Get the most out of DB2 optimizer

Analyze access plans to improve query execution performance

Samir Kapoor
Vincent Corvinelli

October 28, 2010

The access plan that the IBM® DB2® SQL optimizer chooses can significantly influence the execution performance of a SQL statement. In this article, you learn, through examples based on actual scenarios encountered by DB2 users, how to analyze access plans, and tune queries to improve query execution performance.

Introduction

This article shows you ways to improve the performance of SQL statements by analyzing the access plans that are generated by the DB2 SQL Optimizer (referred to as optimizer hereafter). It includes scenarios used by DB2 support and development that show you how problems are diagnosed and resolved, including how you can use the RUNSTATS tool, the REOPT bind option, and the EXPLAIN options in DB2 Version 9.7 to help you increase performance by improving optimizer cardinality estimates.

Scenario 1: Using RUNSTATS to update out-of-date statistics

In this scenario, a query performs poorly on the production system as compared to the development system. To improve the query performance, you analyze the access plan and run RUNSTATS to update any out-of-date statistics.

Listing 1. Original statement

```sql
-- Note: VW is a view
select *
from
db2inst1.vw v,
db2inst1.t1 t1
where
  v.x = t1.x
```
### Listing 2. Optimized statement

```sql
SELECT ...
FROM
  DB2INST1.T4 AS Q1,
  DB2INST1.T3 AS Q2,
  DB2INST1.T2 AS Q3,
  DB2INST1.T1 AS Q4
WHERE
  (Q1.T2ID = Q2.T2ID) AND
  (Q3.T4ID = Q1.T4ID) AND
  (Q1.Y = 'Y') AND
  (Q1.Z = 'N') AND
  (Q3.X = Q4.X) AND
  (Q3.Y IS NULL)
```

### Listing 3. Query access plan

<table>
<thead>
<tr>
<th>9.006</th>
<th>NLJOIN</th>
<th>11476.2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( 2)</td>
<td>2892</td>
</tr>
<tr>
<td>247.66</td>
<td>NLJOIN</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( 3)</td>
<td></td>
</tr>
<tr>
<td>11333.4</td>
<td>NLJOIN</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( 8)</td>
<td></td>
</tr>
<tr>
<td>0.0208</td>
<td>IXSCAN</td>
<td>74416</td>
</tr>
<tr>
<td>0.0199984</td>
<td>IXSCAN</td>
<td>74416</td>
</tr>
<tr>
<td>52</td>
<td>IXSCAN</td>
<td>74416</td>
</tr>
<tr>
<td>0.0104877</td>
<td>IXSCAN</td>
<td>74416</td>
</tr>
<tr>
<td>13</td>
<td>INDEX: DB2INST1</td>
<td>INDEX: DB2INST1</td>
</tr>
<tr>
<td>0.0208</td>
<td>IXSCAN</td>
<td>INDEX: DB2INST1</td>
</tr>
<tr>
<td>0.0194877</td>
<td>IXSCAN</td>
<td>INDEX: DB2INST1</td>
</tr>
<tr>
<td>0</td>
<td>INDEX: DB2INST1</td>
<td>INDEX: DB2INST1</td>
</tr>
</tbody>
</table>

Your first step in analyzing this access plan is to look at the estimated number of rows by the optimizer. If the estimate error is large, then cardinality estimation is likely a major factor in choosing a non-optimal access plan. Estimates less than one row are likely suspects, unless the operator is the inner (right) input of a nested loop join (referred to as $\text{NLJOIN}$ hereafter). The inner input cardinality is a per-outer estimate, and an estimate less than one row is common. Furthermore, when you analyze access plans, you should start from the bottom and work your way up the graph.

The access plan in **Listing 3** shows a cardinality estimate of less than one row, $0.0208$, at operator $\text{IXSCAN}$ (5). This operator is not on the inner input of a $\text{NLJOIN}$, so it is a good candidate for further investigation.

### Listing 4. Predicates applied at IXSCAN(5)

**Predicates:**

- Predicates:
  - $(\text{T2ID} = \text{T2ID})$ AND
  - $(\text{T4ID} = \text{T4ID})$ AND
  - $(\text{Y} = 'Y')$ AND
  - $(\text{Z} = 'N')$ AND
  - $(\text{X} = X)$ AND
  - $(\text{Y IS NULL})$
<table>
<thead>
<tr>
<th></th>
<th>Start Key Predicate</th>
<th>Equal (=)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4)</td>
<td>Relational Operator:</td>
<td>Equal (=)</td>
</tr>
<tr>
<td></td>
<td>Subquery Input Required:</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>Filter Factor:</td>
<td>0.04</td>
</tr>
<tr>
<td></td>
<td>Predicate Text:</td>
<td>(Q1.Y = 'Y')</td>
</tr>
<tr>
<td>4)</td>
<td>Stop Key Predicate</td>
<td>Equal (=)</td>
</tr>
<tr>
<td></td>
<td>Subquery Input Required:</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>Filter Factor:</td>
<td>0.04</td>
</tr>
<tr>
<td></td>
<td>Predicate Text:</td>
<td>(Q1.Y = 'Y')</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Start Key Predicate</th>
<th>Equal (=)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5)</td>
<td>Relational Operator:</td>
<td>Equal (=)</td>
</tr>
<tr>
<td></td>
<td>Subquery Input Required:</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>Filter Factor:</td>
<td>0.04</td>
</tr>
<tr>
<td></td>
<td>Predicate Text:</td>
<td>(Q1.Z = 'N')</td>
</tr>
<tr>
<td>5)</td>
<td>Stop Key Predicate</td>
<td>Equal (=)</td>
</tr>
<tr>
<td></td>
<td>Subquery Input Required:</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>Filter Factor:</td>
<td>0.04</td>
</tr>
<tr>
<td></td>
<td>Predicate Text:</td>
<td>(Q1.Z = 'N')</td>
</tr>
</tbody>
</table>

When you further examine the details of the predicates that are applied at IXSCAN (5) in Listing 4, you notice that the two predicates are shown with $P_1$ and $P_2$, and the filter factor with $FF$.

- $P_1$: $Q1.Y = 'Y'$  $FF=0.04$
- $P_2$: $Q1.Z = 'N'$  $FF=0.04$

Both $P_1$ and $P_2$ have a filter factor of 0.04, so assuming independence, then the cardinality estimate at IXSCAN(5) is calculated as follows.

$$CARD_{AT\_IXSCAN(5)} = FF(P_1) \times FF(P_2) \times INPUT\_CARD$$

where $INPUT\_CARD$ is 13, and represents the number of rows flowing into IXSCAN(5). As a result, the cardinality is estimated as the following:

$$CARD_{AT\_IXSCAN(5)} = 0.04 \times 0.04 \times 13 = 0.0208$$

When computing the filter factor of these predicates, the optimizer can make use of available column distribution statistics to account for non-uniformly distributed data. Or, if the data is uniformly distributed, the optimizer can use the column cardinality (referred to as $col\_card$ hereafter). In the worst case, if statistics are not available, the optimizer fabricates statistics based on the size of the table. It then gives a warning in the $Extended\_Diagnostic\_Information$ section of the $db2exfmt$ output for each table that the optimizer fabricated statistics.
Listing 5. Extended diagnostic information

Diagnostic Identifier:  1
Diagnostic Details:     EXP0045W. The table named "DB2INST1.T4" has fabricated statistics. This can lead to poor cardinality and predicate filtering estimates. The size of the table changed significantly since the last time the RUNSTATS command was run.

The information in Listing 5 shows that since the statistics are outdated, the optimizer fabricates the statistics. If the statistics show a smaller number of records in the table compared to the Update, Delete, and Insert (UDI) counters, then the fabricated statistics are saved in the internal packed descriptor. If the newly fabricated statistics show a larger number of records compared to the UDI counters, then the same fabricated statistics may still remain until it is reset by RUNSTATS. The change in the fabricated statistics may lead to changes in the plan chosen for the query execution.

Listing 6 shows that two count(*) queries are submitted to further confirm the real cardinality estimates before and after the predicates are applied.

Listing 6. COUNT(*) queries to confirm the actual counts

```sql
SELECT COUNT(*) FROM "DB2INST1"."T4";
RESULT:  13 ROWS

SELECT COUNT(*) FROM "DB2INST1"."T4" AS Q1
WHERE
  (Q1.Y = 'Y') AND
  (Q1.Z = 'N');
RESULT:  13 ROWS
```

The count(*) queries show that the actual cardinality of 13 remains the same before and after the predicates are applied. Compared to the estimated cardinality of 0.0208, the error in the estimate is a factor of 13/0.0208 = 625. This is significant since the optimizer estimates less than 1 row at NLJOIN (4), and thus expects to probe the inner of that NLJOIN only once. However, with the counts collected above, that is not true and you can expect the join to result in at least 625 * CARD_EST_NLJOIN(4) =~ 2 rows. As a result, you can expect that the inner is probed twice at NLJOIN (3) as shown in Listing 3. The inner of NLJOIN (3) is a full table scan at TBSCAN (7), although the access may not perform optimally if you have to probe it more than once.

Alternate solution

Starting in Version 9.5, you may enable automatic statement statistics to avoid problems with stale statistics, so manually collecting statistics may not always be a good solution.

Solution: You should execute RUNSTATS on the table DB2INST1.T4 to collect up-to-date statistics. Listing 7 shows the access plan chosen as a result of updating the statistics, and the query performed to your satisfaction.

Listing 7. Access plan after RUNSTATS is executed

```
6191.41
```
The filter factors of the two predicates are estimated as 1 with the updated statistics.

Listing 8. The predicate filter factors for the two predicates at IXSCAN (8) after RUNSTATS executed

Predicates:
--------------
3) Start Key Predicate
   Relational Operator: Equal (=)
   Subquery Input Required: No
   Filter Factor: 1
   Predicate Text:
   ---------------
   (Q1.Z = 'N')

3) Stop Key Predicate
   Relational Operator: Equal (=)
   Subquery Input Required: No
   Filter Factor: 1
   Predicate Text:
   ---------------
   (Q1.Z = 'N')

4) Start Key Predicate
   Relational Operator: Equal (=)
   Subquery Input Required: No
   Filter Factor: 1
   Predicate Text:
   ---------------
   (Q1.Y = 'Y')

4) Stop Key Predicate
   Relational Operator: Equal (=)
   Subquery Input Required: No
   Filter Factor: 1
   Predicate Text:
   ---------------
   (Q1.Y = 'Y')
Using the updated filter factors, the optimizer estimates the cardinality at IXSCAN(8) as

\[ \text{FF(P1)} \times \text{FF(P2)} \times \text{INPUT\_CARD} = 1 \times 1 \times 13 = 13 \]

**Scenario 2: Collecting distribution statistics to improve query execution performance**

In this scenario, the optimizer computes a query access plan with a non-optimal index access.

**Listing 9. Original statement**

```sql
select ...  
from db2inst1.name n1  
where ...  
  n1.lastname like 'BAKER%' and  
  n1.firstname like 'LENNY%'  
```

**Listing 10. Optimized statement**

**Optimized Statement:**

-------------

```sql
SELECT ...  
FROM DB2INST1.NAME AS Q1  
WHERE ... AND (Q1.LASTNAME LIKE 'LENNY%')  
     AND (Q1.FIRSTNAME LIKE 'BAKER%')  
```

**Listing 11. A snippet of the query access plan**

```
  0.48474  
  FETCH  
    ( 3)  
  37.86  
  126.234  
  /---+---\  
  139.289 9.3475e+08  
  RIDSCN  TABLE: DB2INST1  
    ( 4)  NAME  
  22.7192  Q1  
    4  
    |  
  139.289  SORT  
    ( 5)  
  22.7187  
    4  
    |  
  139.289  IXSCAN  
    ( 6)  
  22.6784  
    4  
    |  
  9.3475e+08  INDEX: DB2INST1  
INDEX3  
Q1
```

The **FIRSTNAME** "LENNY" appears 1.14 million times in **DB2INST1.NAME**. But, for the query you looked at in **Listing 9**, the optimizer chooses an index scan access of **DB2INST1.INDX3** with the key
containing only the FIRSTNAME field. Listing 11 shows how the optimizer computes a very low cost estimate for this index access, IXSCAN(6). A second index called DB2INST1.INDX1 with a key that includes both LASTNAME and FIRSTNAME is also defined on the table.

Listing 12. Statistics details

NUM_FREQVALUES  100
NUM_QUANTILES    50

RUNSTATS ON TABLE DB2INST1.NAME WITH DISTRIBUTION AND DETAILED INDEXES ALL

Table Cardinality:

UPDATE SYSSTAT.TABLES
SET CARD=934750120,
     NPAGES=17905266,
     FPAGES=17905266,
     OVERFLOW=0,
     ACTIVE_BLOCKS=0
WHERE TABNAME = 'NAME' AND TABSCHEMA = 'DB2INST1    ';

Column Cardinalities:

<table>
<thead>
<tr>
<th>COLNAME</th>
<th>COLCARD</th>
</tr>
</thead>
<tbody>
<tr>
<td>LASTNAME</td>
<td>6856495</td>
</tr>
<tr>
<td>FIRSTNAME</td>
<td>3829720</td>
</tr>
</tbody>
</table>

Indexes and their associated statistics:

<table>
<thead>
<tr>
<th>INDEX</th>
<th>KEY</th>
<th>NLEAF</th>
<th>NLEVELS</th>
<th>SEQ</th>
<th>PAGES</th>
<th>CLUSTER FACTOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>INDX1</td>
<td>LASTNAME</td>
<td>3829525 5</td>
<td>3828184</td>
<td>0.084789</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FIRSTNAME</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>NAME_MIDDLE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>NUMKEY</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INDX2</td>
<td>LASTNAME</td>
<td>4790072 5</td>
<td>4788313</td>
<td>0.089595</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SIGNIN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INDX3</td>
<td>FIRSTNAME</td>
<td>2596262 5</td>
<td>2595551</td>
<td>0.075572</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>BIRTH</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The index statistics in Listing 12 shows how the clustering of the table is very poor on all the indexes in column CLUSTER FACTOR. If the data stream is not reduced significantly at the index scan level, the random I/O's may cause the large amount of data pages that are fetched to perform poorly.

The first step is to determine the accuracy of this estimate because the output cardinality of the IXSCAN (6) operator in Listing 11 is very small compared to the 934 million rows of the input cardinality.

Listing 13. Details of the predicates applied at IXSCAN (6)

Predicates:

6) Stop Key Predicate
   Comparison Operator: Less Than or Equal (<=)
   Subquery Input Required: No
   Filter Factor: 0.59615
   Predicate Text:
The rewrite phase of query optimization generates equivalent range predicates for each of the LIKE predicates in the statement, which improves query performance by applying the generated range predicates as start/stop keys on the index scan. The range predicate is constructed to search for the range of values between the lowest string with LENNY as the prefix, and the highest string with LENNY as the prefix. The filter factors for the LIKE predicates are shown as follows.

- P4: n1.lastname like 'LENNY%' FF=1.49012e-07
- P5: n1.firstname like 'BAKER%' FF=0.00290581

When the WITH DISTRIBUTION clause is used in the RUNSTATS command, the quantile statistics collected on the column provide the optimizer with the required information to accurately estimate the filter factor of range predicates. For this scenario, Table 1 shows the quantiles of interest for LASTNAME and FIRSTNAME.

Table 1. Quantiles for predicate columns

<table>
<thead>
<tr>
<th>Column</th>
<th>Seqno</th>
<th>Value</th>
<th>Valcount</th>
</tr>
</thead>
<tbody>
<tr>
<td>LASTNAME</td>
<td>1</td>
<td>'ACKLAND'</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>'BAKER'</td>
<td>20656377</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>'BARKLEY'</td>
<td>38993587</td>
</tr>
<tr>
<td>FIRSTNAME</td>
<td>29</td>
<td>'LARRY'</td>
<td>538016212</td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>'LIN'</td>
<td>557252038</td>
</tr>
</tbody>
</table>

The calculation of the filter factor estimate for the range predicate is best shown by using the diagram in Listing 14.
Listing 14. Calculating FF of range predicates

<table>
<thead>
<tr>
<th></th>
<th>'LENNY.....' &lt;= Q1.FIRSTNAME = FF2</th>
<th>FF(RANGE)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>= FF1 + FF2 - 1</td>
</tr>
<tr>
<td>&lt;-------------------------------------------------</td>
<td>Q1.FIRSTNAME &lt;= 'LENNY.....' = FF1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FULL RANGE OF VALUES IN FIRSTNAME</td>
<td></td>
</tr>
</tbody>
</table>

When quantile statistics are available, the filter factor of the LIKE predicate is computed by estimating the individual filter factor of each range (FF1,FF2) individually, and determining the overlap between the two predicates as:

\[
FF(\text{RANGE}) = (FF1 + FF2) - 1.
\]

As a result, the filter factor for the predicate \text{FIRSTNAME LIKE 'LENNY%' is computed as:}

\[
FF1 = FF('LENNY......' <= Q1.FIRSTNAME) = 0.40385 \\
FF2 = FF(Q1.FIRSTNAME <= 'LENNY......') = 0.59615 \\
FF(\text{RANGE}) = FF1 + FF2 - 1 = 0.59615 + 0.40385 - 1 = 0
\]

This shows the estimated filter factor is 0 by using linear interpolation of the quantile statistics, but the optimizer further bounds the filter factor using the frequent value statistics as a lower bound.

Following the same procedure as described in Scenario 1, the cardinality estimate can be verified using \text{count(*)} queries.

Listing 15. \text{COUNT(*)} queries to confirm the actual counts

```sql
select count(*)
from db2inst1.name n8
where n1.lastname like 'BAKER'
RESULT: 2739268

select count(*)
from db2inst1.name n8
where n1.firstname like 'LENNY'
RESULT: 1161991

select count(*)
from db2inst1.name n1
where n1.firstname like 'LENNY' AND n1.lastname like 'BAKER'
RESULT: 3853
```

The counts show that the filter factor estimate for LASTNAME is fine, but FIRSTNAME is underestimated. From Listing 13 you can see that:

\[
FF(\text{n1.lastname like 'BAKER'}) = 0.00290581
\]

Therefore, as seen below, the estimated cardinality is very accurate after applying this predicate.
For **FIRSTNAME**, the optimizer estimates only 139,289 rows, but the actual count is 1,161,991. The error in the estimate is significant.

**Why is the optimizer underestimating the cardinality?** For range predicates, the optimizer considers using the quantile statistics to estimate the filter factor. From the statistics listed in Table 1, there are \(557,252,038 - 538,916,212 = 19,235,826\) rows between positions 29 and 30 that contain values between **LARRY** and **LIN**. The frequent values shown below that one of the values within this range exists in 2.29 million rows.

```
UPDATE SYSSTAT.COLDIST
SET COLVALUE='LESTER', VALCOUNT=2295040
WHERE COLNAME = 'FIRSTNAME' AND TABNAME = 'NAME'
AND TABSCHEMA = 'DB2INST1'
AND TYPE = 'F' AND SEQNO = 58;
```

That value only covers a small fraction of the range, and there are 17 million other rows unaccounted for. **Listing 16** includes a diagram that shows the unknown values.

**Listing 16. Quantile bucket for FIRSTNAME**

```
| LARRY | LESTER | LIN |
|<------UNKNOWN------>|<------2.29M------>|<------UNKNOWN------>|
| ... | +-----------+--------+---------------+---------------------+ ... |
| 538016212 | LENNY? | 557252038 |
```

**Solution:** Since the number of distinct possible values between **LARRY** and **LIN** can be very large, if you can reduce the ranges in the quantiles, then the error in the filter factor estimate can be reduced. Initially, you were collecting 50 quantiles, so if you collect more quantiles, then the error in the optimizer's estimates should be reduced, and should lead to an optimal access plan.

**Listing 17. Snippet of access plan after increasing quantiles**

```
  511.05
  IXSCAN
    (3)
  11111.8
  11242.9
    | 9.3475e+08
INDEX: DB2INST1
  INDEX1
```

Increasing the quantiles collected on **FIRSTNAME** to 200 greatly reduced the error in the cardinality estimate, resulting in the optimizer choosing an index access that led to optimal query execution performance.
Scenario 3: Using REOPT to improve query execution performance

In this scenario, you run a static query from an application that completes in 11 minutes. But if the query is submitted from the command line (CLP), it completes in 10 seconds. Why is there a difference in performance between the two methods of executing the same query?

Listing 18. Query from static package using host variables

```
SELECT ... 
FROM 
  DB2INST1.T1 A, 
  DB2INST1.T2 B, 
  DB2INST1.T3 C, 
WHERE 
  A.T1ID = B.T2ID and 
  B.T2ID = C.T2ID and 
  B.X = :HV00001 :HI00001 and 
  C.Y <> '10' and 
  A.X in ('00') and 
  A.Y in ('000','001','005','006') and 
  A.Z between :HV00002 :HI00002 and :HV00003 :HI00003 
```

Listing 19. Query from CLP using literal values

```
SELECT ... 
FROM 
  DB2INST1.T1 A, 
  DB2INST1.T2 B, 
  DB2INST1.T3 C, 
WHERE 
  A.T1ID = B.T2ID and 
  B.T2ID = C.T2ID and 
  B.X = 'CAD' and 
  C.Y <> '10' and 
  A.X in ('00') and 
  A.Y in ('000','001','005','006') and 
  A.Z between '2007-10-01' and '2007-10-30' 
```

When the optimizer compiles a query with predicates in the `WHERE` clause that contains parameter markers, special registers, or host variables, it does not know the actual value when computing the filter factor. For equality predicates, the `COLCARD` statistic is used, and for range predicates the `HIGH2KEY` and `LOW2KEY` statistics are used. In this scenario, you collected distribution statistics, so the optimizer can estimate a more accurate cardinality when there is significant skew in the column data, and the actual literal values are used in the query when submitted from the CLP.

Solution: Bind the package using the `REOPT ALWAYS` option as follows:

```
db2 BIND <filename> ... REOPT ALWAYS ... 
```

The `REOPT ONCE` option can also be considered, but if there is significant skew in the data, then multiple variations of the plan may be required to achieve optimal performance. With `REOPT ONCE`, if you do not seed it with a representative set of values, then it may not be beneficial. The following guideline shows when to use `REOPT`:

- Non-uniform data distribution
• Distribution statistics are collected
• REOPT ALWAYS: compilation time isn't a concern
• REOPT ONCE: if compilation time is a concern, REOPT-imizing on the first set of values might produce a plan that is good enough for all ranges of values

You can use the db2pd tool to confirm that REOPT is being used, as shown in Listing 20.

**Listing 20. Using db2pd to confirm REOPT is being used**

```
db2pd -db <dbname> -REOPT
```

Database Partition 0 -- Database PCTMS00D -- Active -- Up 15 days 00:28:27 -- Date ...
Dynamic Cache Reoptimization:
Dynamic SQL Statements:
<table>
<thead>
<tr>
<th>Address</th>
<th>AnchID</th>
<th>StmtUID</th>
<th>NumEnv</th>
<th>NumVar</th>
<th>NumRef</th>
<th>NumExe</th>
<th>Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x2E787090</td>
<td>41</td>
<td>94</td>
<td>1</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>SELECT ... FROM DB2INST1.T1 A, DB2INST1.T1 B, DB2INST1.T1 C, WHERE A.T1ID = B.T1ID and B.TZID = C.TZID and B.X = :HV00001 :HI00001 and C.Y &lt;&gt; '10' and A.X in ('00') and A.Y in ('000','001','005','006') and A.Z between :HV00002 :HI00002 and :HV00003 :HI00003</td>
</tr>
</tbody>
</table>

The NumVar value represents the number of variations of the query, and the StmtUID value is used to map to the variations, as shown in Listing 21.

**Listing 21. Dynamic SQL variations**

```
Address     AnchID StmtUID EnvID VarID NumRef Typ ... Time                       ...
0x2F1175E0  41     94      2     4     1      6       2008-11-06-16.34.25.040149 ...
0x2F09E7C50 41     94      2     3     1      6       2008-11-06-16.30.42.854376 ...
0x2F2E6A60  41     94      2     2     1      6       2008-11-06-13.15.17.701584 ...
0x2F2D61C0  41     94      2     1     1      6       2008-11-06-12.48.36.476402 ...
```

Listing 21 shows the different variations for the same query, executed at different times. Due to performance reasons, when you use REOPT ALWAYS, the values illustrated by the empty output in Listing 22 are not tracked internally.

**Listing 22. REOPT values not available under REOPT ALWAYS**

```
Address     AnchID StmtUID EnvID VarID NumRef Typ ... Time ...
```

**Scenario 4: Collecting column group statistics to improve query execution performance**

In this scenario, the optimizer underestimates the cardinality when computing a query access plan, causing the query to run for 2 hours.
The query in **Listing 23** contains multiple equality join predicates between each of the tables in the join.
Group 1:
P2: T2.ACCT_NUM = T3.ACCT_NUM
P3: T2.ID_NUM = T3.ID_NUM

Group 2:
P4: T1.ACCT_NUM = T4.ACCT_NUM
P5: T1.REP_NUM = T4.REP_NUM

Group 3:
P6: T4.ACCT_NUM = T3.ACCT_NUM
P7: T4.ID_NUM = T3.ID_NUM
P8: T4.ACCT_IND = T3.ACCT_IND

The access plan in Listing 24 shows a significant reduction in cardinality at HSJOIN (6) when compared to the two inputs into the join. The detailed predicates section in Listing 25 are from the Group 2 set, shown as P4 and P5. The quantifiers Q1 and Q4 identify tables T4 and T1 respectively.

Listing 25. The predicates applied at HSJOIN(6)

2) Predicate used in Join
   Relational Operator: Equal (=)
   Subquery Input Required: No
   Filter Factor: 2.7307e-07
   Predicate Text:
     ----------------
     (Q4.ACCTNUM = Q1.ACCTNUM) <--- P4

6) Predicate used in Join
   Relational Operator: Equal (=)
   Subquery Input Required: No
   Filter Factor: 2.71267e-05
   Predicate Text:
     ----------------
     (Q1.REP_NUM = Q4.REP_NUM) <--- P5

The estimated cardinality of the hash join is computed as the product of the input cardinalities, and the filter factor of each join predicate, as shown below.

\[
\text{HSJOIN\_CARD} = 2.7307e-07 \times 2.71267e-05 \times 6.69807e+06 \times 1.15013e+06 \approx 57.0648
\]

The following \text{count(*)} query may be used to calculate the error in the estimate of this join cardinality, as shown below.

\[
\begin{align*}
\text{SELECT COUNT(*)} \\
\text{FROM} \\
\text{T4 AS Q1,} \\
\text{T1 AS Q4} \\
\text{WHERE} \\
(Q1.REP_NUM = Q4.REP_NUM) \text{ AND} \\
(Q4.ACCT_NUM = Q1.ACCT_NUM) \text{ AND} \\
('1800-01-01-00.00.00.000000' < Q4.DATE\_MOD) \text{ AND} \\
Q4.COMM \text{ IN ('A', 'D');}
\end{align*}
\]

RESULT: 1,155,273 rows
Column group statistics

This discussion assumes that you are familiar with column group statistics and the terminology associated with it. You can find the details about column group statistics in "Understand column group statistics in DB2".

The individual filter factors for the two local predicates on T1 were verified and confirmed as not being very selective. Column group statistics may be a solution because there are two or more equality join predicates between the pair of tables, and the error in the cardinality estimate of the join is large, so the two join predicates might be statistically correlated. The simple approach is to collect column group statistics on both tables, but only a column group on the parent is required, so the steps necessary to determine the parent side in this join are shown.

Table 2. Column statistics

<table>
<thead>
<tr>
<th>Column</th>
<th>T1</th>
<th>T1</th>
<th>T4</th>
<th>T4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Colcard</td>
<td>High2key</td>
<td>Low2key</td>
<td>Colcard</td>
</tr>
<tr>
<td>ACCT_NUM</td>
<td>3357997</td>
<td>'JERE2'</td>
<td>'00003'</td>
<td>3662064</td>
</tr>
<tr>
<td>REP_NUM</td>
<td>32341</td>
<td>'95517'</td>
<td>'63135'</td>
<td>36864</td>
</tr>
</tbody>
</table>

Database partitioning feature

In a DPF environment, there are more restrictions on how you determine which is the parent table, and which is the child table. Refer to "Understand column group statistics in DB2" for further details.

From the column statistics in Table 2, T4 is the parent in the join because the COLCARD of ACCT_NUM and REP_NUM is greater than T1's respective COLCARD statistics, the HIGH2KEY for each column is greater, and the LOW2KEY are the same (requirement is less than or equal to). As a result, a column group statistic on (T4.ACCT_NUM, T4.REP_NUM) is considered by the optimizer to account for statistical correlation between the two join predicates.

Solution: Collect the column group statistic on T4. If an index is already defined on T4 with (ACCT_NUM, REP_NUM) as the leading columns in the key (the order of the two columns is not important), then the FIRST2KEYCARD statistic of the index would already be available for use by the optimizer to account for the statistical correlation. In this scenario, such an index does not exist, so the RUNSTATS command is executed with the column group included, as shown below.

```
RUNSTATS ON TABLE DB2INST1.T4
  ON ALL COLUMNS AND COLUMNS ((ACCT_NUM, REP_NUM)) WITH DISTRIBUTION
  AND SAMPLED DETAILED INDEXES ALL
```

Listing 26 shows how collecting the column group statistics improves the cardinality estimate computed by the optimizer, which produces a better performing access plan.

Listing 26. Access plan after collecting column group statistics

```
1.6838e+06
```
### Scenario 5: Using Section Actuals when analyzing access plans to improve query performance

Determining the appropriate set of `count(*)` queries can be difficult and time consuming, especially for large queries. You can remedy this by using the new feature called **Section Actuals**, from DB2 Version 9.7 Fix Pack 1.

In this scenario, you repeat **Scenario 1** using **Section Actuals** to illustrate its usefulness. Do the following steps to capture the **Section Actuals**.

1. **Enable Section Actuals**
2. **Create the workload manager and event monitor**
3. **Collect Section Actuals for the statement of interest**
4. **Locate the Application, UOW and Activity ID for the data**
5. **Populate the data into EXPLAIN tables**
6. **Run db2exfmt to generate the access plan**
7. **Examine the output**

### Step 1: Enable Section Actuals

#### Section Actuals restriction

Section Actuals cannot be enabled if automatic statistics profile generation (auto_stats_prof) is enabled. An error with SQLCODE -5153 will be returned if this is attempted.

You must explicitly enable the **Section Actuals** feature by setting the `section_actuals` database configuration parameter to `BASE` as follows.

```
db2 update db cfg for <dbname> using section_actuals base
```

Once enabled, the information is captured using the `EXPLAIN_FROM_ACTIVITY` procedure.
Step 2: Create the workload manager and event monitor

You must create a workload manager and event monitor to use Section Actuals. You can use the default workload manager instead of creating one as described below.

```sql
create workload MYWORKLOAD
  current client_acctng('MYWORKLOAD') service class sysdefaultuserclass
  collect activity data on all database partitions with details, section;

grant usage on workload MYWORKLOAD to public;

create event monitor MYMON for activities write to table;
```

Step 3: Collect Section Actuals for the statement of interest

```sql
delete from ACTIVITYSTMT_MYMON;
call wlm_set_client_info(null, null, null, 'MYWORKLOAD', null);
set event monitor MYMON state 1;

-- a subset of the statement from scenario 1
SELECT * FROM DB2INST1.T4 AS Q1
WHERE (Q1.Y = 'Y')
  AND (Q1.Z = 'N')

set event monitor MYMON state 0;
call wlm_set_client_info(null, null, null, null, null);
```

Step 4: Locate the application, UOW and activity ID for the data

```sql
select appl_id, uow_id, activity_id, substr(stmt_text,1,80) as stmt
  from ACTIVITYSTMT_MYMON ;
```

<table>
<thead>
<tr>
<th>APPL_ID</th>
<th>UOW_ID</th>
<th>ACTIVITY_ID</th>
<th>STMT</th>
</tr>
</thead>
<tbody>
<tr>
<td>*LOCAL.DB2.100530150552</td>
<td>20</td>
<td>1</td>
<td>SELECT * FROM DB2INST1.T4 AS Q1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>WHERE (Q1.Y = 'Y')</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>AND (Q1.Z = 'N')</td>
</tr>
</tbody>
</table>

Step 5: Populate the data into the EXPLAIN tables

The `explain_from_activity` procedure is called to populate the explain tables for the statement of interest. The first three inputs, `APPL_ID`, `UOW_ID`, `ACTIVITY_ID`, are determined for the statement of interest from the output in step 4. The fourth input is the event monitor identifier set in step 3, and the fifth input is the schema name of the EXPLAIN tables. In this example, the schema for the EXPLAIN tables is VCORVINE.

```sql
call explain_from_activity ('*LOCAL.DB2.100530150552',
                           20,1,'MYMON','VCORVINE',?,?,?,?);
```

Value of output parameters

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Parameter Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXPLAIN_SCHEMA</td>
<td>VCORVINE</td>
</tr>
<tr>
<td>EXPLAIN_REQUESTER</td>
<td>VCORVINE</td>
</tr>
<tr>
<td>EXPLAIN_TIME</td>
<td>2010-05-30-11.21.34.250000</td>
</tr>
</tbody>
</table>
Parameter Name: SOURCE_NAME
Parameter Value: SQLC2H21

Parameter Name: SOURCE_SCHEMA
Parameter Value: NULLID

Parameter Name: SOURCE_VERSION
Parameter Value: 

Return Status = 0

Step 6: Run db2exfmt to generate the access plan

db2exfmt -d <dbname> -1 -o ex_activity.out

Screen Output:
DB2 Universal Database Version 9.7, 5622-044 (c) Copyright IBM Corp. 1991, 2008
Licensed Material - Program Property of IBM
IBM DATABASE 2 Explain Table Format Tool

Connecting to the Database.
Connect to Database Successful.
Output is in ex_activity.out.
Executing Connect Reset -- Connect Reset was Successful.

Step 7: Examine the output

The following is a part of the access plan from the db2exfmt output in ex_activity.out.

<table>
<thead>
<tr>
<th>Rows</th>
<th>Rows Actual</th>
</tr>
</thead>
<tbody>
<tr>
<td>RETURN</td>
<td>( 1)</td>
</tr>
<tr>
<td>Cost</td>
<td>0.0208</td>
</tr>
<tr>
<td>I/O</td>
<td>13</td>
</tr>
<tr>
<td>IXSCAN</td>
<td>0.0104877</td>
</tr>
<tr>
<td>INDEX:</td>
<td>DB2INST1</td>
</tr>
<tr>
<td></td>
<td>IDX89</td>
</tr>
</tbody>
</table>

The access plan shows the estimated and actual number of rows at each operator. The Section Actuals are derived from the section Explain, so not all data is available like there is with a regular EXPLAIN of the query. This is identified in the graph of the access plan using the NA notation. For example, IXSCAN(5) does not include the I/O cost value, and the actual base table cardinality is not collected.

Conclusion

Your ability to analyze access plans to tune queries and improve query execution performance is a valuable skill. You can ensure your access plan is optimal, and improve its performance by using
the tools and techniques discussed in this article to identify and correct errors in the optimizer estimated cardinality.
Related topics

• "Comparing real-time cardinality to the optimizer cardinality estimates" (developerWorks, December 2005): A tool to aid in tuning queries.
• "Understand column group statistics in DB2" (developerWorks, December 2006): Learn all about how to use column group statistics.
• "Further Understand column group statistics in DB2" (developerWorks, September 2008): Leverage the extended use of multi-column statistics to improve cardinality estimates.

© Copyright IBM Corporation 2010
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)