



Data Management

DB2 for Linux, UNIX and Windows Query Access Plan Stability

John Hornibrook
IBM Canada

Agenda

- Has this ever happened to you?
- Finding stable ground:
 - Writing SQL Statements
 - Configuring DB2
 - Catalog Statistics
 - Access Plan Lockdown
 - Optimization Profiles

Has this ever happened to you?

- Scenario 1:
 - Query A runs in 25 s
 - A table in the query grows by 3%
 - RUNSTATS ON TABLE
 - Query A runs in 250 s !
- Scenario 2:
 - Query B runs in 15 s
 - WHERE PACKAGE_SIZE = '16 OZ'
 - Query B changes slightly
 - WHERE PACKAGE_SIZE = '17 OZ'
 - Qualifies roughly the same number of rows
 - Query B₁ runs in 500 s !



Has this ever happened to you? You have an important report query that has been running on an IBM® DB2® for Linux®, UNIX®, and Windows® data server consistently in 25 seconds for the past year. Catalog statistics are regularly collected for tables referenced in the query using the RUNSTATS utility. Suddenly, one day this report query takes 250 seconds to execute after statistics have been collected. Or another query in the same reporting application consistently executes in 15 seconds. A small change is made to one of the search conditions in this query that doesn't significantly increase the size of the result set, and execution now takes 500 seconds!

So what do you do?

- Performance troubleshooting:
 - The problem is consistently reproduced
 - Not due to erratic system load
 - Query A runs fast with the previous catalog statistics
 - Query B is always faster than B₁
 - About the same number of rows selected
 - Probably not a system bottleneck
 - Did the access plans change?
 - Get the explain output



Best Practices: Tuning and Monitoring Database System Performance

http://download.boulder.ibm.com/ibmdl/pub/software/dw/dm/db2/bestpractices/DB2BP_System_Performance_1008I.pdf

After following standard performance problem determination procedures, you discover that the access plan has changed in both scenarios. Now what do you do? You can compare the access plan differences to try to understand why the DB2 data server optimizer chose a new access plan. The explain facility is the recommended tool for capturing access plans. The access plan's details are written to a set of explain tables. You can use the db2exfmt tool to format the contents of the explain tables to produce a text file, or you can use the Data Studio Visual Explain tool to provide a graphical view of the access plan. You can read more about the explain facility here:

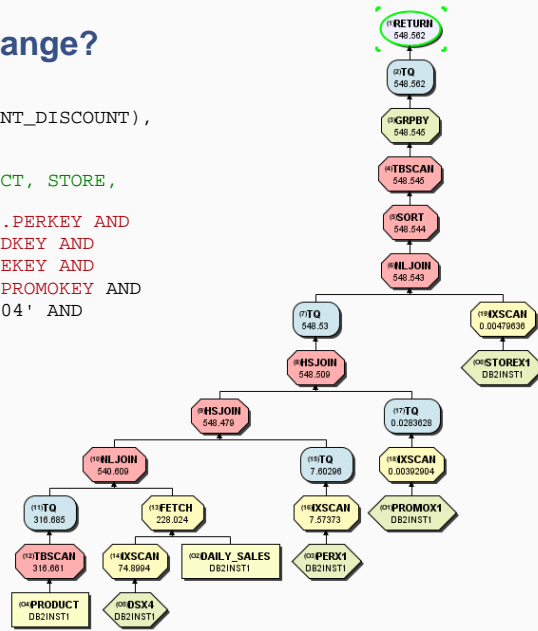
<http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/topic/com.ibm.db2.luw.admin.perf.doc/doc/c0005134.html>

Did the access plan change?

```

SELECT CATEGORY_DESC, SUM(PERCENT_DISCOUNT),
       SUM(EXTENDED_PRICE),
       SUM(SHELF_COST_PCT_OF_SALE)
FROM PERIOD, DAILY_SALES, PRODUCT, STORE,
       PROMOTION
WHERE PERIOD.PERKEY=DAILY_SALES.PERKEY AND
      PRODUCT.PRODKEY=DAILY_SALES.PRODKEY AND
      STORE.STOREKEY=DAILY_SALES.STOREKEY AND
      PROMOTION.PROMOKEY=DAILY_SALES.PROMOKEY AND
      CALENDAR_DATE BETWEEN '04/01/2004' AND
      '04/14/2004' AND
      STORE_NUMBER='01' AND
      PROMODESC = 'Web' AND
      PACKAGE_SIZE = '16 OZ' AND
      SUB_CATEGORY = 747
GROUP BY CATEGORY_DESC ;
    
```

Query B



The access plan graph shown here is produced using Data Studio Visual Explain.

Information Management IBM

The access plan changed. Now what?

```

SELECT CATEGORY_DESC, SUM(PERCENT_DISCOUNT),
       SUM(EXTENDED_PRICE),
       SUM(SHELF_COST_PCT_OF_SALE)
FROM PERIOD, DAILY_SALES, PRODUCT, STORE,
       PROMOTION
WHERE PERIOD.PERKEY=DAILY_SALES.PERKEY AND
       PRODUCT.PRODKEY=DAILY_SALES.PRODKEY AND
       STORE.STOREKEY=DAILY_SALES.STOREKEY AND
       PROMOTION.PROMOKEY=DAILY_SALES.PROMOKEY AND
CALENDAR_DATE BETWEEN '04/01/2004' AND
'04/14/2004' AND
STORE_NUMBER='01' AND
PROMODESC = 'Web' AND
PACKAGE_SIZE = '17 OZ' AND
SUB_CATEGORY = 747
GROUP BY CATEGORY_DESC ;

```

Query B₁

© 2011 IBM Corporation

Note that the access plan for B₁ joins tables in a different order, using different join methods. Tables queues (TQs) are in different positions, compared to access plan B.

The explain facility provides detailed information about the access plan, including each access plan operator, their sequence of execution, the predicates they apply, the number of rows they process, and their estimated cost. The number of rows processed by an access plan operator directly affects its cost. Sometimes the optimizer mis-estimates the number of rows – or cardinality – because of missing or outdated catalog statistics. The inaccurate cardinality estimates may result in inappropriate access plan changes, so verifying them is often a good place for you to start. The cardinality estimates are based on the number of rows in each referenced database object (table, nickname, table function or materialized query table) and the predicates applied to those objects. It is possible that the cardinality catalog statistic for a base object is inaccurate or it is possible that the selectivity estimate for one or more of the predicates applied to the base object is inaccurate. You can verify the optimizer's cardinality estimates by running subsets of the original query to return a count of the actual number of rows processed by specific access plan operators. Alternatively, if DB2 9.7 is being used, you can enable the DB2 data server to automatically capture the actual runtime cardinality of each operator and store them in the explain tables, where you can later format the contents using db2exfmt[1].

[1] At time of writing, Data Studio Visual Explain does not support reporting actual cardinality.

Explain-with-actual cardinality:

<http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/topic/com.ibm.db2.luw.admin.perf.doc/doc/c0056362.html>

Now what do you do?

- Perform a deeper analysis of the access plan
- Start by confirming cardinality estimates
 - Try “explain-with-actual” cardinality feature, new in V9.7 FP1!
- Look for expensive operations
 - Large TEMPs, SORTs, TQs, HSJOIN build tables
 - Inefficient index access (no start/stop keys)
 - TEMPs on NLJOIN inners
 - Large TBSCANs
 - All work done on a single DB partition in DPF
 - Etc.
- **But...do you really have time for this?**

Once you've identified and corrected any inaccurate cardinality estimates - where that is possible - the access plan may still not have changed, or it may have changed but is still not performing as well as the original access plan. You will then need to perform deeper analysis to understand why the optimizer is still not choosing the better access plan.

As you can appreciate by this point, analyzing poorly performing access plans requires considerable time and skill. While access plan analysis skills are very valuable for you to have, performing such a deep analysis for more than a relatively small number of query access plan changes is probably impractical. Quite often, access plan problems need to be corrected quickly, and there isn't time to perform the steps described above. So a more proactive approach is preferable.

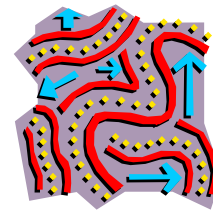
A pro-active approach

- Access plan diagnosis skills are very good to have
 - But it isn't practical to use a reactive approach when too many access plans are changing too often
- Address the fundamental problems in a general way:
 1. Well-written SQL
 2. Proper data server configuration
 3. Statistics, statistics, statistics !
- Provides a 'firm foundation' for all queries
- Handle exceptions with access plan lockdown techniques
 - Access plan lockdown
 - Optimization profiles

Rather than scrambling to fix query access plan problems in panic situations, there are a number of modifications that you can make to your SQL statements, data server configuration, and catalog statistics, to avoid unexpected access plan changes. These modifications allow the optimizer to more accurately cost access plans so that queries run faster. Moreover, when access plan cost estimates are accurate, access plans are less likely to change unexpectedly for the worse, when some aspect of the environment that is considered by the optimizer's cost model changes. For example, a small increase in the number of rows in a table should not result in a new access plan that runs significantly slower. Access plans usually change incorrectly in this type of situation because of other costing inaccuracies. In other words, the optimizer's view of the world is distorted, so it is making decisions based on bad data. The following slides in this presentation describe how you can improve the optimizer's "vision" so it has a clear view of the best access path.

Best practices for writing good SQL

- SQL is a very flexible language
 - There are many ways to get the same correct result
 - This flexibility means that some queries are better than others in taking advantage of the DB2 optimizer's strengths
- The optimizer can perform many query rewrites
 - It can't rewrite all possibilities



SQL is a powerful language that enables you to specify relational expressions in syntactically different but semantically equivalent ways. However, some semantically equivalent variations are easier to optimize than others. Although the DB2 data server optimizer has a powerful query rewrite capability, it might not always be able to rewrite an SQL statement into the most optimal form. There are also certain SQL constructs that can limit the access plans considered by the query optimizer. For example, complex expressions in predicates can limit the optimizer's ability to compute accurate selectivity estimates and can prevent certain access plan operators from being used.

Best practices for writing good SQL

- Avoid complex expressions in search conditions
 - Avoid join predicates on expressions
 - Avoid expressions over columns in local predicates
 - Avoid data type mismatches on join columns
 - Avoid non-equality join predicates
- Avoid unnecessary outer joins
- Use OPTIMIZE FOR N ROWS clause with FETCH FIRST N ROWS ONLY clause
- If you are using the star schema join, ensure your queries fit the required criteria
- Avoid redundant predicates
- Refer to:
 - [The DB2Night Show #52: Writing Optimized DB2 LUW SQL Queries](#)
 - <http://www.dbisoftware.com/blog/db2nightshow.php?id=268>
 - [Best Practices: Writing and Tuning Queries for Optimal Performance](#)
 - <http://www.ibm.com/developerworks/data/bestpractices/querytuning/>

Proper data server configuration

- Optimizer is affected by the following database manager configuration parameters:
 - ▶ CPUSPEED – set automatically by DB2
 - ▶ COMM_BANDWIDTH – set automatically by DB2. Used for costing in DPF system
 - ▶ INTRA_PARALLEL – intra-partition parallelism
- Optimizer is affected by the following database configuration parameters:
 - ▶ DFT_QUERYOPT
 - ▶ SORTHEAP
 - ▶ STMTHEAP
 - ▶ LOCKLIST
 - ▶ AVG_APPLS
 - ▶ MAXLOCKS
 - ▶ DFT_DEGREE (Inter-partition parallelism)
- Tablespace configuration also matters:
 - ▶ **OVERHEAD – should be updated to reflect I/O subsystem characteristics**
 - ▶ **TRANSFERRATE – should be updated to reflect I/O subsystem characteristics**
 - ▶ **Should be adjusted to account for table space page size:**
 - ▶ See: <http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/topic/com.ibm.db2.luw.admin.perf.doc/doc/c0005051.html>
 - ▶ PREFETCHSIZE – considered by cost model
 - ▶ EXTENTSIZE – considered by cost model

There are a number of configuration options that affect the DB2 data server's performance, such as database and database manager configuration parameters, registry variables, bind options, special registers, and database object attributes. Some of these configuration options have a direct effect on the DB2 data server optimizer because they provide information about the system's capabilities that is considered when estimating the cost of access plans. Therefore, access plans will tend to be more stable if configuration options important to the optimizer's cost model are set accurately.

To update the Database Manager configuration parameters, use the command:

update dbm config using *cfg_parm value*

To update the Database configuration parameters for database *dbname*, use the command:

update db config for *dbname* using *cfg_parm value*

OVERHEAD provides an estimate of the time (in milliseconds) that is required by the container before any data is read into memory. This overhead activity includes the container's I/O controller overhead as well as the disk latency time, which includes the disk seek time.

TRANSFERRATE provides an estimate of the time (in milliseconds) that is required to read one page of data into memory. If each table space container is a single physical disk, you can use the following formula to estimate the transfer cost in milliseconds per page: $TRANSFERRATE = (1 / spec_rate) * 1000 / 1024000 * page_size$ where:

- You divide by *spec_rate*, which represents the disk specification for the transfer rate (in megabytes per second), to get seconds per megabyte
- You multiply by 1000 milliseconds per second
- You divide by 1 024 000 bytes per megabyte
- You multiply by the page size (in bytes); for example, 4096 bytes for a 4-KB page

Proper data server configuration

- Additional considerations:
 - Consider using **constraints** to improve query optimization
 - Allows more semantic query rewrites
 - Choose the best **optimization class** for your workload
 - Default is 5
 - 0,1,2,3,5,7 and 9 available
 - Certain DB2 registry variables can provide improved optimization
 - See <http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/topic/com.ibm.db2.luw.admin.regvars.doc/doc/r0005664.html>

The DB2 data server allows you to control the level of optimization performed by the query optimizer. Different levels of optimization are grouped into classes. You can specify the query optimization class using the CURRENT QUERY OPTIMIZATION special register or the QUERYOPT bind option. Setting the optimization class can provide some of the advantages of explicitly specifying optimization techniques, particularly for the following reasons:

- To manage very small databases or very simple dynamic queries
- To accommodate memory limitations at compile time on your database server
- To reduce the query compilation time, such as PREPARE.

Most statements can be adequately optimized with a reasonable amount of resources by using optimization class 5, which is the default query optimization class. At a given optimization class, the query compilation time and resource consumption is primarily influenced by the complexity of the query, particularly the number of joins and subqueries. However, compilation time and resource usage are also affected by the amount of optimization performed.

Query optimization classes 1, 2, 3, 5, and 7 are all suitable for general-purpose use. Consider class 0 only if you require further reductions in query compilation time and you know that the SQL statements are extremely simple.

Be aware that if you choose an optimization class that is too low for your workload, the estimated cost of the access plans may be inaccurate because certain types of statistics aren't used. In particular, frequent value and histogram statistics aren't used at optimization class 0 and 1 and histogram statistics aren't used at optimization class 3. The DB2 Information Center provides more details on the different optimization classes and how to choose the best optimization class for your workload:

<http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/topic/com.ibm.db2.luw.admin.perf.doc/doc/c0005277.html>

Statistics, statistics, statistics!

- Always collect catalog statistics!
 - RUNSTATS command (manual)
 - Use these options:



```
RUNSTATS ON TABLE <schema>.<table_name> WITH DISTRIBUTION
AND SAMPLED DETAILED INDEXES ALL
```

- Consider **automatic** and **real-time statistics** collection
- Consider advanced statistics for specific query characteristics:
 - Column group statistics
 - LIKE statistics
 - Statistical views

Accurate database statistics are critical for query optimization. You should perform RUNSTATS regularly on any tables critical to query performance. You might also want to collect statistics on system catalog tables, if an application queries these tables directly and if there is significant catalog update activity such as DDL statements. You can enable automatic statistics collection to allow the DB2 data server to automatically perform RUNSTATS. You can also enable real-time statistics collection to allow the DB2 data server to provide even more timely statistics by collecting them immediately before queries are optimized.

If collecting statistics manually using RUNSTATS, you should use the following options at a minimum:

```
RUNSTATS ON TABLE DB2USER.DAILY_SALES WITH DISTRIBUTION
AND SAMPLED DETAILED INDEXES ALL
```

Distribution statistics make the optimizer aware of data skew. Detailed index statistics provide more details about the I/O required to fetch data pages when the table is accessed using a particular index. Collecting detailed index statistics consumes considerable CPU and memory for large tables. The SAMPLED option provides detailed index statistics with nearly the same accuracy but requires a fraction of the CPU and memory. These defaults are also used by automatic statistics collection when a statistical profile has not been provided for a table.

To improve query performance, consider collecting more advanced statistics such as column group statistics, LIKE statistics or creating statistical views.

Column group statistics

- Column group statistics are used to detect and correct for data correlation

POLICY

Primary key (POLICY_NO, POLICY_REV)

Column	Cardinality
POLICY_NO	10000
POLICY_REV	20

CLAIMS

Column	Cardinality
POLICY_NO	5000
POLICY_REV	10
CLAIM_ID	40000
CLAIMANT_ID	8000

Strong correlation -> not every policy has 20 revisions

Provided by FULLKEYCARD of index:
(POLICY_NO, POLICY_REV) = 40000

Strong correlation -> not every claimant has a claim for every policy

Provided by column group statistic:
(CLAIM_ID, CLAIMANT_ID)

Column group cardinality statistics represent the number of distinct values in a group of columns. This statistic helps the optimizer recognize when the data in columns is correlated. Without it, the optimizer assumes that the data in columns is independent.

Correlation – local equality predicates

Consider local predicates:

P3: CLAIM.CLAIM_ID = 90176899

P4: CLAIM.CLAIMANT_ID = 00799

Individual selectivities are:*

$\text{Sel}(P3) = 1 / \text{colcard}(\text{CLAIM.CLAIM_ID})$

$= 1 / 40000$

$\text{Sel}(P4) = 1 / \text{colcard}(\text{CLAIM.CLAIMANT_ID})$

$= 1 / 8000$

Assuming the columns are independent, combined selectivity is:

$\text{Sel}(P3) * \text{Sel}(P4) = 1/32000000$

No index, no FULLKEYCARD available. But there are only 35000 combinations, in reality.

Table access cardinality is underestimated by a factor of ~9100!!

* Assuming uniformity

Statistical views

- Statistics are associated with the view
- Query does not need to reference the view
- Materialized Query Table (MQT) matching technology matches query to statistical view
- View is NOT materialized
- Statistics on the view are used to 'adjust' selectivity estimates for predicates in query

The DB2 cost-based optimizer uses an estimate of the number of rows – or cardinality – processed by an access plan operator to accurately cost that operator. This cardinality estimate is the single most important input to the optimizer's cost model, and its accuracy largely depends upon the statistics that the RUNSTATS utility collects from the database. The statistics described above are all important for computing an accurate cardinality estimate, however there are some situations where more sophisticated statistics are required. In particular, more sophisticated statistics are required to represent more complex relationships, such as comparisons involving expressions (for example, $\text{price} > \text{MSRP} + \text{Dealer_markup}$), relationships spanning multiple tables (for example, $\text{product.name} = \text{'Alloy wheels'}$ and $\text{product.key} = \text{sales.product_key}$), or anything other than predicates involving independent attributes and simple comparison operations. Statistical views are able to represent these types of complex relationships because statistics are collected on the result set returned by the view, rather than the base tables referenced by the view.

When a query is compiled, the optimizer matches the query to the available statistical views. When the optimizer computes cardinality estimates for intermediate result sets, it uses the statistics from the view to compute a better estimate.

Queries do not need to reference the statistical view directly in order for the optimizer to use the statistical view. The optimizer uses the same matching mechanism used for materialized query tables (MQTs) to match queries to statistical views. In this respect, statistical views are very similar to MQTs, except they are not stored permanently, so they do not consume disk space and do not have to be maintained.

Information Management IBM

So how do we fix query B?

```

SELECT CATEGORY_DESC, SUM(PERCENT_DISCOUNT),
      SUM(EXTENDED_PRICE),
      SUM(SHELF_COST_PCT_OF_SALE)
FROM PERIOD, DAILY_SALES, PRODUCT, STORE,
      PROMOTION
WHERE PERIOD.PERKEY=DAILY_SALES.PERKEY AND
      PRODUCT.PRODKEY=DAILY_SALES.PRODKEY AND
      STORE.STOREKEY=DAILY_SALES.STOREKEY AND
      PROMOTION.PROMOKEY=DAILY_SALES.PROMOKEY AND
CALENDAR_DATE BETWEEN '04/01/2004' AND
      '04/14/2004' AND
STORE_NUMBER='01' AND
PROMODESC = 'Web' AND
PACKAGE_SIZE = '16 OZ' AND
SUB_CATEGORY = 747
GROUP BY CATEGORY_DESC ;

```

- Notice the query is over a star schema
- There is often:
 - Skew in the fact table foreign keys
 - Many more primary keys than foreign keys
- **Statistical views** will allow the optimizer to see these characteristics

Query B

17 © 2011 IBM Corporation

Start by understanding the underlying schema used by the query. In this example, it is a star schema, which has characteristics that can sometimes pose query optimization challenges. You can use statistical views to capture more complex relationships across the fact and dimension tables. This makes the optimizer aware of skew in the fact table foreign key columns and that the fact table foreign keys only contain a subset of the dimension primary key values.

Fixing query B1

- **A simple approach for statistical views in a star schema***:

- Create a statistical view for each dimension-fact join
- Limit to dimensions with:
 - Skew in the fact table foreign key columns
 - Many more dimension ids than exist in fact table

- **Correct for data correlation between columns**

```
PACKAGE_SIZE = '16 OZ' AND  
SUB_CATEGORY = 747
```

- PACKAGE_SIZE has 3680 distinct values
- SUB_CATEGORY has 160 distinct values
- There are 5000 distinct combinations
- **The optimizer thinks there are $3680 * 160 = 588800$!**
- **Collect column group statistics on the PRODUCT table**

*Statistical views are very general and can be used for many types of schemas and queries

Fixing query B1 with statistical views

Create a statistical view for:

- (store - daily_sales)
- (product - daily_sales)
- (period - daily_sales)

```
CREATE VIEW DB2DBA.SV_STORE AS
(SELECT S.*
 FROM STORE S, DAILY_SALES F
 WHERE S.STOREKEY = F.STOREKEY)

CREATE VIEW DB2DBA.SV_PRODUCT AS
(SELECT P.*
 FROM PRODUCT P, DAILY_SALES F
 WHERE S.PRODKEY = F.PRODKEY)

CREATE VIEW DB2DBA.SV_PERIOD AS
(SELECT P.*
 FROM PERIOD P, DAILY_SALES F
 WHERE S.PERKEY = F.PERKEY)
```

Include all dimension columns
Don't need to include fact table columns

A statistical view is created by first creating a view and then enabling it for optimization using the ALTER VIEW statement. RUNSTATS is then run on the statistical view, populating the system catalog tables with statistics for the view.

Fixing query B1 with statistical views

Enable statistical views for query optimization

```
ALTER VIEW DB2DBA.SV_STORE ENABLE QUERY OPTIMIZATION  
ALTER VIEW DB2DBA.SV_PRODUCT ENABLE QUERY OPTIMIZATION  
ALTER VIEW DB2DBA.SV_PERIOD ENABLE QUERY OPTIMIZATION
```

Gather statistics for the statistical views:

```
RUNSTATS ON TABLE DB2DBA.SV_STORE WITH DISTRIBUTION  
RUNSTATS ON TABLE DB2DBA.SV_PRODUCT WITH DISTRIBUTION  
RUNSTATS ON TABLE DB2DBA.SV_PERIOD WITH DISTRIBUTION
```

Fixing query B1 with a column group statistic

- Note the nested parentheses

```
RUNSTATS ON TABLE DB2INST1.PRODUCT ON ALL COLUMNS  
AND COLUMNS ((PACKAGE_SIZE, SUB_CATEGORY))  
WITH DISTRIBUTION  
AND SAMPLED DETAILED INDEXES ALL
```

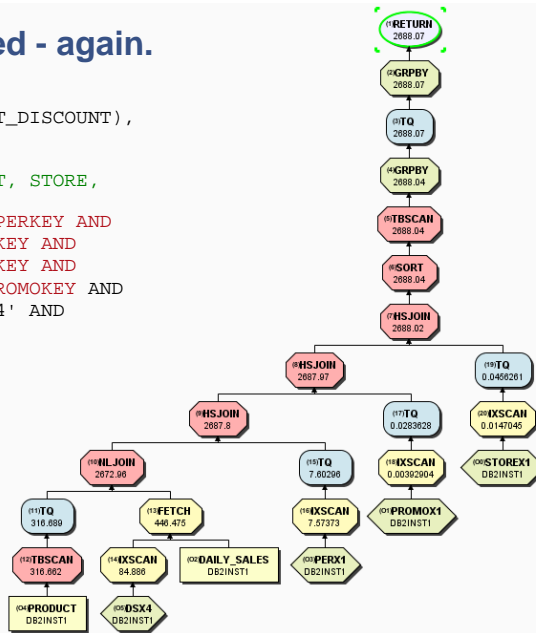
- **Now query B and B₁ run in 10s !!**
 - Previously B was 15s and B₁ was 500s

The access plan changed - again.

```

SELECT CATEGORY_DESC, SUM(PERCENT_DISCOUNT),
       SUM(EXTENDED_PRICE),
       SUM(SHELF_COST_PCT_OF_SALE)
FROM PERIOD, DAILY_SALES, PRODUCT, STORE,
       PROMOTION
WHERE PERIOD.PERKEY=DAILY_SALES.PERKEY AND
      PRODUCT.PRODKEY=DAILY_SALES.PRODKEY AND
      STORE.STOREKEY=DAILY_SALES.STOREKEY AND
      PROMOTION.PROMOKEY=DAILY_SALES.PROMOKEY AND
      CALENDAR_DATE BETWEEN '04/01/2004' AND
      '04/14/2004' AND
      STORE_NUMBER='01' AND
      PROMODESC = 'Web' AND
      PACKAGE_SIZE = '17 OZ' AND
      SUB_CATEGORY = 747
GROUP BY CATEGORY_DESC ;
    
```

The same access plan is chosen for both queries.



Access plan stability

- Unfortunately, there isn't a happy ending to every story
- Sometimes a more direct approach is required to prevent access plans from changing

Unfortunately, occasionally access plans continue to unexpectedly change for the worse, even though you've taken the tuning actions described above. Sometimes it is necessary for you to override the optimizer's decisions, in order to quickly correct a poorly performing access plan or to reduce the risk of an existing access plan changing. If you've properly tuned your SQL statements and your DB2 data server, you should only need to override the optimizer in exceptional situations. Nonetheless, it is important for you to be aware of the different ways to override the optimizer, so that you can react quickly in emergency situations, or so that you can prevent emergency situations from occurring at all.

Access plan stability

- Control access plan changes to avoid performance degradation
- Near-term strategy:
 - Prevent access plan changes during statement execution
 - Provides more consistent performance
 - But may miss opportunity for better access plan
- Longer-term strategy:
 - Allow access plans changes but automatically revert to a previously known good access plan
- Ultimate goal
 - All access plan changes should be good
 - This is not likely achievable for all scenarios
 - Access plan stability 'safety net' will still be necessary



'Access plan stability' refers to mechanisms used to control when access plans change. In the near-term, the DB2 data server provides a way to prevent access plans from changing from their last known access path. This essentially "locks down" the access path. The mechanism is different for static and dynamic SQL and the details are provided on later slides. While locking down an access plan provides more consistent performance, it may miss future opportunities for choosing better performing access plans. In the longer-term, the DB2 data server strategy will be to let access plans change based on changes in statistics and system configuration, in order to produce better performing access plans. However, if the DB2 data server detects that a new access plan is sub-optimal, it will automatically revert to the previous access plan.

The ultimate goal is that access plan changes never result in performance regressions, however this isn't likely achievable for all scenarios, considering the infinite complexity of SQL. So some form of access plan stability 'safety net' will always be necessary.

Access plan stability

- Prevent access plan changes
 - 'Lockdown' access plans for workloads
 - **Dynamic SQL - IBM Optim pureQuery Runtime for Linux, UNIX and Windows**
 - Automatically convert dynamic SQL to static
 - Supports Java and .NET applications
 - Available now
 - **Static SQL**
 - Inherently stable, except across rebind and bind
 - The DB2 data server provides an option to reuse access plans
 - **New DB2 9.7 feature**
 - 'Repair' access plans for individual queries
 - Use native data server 'hints' capabilities
 - DB2 – optimization profiles

Access plans for dynamic SQL statements can be 'locked down' using the IBM Optim pureQuery Runtime application. This application automatically converts dynamic SQL to static SQL based on running your application in a profiling mode. You can read more about it here:

- <http://www-01.ibm.com/software/data/optim/purequery-runtime/>

- <http://www-01.ibm.com/software/data/optim/purequery-platform/>

Previously known as IBM Data Studio pureQuery Runtime, Optim pureQuery Runtime provides a high-performance data access platform.

- Optim pureQuery Runtime provides an innovative approach to building high quality, better performing database applications.
- Simplifies the use of best practices for SQL and JDBC, improving application performance.
- Facilitates developer and DBA collaboration to improve the security, performance, and manageability of your Java or .NET applications.
- Increases database system throughput and reduces costs.
- With 2.2, Optim pureQuery Runtime supports literal consolidation and Oracle databases.
- [Also available for native z/OS deployments](#)
- In house testing shows pureQuery Static APIs doubled the throughput of the DB2 LUW database server in comparison to JDBC. In addition, compared to JDBC, pureQuery Client Optimization Static and pureQuery API Static (Method Style) increased the throughput of the DB2 LUW database server by 60 and 66% respectively.

Lock in access plans for guaranteed service levels

Access plans for static SQL are inherently more stable than those for dynamic SQL, except across binding and rebinding. The 'access plan reuse' option provides access plan stability for static SQL.

Access plan stability for static SQL

- APREUSE BIND/REBIND option
 - Preserve the access path across BIND and REBIND:

```
REBIND DB2USER.APP1 APREUSE YES
BIND APP1 APREUSE YES
CALL SYSPROC.REBIND_ROUTINE_PACKAGE
('P','UPDATE_EMPLOYEE', 'APREUSE YES')
```
 - Why do a REBIND?
 - Rebinding is necessary after a major release upgrade
 - Physical design change e.g. new index, change in table organization scheme
 - Take advantage of runtime improvements
 - More typically – to update the access plans based on new statistics
 - Can preserve access paths across BIND
 - If statements aren't added or removed from the package
 - Statement text must match exactly

Access plan reuse is a new feature available in DB2 9.7 that allows you to request that the access plans for static SQL statements be preserved across binds or rebinds.

In some situations, you might choose to rebind your packages in order to update the access plans to reflect changes in catalog statistics or changes in the DB2 data server configuration. However, sometimes existing packages need to be rebound because of functional changes, such as a release upgrade or the installation of a new service level, but you don't want the existing access plans to change. When a new DB2 data server release is installed, existing packages must be rebound in order for access sections to be rebuilt, so that they are compatible with the runtime component of the new data server release. When a new service level is installed, existing packages don't need to be rebound because access sections are compatible across service levels for the same release. However, you may want to rebuild existing access sections, to take advantage of runtime improvements in the new service level.

Sometimes existing packages need to be rebound because of changes in database objects referenced by SQL statements in the packages. Changes to the database objects may have implicitly invalidated existing packages, so they must be rebound before they can be used. You can rebind invalidated packages yourself using the BIND or REBIND commands or you can let the DB2 data server automatically rebind the packages the next time the application attempts to use the package.

Ideally, access plans should change for the better when a package is rebound, but sometimes you cannot take that risk, because you aren't able to react fast enough to unexpected access plan changes.

APREUSE option

- Important aspects of access paths are preserved:
 - Base access methods (including which index)
 - Join order, join method
 - TQ method
 - Aggregation method
 - Star join plans
- Access paths may not be preserved, if schema changes too much
 - e.g. drop an index, change MQT definition, compile at different opt. level
- APREUSE is not supported for upgrade to DB2 9.7
- If access path can't be preserved:
 - Compilation succeeds with SQL20516W and a reason code
 - Explain diagnostic messages may provide more information
- Catalogs indicate APREUSE packages
 - SYSCAT.PACKAGES.APREUSE = 'Y'

You can enable access plan reuse through the ALTER PACKAGE statement, or by using the APREUSE option on the BIND, REBIND, or PRECOMPILE command. Packages that are subject to access plan reuse have the value 'Y' in the APREUSE column of the SYSCAT.PACKAGES catalog view.

Access plan reuse is most effective when changes to the schema and compilation environment are kept to a minimum. If significant changes are made, it might not be possible to recreate the previous access plan. Examples of such significant changes include dropping an index that is being used in an access plan, or recompiling an SQL statement at a different optimization level. Significant changes to the query compiler's analysis of the statement can also result in the previous access plan no longer being reusable.

Access plans from packages that were produced by releases prior to DB2 9.7 cannot be reused.

If an access plan cannot be reused, compilation continues, but a warning (SQL20516W) is returned with a reason code that indicates why the attempt to reuse the access plan was not successful. Additional information is sometimes provided in the diagnostic messages that are available through the explain facility.

You can read more about access plan reuse in the DB2 Information Center:

<http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/topic/com.ibm.db2.luw.admin.perf.doc/doc/c0055383.html>

Access plan stability

- Ability to alter a package to specify:
 - Access plan reuse option (APREUSE)
`ALTER PACKAGE DB2USER.EMPADMIN APREUSE`
`REBIND DB2USER.EMPADMIN`
 - Optimization profile
`ALTER PACKAGE DB2USER.EMPADMIN`
`OPTIMIZATION PROFILE DB2USER.JOINHINT`
 - Immediately affects subsequent dynamic SQL for that package
 - Affects static SQL on next REBIND

The ALTER PACKAGE statement can also be used to change the optimization profile for a specific package.

Access plan stability

- **Routines, functions and triggers have packages too!**
 - New to V9.7:
 - PL/SQL triggers and functions are compiled and have packages
 - Regular triggers and functions are compiled if the ATOMIC keyword is not specified
 - Use the ALTER_ROUTINE_PACKAGE stored procedure to alter packages for compiled routines, triggers and functions e.g.
 - Alter the package for the UPDATE_EMPLOYEE stored procedure to specify access plan reuse and an optimization profile

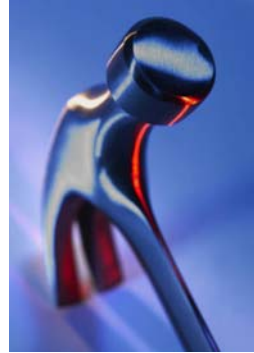
```
CALL SYSPROC.ALTER_ROUTINE_PACKAGE
('P','','','UPDATE_EMPLOYEE',
 'ACCESS PLAN REUSE YES
  OPTIMIZATION PROFILE DB2USER.INDEXHINTS')
```

The ALTER_ROUTINE_PACKAGE procedure is a convenient way for you to enable access plan reuse for compiled SQL objects, such as routines, functions, and triggers.

You can combine access plan reuse with optimization guidelines. A statement-level guideline takes precedence over access plan reuse for the static SQL statement to which it applies. Access plans for static statements that do not have statement-level guidelines can be reused if they do not conflict with any general optimization guidelines that have been specified. A statement profile with an empty guideline can be used to disable access plan reuse for a specific statement, while leaving plan reuse available for the other static statements in the package.

Access plan stability

- And sometimes you need a really big hammer...



Optimization profiles

- **Mechanism to control statement optimization**
 - Can control both query rewrite optimization and access path optimization
- **Sets of explicit optimization guidelines (DML statements)**
 - “For app1.0, only consider routing to MQTs: Newt.AvgSales and Newt.SumSales”
 - “Use index ISUPPKEY to access SUPPLIERS in the subquery of query g”
- **Can be put into effect without editing application code**
 - Compose optimization profile, add to DB, rebind targeted packages
- **Should only be used after all other tuning options exhausted**
- **Available since DB2 9**

The DB2 data server supports an even more direct way to influence access plans using optimization profiles. Optimization profiles allow you to specify access plan details such as base access and join methods and join order. For example, you can specify that access to a particular table should use a particular index or you can specify that two tables should be joined using the hash join method. Optimization profiles also allow you to control query rewrite optimizations such transforming certain types of subqueries to joins. You can specify the base table access methods, join methods, and join order for the entire access plan, or just a subset of the access plan.

Optimization profiles are a powerful tool for controlling access plans; however, they should be used with caution. Optimization profiles prevent access plans from adjusting to changes in your data and your environment. While this does accomplish access plan stabilization, it may be a bad approach when used for extended periods of time, because the performance improvements resulting from better access plans will never be realized. Optimization profiles are best used for exceptional situations when the tuning actions described previously in the presentation are unsuccessful in improving or stabilizing access plans.

Optimization profiles: anatomy

- **XML document**

- Elements and attributes understood as explicit optimization guidelines
- Composed and validated with Current Optimization Profile Schema (COPS)
 - sqllib/misc/DB2OptProfile.xsd
- **Profile Header** (exactly one)
 - Meta data and processing directives
- **Global optimization guidelines** (at most one)
 - Applies to all statements for which profile is in effect
 - E.g. eligible MQTs guideline defining MQTs to be considered for routing
- **Statement-level optimization guidelines** (zero or more)
 - Applies to a specific statement for which profile is in effect
 - Specifies aspects of desired execution plan

An optimization profile is specified as an XML document that you create and store in the SYSTOOLS.OPT_PROFILE table.

An optimization profile contains optimization guidelines that specify the access plan details. An optimization profile can contain optimization guidelines for one or more SQL statements. The SQL statement text is stored in the optimization profile along with the optimization guidelines. When an optimization profile is in effect for your application, each SQL statement compiled by your application will be matched to the SQL statements specified in the optimization profile. When a matching SQL statement is found in the optimization profile, the SQL compiler will use the optimization guidelines for that SQL statement while optimizing it.

<http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/topic/com.ibm.db2.luw.admin.perf.doc/doc/r0024580.html>

Sample optimization profile

```

<?xml version="1.0" encoding="UTF-8"?>
<OPTPROFILE VERSION="9.7.0">
<!--
  Global optimization guidelines section.
  Optional but at most one.
-->
<OPTGUIDELINES>
  <MQT NAME="DBA.AvgSales"/>
  <MQT NAME="DBA.SumSales"/>
</OPTGUIDELINES>
<!--
  Statement profile section.
  Zero or more.
-->
<STMTPROFILE ID="Guidelines for TPCD Q9">
  <STMTKEY SCHEMA="TPCD">
    <![CDATA[SELECT S.S_NAME, S.S_ADDRESS, S.S_PHONE,
S.S_COMMENT FROM PARTS P, SUPPLIERS S, PARTSUPP PS
WHERE P_PARTKEY = PS.PS_PARTKEY AND S.S_SUPPKEY = PS.PS_SUPPKEY AND P.P_SIZE = 39
AND P.P_TYPE = 'BRASS' AND S.S_NATION = 'MOROCCO' AND S.S_NATION IN ('MOROCCO', 'SPAIN')
AND PS.PS_SUPPLYCOST = (SELECT MIN(PS1.PS_SUPPLYCOST) FROM PARTSUPP PS1, SUPPLIERS S1
WHERE P.P_PARTKEY = PS1.PS_PARTKEY AND S1.S_SUPPKEY = PS1.PS_SUPPKEY AND
S1.S_NATION = S.S_NATION)]]>
  </STMTKEY>
  <OPTGUIDELINES>
    <IXSCAN TABID="Q1" INDEX="I_SUPPKEY"/>
  </OPTGUIDELINES>
</STMTPROFILE>
</OPTPROFILE>

```

Putting an optimization profile into effect

- Create the OPT_PROFILE table in the SYSTOOLS schema:

```
CALL SYSPROC.SYSINSTALLOBJECTS('OPT_PROFILES', 'C',  
    CAST (NULL AS VARCHAR(128)), CAST (NULL AS VARCHAR(128)))
```

- Prior to DB2 9.5:

```
CREATE TABLE SYSTOOLS.OPT_PROFILE (  
    SCHEMA VARCHAR(128) NOT NULL,  
    NAME VARCHAR(128) NOT NULL,  
    PROFILE BLOB (2M) NOT NULL,  
    PRIMARY KEY ( SCHEMA, NAME ));
```

- Compose document, validate, insert into table with qualified name

Inserts inventory_db.xml from current directory into the SYSTOOLS.OPT_PROFILE table with qualified name "DBA"."INVENTDB"

```
File profiledata:  
"DBA","INVENTDB","inventory_db.xml"
```

```
IMPORT FROM profiledata OF DEL MODIFIED BY LOBSINFILE  
INSERT INTO SYSTOOLS.OPT_PROFILE;
```

Putting an optimization profile into effect (cont.)

- At the package level using the *optprofile* bind option
- In DB2 9.7 use ALTER PACKAGE:

```
ALTER PACKAGE DB2USER.EMPADMIN OPTIMIZATION PROFILE  
DB2USER.JOINHINT
```

Bind optimization profile "DBA"."INVENTDB" to the package "inventapp"

```
db2 prep inventapp.sqc bindfile optprofile DBA.INVENTDB  
db2 bind inventapp.bnd
```

- At the dynamic statement level: using *current optimization profile* special register

```
EXEC SQL SET CURRENT OPTIMIZATION PROFILE = 'DBA.INVENTDB';  
/* The following statements are both optimized with 'DBA.INVENTDB' */  
EXEC SQL PREPARE stmt FROM SELECT ... ; EXEC SQL EXECUTE stmt;  
EXEC SQL EXECUTE IMMEDIATE SELECT ... ;
```

```
EXEC SQL SET CURRENT SCHEMA = 'JON';  
EXEC SQL SET CURRENT OPTIMIZATION PROFILE = 'SALES';  
/* This statement is optimized with 'JON.SALES' */  
EXEC SQL EXECUTE IMMEDIATE SELECT ... ;
```

Optimization Guidelines

- **Access path guidelines**
 - Base access request
 - **Method to access a table e.g. TBSCAN, IXSCAN**
 - Join request
 - **Method and sequence for performing a join e.g. HSJOIN, NLJOIN, MSJOIN**
 - **IXAND star joins**
- **Query rewrite guidelines**
 - IN-list to join
 - Subquery to join
 - NOT EXISTS subquery to anti-join
 - NOT IN subquery to anti-join
- **General optimization guidelines**
 - REOPT (ONCE/ALWAYS/NONE)
 - DEGREE
 - QUERYOPT
 - RTS
 - MQT choices

Access Path Optimization Guidelines

- **Example:**

```
SELECT S.S_NAME, S.S_ADDRESS, S.S_PHONE, S.S_COMMENT
FROM "Tpcd".PARTS, "Tpcd".SUPPLIERS S, "Tpcd".PARTSUPP PS
WHERE
  P_PARTKEY = PS.PS_PARTKEY AND
  S.S_SUPPKEY = PS.PS_SUPPKEY AND
  P_SIZE = 39 AND P_TYPE = 'BRASS' AND
  S.S_NATION IN ('MOROCCO','SPAIN') AND
  PS.PS_SUPPLYCOST =
    (SELECT MIN(PS1.PS_SUPPLYCOST)
     FROM "Tpcd".PARTSUPP PS1, "Tpcd".SUPPLIERS S1
     WHERE "Tpcd".PARTS.P_PARTKEY = PS1.PS_PARTKEY AND
           S1.S_SUPPKEY = PS1.PS_SUPPKEY AND
           S1.S_NATION = S.S_NATION)
ORDER BY S.S_NAME;
```

```
<OPTGUIDELINES><IXSCAN TABLE='S' INDEX='I_SUPPKEY'> </OPTGUIDELINES>
```

- Choose an index access using index 'I_SUPPKEY' for access to SUPPLIERS table in main subselect
- Table is referenced using correlation name 'S'
 - TABLE attribute must reference the 'exposed' name

Access Path Optimization Guidelines

- **Example:**

```
SELECT S.S_NAME, S.S_ADDRESS, S.S_PHONE, S.S_COMMENT
FROM "Tpcd".PARTS, "Tpcd".SUPPLIERS S, "Tpcd".PARTSUPP PS
WHERE
  P_PARTKEY = PS.PS_PARTKEY AND
  S.S_SUPPKEY = PS.PS_SUPPKEY AND
  P_SIZE = 39 AND P_TYPE = 'BRASS' AND
  S.S_NATION IN ('MOROCCO', 'SPAIN') AND
  PS.PS_SUPPLYCOST =
  (SELECT MIN(PS1.PS_SUPPLYCOST)
   FROM "Tpcd".PARTSUPP PS1, "Tpcd".SUPPLIERS S1
   WHERE "Tpcd".PARTS.P_PARTKEY = PS1.PS_PARTKEY AND
         S1.S_SUPPKEY = PS1.PS_SUPPKEY AND
         S1.S_NATION = S.S_NATION)
ORDER BY S.S_NAME;
```

<OPTGUIDELINES>

<NLJOIN>

<IXSCAN TABLE="Tpcd".Parts'>

<IXSCAN TABLE="PS" />

</NLJOIN>

</OPTGUIDELINES>

Index name not provided so optimizer
chooses based on cost

- Join requests contains 2 elements – inner and outer
- Elements can be base accesses or other join requests

Nesting join requests

- **Example**

```
<OPTGUIDELINES>  
  <MSJOIN>  
    <NLJOIN>  
      <IXSCAN TABLE='Tpcd'.Parts'/>  
      <IXSCAN TABLE="PS"/>  
    </NLJOIN>  
  <IXSCAN TABLE='S'/>  
</MSJOIN>  
</OPTGUIDELINES>
```

- Nested loop join is on outer of merge scan join
- Nested access request elements inside a join request must reference tables in the same FROM clause of the optimized statement.
- Query rewrite optimization guidelines, as well as general optimization guidelines, can affect the optimized statement.

Forming table references

- 2 methods
 - Reference 'exposed' name in the original SQL statement
 - Use 'TABLE' attribute
 - Rules for specifying SQL identifiers apply to 'TABLE' attribute
 - Reference correlation name in the optimized SQL statement
 - Use 'TABID' attribute
 - 'Optimized' SQL is the semantically equivalent version of the statement after it has been optimized by query rewrite
 - Use the explain facility to get the optimized SQL statement
 - **NOTE: There is no guarantee that correlation names in the optimized SQL statement are stable across new releases**
- Table references must refer to a single table or they are ignored
 - i.e. no ambiguous references
- Unqualified table references are implicitly qualified by the current schema
- If both 'TABLE' and 'TABID' are specified, they must refer to the same table or they are ignored.
- Derived tables are referenced using 'TABID'

Conflicting optimization guidelines

- Example

```
<OPTGUIDELINES>
```

```
<IXSCAN TABLE='Tpcd'.PARTS' INDEX='I_PTYPE'/>
```

```
<IXSCAN TABLE='Tpcd'.PARTS' INDEX='I_SIZE'/>
```

```
</OPTGUIDELINES>
```

- Multiple optimization guidelines can't reference the same table
- The first reference is applied and the others are ignored
- If I_PTYPE doesn't exist but I_SIZE does, the guideline is still ignored

Query Rewrite Guidelines

- **Example:**

```
SELECT S.S_NAME, S.S_ADDRESS, S.S_PHONE, S.S_COMMENT
FROM "Tpcd".PARTS P, "Tpcd".SUPPLIERS S, "Tpcd".PARTSUPP PS
WHERE
  P_PARTKEY = PS.PS_PARTKEY AND
  S.S_SUPPKEY = PS.PS_SUPPKEY AND
  P_SIZE IN (35, 36, 39, 40) AND
  S.S_NATION IN ('INDIA', 'SPAIN')
ORDER BY S.S_NAME;
```

```
<OPTGUIDELINES><INLIST2JOIN TABLE='P' /></OPTGUIDELINES>
```

- INLIST2JOIN specifies that list of constants in IN list predicate should be transformed to a table expression
- Table expression can then be joined to "Tpcd".PARTS using an indexed NLJN
- Target IN-list identified by specifying table to which predicate is applied
- If there are multiple IN-lists, guideline can be further qualified with COLUMN attribute

General Optimization Guidelines

- **Example**

```
SELECT S.S_NAME, S.S_ADDRESS, S.S_PHONE  
FROM "Tpcd".SUPPLIERS S  
WHERE S.S_NATION IN (?, ?) AND S.S_SUPPKEY = ?  
ORDER BY S.S_NAME ;
```

```
<OPTGUIDELINES> <REOPT VALUE='ONCE'> </OPTGUIDELINES>
```

- 'ONCE' indicates that optimization should be deferred until the first set of variable values is provided.
- This allows the optimizer to compare the input values to the statistics to get a better selectivity estimate and a better query execution plan

Conclusion

- The key to achieving access plan stability is to address the fundamental problems in a general way:
 1. Well-written SQL
 2. Proper system configuration
 3. Statistics, statistics, statistics !
- Provides a 'firm foundation' for all queries
- Handle exceptions with access plan lockdown techniques
 - Access plan lockdown
 - Optimization profiles

The DB2 data server query optimizer is more likely to choose optimal access plans consistently if you provide it accurate information about your data and system configuration. Access plans will also tend to be more stable because they will be less sensitive to small changes in your data and system configuration. How you write your SQL statements also affects the optimizer's ability to choose optimal access plans. Following the tuning actions described earlier in this presentation should help you achieve consistently good query performance. However, for the exceptional situations in which the described tuning actions don't have the desired effect, you can override the optimizer using access plan reuse or optimization profiles. The overall combination of system tuning actions and selective optimizer overrides should help you achieve your DB2 data server query performance objectives.

Appendix

Putting an optimization profile into effect

- Clearing the special register:

```
EXEC SQL SET CURRENT OPTIMIZATION PROFILE = NULL;  
/* The following statement is optimized with the setting of the OPTPROFILE bind option */  
EXEC SQL PREPARE stmt FROM SELECT ... ; EXEC SQL EXECUTE stmt;
```

```
EXEC SQL SET CURRENT OPTIMIZATION PROFILE = "";  
/* The following statement is optimized with no optimization profile */  
EXEC SQL PREPARE stmt FROM SELECT ... ; EXEC SQL EXECUTE stmt;
```

- At the dynamic statement level: using *db2_optprofile* CLI option

-- after each successful connect to the SANFRAN database, the CLI client would issue the command:

```
SET CURRENT OPTIMIZATION PROFILE=JON.SALES.
```

```
[SANFRAN]  
DB2_OPTPROFILE JON.SALES
```

Putting an optimization profile into effect

- SQL procedures

```
CALL SET_ROUTINE_OPTS('OPTPROFILE DBA.INVENTDB ') %
```

```
CREATE PROCEDURE MY_PROC  
BEGIN  
    DECLARE CUR1 CURSOR FOR SELECT ...  
END %
```

- SQL may be modified during CREATE PROCEDURE processing
- Use explain facility or query system catalogs to get modified SQL statements to include in optimization profile STMTKEY element for profile statement matching

```
SELECT STMTNO, SEQNO, SECTNO, TEXT  
FROM SYSCAT.STATEMENTS AS S,  
     SYSCAT.ROUTINEDEP AS D,  
     SYSCAT.ROUTINES AS R  
WHERE PKGSHEMA = BSHEMA  
      AND PKGNAME = BNAME;  
      AND BTYPE = 'K'  
      AND R.SPECIFICNAME = D.SPECIFICNAME  
      AND R.ROUTINESCHAME = D.ROUTINESCHAMA  
      AND ROUTINENAME = ?  
      AND ROUTINESCHAMA = ?  
      AND PARM_COUNT = ?  
ORDER BY STMTNO
```

- STMTNO should be the line number in the source code of the CREATE PROCEDURE, relative to the beginning of the procedure statement (line number 1)

Table references in views

- **Example**

```
CREATE VIEW "DBGuy".V1 as (SELECT * FROM EMPLOYEE A WHERE SALARY > 50,000) ;
```

```
CREATE VIEW DB2USER.V2 AS (SELECT * FROM "DBGuy".V1 WHERE DEPTNO IN ('52', '53','54') ;
```

```
SELECT * FROM DB2USER.V2 A WHERE V2.HIRE_DATE > '01/01/2004' ;
```

```
<OPTGUIDELINES><IXSCAN TABLE='A'/"DBGuy".V1/A'></OPTGUIDELINES>
```

- Extended syntax allows unambiguous table references in views
 - 'A' is ambiguous
- Extended name consists of exposed names in the path from the statement reference to the nested reference separated by slashes
- Same rules for exposed names apply to extended syntax

Table references in views

- Extended syntax is not necessary if references are unique with respect to all table references in the query

- **Example**

```
CREATE VIEW "DBGuy".V1 AS (SELECT * FROM EMPLOYEE E WHERE SALARY > 50,000) ;
```

```
CREATE VIEW DB2USER.V2 AS (SELECT * FROM "DBGuy".V1 WHERE DEPTNO IN ('52', '53','54') ;
```

```
SELECT * FROM DB2USER.V2 A WHERE V2.HIRE_DATE > '01/01/2004' ;
```

```
<OPTGUIDELINES><IXSCAN TABLE='E'></OPTGUIDELINES>
```

Ambiguous table references

- **Example**

```
CREATE VIEW V1 AS  
(SELECT * FROM EMPLOYEE WHERE SALARY >  
(SELECT AVG(SALARY) FROM EMPLOYEE);
```

```
SELECT * FROM V1 WHERE DEPTNO IN ('M62', 'M63') ;  
<OPTGUIDELINES><IXSCAN TABLE='V1/EMPLOYEE'></OPTGUIDELINES>
```

- Which EMPLOYEE reference?
- The IXSCAN request is ignored
- Uniquely identify EMPLOYEE by adding correlation names in the view
- Use TABID
 - Correlation names in the optimized SQL are always unique