“This query used to run in 5 seconds and now takes an hour! How could the Optimizer do this to me?” Sounds familiar? But are you sure the Optimizer is at fault?

In this session, we will examine the challenges faced by cost-based optimization and identify steps to minimize, if not eliminate, such behavior. We will also cover various ways of influencing and stabilizing the optimal access path.

We will transition from the “Why doesn’t IBM?” complaint to the more positive “What can I do TODAY?” action plan.
Session Outline

1. Cost-based optimizer basics
   • Filter factors
   • Issues in access path creation

2. Stats, stats and even more stats
   • Basic stats
   • NUD
   • Correlation
   • Histogram

3. Case studies
   • Case 1 – Who's on first? (Basic stats)
   • Case 2 – Filtering matters! (Basic stats)
   • Case 3 – Averages can be deceiving! (NUD stats)
   • Case 4 – Are we related? (Correlation stats)
   • Case 5 – Ranges are skewed too! (Histogram stats)
   • Case 6 - The "Pick me!" index (Index design)

4. Access path management - Influencing, stabilizing and falling back
   • Hints
   • Stats
   • Tricks
   • Package stability

What does the future hold?
   • Directions
   • My wish list
Warning!

- 89 slides, oh my!
- I do not intend to cover each one – several, which show the detail or are basic in nature, will be skipped.

This is my excuse for fast-forwarding when I am in time trouble (as always!).
Suresh Sane is an IDUG Hall of Fame speaker with two Best User Speaker awards and numerous top 10 finishes. He has lectured worldwide and co-authored 4 IBM Redbooks (Dynamic SQL, Stored Procedures, Data Integrity and DB2 Packages). He was recognized as an IBM Information Champion in 2009 and in 2010.

He is a long time IDUG volunteer and was the Conference Chair for IDUG NA 2008. He currently serves on the IDUG Board of Directors.

Contact Information:

sssane@dstsystems.com

Suresh Sane
DST Systems, Inc.
1055 Broadway
Kansas City, MO 64105
USA

(816) 435-3803
If you have ever invested in a mutual fund, have had a prescription filled, or are a cable or satellite television subscriber, you may have already had dealings with our company.

DST Systems, Inc. is a publicly traded company (NYSE: DST) with headquarters in Kansas City, MO. Founded in 1969, it employs about 10,000 associates domestically and internationally.

The three operating segments - Financial Services, Output Solutions and Customer Management - are further enhanced by DST’s advanced technology and e-commerce solutions.
In this section we will cover the basics and explain how the optimizer determines the optimal access path.
The basics of how the optimizer creates an access path and externalizes it in a plan_table(s) when using the EXPLAIN option.
Some of the considerations used by the optimizer for access path selection.

The basic steps of the optimization process. It is easy to underestimate the magnitude of this task. The number of access paths which are theoretically possible increases exponentially as the number of tables and indexes increase.

To me, it is not surprising that it picks a less than optimal access path in rare cases. What is surprising is that it picks the right access path almost all the time (97% or better according to one estimate)!
Simple Filter Factor Example

Filter Factor = \frac{\text{Number of result rows}}{\text{Number of source rows}}

- Calculate the filter factor:
  1/\text{COLCARD of LAST\_NAME}
  1/1000 = .001
  1/\text{COLCARD of DEPT}
  1/10 = .1
- AND (multiply) the predicates together:
  .001 \times .1 = .0001

Calculations the optimizer uses for combining filter factors.
Clustering has to be taken into account since a clustered index matches the order of data in the table and can result in a far fewer page “touches” – a concept we will explore further later on.

**Effect of clustering**

- Filtering using an index affects the number of data rows that must be accessed (assuming no index-only access)
- Clustering has a huge impact on the number of data pages that must be accessed
- Example – 50 rows using a non-clustering index could mean 50 getpages while 100 rows using a clustering index could mean just 1 or 2 getpages
Effect of working with an unknown value (host variable or parameter marker) vs. a known value (a literal).

**Literals and REOPT**
- With host variables for static SQL and parameter markers for dynamic SQL, the exact value at bind/prepare time is not known. REOPT allows the creation of an access path at runtime:
  - REOPT (NONE) – default
  - REOPT (ALWAYS) – for static and dynamic SQL
  - REOPT (ONCE) – for dynamic SQL
  - REOPT (AUTO) – DB2 9 for dynamic SQL
- Specifying literals:
  - For static, better performance but hard-coding may necessitate SQL changes when business needs change
  - For dynamic, better access path but may lead to no cache reuse
In this section, we will cover what stats can be gathered with the RUNSTATS utility and the scenarios where they are useful.

I will cover this section quickly, emphasizing a few key issues. I expect most of you are already familiar with the basics and the discussion is somewhat dry.
Why stats?

“Measure what is measurable, and make measurable what is not so.” - Galileo Galilei

“Not everything that can be counted counts, and not everything that counts can be counted” - Albert Einstein

“There are three kinds of lies: lies, damned lies, and statistics.” - Benjamin Disraeli

…and more to the point.. for DB2:

“What I mean by ‘all optimizers are hungry’... We push collection of statistics so the optimizer more accurately estimates the cost ... The better we estimate the available choices, the better able we are to choose the cheapest one.” - Patrick Bossman, IBM, on IDUG DB2-L June 26, 2007

Why should you care about stats?
Table statistics

- CARDF
  - Number of rows in the table or partition
- NPAGESF
  - Number of pages containing data
- NACTIVEF
  - Number of active pages in the tablespace
  - Only used for single table simple table spaces
- PCTROWCOMP
  - Percentage of compressed rows

Some of the critical catalog statistics for tables and table spaces. Notice that the older columns in the table (CARD, NPAGES etc) are now obsolete.
### Index statistics

- **NLEAF**
  - Number of active leaf pages
- **NLEVELS**
  - Number of levels in the index tree
- **CLUSTERRATIOF**
  - Percentage of rows in clustering order
- **CLUSTERING**
  - Is the index the clustering index?
- **FIRSTKEYCARDF**
  - Number of distinct values for the first index column
- **FULLKEYCARDF**
  - Number of distinct values for the full index key

Important catalog statistics for the index.
Explaining the concept of non-uniform distribution and correlation.

Non-uniform distribution and correlation

- Non-uniform distribution (data skew)
  - Data is not uniformly distributed
    - Point-skewed (e.g. active = 95%, retired = 5%)
    - Range-skewed (e.g. salary range is 30K to 300K, with 90% earning less than 100K) – see Histogram statistics

- Correlation
  - Data in two or more columns is not independent
  - Affects how filter factors are multiplied
  - E.g. State and zip code (64105 exists only in MO) or make and model (Civic is made only by Honda) – 100% correlation
Catalog statistics that determine the selectivity of predicates. We will discuss each of these in detail in the following slides.
Stats that can be gathered for a single column.
Somewhat primitive measures (useful nonetheless) are the LOW2KEY and HIGH2KEY. Finer histograms are the way of the future that make these range stats more accurate.

**HIGH2KEY/LOW2KEY**

- Stored as SYSCOLUMNS.LOWER2KEY, HIGHER2KEY
- Used when
  - Interpolation for range predicates
  - LIKE, BETWEEN, < , <=, > , >=
  - Literal value is known (not for host variables)
- Can be used in combination with single column frequencies for more accurate estimates
- Histogram stats covered later
Single column frequency

- Stored as SYSCOLDIST.FREQUENCYF (TYPE = F, NUMCOLS = 1)
- Information about non-uniform distribution (data skew)
- Used when
  - Literal value is known (not for host variables)
  - EQUAL, IS NULL, IS NOT NULL
  - LIKE, BETWEEN, <, <=, >, >=
- Collected by issuing:
  - RUNSTATS INDEX(..) COLUMNS(..) – leading cols
  - RUNSTATS INDEX(.. FREQVAL NUMCOLS(..) COUNT(..) how many columns and how many values
  - V8 allows this for any columns with the COLGROUP keyword

Keep in mind the COLGROUP keyword introduce in V8 which makes this more powerful.
Keep in mind the COLGROUP keyword introduced in V8 which makes this more powerful.
Keep in mind the COLGROUP keyword introduced in V8 which makes this more powerful.
Example of how the additional stats help the Optimizer.
Details of what stats are collected.
Note – these are equal-height, not equal-width.

The graph at the left is a visual representation of what the optimizer “sees” – even though the catalog does not contain such slices of data.
General recommendations for RUNSTATS

• Collect basic stats (table and index) always.
• Focus on columns in WHERE clauses and in ORDER BY clauses (DSN_PREDICAT table of extended explain).
• Collect non-uniform distribution stats when data skew is present.
• Collect correlation stats when two or more columns are highly correlated.
• Collect histogram stats when range skew and range predicates exist.
• Remember how “hungry” the Optimizer is!

General advice on what stats should be collected.
In this section, we will cover 6 examples. We will begin with a simple example to show why basic stats are necessary. We will build on this to explore the impact of non-uniform distribution, correlation and range skew. Finally, we will cover how an “irresistible” index – one good for all seasons – can be created.

Keep in mind that the cases discussed here are sterile in nature – real life scenarios are far more complex. They are intended to illustrate the need for accurate and detailed stats and to show how a “perfect” index can be discovered.
Case studies

- Case 1 – Who’s on first? (Basic stats)
- Case 2 – Filtering matters! (Basic stats)
- Case 3 – Averages can be deceiving! (NUD stats)
- Case 4 – Are we related? (Correlation stats)
- Case 5 – Ranges are skewed too! (Histogram stats)
- Case 6 - The "Pick me!" index (Index design)
Case 1 – Who's on first?

OPT1ALL

<table>
<thead>
<tr>
<th>EMPID</th>
<th>EMPNAME</th>
<th>...</th>
</tr>
</thead>
</table>

K0, PK, Cluster EMPID

All employees - 262,144 rows

OPT1VIP

<table>
<thead>
<tr>
<th>EMPID</th>
<th>EMPNAME</th>
<th>...</th>
</tr>
</thead>
</table>

K0, PK, Cluster EMPID

All "highly compensated" employees - 100 rows

Table structure and stats for Case 1.
Case 1 – Who's on first?

SQL

```
SELECT *  
FROM   OPT1ALL A, OPT1VIP B  
WHERE  A.EMPID = B.EMPID
```

RUNSTATS BEFORE
Nonc

RUNSTATS AFTER

```
RUNSTATS TABLESPACE(..) INDEX(ALL)
```
Without basic RUNSTATS:

The bad access path which scans the larger table (262,144 rows) instead of the smaller table (100 rows)

With basic RUNSTATS:

The better access path scanning the smaller table first.
Table structure and stats for Case 2.
Case 2 – Filtering matters!

```sql
SELECT * FROM OPT2CRS C, OPT2ENR E, OPT2STUS
WHERE C.COURSE_ID = E.COURSE_ID
AND S.STUDENT_ID = E.STUDENT_ID
AND C.COURSE_NAME LIKE 'DB2%'
AND S.STUDENT_NAME LIKE 'S%'

RUNSTATS BEFORE
None

RUNSTATS AFTER
RUNSTATS TABLESPACE(--) INDEX(ALL)
```

SQL used for testing.
Without basic RUNSTATS:

The bad access path using a non-matching index scan against the largest table (OPT2ENR) due to the lack of knowledge about its size.

With basic RUNSTATS:

The better access path which identifies the qualifying students first, the qualifying courses next and then access the largest table using an index.
Table structure and stats for Case 3. Note the non-uniform distribution of columns STATUS and CLASS_TYPE.
Case 3 – Averages can be deceiving!

SQL

```
SELECT * 
FROM OPTCASE3 
WHERE LOCATION_ID = ? 
AND COURSE_ID = ? 
AND STATUS = 'X' 
AND CLASS_TYPE = 'L'
```

All cancelled classes (status = X) which are labs (class_type = L)

SQL used for testing.
Case 3 – Averages can be deceiving!

RUNSTATS BEFORE

RUNSTATS TABLESPACE(..) INDEX(ALL)

RUNSTATS AFTER

RUNSTATS TABLESPACE(…) TABLE(…)
COLGROUP(STATUS) FREQVAL COUNT 10
COLGROUP(CLASS_TYPE) FREQVAL COUNT 10
Case 3 – Averages can be deceiving!

Without NUD RUNSTATS

<table>
<thead>
<tr>
<th>OB</th>
<th>PL</th>
<th>M X</th>
<th>M T</th>
<th>TAB</th>
<th>AC</th>
<th>INDEX</th>
<th>MC</th>
<th>IO</th>
<th>PF</th>
<th>UJOGUJOG</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>OPTCASE3</td>
<td>M</td>
<td></td>
<td>0</td>
<td>L</td>
<td></td>
<td>NNNNNNNNN</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>OPTCASE3</td>
<td>MX</td>
<td>OPTCASE3K2</td>
<td>3</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td>NNNNNNNNN</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>OPTCASE3</td>
<td>MX</td>
<td>OPTCASE3K1</td>
<td>2</td>
<td>Y</td>
<td>B</td>
<td></td>
<td></td>
<td>NNNNNNNNN</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>OPTCASE3</td>
<td>MI</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>NNNNNNNNN</td>
</tr>
</tbody>
</table>

With NUD RUNSTATS

<table>
<thead>
<tr>
<th>OB</th>
<th>TIL</th>
<th>M X</th>
<th>M T</th>
<th>TAB</th>
<th>AC</th>
<th>INDEX</th>
<th>MC</th>
<th>IO</th>
<th>IT</th>
<th>UJOGUJOG</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>OPTCASE3</td>
<td>I</td>
<td>OPTCASE3K2</td>
<td>3</td>
<td></td>
<td></td>
<td>NNNNNNNNN</td>
</tr>
</tbody>
</table>

Without basic RUNSTATS:

The bad access path using a multi-index access with indexes K2 and K1 due to the lack of knowledge about the non-uniform distribution of the data.

With basic RUNSTATS:

The better access path which uses only index K2, which is now known to qualify only a few rows.
Case 4 – Are we related?

**OPTCASE4**

<table>
<thead>
<tr>
<th>VIN</th>
<th>K0, PK, Cluster VIN</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAKE_ID</td>
<td>K1, MAKE_ID, MODEL_ID, VIN</td>
</tr>
<tr>
<td>MODEL_ID</td>
<td></td>
</tr>
<tr>
<td>COLOR_ID</td>
<td>K2, COLOR_ID, VIN</td>
</tr>
</tbody>
</table>

Cars
262,144 rows

MAKE_ID - 5 values
MODEL_ID - 5 values (100% correlation with MAKE_ID)
COLOR_ID – 1,000 values

Table structure and stats for Case 4. Note the high correlation between the columns MAKE_ID and MODEL_ID.
Case 4 – Are we related?

SQL

```sql
SELECT * 
FROM OPTCASE4 
WHERE MAKE_ID = ? 
AND MODEL_ID = ? 
AND COLOR_ID = ?
```

SQL used for testing.
Case 4 – Are we related?

**RUNSTATS BEFORE**

<table>
<thead>
<tr>
<th>Command</th>
<th>Settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>RUNSTATS TABLESPACE(...) TABLE(..)</td>
<td></td>
</tr>
<tr>
<td>COLGROUP(MODEL_ID) FREQVAL COUNT 10</td>
<td></td>
</tr>
<tr>
<td>COLGROUP(MAKE_ID) FREQVAL COUNT 10</td>
<td></td>
</tr>
<tr>
<td>INDEX(ALL)</td>
<td></td>
</tr>
</tbody>
</table>

**RUNSTATS AFTER**

<table>
<thead>
<tr>
<th>Command</th>
<th>Settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>RUNSTATS TABLESPACE(...) TABLE(..)</td>
<td></td>
</tr>
<tr>
<td>COLGROUP(MODEL_ID) FREQVAL COUNT 10</td>
<td></td>
</tr>
<tr>
<td>COLGROUP(MAKE_ID) FREQVAL COUNT 10</td>
<td></td>
</tr>
<tr>
<td>INDEX(ALL) KEYCARD FREQVAL NUMCOLS 2 COUNT 10</td>
<td></td>
</tr>
</tbody>
</table>
Without basic RUNSTATS:

The bad access path using a multi-index access with indexes K2 and K1 due to the lack of knowledge about the column correlation between MAKE and MODEL.

With basic RUNSTATS:

The better access path which uses only index K2, since K1 provides little filtering.
Table structure and stats for Case 5. Note the range skew.
Case 5 – Ranges are skewed too!

SQL

```
SELECT *
FROM OPTCASE5
WHERE SALARY >= 100,000
```

RUNSTATS BEFORE

```
RUNSTATS TABLESPACE(..) INDEX(ALL)
```

RUNSTATS AFTER

```
RUNSTATS INDEX(....OPTCASE5K1)
HISTOGRAM NUMCOLS 1
```

SQL used for testing.
Without basic RUNSTATS:

The bad access path, not using K1 because it expects 92% of rows to qualify
\[(1,000,000 - 100,000) / (1,000,000 - 20,000) = 92\%\]

With basic RUNSTATS:

The better access path using K1 because it knows only 3.35 of rows qualify. Note that the calculated filter factor is NOT exact – actual data distribution is such that only 1% lie above 100,000 but depending on where the last quantile starts, DB2 over-estimates this value leading to a conservative estimate of the filtering. In my example, using the default value of 100 quantiles, DB2 chose 76.

In general, such conservative estimates are inevitable due to the “height-balanced” nature of histograms – the sparsely populated intervals tend to be wider as a result.
A review of the basic terminology. The stage at which filtering occurs determines what data must be accessed.

Case 6 – The “Pick me!” index

- Index and table access review
  - Matching
    - Defines the index slice which must be scanned
    - Based on leading columns of an index only
  - Screening
    - Reduces the number of table touches
    - Table is accessed only if all screening predicates are true (if accessed at all)
    - Based on non-leading columns of the index or any non-indexable columns
  - Result Rows
    - Actual number of rows returned to the application
Case 6 – The “Pick me!” Index

- What is a “Touch”?  
  - A single index row or single table row = 1 Touch  
  - Slice of an index = 1\textsuperscript{st} row is random touch, all others are sequential touches  
- Response time estimates from QUBE (Quick Upper Bound Estimate)  
  - Random read = 10 ms per touch  
  - Sequential read (index or table) = 10 ms + 0.01 ms per touch (this accounts for multiple rows per page and sequential prefetch)

This case is based on ideas provided in the book on index design by Lahdenmaki and Leach (see ref #5). We use the estimation formulas from QUBE (Quick Upper Bound Estimate).
Case 6 – The “Pick me!” index

Table structure for Case 6.

Note that the detailed calculations for this case cannot possibly covered in the time allotted. So, we will visit them but focus on the conclusions only.
Case 6 – The “Pick me!” Index

SQL

```
SELECT COL5
FROM OPTCASE6
WHERE COL1 = ?
AND COL3 = ?
AND COL4 = ?
ORDER BY COL5
```

SQL for testing.
Table and column cardinalities.

In general, DB2 will know (via RUNSTATS), \( \text{CARD(COL1)} \), \( \text{CARD(COL1,COL2)} \), and \( \text{CARD(COL1,COL2,COL3)} \) but probably not \( \text{CARD(COL1,COL3)} \), which is the one that is really needed to determine the correct filter factor.

The optimizer must assume \( \text{CARD(COL1,COL3)} \) is equal \( \text{CARD(COL1)} \times \text{CARD(COL3)} = 500 \times 300 = 150,000 \) while the actual \( \text{CARD(COL1,COL3)} \) is 5000.
Actual filter factors for the average case and the worst case.

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Average</th>
<th>Worst case (given)</th>
</tr>
</thead>
<tbody>
<tr>
<td>COL1=?</td>
<td>1/500 = 0.2%</td>
<td>2%</td>
</tr>
<tr>
<td>COL1=?</td>
<td>1/5000 = 0.02%</td>
<td>0.5%</td>
</tr>
<tr>
<td>AND COL3=?</td>
<td>1/250,000 = 0.0004%</td>
<td>0.005%</td>
</tr>
</tbody>
</table>
Case 6 – The “Pick me!” Index

<table>
<thead>
<tr>
<th>Step</th>
<th>TS scan</th>
<th>Index – Avg case</th>
<th>Index – Worst case</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># Touch</td>
<td>Est time</td>
<td># Touch</td>
</tr>
<tr>
<td>IX</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>IX</td>
<td></td>
<td>IX</td>
</tr>
<tr>
<td>TS</td>
<td>1M</td>
<td>1M * 0.01 ms = 10.00</td>
<td>0.02% * 1M = 200</td>
</tr>
<tr>
<td>Sort</td>
<td>0.005 * 1M = 50</td>
<td>Small</td>
<td>0.0004 % * 1M = 4</td>
</tr>
<tr>
<td>Fetch</td>
<td>50</td>
<td>Small &amp; indep.</td>
<td>4</td>
</tr>
<tr>
<td>Total</td>
<td>10.00</td>
<td>2.020</td>
<td>50.20</td>
</tr>
</tbody>
</table>

An estimate of the response time with various access path choices.
This poses a dilemma – an access path that is good on average but bad for the worst case!

Also see my discussion of “aggressive vs. defensive optimization” in the next section.
Creating the proposed “minimal-cost-fat-index”. Since this leads to an index-only access, we should fare better.

For the query in question an index by columns COL1, COL3, COL4, COL5 would be the best but we are assuming that index consisting of columns COL1, COL2, COL3 is needed and we can only add columns to the end without impacting queries which access by COL1, COL2 or by COL1, COL2, COL3.
An estimate of the response time with an index scan for the worst case – with the new index.
A good access path for all seasons! Life is good!

As explained earlier, real-life situations are far more complex and a minimal-cost-fat-index” may not always be possible or even desirable. However, the example does illustrate how a systematic analysis can lead to a consistently good performance.

It should be noted that we have assumed the Optimizer has full knowledge of the data distribution. In reality, it is likely to know even less making worse choices. The “irresistible” index virtually makes sure it cannot get it wrong!
This section will cover various methods of influencing and stabilizing the access path.
Optimization hints – why?

- Want consistency of response times across rebinds and across code migrations ("I hate change") – covered in (a) freezing the access path

- Want to temporarily bypass the access path chosen by DB2 ("I know better") – covered in (b) Obtaining a better access path.

The business drivers for hints.
While this is the normal mode of operation, you can also construct the plan_table rows from scratch and use them in a bind operation.
Pre-work for Hints

- Specify YES for the DSNZPARM OPTHINTS. If this is not set, all optimization hints are ignored by DB2.
- But... once authorized, cannot limit the use – anyone who can bind the package can apply a hint.
- Before giving hints to DB2, make sure your PLAN_TABLE is of the correct format (see ref #4).
- For best performance, create an ascending index on the following columns of PLAN_TABLE – in this order:
  - QUERYNO
  - APPLNAME
  - PROGNAME
  - VERSION
  - COLLID
  - OPTHINT

What is needed to implement hints.
QUERYNO is critical!

- For DB2 hints, the query number clause is optional but...
- If no query number is specified, DB2 uses the statement number.
- Query numbers are critical in the long run, especially for static SQL.
- Dynamic SQL - Statement # is based on application preparing it – e.g. for DSNTEP2/4, same statement # is used.
- Static SQL - In a program with embedded static SQL (e.g. CORO1), any program change (e.g. addition of a few comment lines at the top) is likely to affect the statement number and make the hint inapplicable.
### Simple Hint Examples – Steps

(a) Freezing the access path

<table>
<thead>
<tr>
<th></th>
<th>QNO</th>
<th>IX</th>
<th>MC</th>
<th>OPTHINT</th>
<th>HINT_USED</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>59</td>
<td>K2</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Apply</td>
<td>59</td>
<td>K2</td>
<td>1</td>
<td>I_HATE_CHANGE</td>
<td></td>
</tr>
<tr>
<td>Verify</td>
<td>59</td>
<td>K2</td>
<td>1</td>
<td></td>
<td>I_HATE_CHANGE</td>
</tr>
</tbody>
</table>

(b) Obtaining a better access path

<table>
<thead>
<tr>
<th></th>
<th>QNO</th>
<th>IX</th>
<th>MC</th>
<th>OPTHINT</th>
<th>HINT_USED</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>59</td>
<td>K2</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Apply</td>
<td>59</td>
<td>K1</td>
<td>1</td>
<td>I_KNOW_BETTER</td>
<td></td>
</tr>
<tr>
<td>Verify</td>
<td>59</td>
<td>K1</td>
<td>1</td>
<td></td>
<td>I_KNOW_BETTER</td>
</tr>
</tbody>
</table>

What the plan_table looks like before and after.
Indicating what hint is needed

- If dynamic, issue: SET CURRENT OPTIMIZATION HINT = ‘hint_name’;
- If static, rebind the package or plan with OPPTHINT set to hint_name.

How you indicate to DB2 what hint is to be applied.
Locating the hint

• For a PLAN_TABLE row, the QUERYNO, APPLNAME, PROGNAME, VERSION, and COLLID values must match the corresponding values for an SQL statement.

• In addition:
  • If the SQL statement is executed dynamically, the OPTHINT value for that row must match the value in the CURRENT OPTIMIZATION IIINT special register.
  • If the SQL statement is executed statically, the OPTHINT value for the row must match the value of bind option OPTHINT for the package or plan that contains the SQL statement.

How does DB2 locate the hint in the plan_table?
Validating the hint

- DB2 validates only the following PLAN_TABLE columns:
  - METHOD
  - CREATOR and TNAME
  - TARNO
  - ACCESTYPE
  - ACCESSCREATOR and ACCESSNAME
  - SORTN_JOIN and SORTC_JOIN
  - PREFETCH
  - PAGE_RANGE
  - PARALLELISM_MODE
  - ACCESS_DEGREE and JOIN_DEGREE
  - WHEN_OPTIMIZE
  - PRIMARY_ACCESSSTYPE

- If the access path you suggest cannot be enforced, all hints of that QBLOCK are discarded.
**Hint Limitations**

- Hints cannot force or undo query transformations, such as subquery transformation to join or materialization or merge of a view or table expression.
- If a query is not transformed in that release, but in a later release of DB2 it is transformed, DB2 does not use the hint in the later release.
- Be aware that a hint supplied on the PLAN_TABLE which was ignored (did not match the OPPTHINT specified) is also shown but results in return code 0 for the BIND – **must check for +394 not 0!**

```sql
D$NT222I :DBxx D$NTBBP2 REBIND WARNING
FOR PACKAGE = DBxx.xxxxxx.I HINTNEW.
USE OF OPTHINT RESULTS IN
1 STATEMENTS WHERE OPTHINT FULLY APPLIED
0 STATEMENTS WHERE OPTHINT NOT APPLIED OR PARTIALLY APPLIED
1 STATEMENTS WHERE OPTHINT IS NOT FOUND
```

Some of the limitations.
**Catalog stats manipulation**

- If you update stats, make sure you update all related stats – e.g. if the table card is increased, the stats for the associated indexes and column should be updated also.
- Can help your specific query, but other queries can be affected adversely.
- The UPDATE statements must be repeated after RUNSTATS resets the catalog values.
- If you are using dynamic statement caching, you must invalidate statements in the cache that access those tables or indexes.
  - For this, you can use:

```
RUNSTATS ..REPORT NO UPDATE NONE
```

Manipulating the catalog stats to influence the access path.
SQL “tricks”

- Some of the examples of such “tricks” are:
  - Add predicates to turn indexable predicates into stage 2 (bad) e.g.
    OR 0=1
  - Add predicates to turn indexable predicates into stage 1 non-
    indexable (good) e.g. col +0
  - Add fake redundant predicates to favor one access path over
    another
  - Add OPTIMIZE FOR n ROWS where n is artificially small or large
  - Define a table as VOLATILE to encourage index usage
  - Use CARDINALITY or CARDINALITY MULTIPLIER clause of a
    user-defined function
- “Tricks” can cause significant performance degradation if they are not carefully implemented and monitored. In case of a query re-write, the “trick” may become ineffective in a future release of DB2.

Using “tricks” to influence the access path.
Package Stability - Framework

- Ability to easily fallback to a previous (or original) copy of a version of a package (Note: Each version can have up to 3 copies, version is **NOT** the same as copy!)
- The support applies to packages — not plans — and includes non-native SQL procedures, external procedures, and trigger packages.
- Some IBM manuals refer to this feature as “Plan stability” since it deals with the stability of access paths (an “access plan”).
- I prefer the term package stability since this option is available for packages (not plans).

The infrastructure needed.
Controlling package stability

- The option can be controlled at two levels:
  - Subsystem level via a new DSNZPARM PLANMGMT (suggest not setting this to use BASIC or EXTENDED!)
  - BIND level with new options for REBIND
- Possible settings:
  - PLANMGMT(OFF) 1 copy
  - PLANMGMT(BASIC) – 2 copies
  - PLANMGMT(EXTENDED) – 3 copies

How do you control it (and at what level)?
How does it work?
PLANMGMT(BASIC)

- The package has one active ("current") copy, and one additional ("previous") copy is preserved.
- At each REBIND:
  - Any previous copy is discarded
  - The current copy becomes the previous copy
  - The incoming copy becomes the current copy
- If you issue two or more rebinds after migration to a new version, you will wipe out the access path for packages from previous version which you might want to preserve.
PLANMGMT(EXTENDED) – REBIND and SWITCH

<table>
<thead>
<tr>
<th>REBIND</th>
<th>REBIND…SWITCH (PREVIOUS)</th>
<th>REBIND…SWITCH (ORIGINAL)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Incoming**
  - Copy
  - Current
  - Move
  - Previous
  - Delete
  - Trash
  - Original

- **Current**
  - Move
  - Previous
  - Original

- **Copy if no Original**

- **Current**
  - Move
  - Previous
  - Delete
  - Original
  - Trash

- **Copy**
PLANMGMT(EXTENDED)

- Retains up to three copies of a package: one active copy and two additional old copies (PREVIOUS and ORIGINAL) are preserved.
- At each REBIND:
  - Any previous copy is discarded
  - If there is no original copy, the current copy is saved as the original copy
  - The current copy becomes the previous copy
  - The incoming copy becomes the current copy
- Unlike the case when you use PLANMGMT(BASIC), the original copy is the one that existed from the “beginning”, it is saved once and never overwritten (it could be the copy you wish to preserve from a prior version).
In DB2 10, this visibility for non-active copies has been extended, so no longer an issue in DB2 10.
How do you tell which flavor of package stability (if any) applies to a package?

```sql
SELECT SP.COLLID, SP.NAME, SP.VERSION,
       COUNT(DISTINCT SPD.DTYPE) AS COPY_COUNT
FROM SYSIBM.SYSPACKAGE SP
   , SYSIBM.SYSPACKDEP SPD
WHERE SP.COLLID = SPD.DCOLLID
AND SP.NAME = SPD.DNAME
GROUP BY SP.COLLID, SP.NAME, SP.VERSION

- COPY_COUNT=1: OFF (DTYPE = blank)
- COPY_COUNT=2: BASIC (DTYPE = blank and P)
- COPY_COUNT=3: EXTENDED (DTYPE = blank, P and O)
```
Deleting old copies

- A new FREE PACKAGE option called PLANMGMT SCOPE allows you to free older copies that are no longer necessary.
  - PLANMGMT SCOPE(ALL) - To free the entire package including all copies. This is the default.
  - PLANMGMT SCOPE(INACTIVE) - To free all old copies only (i.e. original and previous, if any).
- The existing FREE PACKAGE command and DROP TRIGGER SQL statement drops the specified package and trigger as well as all associated current, previous and original copies – i.e. it behaves like SCOPE(ALL).

Clean up of old copies.
Comparing the options. The best option depends entirely on your setup.

Another option worth mentioning is the ability to bind into a dummy collection. A more detailed discussion on managing access path appears in one of my other sessions (see ref # 7).
In this final section, we will present what the future holds (announced) and what I would like it to hold.
Some of the many enhancements which will be covered in other IBM sessions. I can mention only those features which have been announced by IBM.

Safe optimization is a great start! But also see my discussion of defensive optimization.

The ability to retain multiple copies of access paths extends what Package Stability provided much further – I am excited about this one!
My wish list (Are you listening, IBM?)

- Aggressive vs. defensive optimization
- Scrolling/Re-positioning issues – fixed in DB2 10
- Externalization of alternatives and reasons
- “Anti-hints”

A list of enhancements I would like to see some day… we will discuss each one in detail in the following slides.
You want minimum cost, but what cost? Average or worst-case?

This tradeoff between performance and predictability has been discussed in various papers, most notably by Brian Babcock and Surajit Chaudhuri (see ref #8). In their own words:

“...a robust query optimizer is one that generates plans that work reasonably well even when optimizer assumptions fail to hold.” – yes, DB2 10 attempts to do so.

“Because robustness sometimes comes at the cost of performance, users should be allowed to prioritize these competing objectives.” – no, this is hidden within DB2.
Range predicates used in scrolling logic are notoriously hard for the optimizer and the filter factors tend to be too aggressive. The optimizer really has no way know if proper filtering is going to occur without a special indication which indicates scrolling.

In DB2 10, this issue has been addressed squarely by the Optimizer recognizing such constructs and using a new access type NR - (see my session A12 for details),
Externalization of alternatives and reasons

- An extended EXPLAIN command to show not only the access path chosen but also the alternatives considered and the reasons they were discarded.
- Externalization of i/o cost, cpu cost etc.

Finally, making the optimizer translucent (if not transparent). This would also help in picking the alternative when using OPTHINTs, which is not always clear.
“Anti-hints”

- Knowing the desirable access path is not always obvious
- Creating a desirable access path can be cumbersome
- This would allow you to say “not this, try something else” and let the optimizer pick an alternative

The lazy man’s alternative.
A summary of what we discussed.

1. Cost-based optimizer basics
   - Filtering, Effect of NUD, Effect of correlation, Effect of clustering, Literals and REOPT
2. Stats, stats and even more stats
   - Basic stats, Distribution stats, Correlation stats, Histogram
3. Case studies – searching for the “irresistible” index
   - Table sizes (basic stats), Table filters (basic stats), Non uniform distribution, correlation, range skew (Histogram stats), The “Pick me!” index
4. Influencing the access path
   - Stats plugging, Tricks, Hints, Package stability
5. What does the future hold?
   - V10 announced changes, My wish list
I trust this session has empowered you with the knowledge to exploit the optimizer fully. Good Luck!
Some of the many useful references. While reference #6 is now dated, the ideas presented in it are still useful for a quick analysis.

ACKNOWLEDGEMENTS:

I would like to express my sincere thanks to Adarsh Pannu (IBM Optimizer team) for his review of a draft copy and suggestions for improvement. Thank you Adarsh!
Thank you and good luck with access paths… the Optimizer is not perfect, but it is your friend – treat it like one!