The Db2 Access Plan Troubleshooting Handbook

John Hornibrook, IBM Canada

Themes

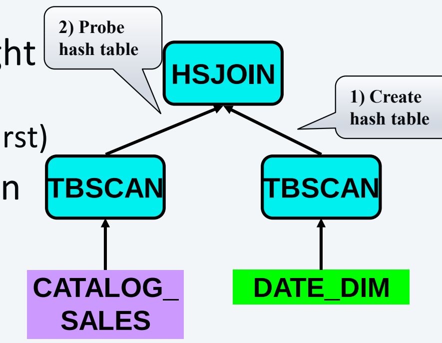
IBM

- Finding the access plan troublemakers
- Getting to the truth: optimizer fantasy vs. runtime reality
- Bringing the optimizer back to reality
- When the optimizer has done its best
- Desperate measures

Before we begin – what is an access plan?

IBM

- An ***Access Plan*** represents a sequence of runtime operators used to execute the SQL statement
- Represented as a graph where each node is an operator and the edges represent the flow of data
- The order of execution is generally left to right
  - But there are some exceptions
  - (Hash join build table is on the RHS and is created first)
- Use the ***explain facility*** to see the access plan



3

The explain facility – what is it?

IBM

- Internal phase of the optimizer that captures critical information used in selecting the query access plan
- Access plan information is written to a set of tables
- External tools to format explain table contents:
  - Db2 Data Management Console Visual Explain
    - GUI to render and navigate query access plans
    - Supersedes Data Server Manager Visual Explain
  - db2exfmt
    - Text-based output from the explain tables
    - Command-line interface

They show the same information

4

The explain facility is used to display the query access plan chosen by the query optimizer to run an SQL statement. It contains extensive details about the relational operations used to run the SQL statement such as the plan operators, their arguments, order of execution, and costs. Since the query access plan is one of the most critical factors in query performance, it is important to be able to understand the explain facility output in order to diagnose query performance problems.

Explain information is typically used to:

- understand why application performance has changed
- evaluate performance tuning efforts

Decide where to look in the db2exfmt

- What section to check first depends on the situation
  - Are you familiar with the system config?
    - Does the optimizer have the correct information?
    - Does the system have enough memory for this query?
  - Are you sure the statistics are current?
  - Check the access plan graph
  - Check the access plan details

Understand the db2exfmt layout

IBM

- Main sections:
  1. System configuration summary
  2. Original statement
  3. Optimized statement (after query transformations)
  4. Access plan graph
  5. Extended diagnostic information
  6. Plan details
  7. Objects used in access plan (and their statistics)

6

db2exfmt - System configuration summary

IBM

```
Database Context:
----------------
        Parallelism:          Inter-Partition Parallelism
        CPU Speed:            3.188324e-07
        Comm Speed:           100
        Buffer Pool size:     1202128
        Sort Heap size:       429252
        Database Heap size:   4633
        Lock List size:       6200
        Maximum Lock List:    60
        Average Applications: 1
        Locks Available:      119040

Package Context:
---------------
        SQL Type:           Dynamic
        Optimization Level: 5
        Blocking:           Block All Cursors
        Isolation Level:    Cursor Stability
```

Does the CPUSPEED DBM config look reasonable? (Set automatically by Db2)

Does the COMM_BANDWIDTH DBM config look reasonable? (DPF only - set automatically by Db2)

Enough memory (buffer pool and sort heap) ?

Leave this set to 1

You should understand if this is something other than 5 (the default)

7

db2exfmt – Access plan graph

IBM

Cardinality (rows)
Operator name
(Operator ID)
Cost (timerons)
I/O (pages)

```
                          Rows
                         RETURN
                         (   1)
                          Cost
                           I/O
                           |
                            4
                          NLJOIN
                         (  13)
                         185.206
                            7
                  /-------+------\
                 1                 4
              IXSCAN             FETCH
              (  14)             (  15)
              75.0966            100.118
                 3                  4
                 |             /----+---\
           4.99966e+06        4         1.99987e+07
         INDEX: TPCD        IXSCAN      TABLE: TPCD
         UXP_NMPK           (  16)     PARTSUPP
                            75.1018
                               3
                               |
                          1.99987e+07
                     INDEX: TPCD.UXPS_PK2KSC
```

We will be paying close attention to this number

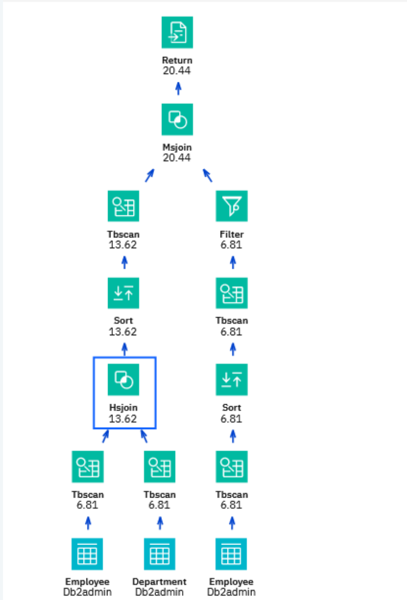
Cost estimates are cumulative i.e. represents cost of operator + its input plans

Base table cardinality from catalog statistics

8

Db2 Data Management Console Visual Explain

IBM

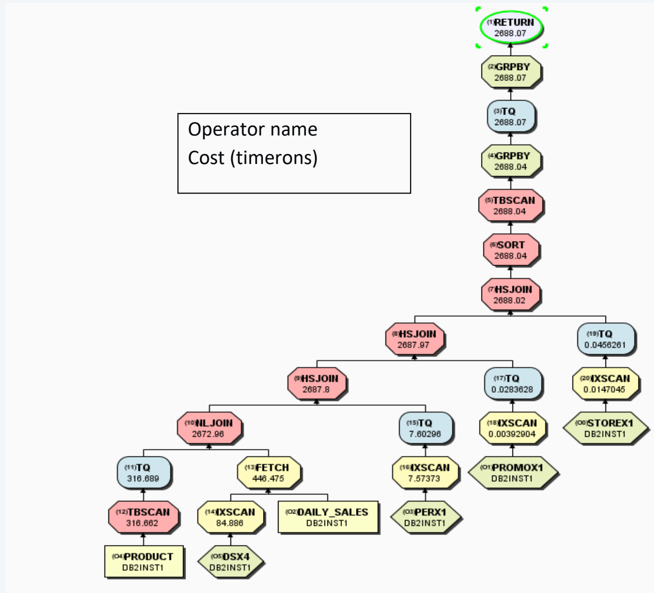


Description

Properties

Name	Value
Operator identifier	5
Predicate text	(Q1.MGRNO = Q2.EMPNO)
Operator type	Hash join
Early out flag	RIGHT
Hash join bit filter used	FALSE
Temporary table page size	32768
Hash code size	24 BIT
HASHTBSZ	9
TUPBLKSZ	4000
Cost Information	
Estimated output cardinality	14.00
Cumulative total cost	13.62
Cumulative CPU cost	214,641.66
Cumulative I/O cost	2.00
Cumulative first row total	13.62

Data Server Manager Visual Explain



Check the Cardinality Estimates

- *Cardinality* = number of rows
- The optimizer estimates the number of rows processed by each access plan operator
- Based on the number of rows in the table and the filter factors of applied predicates.
- **This is the biggest impact on estimated cost!**
- Catalog statistics are used to estimate filter factors and cardinality

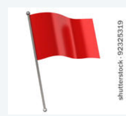
Checking Cardinality Estimates

IBM

```
                 |
             0.0327916
              ^HSJOIN
              (   4)
            1.13515e+06
              677451
        /-------+-------\
2.88279e+09           3.4184e-06
  TBSCAN                TBSCAN
  (   5)                (   6)
1.10132e+06            1839.17
  675933                 1518
    |                     |
2.88279e+09             300520
CO-TABLE: TPCDS      CO-TABLE: TPCDS
STORE_SALES               ITEM
    Q1                     Q2
```

They propagate up the plan and could cause bad plan choices later

Cardinalities < 1 could be under-estimations

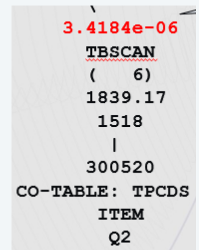


12

Cardinality estimates < 1 should be treated with suspicion because they could be under-estimated. That being said, it is expected for the cardinality to be < 1 if the probability of at least 1 row being returned is small. But this tends to be a rare situation. In order to understand the cardinality estimate, we need to understand what predicates were applied. So let's check the details for TBSCAN(6) …

Checking Predicates

IBM



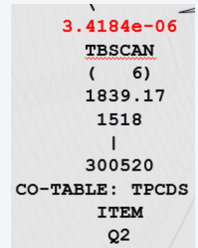
- Check the operator details to see predicates and their filter factors
  - 4 equality predicates with literals

```
6) TBSCAN: (Table Scan)
Predicates:
----------
2) Sargable Predicate,
        Comparison Operator:        Equal (=)
        Filter Factor:              2.52819e-05
        Predicate Text:
        --------------
        (Q2.I_PRODUCT_NAME = 'Zoom ')

3) Sargable Predicate,
        Comparison Operator:        Equal (=)
        Filter Factor:              0.000594525
        Predicate Text:
        --------------
        (Q2.I_BRAND = 'Swoosh ')

4) Sargable Predicate,
        Comparison Operator:        Equal (=)
        Filter Factor:              0.00798552
        Predicate Text:
        --------------
        (Q2.I_CLASS = 'athletic shoes')

5) Sargable Predicate,
        Comparison Operator:        Equal (=)
        Filter Factor:              0.0947691
        Predicate Text:
        --------------
        (Q2.I_CATEGORY = 'Sports ')
```

13

These are the predicates applied at TBSCAN(6) and their filter factors.

Checking Combined Predicate Filtering



- Are the predicates independent or correlated?
- The optimizer assumes they are independent
- In this situation, they appear correlated
  - 'Swoosh' and 'Zoom' probably only occur with 'Sports' and 'athletic shoes'

```
6) TBSCAN: (Table Scan)                       3.4184e-06 =  -- TBSCAN cardinality
Predicates:
----------
(Q2.I_PRODUCT_NAME = 'Zoom ') AND             2.52819e-05 *
(Q2.I_BRAND = 'Swoosh ') AND                  0.000594525 *
(Q2.I_CLASS = 'athletic shoes') AND           0.00798552 *
(Q2.I_CATEGORY = 'Sports ')                   0.0947691 *
                                              300520        -- Table cardinality
```

Verifying Cardinality Estimates

- Confirm the cardinality estimates
- Method 1: COUNT(*) queries
  - This can be tedious and tricky, especially for multiple joins and complex queries

```
SELECT COUNT(*) FROM TPCDS.ITEM AS Q2
WHERE
(Q2.I_PRODUCT_NAME = 'Zoom ') AND
(Q2.I_BRAND = 'Swoosh ') AND
(Q2.I_CLASS = 'athletic shoes') AND
(Q2.I_CATEGORY = 'Sports ')
```

- Method 2: Explain with actual cardinality

Explain with Actual Cardinality

IBM

- Capture cardinality processed by each access plan operator at runtime
- Compare with the optimizer's estimates to identify possible access plan problems
  - **Estimated cardinality is most important input to cost model**
- Use explain from access section mechanism
- Easiest method:

  db2caem -d <dbname> -st "SQL stmt"
  - Db2 Capture Activity Event Monitor data tool

- Fine print: doesn't collect actuals for column-organized processing
- Use Method 1

16

https://www.ibm.com/docs/en/db2/11.5?topic=information-capturing-accessing-section-actuals

Alternative method:

1) WLM setup:

Create workload or use default workload (to collect activity data)

2) Use WLM_SET_CONN_ENV stored procedure for the current connection

call wlm_set_conn_env(null, '<collectactdata>with details, section </collectactdata><collectsectionactuals>base</collectsectionactuals>');

Activate activity event monitor

SET EVENT MONITOR ACTEVMON STATE 1;

Execute SQL statement

Locate SQL statement information in event monitor table to pass to EXPLAIN_FROM_ACTIVITY stored procedure:

```
SELECT APPL_ID,      UOW_ID,      ACTIVITY_ID,      STMT_TEXT FROM ACTIVITYSTMT_ACTEVMON;
-- APPL_ID                  UOW_ID   ACTIVITY_ID   STMT_TEXT
-- ------------------------  --------  --------------  ---------------
-- *N2.DB2INST1.0B5A12222841        1             1  SELECT * FROM ...
```

Populate the explain tables:

CALL EXPLAIN_FROM_ACTIVITY('*N2.DB2INST1.0B5A12222841', 1, 1, 'ACTEVMON', 'MYSCHEMA', ?, ?, ?, ?, ?);

Format the explain tables as usual e.g. db2exfmt

Actual Cardinality Example

IBM

```
    Rows
Rows Actual
  OPERATOR
  (   1)
    Cost
    I/O
```

```
                           |
                       0.0327916
                         75233
                        ^HSJOIN
                        (   4)
                      1.13515e+06
                        677451
                   /-------+-------\
          2.88279e+09              3.4184e-06
          2.88279e+09                  8
            TBSCAN                   TBSCAN
            (   5)                   (   6)
          1.10132e+06               1839.17
            675933                    1518
              |                        |
          2.88279e+09                300520
      CO-TABLE: TPCDS          CO-TABLE: TPCDS
          STORE_SALES                ITEM
              Q1                      Q2
```

Error propagates throughout the plan (Data flows upward in explain graph)

Significant under-estimation!!

17

Correcting for Data Correlation

IBM

- For equality predicates, use *column group statistics* to tell the optimizer about the correlation:

```
RUNSTATS ON TABLE TPCDS.ITEM
       ON COLUMNS ( (I_CATEGORY,I_CLASS,I_BRAND,I_PRODUCT_NAME) )
WITH DISTRIBUTION AND DETAILED INDEXES ALL

SYSSTAT.COLGROUPS.COLGROUPCARD = 37120
```

Note the extra set of parentheses.

Best practice RUNSTATS options

- Represents the number of distinct values in the set of columns
- Statistics and columns are stored in:
  - SYSSTAT.COLGROUPS
  - SYSCAT.COLGROUPCOLS

18

Corrected Cardinality Estimates

IBM

```
      Rows
  Rows Actual
    OPERATOR
    (   1)
      Cost
      I/O

                       |
                    72882.4
                    75233
                   ^HSJOIN
                   (   4)
                 1.13515e+06
                   677451
             /-------+-------\
   2.88279e+09                 7.5977
   2.88279e+09                    8
     TBSCAN                    TBSCAN
     (   5)                    (   6)
   1.10132e+06                 1839.17
     675933                     1518
       |                          |
   2.88279e+09                 300520
CO-TABLE: TPCDS           CO-TABLE: TPCDS
    STORE_SALES                 ITEM
       Q1                        Q2
```

More accurate after the join too.

Estimate much closer to actual with column group statistics

7.5977 = MIN(2.52819e-05, 0.000594525, 0.00798552, 0.0947691, 1 / 37120) * 300520 = 2.52819e-05 * 300520

(Only includes the filter factor of the most filtering predicate)

19

Automatic Column Group Statistics (Db2 11.5)

IBM

- Identifying correlation and specifying RUNSTATS options requires effort
  - IBM Data Management Console provides a **statistics advisor**
  - Recommends statistics options based on SQL statements
  - https://www.ibm.com/docs/en/db2-data-mgr-console/3.1.x?topic=new-version-316
- Db2 does this automatically as part of automatic statistics collection
  - Performs an **automatic discovery** of pair-wise column group statistics
  - Registers a *statistics profile* with the column group statistics options
  - Later automatic statistics collection will use the statistics profile
  - Automatic discovery only occurs during asynchronous (background) collection
  - Controlled by the AUTO_CG_STATS DB configuration parameter
    - OFF by default

20

https://www.ibm.com/docs/en/db2/11.5?topic=oap-collecting-accurate-catalog-statistics-including-advanced-statistics-features

The optimizer uses column group statistics to account for statistical correlation when estimating the combined selectivity of multiple predicates and when computing the number of distinct groupings for operations that group data such as GROUP BY or DISTINCT. Gathering column group statistics can be automated through the automatic statistics collection feature in Db2. Enabling or disabling the automatic collection of column group statistics is done by using the auto_cg_stats database configuration parameter. To enable this function, issue the following command: update db cfg for *dbname* using auto_cg_stats on

The automatic collection of column group statistics will generate a profile describing the statistics that need to be collected. If a user profile does not exist, the background statistics collection will initially perform an automatic discovery of pair-wise column group statistics within the table and set a statistics profile. After the discovery is completed, statistics are gathered on the table using the existing statistics profile feature. The set of column groups discovered is preserved across subsequent discoveries.

If a statistics profile is already manually set, it will be used as is and the discovery is not performed. The automatically generated statistics profile can be used together with any PROFILE option of the RUNSTATS command. If the profile is updated using the UPDATE PROFILE option, any further discovery is blocked on the table, but the set of column group statistics already set in the profile will continue to be collected automatically as well as with a manual RUNSTATS that includes the USE PROFILE option.

The UNSET PROFILE command can be used to remove the statistics profile to restart the discovery process.

To disable this feature, issue the following command: update db cfg for *dbname* using auto_cg_stats off

Disabling this feature will prevent any further discovery, but the statistic profiles will persist and will continue to be used. If there is a need to remove the profile, use the UNSET PROFILE option of RUNSTATS.

Correcting Cardinality Estimates

IBM

- Column group statistics help for equality predicates only
- Try *statistical views* for more complex situations:
  - Correlation among inequality predicates
  - Skew across joins
  - Predicates with expressions
- Use SELECTIVITY clause for more stubborn situations:
  - `WHERE <complex predicate> SELECTIVITY 0.1234`
  - Must set DB2_SELECTIVITY=ALL registry variable
- Future:
  - Machine learned cardinality estimation models
  - Available in tech preview in 11.5.5+

21

https://www.ibm.com/docs/en/db2/11.5?topic=optimization-statistical-views

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 earlier in this presentation 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, price > MSRP + Dealer_markup), relationships spanning multiple tables (for example, product.name = 'Alloy wheels' and product.key = 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.

Correlated Sub-plans

- "Correlated" means referencing columns outside the current sub-select

```
SELECT T1.CODE,
 (SELECT A.CDATE FROM T2 A WHERE A.PID = T1.PID AND A.VERS =
  (SELECT MIN(B.VERS) FROM T2 B
   WHERE B.PID = T1.PID AND B.CODE = T1.CODE) AS MINVERS)
FROM T1
```

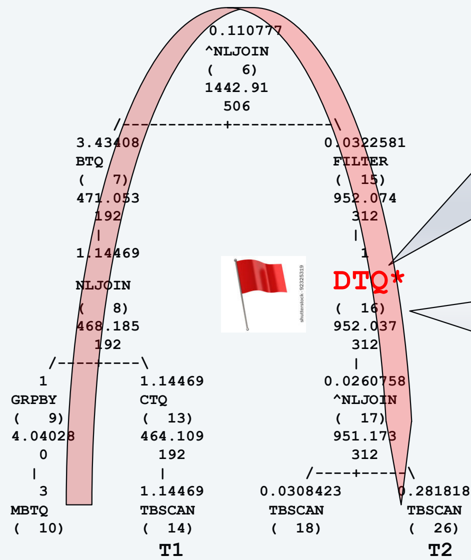
- Correlation is often expensive to process
- The optimizer tries to remove correlation by automatically rewriting the query
  - It can't do it in every situation

22

The query on this page has a scalar sub-select (in purple) in the select-list of the outermost select (in black). The scalar sub-select applies a scalar subquery (in teal, or whatever that bluish colour is ;-)). Both scalar sub-selects reference columns from T1 which is referenced in the outermost select. These references to T1 are correlated references which means that the scalar sub-selects must be executed for every qualifying T1 row.

Looking for Correlated Sub-plans

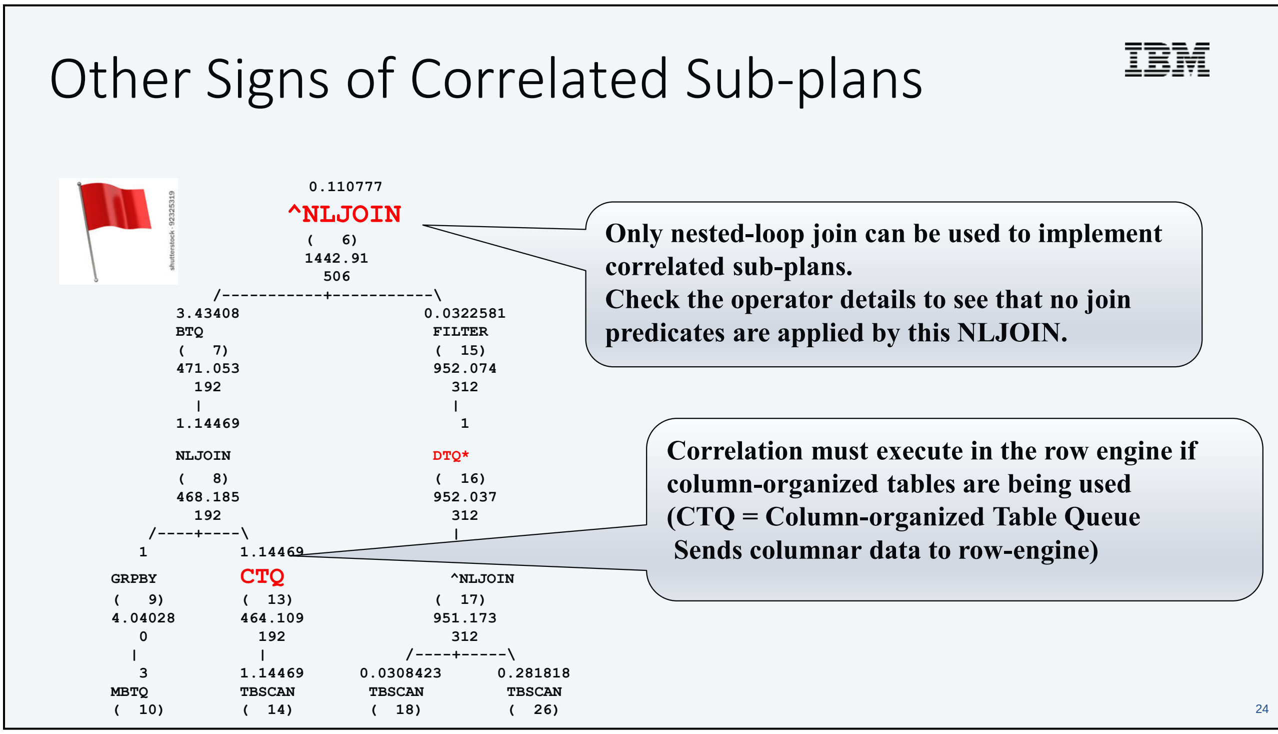
IBM



Correlation is very expensive in a partitioned DB (MPP) system.
Look for "listener table queues" (denoted by *).
The sub-plan below the TQ* is re-executed for every outer row.

T1.PID must be passed from one side of the plan to the other.
The sub-plan from FILTER(15) and below must be executed for every T1 row.

T1.PID = T2.PID



Another clue that there are correlated sub-plans is nested-loop join operators (NLJOIN) with no join predicates. This means that the inner (RHS) of NLJOIN is re-executed for every outer row. The correlated references are somewhere in the NLJOIN inner and they could be in predicates or select-list items. Identify the correlated references in the optimized SQL and then check the operator details to locate them.

One issue with correlated sub-plans is that they cannot execute in the columnar runtime if the statement references column-organized tables. This means that the NLJOIN that drives the correlated sub-plan executes in the row-organized runtime. This can be identified by looking for column-organized table queue (CTQ) operators lower in the access plan. The CTQ operator passes data from the columnar runtime to the row-organized runtime. Ideally there should only be one CTQ in the access plan and it should be near the very top of the plan.

Dealing with Correlated Sub-plans

- Create indexes if correlated references are in predicates
  - E.g. create an index on T2.PID in previous example
  - This allows the T2.PID=T1.PID predicate to be applied more efficiently
- Try a higher optimization level
  - The optimizer might be able to decorrelate
- Manually rewrite the query to remove the correlation

Check for Risky HSJOIN Build Tables

IBM

```
                                0.000173087
                                  HSJOIN
                                  (   8)
                                  456522
                                  122652
           /--------------------------------+--------------------------------\
      1000                                                              0.000173087
     TBSCAN                                                               ^HSJOIN
     (   9)                                                               (  10)
     62.0912                                                              456460
        9                                                                 122643
        |                                                     /------------+------------\
      1000                                              0.00106296                   47.0591
CO-TABLE: DBUSER                                          ^HSJOIN                    TBSCAN
    CUSTOMER                                              (  11)                     (  22)
       Q4                                                 456398                     62.0365
                                                          122634                        9
                                          /------------+------------\                   |
                                    2.88279e+09                    1.00003             289
                                      TBSCAN                       TBSCAN        CO-TABLE: DBUSER
                                      (  12)                       (  21)             STORE
                                      456336                       62.018               Q3
                                      122625                          9
                                        |                             |
                                    2.88279e+09                      208
                                 CO-TABLE: DBUSER              CO-TABLE: DBUSER
                                    DAILY_SALES                      PROMO
                                        Q1                            Q2
```

Plan tree starts with a large table

Cardinality is < 1 and the plan tree is on the build side of HSJOIN(8)

This could perform badly if the actual cardinality is large, due to excessive memory usage and/or spilling to a system temporary table

26

Check for Spilling SORTs

IBM

```
                 TBSCAN
                 (   3)
               8.90346e+07
               2.08077e+07
                    |
               2.71211e+09
                  SORT
                 (   4)
               7.53051e+07
             1.07422e+07
                    |
               2.71211e+09
                 ^HSJOIN
                 (   5)
               1.14095e+06
                 676797
          /-------+-------\
  2.88279e+09              282726
    TBSCAN                 TBSCAN
    (   6)                 (   7)
  1.10132e+06              1044.65
    675933                   864
      |                       |
  2.88279e+09              300520
CO-TABLE: TPCDS        CO-TABLE: TPCDS
  STORE_SALES               ITEM
      Q1                     Q2
```

SORT's I/O estimate increases
1.07422e+07 – 676,797 = 10,065,403 pages

Check SORT operator details:

ROWWIDTH: (Estimated width of rows)
112.250000
SPILLED : (Pages spilled to bufferpool or disk)
1.00654e+07
TEMPSIZE: (Temporary Table Page Size)
32768

Optimizer doesn't choose the specific temp tablespace but it does choose the page size

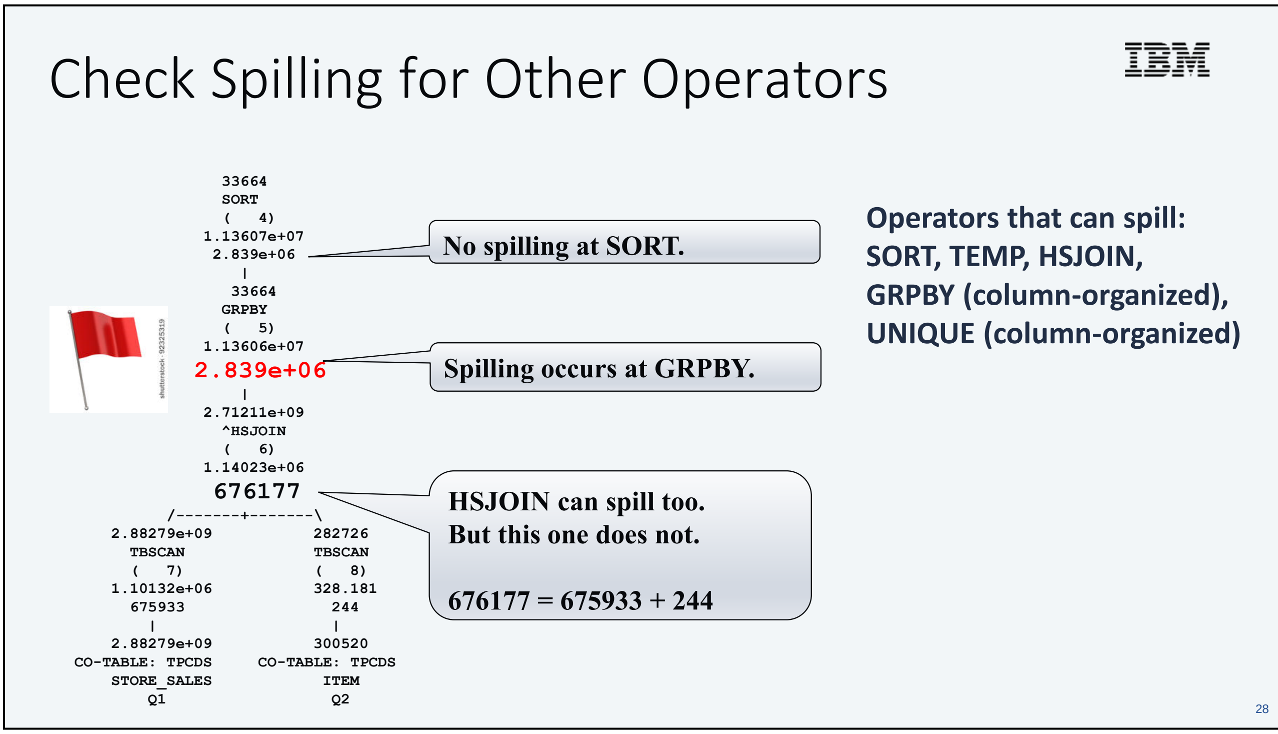
27

Spilling SORTs are those that can't fit in memory (sortheap) so they are written to temporary tables. If the temporary table doesn't fit in the buffer pool it will be written to disk. The optimizer tries to model this and will include the extra I/O cost.

Check for increases in I/O estimate at the SORT operator.

It might be necessary to increase the sortheap or bufferpool size to avoid spilling to disk.

The optimizer doesn't choose a specific system temporary tablespace but it does choose the page size which indirectly determines the tablespace. Check that the tablespaces and bufferpools for the chosen page size have enough space.



GROUP BY and UNIQUE operations processing column-organized data use a hash table that is stored in sortheap memory. Their memory consumption can be significant and could spill to bufferpool and disk. Check them out too.

Expanding (M:N) Joins

IBM

```
                  |
             2.13973e+10
               HSJOIN
               (   4)
             1.42235e+06
               791354
          /-------+-------\
 2.88279e+09             2.54618e+06
    TBSCAN                  TBSCAN
    (   5)                  (   6)
 1.10132e+06               242310
    675933                  115421
      |                       |
 2.88279e+09             7.83762e+08
CO-TABLE: TPCDS        CO-TABLE: TPCDS
   STORE_SALES            INVENTORY
      Q1                     Q2
```

Join cardinality is larger than either of its input plans.

Does this make sense considering the schema?

Is a join specification (predicate) missing?

Should another table have been joined first?

Is a DISTINCT needed to remove duplicates?

Nested-loop Join with Inner Scan

IBM

```
                 |
             2.72162e+10
               NLJOIN
               (   3)
             3.71737e+09
             2.07853e+09
           /-----+------\
        475              5.72972e+07
      TBSCAN               TBSCAN
      (   4)               (   5)
      301.283            7.82605e+06
        215              4.37585e+06
         |                   |
       73049             2.88279e+09
 TABLE: TPCDS         TABLE: TPCDS
   DATE_DIM_R           STORE_SALES_R
        Q1                   Q2
```

NLJOINs with inner TBSCANs can be expensive because the TBSCAN occurs for every outer row (475 in this example).
Very expensive if the inner table is large.

Why isn't HSJOIN used?
- No equality predicates?
If only inequality predicates, is there an index?
If there is an index, why wasn't it chosen?
- A FETCH is required and index is poorly clustered?
- Index can't apply predicates using start/stop keys?

Expensive Index Scans (1|3)

IBM

```
                    37654.1
                    NLJOIN
                    (   3)
                  4.2062e+06
                  1.70648e+06
             /--------+--------\
       0.954141                39463.8
        TBSCAN                  FETCH
        (   4)                  (   5)
       301.283                4.2059e+06
         215                  1.70627e+06
          |                  /----+-----\
        73049        39463.8          2.88279e+09
    TABLE: TPCDS      IXSCAN        TABLE: TPCDS
      DATE_DIM_R      (   6)          STORE_SALES_R
         Q1        3.92896e+06
                   1.66698e+06
                        |
                   2.88279e+09
                  INDEX: TPCDS
                     SSR_IX1
                       Q2
```

```
3) Sargable Predicate,
    Comparison Operator:      Equal (=)
    Subquery Input Required:  No
    Filter Factor:            1.36894e-05
    Predicate Text:
    --------------
    (Q2.SS_SOLD_DATE_SK = Q1.D_DATE_SK)

Name:      SSR_IX1
Type:      Index
Columns in index:
    SS_ITEM_SK(A)
    SS_SOLD_DATE_SK(A)
```

Index filters well but I/O is very high.
Check operator details to see how the predicates are applied.
Start/stop keys should be used unless index is used to avoid a SORT.

31

The index definition shows that the join column is not leading in the index and there is no other predicate to cover the leading column. Consider reversing the columns in the existing index or creating a new index with the columns reversed.

Expensive Index Scans (2|3)

IBM

Expensive FETCH that doesn't reduce the cardinality. Can it be avoided by adding more columns to the index?

```
                37654.1
                NLJOIN
                (   3)
               4.2062e+06
               1.70648e+06
           /--------+--------\
      0.954141               39463.8
       TBSCAN                 FETCH
       (   4)                 (   5)
      301.283              4.2059e+06
        215                1.70627e+06
         |                /----+-----\
       73049        39463.8        2.88279e+09
   TABLE: TPCDS      IXSCAN        TABLE: TPCDS
     DATE_DIM_R      (   6)         STORE_SALES_R
        Q1         3.92896e+06          Q2
                   1.66698e+06
                        |
                   2.88279e+09
                 INDEX: TPCDS
                     SSR_IX1
                       Q2
```

```
5) FETCH  : (Fetch)
Input Streams:
-------------
4) From Operator #6
     Column Names:
     ------------
     +Q2.$RID$
5) From Object TPCDS.STORE_SALES_R
     Column Names:
     ------------
     +Q2.SS_SALES_PRICE

Output Streams:
--------------
6) To Operator #3
     Column Names:
     ------------
     +Q2.SS_SALES_PRICE
```

32

The FETCH operator details show that it doesn't apply any predicates so it must only exist to retrieve columns. The stream information shows that SS_SALES_PRICE is being fetched because it isn't included in the index.

Expensive Index Scans (3|3)

IBM

```
    37654.1
     NLJOIN
     (   3)
   4.2062e+06
   1.70648e+06
```

```
                  37654.1
                  NLJOIN
                  (   3)
                  388.342
                  237.78
              /-----+------\
       0.954141             39463.8
        TBSCAN              IXSCAN
        (   4)              (   5)
        301.283             89.9253
          215               23.7796
           |                  |
         73049           2.88279e+09
     TABLE: TPCDS        INDEX: TPCDS
       DATE_DIM_R           SSR_IX2
          Q1                  Q2
```

Creating a better index with the join column leading and including the fetched column avoids the FETCH and results in a much cheaper IXSCAN.

Name: SSR_IX2
Type: Index
Columns in index:
- SS_SOLD_DATE_SK(A)
- SS_SALES_PRICE(A)

33

A good index can make a world of difference. The NLJOIN cost has dropped dramatically. Use explain to verify that it worked.

db2exfmt - Extended diagnostic information

- Explain diagnostic messages could indicate problems:

```
EXP0020W Table has no statistics. The table "DB2DBA"."SALES" has not had runstats run on
  it. This may result in a sub-optimal access plan and poor performance.

EXP0060W The following materialized query table (MQT) or statistical view was not eligible
  for query optimization: "DB2DBA"."SV_STORE". The MQT cannot be used for query
  optimization because one or more tables, views or subqueries specified in the MQT could
  not be found in the query that is being explained.

EXP0147W  The following statistical views may have been used by the optimizer to estimate
cardinalities: "DB2DBA"."SV_STORE".
```

Explain Diagnostic Messages

IBM

- Explain can provide helpful information such as:
  - Notification about missing statistics
  - Information about whether or not materialized query tables (MQTs) or statistical views could be matched
  - Syntax errors when using optimization profiles
  - More will be added in future releases
- Messages are recorded in:
  - EXPLAIN_DIAGNOSTICS
  - EXPLAIN_DIAGNOSTICS_DATA

Check the RETURN operator details

IBM

```
1) RETURN: (Return Result)
Arguments:
---------
BLDLEVEL: (Build level)
        DB2 v11.1.9.0 : s1901181500
ENVVAR  : (Environment Variable)
        DB2_ANTIJOIN=EXTEND
        DB2_REDUCED_OPTIMIZATION=YES[Embedded Optimization Guidelines]
HEAPUSE : (Maximum Statement Heap Usage)
        6240 Pages
PLANID  : (Access plan identifier)
        3ecc6fdf9ece8198
PREPTIME: (Statement prepare time)
        2856 milliseconds
SQLCA   : (Warning SQLCA from compile)
        SQLCODE 437; Function SQLNO26D; Message token '3'; Warning 'None'
STMTHEAP: (Statement heap size)
        16384
```

Registry variables that affect query optimization. Indicates how they are set.

Is STMTHEAP use reasonable considering the query complexity? (Try reducing opt level)

Is prepare time reasonable considering the query complexity? (Try reducing opt level)

SQL0437W rc 3 indicates an optimizer cost underflow. Likely due to cardinality under-estimation.

Speaker: John Hornibrook
Company: IBM Canada
Email Address: jhornibr@ca.ibm.com

John is a Senior Technical Staff Member responsible for relational database query optimization on IBM's distributed platforms. This technology is part of Db2 for Linux, UNIX and Windows, Db2 Warehouse, Db2 on Cloud, IBM Integrated Analytics System (IIAS) and Db2 Big SQL. John also works closely with users to help them fully realize the benefits of IBM's relational DB technology products.