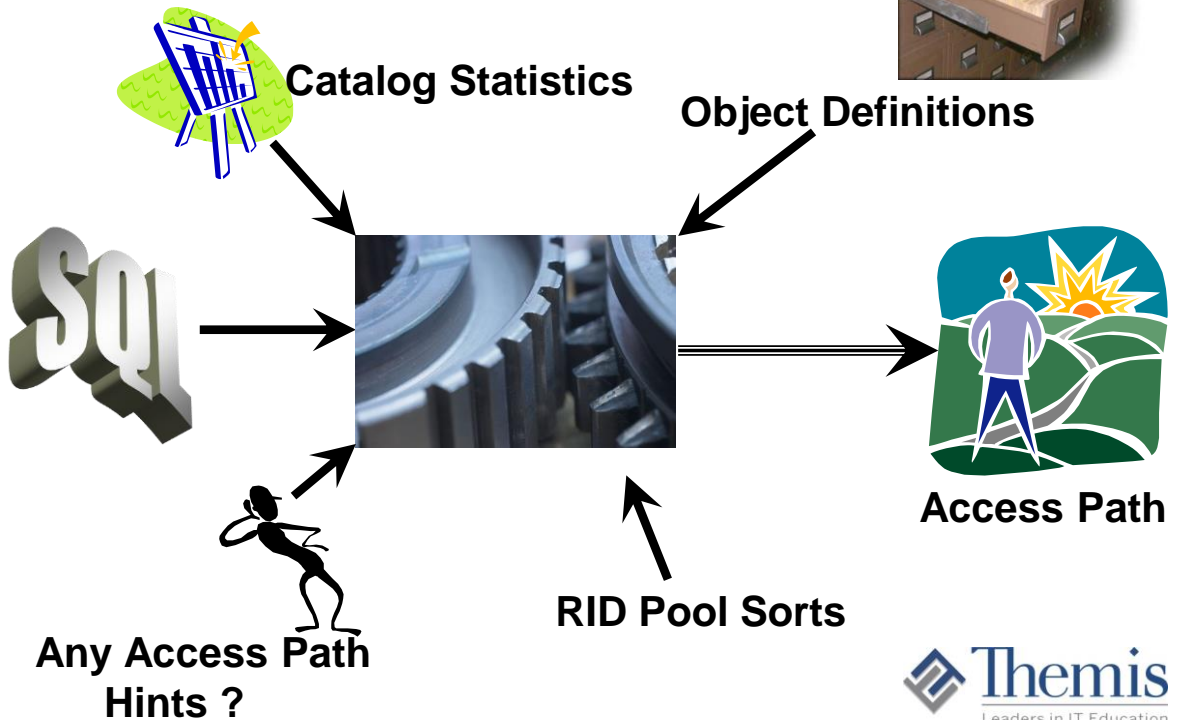
Reducing I/O within Db2 is one of the easiest ways to improve the performance of SQL within an application.

Creating appropriate indexes on columns or groups of columns that are commonly used to identify needed data can significantly reduce the I/O and CPU needed to retrieve a result.

It is also important that the Db2 system itself be configured optimally for the workload that must be supported. One important component of system tuning is the bufferpools. Bufferpools exist to reduce the amount of I/O needed by applications. Bufferpools should be configured to allow for as much reuse of cached data as possible. Objects may be grouped by sequential and random access patterns and the settings of the pools adjusted accordingly. In some cases a system tuning effort can reap significant rewards.

No amount of system tuning, however, can recover the resources wasted by a poor database design or poor access paths generated by the Db2 optimizer. This presentation focuses on optimization and tuning at the SQL level. In general, we want the optimizer to generate an access path that eliminates as much data from consideration as early as possible in the process.

The Db2 Optimizer



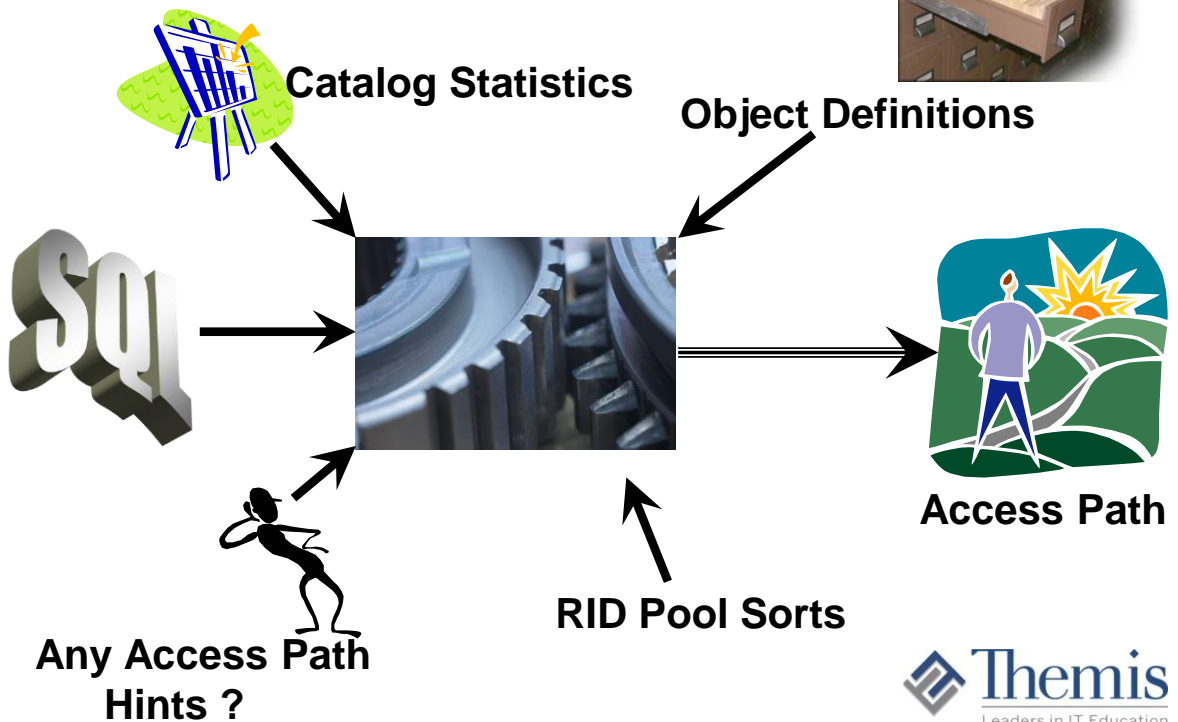
Through the Data Manipulation Language (DML) the user of a Db2 database supplies the “WHAT”; that is, the data that is needed from the database to satisfy the business requirements. Db2 then uses the information in the Db2 Catalog to resolve “WHERE” the data resides. The Db2 Optimizer is then responsible for determining the all important “HOW” to access the data most efficiently.

Ideally, the user of a relational database is not concerned with how the system accesses data. This is probably true for an end user of Db2, who writes SQL queries quickly for one-time or occasional use. It is less true for developers who write application pro-grams and transactions, some of which will be executed thou-sands of times a day. For these cases, some attention to Db2 access methods can significantly improve performance.

Db2’s access paths can be influenced in four ways:

- ◆ By rewriting a query in a more efficient form.
- ◆ By creating, altering, or dropping indexes.
- ◆ By updating the catalog statistics that Db2 uses to estimate access costs.
- ◆ By utilizing Optimizer Hints.

The Db2 Optimizer Cont'd



Watch out for different RID pool sizing from production and test environments. The row id (RID) pool is used for the RID sorts that accompany optimizer access path techniques such as list pre-fetch, hybrid join, and multi-index access. These pool sizes may vary from production environments to test environments with typically more RID pool sort area in production. This can at times cause the access path to be different in different environments

Other items affecting optimization: ♦ Buffer Pools, ♦ Rid Pools.

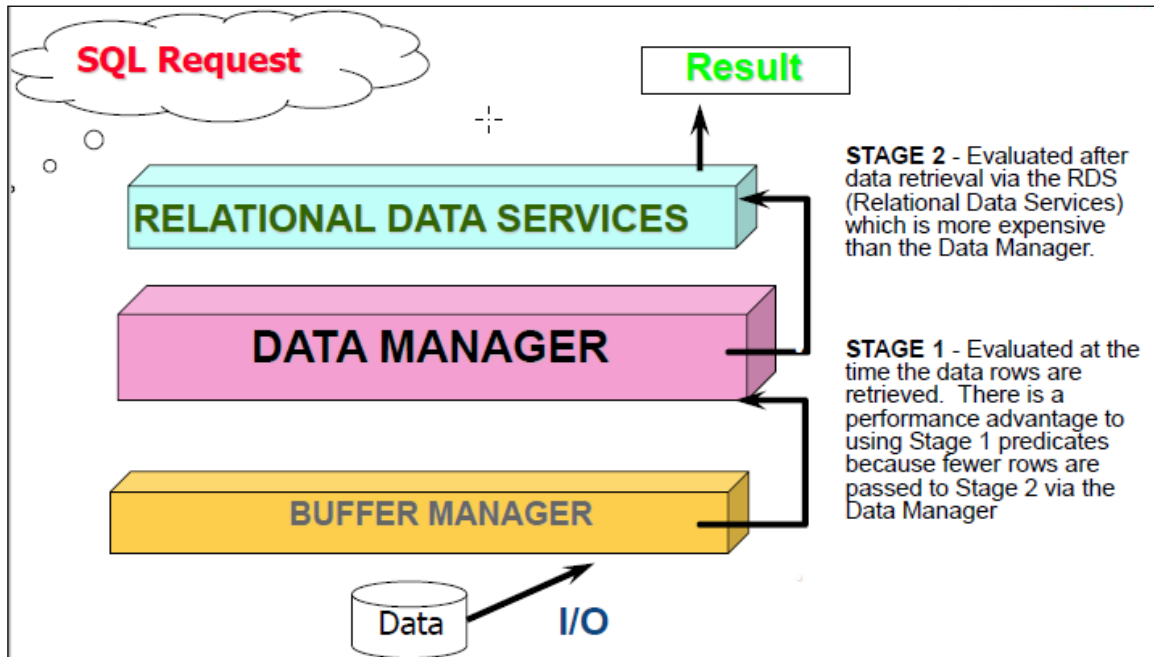
You can improve access path testing by updating the catalog statistics on your test system to be the same as your production system.

There are various ways to accomplish this.

You can improve the accuracy of test access path by modeling the configuration and settings of your production subsystem in your test subsystem. The test system uses values that you specify in profile tables for:

- processor configuration
- RID pool, Sort pool
- Buffer pool settings.

Stage 1 versus Stage 2 Predicates



If rows can be eliminated being passed to stage 2, the better the performance since that is 1 less step taken to apply logic.

Stage 2 = Non Sargeable. Predicate processing by this RDS are of Db2 is much more expensive than the RDS Stage 1 area. Additional processing, additional code path, much more expensive then stage 1.

Indexable predicates evaluated first, Stage 1 predicates, next, and Stage 2 predicates last.

Stage 1 versus Stage 2 Predicates

- **Stage 1** (Db2 Data Manager) is responsible for translating the data stored on pages into a result set of rows and columns. Predicates that are written in a fairly straightforward way can usually be evaluated by the Data Manager with relatively little expense.
- **Stage 2** (Relational Data Services) handle more complex predicates, data transformations, and computations. These Stage 2 predicates are much more expensive for Db2 to resolve than Stage 1 due to additional processing and additional code path. Additionally, Stage 2 predicates cannot make effective use of indexes.

Luckily: There are far less stage 2 predicates as of V11



Stage 1 = Sargeable

Stage 2 = Non Sargeable. Predicate processing by this RDS are of Db2 is much more expensive than the RDS Stage 1 area. Additional processing, additional code path, much more expensive then stage 1.

Indexable predicates evaluated first, Stage 1 predicates, next, and Stage 2 predicates last.

Stage 2 Predicates

Use the Visual Explain in IBM Data Studio or query directly the `DSN_PREDICAT_TABLE` to see any stage 2 predicates. Note the filter factor information also.

The screenshot shows the Visual Explain interface in IBM Data Studio. On the left, the 'Node Descriptor' pane displays the query plan for a 'FETCH' operation. The plan includes a 'Stage2_Predicates' node with a filter factor of 0.01, and a 'Page_Ranges' node with a filter factor of 1.073. The 'Attribute explanation' pane shows the following table:

Name	Value
Input Cardinality	8.3161
Scanned Rows	8.3161
Stage 1 Returned Rows	8.3161
Stage 2 Predicates	Filter Factor
'1990-01-01' BETWEEN THE...	0.01
Stage 2 Returned Rows	1.073
Output Cardinality	1.073
Estimated Total Cost	N/A

On the right, the query plan diagram shows a hierarchical structure starting with a 'QUERY' node, followed by 'QBI 1', 'FETCH 1.073', and 'MIXSCAN 8.3161'. The 'MIXSCAN' node branches into two 'SORTED' nodes, which then branch into two 'IXSCAN' nodes, and finally into two 'XEMP' nodes.



- 1) Click on the FETCH box to see any/all predicates not associated with the index chosen
- 2) Click on the IXSCAN boxes to see matching index and screening index predicate information

What Makes an Inefficient SQL Query?

- 1). **SQL statements have poorly coded predicates. Stage 2 and/or non indexable. Db2 LUW has RESIDUAL Predicates**
- 2). **SQL is doing more work than is needed. For example:**
 - Sorts that are not needed
 - Distinct / Group By / Order By that are not needed
 - UNION versus UNION ALL
 - Extra tables that are not needed
 - Table not being joined to (causes Db2 to do its own joining called a Cartesian join).
- 3). **SQL not using an index, instead executing table scans**
- 4). **SQL not using an index to full capacity (Index scan, too much screening)**
- 5). **Extra columns / Extra rows being returned**



Developers should review their own SQL code and make sure it is:

- Not doing more work than needed
- Not bring back more data (columns and/or rows) needed
- Not doing unneeded sorts

What Makes an Inefficient Program that Contains SQL?

Too many trips between the program code and Db2. Minimize the SQL requests to Db2!! In general let Db2 do the work !!

Code relationally, not procedurally !!

This is huge in performance tuning of programs, especially batch programs because they tend to process more data. Every time an SQL call is sent to the database manager, there is overhead in sending the SQL statement to Db2, going from one address space in the operating system to the Db2 address space for SQL execution.

In general developers need to minimize:

- The number of time cursors are Opened/Closed
- The number of random SQL requests (noted as synchronized reads in Db2 monitors).

V8: Multi Row Fetch, Update, and Inserting. Recursive SQL. Select from Insert.

V9: 'Upsert' processing. Fetch First / Order By within subqueries.



1) Minimize programming API (Application Programming Interface) to Db2. There is overhead involved in API calls, no matter how efficient the call may be.

2) Minimize the number of fetches by using multi row fetch. Take advantage of multi row deletes/inserts also.

3) Distributed Apps: Once in compatibility mode in V8 the blocks used for block fetching are built using the multi-row capability without any code change. This results in automatic savings for example distributed SQLJ applications.

Multi Row Fetch – Should be a Standard

When using cursors, use ROWSET positioning and fetching using multi row fetch, multi row update, and multi row insert.

Db2 V8/V9 introduced support for the manipulation of multiple rows on fetches, updates, and insert processing. Prior versions of Db2 would only allow for a program to process one row at a time during cursor processing. Now having the ability to fetch, update, or insert more than 1 row at a time reduces network traffic and other related costs associated with each call to Db2. CPU is greatly reduced. Going across address spaces with requests and the movement of data is expensive.

The recommendation is to start with 100 row fetches, inserts, or updates, and then test other numbers. **It has been proven many times that this process reduces runtime on average of 35%.** Consult the IBM Db2 manuals for further detail and coding examples. *At times up to 50% faster!*



- 1) Check 'Get Diagnostics' only after receiving a SQLCODE \neq 0
- 2) For multi-row fetch you get a +354 SQLCODE which says "one or more errors may have occurred."
- 3) For a non atomic multi-row insert you get a -253 if some of the rows fail, -254 if all fail
- 4) For an atomic multi-row insert you get the real SQLCODE of the first failure (since atomic means any failure backs out the entire insert), but you still need the diagnostics to determine which row actually tripped the error.
- 5) If you get multiple errors, they will be returned in reverse order. e.g. when inserting 5 rows, row no. 2 and row no. 4 had errors. GET DIAGNOSTICS shows three errors (not two!) - first is the generic, 2nd is error for row #4 and third is error for row #2.
- 6) If you perform a multi-row operation and receive an SQLCODE of 0 then you generally have no need for GET DIAGNOSTICS, which is VERY expensive. So, check SQLCODE first and only go to the DIAGNOSTICS when errors are encountered.
- 7) Multi Row for even a small number of rows returned can be beneficial. When searching for a "break-even" (runtime), don't forget to take into consideration the length of the data rows being fetched. Multi-row fetch will be especially beneficial (even for just a few rows) if the rows are short in length.
- 8) Seen best results with large amounts of data being returned from a cursor. Fetches of 100 rows each.

Generate the Following Report: EMP Detail Data, Along with Aggregate Data

Detail data

Aggregate Data

EMPNO	LASTNAME	DEPTNO	SALARY	DEPT_AVG_SAL
000120	O'CONNELL	A00	29250.00	45312.50000000
000110	LUCCHESI	A00	46500.00	45312.50000000
000010	HAAS	A00	52750.00	45312.50000000
000020	THOMPSON	B01	41250.00	41250.00000000
000130	QUINTANA	C01	23800.00	30156.66666666
000140	NICHOLLS	C01	28420.00	30156.66666666
000030	KWAN	C01	38250.00	30156.66666666
000210	JONES	D11	18270.00	24677.77777777
000190	WALKER	D11	20450.00	24677.77777777
000180	SCOUTTEN	D11	21340.00	24677.77777777
000160	PIANKA	D11	22250.00	24677.77777777
000170	YOSHIMURA	D11	24680.00	24677.77777777
000150	ADAMSON	D11	25280.00	24677.77777777
000200	BROWN	D11	27740.00	24677.77777777
000220	LUTZ	D11	29840.00	24677.77777777
000060	STERN	D11	32250.00	24677.77777777



Not stating that this is the best option, but at times it may be. But it gives developers another way to code for certain results.

Other Options:

- ```

1) SELECT E1.EMPNO, E1.LASTNAME, E1.WORKDEPT, E1.SALARY, X.DEPT_AVG_SAL
 FROM EMPLOYEE E1 INNER JOIN
 (SELECT E2.DEPTNO, DEC(ROUND(AVG(E2.SALARY),2),9,2)
 AS DEPT_AVG_SAL
 FROM EMP E2
 GROUP BY E2.WORKDEPT) AS X ON E1.DEPTNO = X.DEPTNO
 ORDER BY E1.WORKDEPT, E1.SALARY

2) WITH X AS
 (SELECT E2.WORKDEPT, DEC(ROUND(AVG(E2.SALARY),2),9,2)
 AS DEPT_AVG_SAL
 FROM EMPLOYEE E2
 GROUP BY E2.WORKDEPT)

 SELECT E1.EMPNO, E1.LASTNAME, E1.WORKDEPT, E1.SALARY, X.DEPT_AVG_SAL
 FROM EMP E1 INNER JOIN
 X ON E1.DEPTNO = X.DEPTNO
 ORDER BY E1.WORKDEPT, E1.SALARY

3) SELECT E1.EMPNO, E1.LASTNAME, E1.WORKDEPT, E1.SALARY, AVG(E2.SALARY) AS DEPT_AVG_SAL
 FROM EMP E1 INNER JOIN
 EMP E2 ON E1.DEPTNO = E2.DEPTNO
 GROUP BY E1.EMPNO, E1.LASTNAME, E1.WORKDEPT, E1.SALARY
 ORDER BY E1.WORKDEPT, E1.SALARY

```

# Query Example of a Program Coded Relational vs Procedural

**Take advantage of Scalar Fullselects within the Select clause whenever possible.**

Many times the output needed from SQL development requires a combination of detail and aggregate data together. There are typically a number of ways to code this with one SQL, and the Scalar Fullselect provides a newer way to get both in the same query. .

For Example: Individual Employee Report with Aggregate Department Averages

```
SELECT E1.EMPNO, E1.LASTNAME,
E1.DEPTNO, E1.SALARY, (SELECT AVG(E2.SALARY)
FROM EMP E2
WHERE E2.DEPTNO = E1.DEPTNO)
AS DEPT_AVG_SAL
FROM EMP E1
ORDER BY E1.DEPTNO, E1.SALARY
```



Not stating that this is the best option, but at times it may be. But it gives developers another way to code for certain results.

Other Options:

- 
- 1) 

```
SELECT E1.EMPNO, E1.LASTNAME, E1.WORKDEPT, E1.SALARY, X.DEPT_AVG_SAL
FROM EMPLOYEE E1 INNER JOIN
 (SELECT E2.DEPTNO, DEC(ROUND(AVG(E2.SALARY),2),9,2)
 AS DEPT_AVG_SAL
 FROM EMP E2
 GROUP BY E2.WORKDEPT) AS X ON E1.DEPTNO = X.DEPTNO
ORDER BY E1.WORKDEPT, E1.SALARY
```
  - 2) 

```
WITH X AS
(SELECT E2.WORKDEPT, DEC(ROUND(AVG(E2.SALARY),2),9,2)
 AS DEPT_AVG_SAL
 FROM EMPLOYEE E2
 GROUP BY E2.WORKDEPT)

SELECT E1.EMPNO, E1.LASTNAME, E1.WORKDEPT, E1.SALARY, X.DEPT_AVG_SAL
FROM EMP E1 INNER JOIN
 X ON E1.DEPTNO = X.DEPTNO
ORDER BY E1.WORKDEPT, E1.SALARY
```
  - 3) 

```
SELECT E1.EMPNO, E1.LASTNAME, E1.WORKDEPT, E1.SALARY, AVG(E2.SALARY) AS DEPT_AVG_SAL
FROM EMP E1 INNER JOIN
 EMP E2 ON E1.DEPTNO = E2.DEPTNO
GROUP BY E1.EMPNO, E1.LASTNAME, E1.WORKDEPT, E1.SALARY
ORDER BY E1.WORKDEPT, E1.SALARY
```

# Query Example of a Program Coded Relational vs Procedural

## Take advantage of Common or Nested Table Expressions.

Many times the output needed from SQL development requires a combination of detail and aggregate data together. There are typically a number of ways to code this with one SQL, and the coding a table expressions is another way.

For Example: Individual Employee Report with Aggregate Department Averages

```
SELECT E1.EMPNO, E1.LASTNAME, E1.WORKDEPT, E1.SALARY,
 X.DEPT_AVG_SAL
FROM EMPLOYEE E1 INNER JOIN
 (SELECT E2.DEPTNO, DEC(ROUND(AVG(E2.SALARY),2),9,2)
 AS DEPT_AVG_SAL
FROM EMP E2
 GROUP BY E2.WORKDEPT) AS X
ON E1.DEPTNO = X.DEPTNO
ORDER BY E1.WORKDEPT, E1.SALARY
```



Not stating that this is the best option, but at times it may be. But it gives developers another way to code for certain results.

Other Options:

- 1) 

```
SELECT E1.EMPNO, E1.LASTNAME, E1.WORKDEPT, E1.SALARY, X.DEPT_AVG_SAL
FROM EMPLOYEE E1 INNER JOIN
 (SELECT E2.DEPTNO, DEC(ROUND(AVG(E2.SALARY),2),9,2)
 AS DEPT_AVG_SAL
FROM EMP E2
 GROUP BY E2.WORKDEPT) AS X ON E1.DEPTNO = X.DEPTNO
ORDER BY E1.WORKDEPT, E1.SALARY
```
- 2) 

```
WITH X AS
(SELECT E2.WORKDEPT, DEC(ROUND(AVG(E2.SALARY),2),9,2)
 AS DEPT_AVG_SAL
FROM EMPLOYEE E2
 GROUP BY E2.WORKDEPT)

SELECT E1.EMPNO, E1.LASTNAME, E1.WORKDEPT, E1.SALARY, X.DEPT_AVG_SAL
FROM EMP E1 INNER JOIN
 X ON E1.DEPTNO = X.DEPTNO
ORDER BY E1.WORKDEPT, E1.SALARY
```
- 3) 

```
SELECT E1.EMPNO, E1.LASTNAME, E1.WORKDEPT, E1.SALARY, AVG(E2.SALARY) AS DEPT_AVG_SAL
FROM EMP E1 INNER JOIN
 EMP E2 ON E1.DEPTNO = E2.DEPTNO
GROUP BY E1.EMPNO, E1.LASTNAME, E1.WORKDEPT, E1.SALARY
ORDER BY E1.WORKDEPT, E1.SALARY
```

# Query Example of a Program Coded Relational vs Procedural

## Take advantage of Joins with Group By.

Many times the output needed from SQL development requires a combination of detail and aggregate data together. There are typically a number of ways to code this with one SQL, and the 2 table join with a group by can be used. same query. .

For Example: Individual Employee Report with Aggregate Department Averages

```
SELECT E1.EMPNO, E1.LASTNAME, E1.WORKDEPT, E1.SALARY,
 AVG(E2.SALARY) AS DEPT_AVG_SAL
FROM EMP E1 INNER JOIN
 EMP E2 ON E1.DEPTNO = E2.DEPTNO
GROUP BY E1.EMPNO, E1.LASTNAME, E1.WORKDEPT, E1.SALARY
ORDER BY E1.WORKDEPT, E1.SALARY
```



Not stating that this is the best option, but at times it may be. But it gives developers another way to code for certain results.

Other Options:

- 
- 1) 

```
SELECT E1.EMPNO, E1.LASTNAME, E1.WORKDEPT, E1.SALARY, X.DEPT_AVG_SAL
FROM EMPLOYEE E1 INNER JOIN
 (SELECT E2.DEPTNO, DEC(ROUND(AVG(E2.SALARY),2),9,2)
 AS DEPT_AVG_SAL
FROM EMP E2
GROUP BY E2.WORKDEPT) AS X ON E1.DEPTNO = X.DEPTNO
ORDER BY E1.WORKDEPT, E1.SALARY
```
  - 2) 

```
WITH X AS
(SELECT E2.WORKDEPT, DEC(ROUND(AVG(E2.SALARY),2),9,2)
AS DEPT_AVG_SAL
FROM EMPLOYEE E2
GROUP BY E2.WORKDEPT)

SELECT E1.EMPNO, E1.LASTNAME, E1.WORKDEPT, E1.SALARY, X.DEPT_AVG_SAL
FROM EMP E1 INNER JOIN
 X ON E1.DEPTNO = X.DEPTNO
ORDER BY E1.WORKDEPT, E1.SALARY
```
  - 3) 

```
SELECT E1.EMPNO, E1.LASTNAME, E1.WORKDEPT, E1.SALARY, AVG(E2.SALARY) AS DEPT_AVG_SAL
FROM EMP E1 INNER JOIN
 EMP E2 ON E1.DEPTNO = E2.DEPTNO
GROUP BY E1.EMPNO, E1.LASTNAME, E1.WORKDEPT, E1.SALARY
ORDER BY E1.WORKDEPT, E1.SALARY
```

# Data Statistics

**Make sure the data distribution statistics are current in the tables being processed.**

This is done by executing the Runstats utility on each specific table and associated indexes. This utility loads up the system catalog tables with data distribution information that the optimizer looks for when selecting access paths. Some of the information that the Runstats utility can provide is:

- The size of the tables (# of rows)
- The cardinalities of columns
- The percentage of rows (frequency) for those uneven distribution of column values
- The physical characteristics of the data and index files. Rows / Pages etc.
- Information by partition

Pay attention to the 'Statstime' column in the catalog tables as it will state when the last time Runstats has been executed on each table.

**Volatile Tables** - Db2 considers using Index access no matter the statistics

**GTTs** – Created can have manual statistics added. Declared can have indexes.



1) FREQVAL Statistics are important for any columns containing uneven distribution of data values. For example some tables may contain a status code column containing multiple values. If any of the values contains a high or low percentage of rows in the table, then it should have FREQVAL statistics run on that column.

2) Statistics are typically up to date in production, but many time are behind or even reset in test environments.

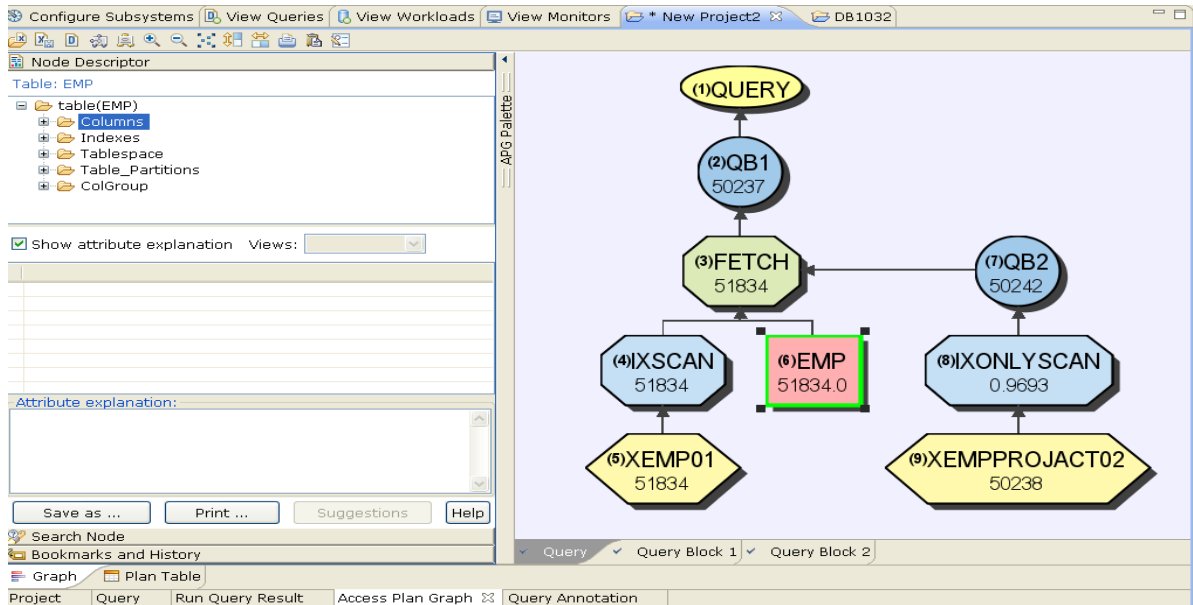
3) Volatile tables are always an issue. Statistics only reflect a point in time.

By declaring the table volatile, the optimizer will consider using index scan rather than table scan. The access plans that use declared volatile tables will not depend on the existing statistics for that table.

4) By creating a Global Temporary Table, manual statistics can then be added to the catalog tables for the average number of rows, average cardinalities, etc...

# Data Statistics

**Make sure the data distribution statistics are current in the tables being processed.**



1) FREQVAL Statistics are important for any columns containing uneven distribution of data values. For example some tables may contain a status code column containing multiple values. If any of the values contains a high or low percentage of rows in the table, then it should have FREQVAL statistics run on that column.

2) Statistics are typically up to date in production, but many time are behind or even reset) in test environments.

3) Volatile tables are always an issue. Statistics only reflect a point in time.

By declaring the table volatile, the optimizer will consider using index scan rather than table scan. The access plans that use declared volatile tables will not depend on the existing statistics for that table.



## Db2 Runstats Utility

- Optimizer takes into account table statistics for access path optimization
- Cardinality statistics should be generated for each column that is used in 'Where' logic.
- Cardinality of total rows are generated for the table
- Frequency value statistics are great for data with very uneven distribution of values (and used in 'Where' logic) (and used in 'Where' logic).
- Histogram statistics great for range type predicates
- Hard coding and/or dynamic queries needed to take advantage of Frequency / Histogram statistics



The optimizer assumes that the data is uniformly distributed. Therefore, frequency distribution statistics are needed on columns where there is a skew in the data values that may cause poor decisions by the optimizer. Histogram helps data estimation when certain data ranges have a lot of data and others are sparse (hot spots in data across many values).

The best statistics do not help if the queries are optimized with host variables. At times, queries need to have some hard coded values or have the query executed dynamically.

# Db2 Runstats Utility – Freqval Stats

Example: **COMM** column in **EMP** table has 31 values, but the value 0 is 99% of the data. Frequency value runstats can load this information for Db2. *Developers have to code for it.*

```
SELECT *
FROM EMP
WHERE LASTNAME LIKE 'S%'
AND COMM = 0
```

```
SELECT *
FROM EMP
WHERE LASTNAME LIKE 'S%'
AND COMM = ?
```

```
SELECT *
FROM EMP
WHERE LASTNAME LIKE 'S%'
AND COMM = ?
AND COMM <> 0
```



The optimizer assumes that the data is uniformly distributed. Therefore, frequency distribution statistics are needed on columns where there is a skew in the data values that may cause poor decisions by the optimizer. Histogram helps data estimation when certain data ranges have a lot of data and others are sparse (hot spots in data across many values).

# Db2 Runstats Utility – Histogram Stats

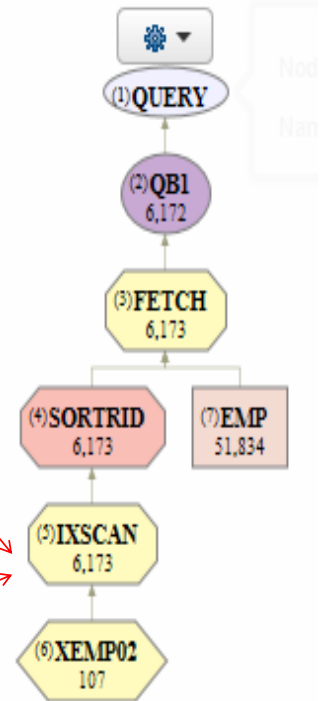
**Example: DEPTNO has over 100 values, and there are queries with ‘Range’ type predicates often applied (Between, >, <, Like).**

```
SELECT COUNT(*) FROM EMP => 51,803
WHERE DEPTNO BETWEEN 'P00' AND 'P99'
```

Filter Factor

0.11908262968063354

```
SELECT LASTNAME FROM EMP
WHERE DEPTNO BETWEEN 'P00' AND 'P99'
```



The optimizer assumes that the data is uniformly distributed. Therefore, frequency distribution statistics are needed on columns where there is a skew in the data values that may cause poor decisions by the optimizer. Histogram helps data estimation when certain data ranges have a lot of data and others are sparse (hot spots in data across many values). And especially when there are range predicates against the column.

Histogram statistics are only good if the queries sent to the optimizer have hard coded values or are dynamic, or are being REPOPT'd at runtime. ***Histogram statistics will not help queries with host variables.***

If no NUMQUANTILES is specified, Db2 will choose based on the cardinality of the columns and number of rows up to 100 max. If you know the application and the typical ranges that get executed, set the number accordingly. For example, if many of the ranges get 5% or less, then specify a higher number of quantiles. But if the range predicates are always on a larger range, less quantiles is needed.

# Db2 Runstats Utility – Histogram Stats

```
RUNSTATS INDEX(THEMIS82.XEMP02)
HISTOGRAM NUMCOLS 1 NUMQUANTILES 20
```

*For Indexed Columns or Column Groups*

```
RUNSTATS TABLESPACE DTHM82.TS00EMP
TABLE(THEMIS82.EMP)
COLGROUP(SALARY)
HISTOGRAM NUMQUANTILES 20
```

*For ANY Column or Column Group*



These two control cards gather the new histogram statistics. The first example gathers the statistics on the leading column of the XEMP02 index (the DEPTNO column). Histogram statistics break the data into ranges of values with an estimate of the percentage of rows that fall into each range. The NUMQUANTILES option specifies how many ranges should be used (or how much granularity will be present in the statistics). This number is only an estimate, and it may be necessary for the utility to use a few more or less quantiles than requested.

The second example gathers the histogram statistics on a non-indexed column. This invocation of the utility requires a sort of the data to determine the appropriate groupings making this collection rather expensive. Although this syntax may be used for indexed columns, the syntax used in the first example should be used whenever possible to avoid sorting.

It is possible to gather these statistics infrequently even if RUNSTATS is run regularly without the HISTOGRAM option. When the HISTOGRAM option is not used, these statistics are not overlaid in the catalog.

# Db2 Runstats Utility – Histogram Stats

```
SELECT CHAR(NAME,18) AS NAME,
 CHAR(STRIP(LOWVALUE,B),10) AS LOW,
 CHAR(STRIP(HIGHVALUE,B),18) AS HI,
 CAST (FREQUENCYF * 100 AS DECIMAL(5,2))
 AS PCT_IN_RANGE

FROM SYSIBM.SYSCOLDIST

WHERE TBNAME = 'EMP'
 AND TBOWNER = 'THEMIS82'
 AND TYPE = 'H'

ORDER BY NAME, LOW
```



Histogram statistics are stored in SYSIBM.SYSCOLDIST with a TYPE code of “H”. This query may be used to view and analyze histogram statistics for a particular column. The frequency values are stored as floating point decimals, so this query converts them to decimals for ease of viewing.

# Db2 Runstats Utility – Histogram Stats

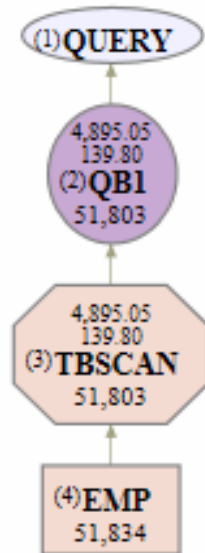
| <b>NAME</b> | <b>LOW</b> | <b>HI</b> | <b>PCT IN RANGE</b> |
|-------------|------------|-----------|---------------------|
| DEPTNO      | A00        | E21       | 0.05                |
| DEPTNO      | P01        | P04       | 3.96                |
| DEPTNO      | P05        | P08       | 2.98                |
| DEPTNO      | P08        | P12       | 3.76                |
| DEPTNO      | P12        | P16       | 3.45                |
| DEPTNO      | P16        | P20       | 3.35                |
| DEPTNO      | P20        | P24       | 3.44                |
| DEPTNO      | P24        | P29       | 3.73                |
| DEPTNO      | P30        | P35       | 5.28                |
| DEPTNO      | P35        | P39       | 3.11                |
| DEPTNO      | P39        | P43       | 3.78                |
| DEPTNO      | P43        | P50       | 5.46                |
| DEPTNO      | P50        | P54       | 3.83                |
| DEPTNO      | P54        | P59       | 3.56                |
| DEPTNO      | P59        | P62       | 2.67                |
| DEPTNO      | P62        | P66       | 3.49                |
| DEPTNO      | P66        | P70       | 3.46                |
| DEPTNO      | P70        | P74       | 3.80                |
| DEPTNO      | P74        | P80       | 5.45                |
| DEPTNO      | P80        | P84       | 3.39                |
| DEPTNO      | P84        | P90       | 4.99                |
| DEPTNO      | P90        | P94       | 3.82                |
| DEPTNO      | P94        | P98       | 3.66                |
| DEPTNO      | P98        | P99       | 1.06                |



Here are the results of the SYSCOLDIST query. Note that the data is not evenly distributed across the potential values.

# Db2 Runstats Utility – Histogram Stats

```
SELECT LASTNAME FROM EMP
WHERE DEPTNO BETWEEN 'P00' AND 'P99'
```



***With XEMP02  
Histogram Stats***



Queries that use range predicates (<, >, BETWEEN, LIKE) are most likely to take advantage of available histogram statistics. This example shows a query retrieving all employees for departments less than or equal to “M99”. The access path graphs from Optimization Service Center reflect the access path before and after the addition of histogram statistics on the DEPTNO column. Without the histogram statistics, the optimizer only knows the LOW2KEY and HI2KEY of the column and uses an interpolation formula to estimate the number of records that will be returned. This estimate is 5,699 rows out of 51,834 rows on the table and the resulting access path is index access with list prefetch. With the addition of histogram statistics on the DEPTNO column, the optimizer knows that the data is skewed; in this case, most employees work in a department that begins with “P” (see previous page for actual statistics). The estimate of rows is reduced to 51,802 which is much closer to reality. The access path that is chosen when this information is known is a tablespace scan.

# Db2 Runstats Utility – Histogram Stats

```
SELECT LASTNAME, FIRSTNME
WHERE DEPTNO BETWEEN ? AND ?
```

## What if?

- Sometimes the range is narrow
- Sometimes the range is wide
- The variables are different on each execution



Actual count = 51,802 rows. The optimizer does not have specific information on ranges of values. Need to write the code as dynamic SQL or use a REOPT option.

JAVA Dynamic example:

```
// Variables to hold the deptno values
String v1 = null;
String v2 = null;
// calls to methods (not shown) that deliver input values
v1 = getInput1(); // e.g. receives D01
v2 = getInput2(); // e.g. receives D11
// NOTE: single quotes embedded around each of the two deptno values
String sql = "SELECT LASTNAME, FIRSTNME, HIREDATE FROM EMP "
 + "WHERE DEPTNO BETWEEN '" + v1 + "' AND '" + v2 + "'";
// conn is the Connection to Db2 assumed to have been made already
// Statement is the vehicle for passing SQL to Db2 and getting back results
Statement stmt = conn.createStatement();
// The sql statement is passed to Db2 in this Java statement
// Since a SELECT statement is passed, a ResultSet is returned
ResultSet rs = stmt.executeQuery(sql);
// This method would process row by row and field by field (not shown) processResults(rs);
```



# Clustering Order of Data

**Make sure of the clustering order of data in your tablespaces.**

Tables should be physically clustered in the order that they are typically processed by queries processing the most data. This ensures the least amount of 'Getpages' when processing.

- 1) Indexes specify the physical order.
- 2) Cluster = 'YES' or first index created

Long running queries with 'List Prefetch' and 'Sorts' in many join processes are good indicators that maybe a table is not in the correct physical order.

How is the data accessed? How is the table joined to most often? What queries bring back the most data, most often?



Getting the correct physical clustering can save sorts, and I/Os.

# Indexes for Clustering

**A ‘Clustering Index’ specifies how data is physically ordered in the table space file. Operations that benefit from a good clustering design are:**

- Queries that return a high number of rows
- Grouping operations
- Ordering operations
- Join operations (especially on Foreign Keys)
- Range type predicates



The way the data is clustered in a table space should be specific to the queries that process through and/or return many rows of data. ra I/O to the index files. The more indexes, the longer REORG utility processes takes.

Know you data, and know your processing.

# EMP Table Clustered by EMPNO

|                            |                            |  |
|----------------------------|----------------------------|--|
| 000010 HAAS ..... A00      | 000100 SPENSER ..... E21   |  |
| 000020 THOMPSON ..... B01  | 000110 LUCHESI ..... A00   |  |
| 000030 KWAN ..... C01      | 000120 O'CONNELL ..... A00 |  |
| 000050 GEYER ..... E01     | 000130 QUINTANA ..... C01  |  |
| 000060 STERN ..... D11     | 000140 NICHOLLS ..... C01  |  |
| 000070 PULASKI ..... D21   | 000150 ADAMSON.. ..... D11 |  |
| 000090 HENDERSON ..... E11 | 000160 PIANKA ..... D11    |  |
|                            |                            |  |

Should this table be in EMPNO Primary Key order?

It Depends.....

This is an example of the EMP table being in EMPNO primary key order.

# EMP Table Clustered by EMPNO

|                                                                                                                                                                                           |                                                                                                                                                                                                       |  |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--|
| 000010 HAAS ..... A00<br>000020 THOMPSON ..... B01<br>000030 KWAN ..... C01<br>000050 GEYER ..... E01<br>000060 STERN ..... D11<br>000070 PULASKI ..... D21<br>000090 HENDERSON ..... E11 | 000100 SPENSER ..... E21<br>000110 LUCHESI ..... A00<br>000120 O'CONNELL ..... A00<br>000130 QUINTANA ..... C01<br>000140 NICHOLLS ..... C01<br>000150 ADAMSON.. ..... D11<br>000160 PIANKA ..... D11 |  |
|                                                                                                                                                                                           |                                                                                                                                                                                                       |  |

What happens here?

```
SELECT *
FROM EMP
WHERE DEPTNO = 'A00'
```

Where are all the rows that have 'A00' as a DEPTNO value?

IF there were 100 rows that contain this value, they could be on 100 pages of data. Yes?



With the table being in EMPNO order, any queries on DEPTNO, or any joins by DEPTNO will cause many getpages due to the DEPTNOs being scattered across the file.

# Indexing

**Make sure indexes are being chosen by optimizer for queries.**

At times (rare), it can be better to have the optimizer execute a query with a full table space scan versus an index. But I would say in my experience that 99% of the time developer SQL code should be using indexes for the retrieval of data.

**Why are table spaces scans problematic?**

**Why would the optimizer not choose an index, instead choosing a table space scan?**



Getting the correct physical clustering can save sorts, and I/Os. A very small percentage of queries should be doing table scans, so it is important for developers to make sure their queries are using indexes, or understand why the optimizer chose a table scan.

# Why are Table Space Scans Problematic at Times?

- Increase of CPU time when it comes to checking every row in a table against the query logic.
- Disk I/O: All data in a table may not be in a buffer pool, so physical I/O is needed. There is something called Sequential Prefetch that will help minimize the I/O needed when table scans are occurring, but for some tables there can still be a lot of I/O.
- Filling up its assigned buffer pool. Data is always brought from the physical disk to an area in memory called a buffer pool. Sometimes a buffer pool is shared by a number of tables. Once data is brought into a buffer pool, it will stay until the buffer pool area gets filled up, and then some pages begin dropping out. Db2 knows what data pages for a table are in the buffer pool, and ones that are not. Obviously the more data pages for a table in buffer pool memory, the better. Having one table take up most of a buffer pool can then hurt any other table's response times.

# Why Would the Optimizer Choose a Table Space Scan?

1. Are any predicate(s) poorly coded in a non-indexable way that takes away any possible index choices from the optimizer?
2. Do the predicates in the query not match any available indexes on the table? ***Know your indexes on a table!***
3. The table could be small and Db2 decides a table scan may be faster than index processing.
4. The catalog statistics could say the table is small. This is more common in test environments where the Runstats utility is not executed very often.
5. Are the predicates such that Db2 thinks the query is going to retrieve a large enough amount of data that would require a table scan? Some explain tools will show the number of rows Db2 thinks will be returned in the execution of a query (the IBM Data Studio tool is very good at this).



Developers should:

- Know their data (cardinalities, columns with uneven distributions of data, etc.)
- Know the indexes on tables involved in their queries
- Know the clustering order of data in their tables
- Know the partitioning of data in their tables

# Why Would the Optimizer Choose a Table Space Scan?

6. Are the predicates such that Db2 picks a non-clustered index, and the rows needed are scattered throughout the table file such that the number of data pages to retrieve is high enough based on total number of pages in the table to require a table scan? ***Know how the data is physically clustered in the tablespace!***
7. Are the tablespace files or index files physically out of shape and need a REORG?
8. Are there no predicates? So the query wants all the rows.
9. Sometimes there are just too many conditions in the logic to return the results needed any other way. This is quite typical with many predicates that are OR'd together.



# Ten Steps to Tuning a Query

- 1) Check every predicate. Are they indexable and Stage 1?
- 2) Is there a 'Distinct' or 'Group By'? Is it needed? Can it be rewritten?
- 3) Are there subqueries? Rewrite 'In' as 'Exists' and vice versa.
- 4) Can the query be rewritten any different way. Predicates rewritten differently?
- 5) Check Db2 statistics. Are they accurate to reality
- 6) Check the number of times every SQL statement is getting executed. Can the logic be changed to cut down the number of times requests are sent?
- 7) Check the Db2 Explain. Are there any table scans? Index Scans? Sorts?
- 8) Check the Db2 VE. Are there any predicates with crazy Filter Factors?
- 9) Check the Db2 Explain. If there is a join, are indexes being chosen on joined columns?
- 10) Check any correlated subqueries? How many times are they executed? Can they be Index-Only?
- 11) Check for any columns in predicates with uneven distribution statistics? Should values be hard coded in the query?
- 12) Are there any range predicates. Could histogram statistics help?

Thank you for allowing me and Themis  
to share some of our experience and  
knowledge today!

*Tony Andrews*

[tandrews@themisinc.com](mailto:tandrews@themisinc.com)

I hope that you learned something new today !!!!



The material in this presentation is further developed in the following Themis courses:

DB1041 – Advanced SQL FOR Db2

SQ1010 – Dealing With Complex Queries  
Cross Platform SQL

DB1037 – Advanced Query Tuning With IBM Data Studio  
on z/OS

DB1032 – Db2 for z/OS Optimization Performance  
and Tuning

DB1006 – Db2 LUW Query Tuning With IBM Data Studio

Links to these courses may be found at: [www.themisinc.com](http://www.themisinc.com)

Tony's Email: [tandrews@themisinc.com](mailto:tandrews@themisinc.com)

Twitter: [@ThemisTraining](https://twitter.com/ThemisTraining)

The material in this presentation is further developed in the following Themis courses:

OR5451 – Oracle Advanced SQL

OR5470 – Oracle Advanced PL/SQL

OR5480 – Oracle SQL Optimization for Developers and DBA's

Links to these courses may be found at: [www.themisinc.com](http://www.themisinc.com)

Tony's Email: [tandrews@themisinc.com](mailto:tandrews@themisinc.com)

Twitter: [@ThemisTraining](https://twitter.com/ThemisTraining)

Education. Check out  
[www.db2sqltuningtips.com](http://www.db2sqltuningtips.com)  
[www.ibmpress.com](http://www.ibmpress.com)  
[www.amazon.com](http://www.amazon.com)

**Finally! A book of Db2 SQL tuning tips for developers, specifically designed to improve performance.**

**Db2 SQL developers now have a handy reference guide with tuning tips to improve performance in queries, programs and applications.**



 **Themis**  
Leaders in IT Education

Education. Check Out  
[www.themisinc.com](http://www.themisinc.com)

- On-site and Public
- Instructor -led
- Hands-on
- Customization
- Experience
- Over 30 Db2 courses
- Over 400 IT courses



**Themis**

Leaders in IT Education

**US 1-800-756-3000**

**Intl. 1-908-233-8900**



**Themis**

Leaders in IT Education