15
hroug

19.-23.10.2010, Rovinj, Croatia

Query Transformations

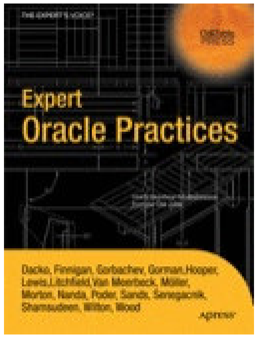
Jože Senegačnik
Oracle ACE Director
joze.senegacnik@dbprof.com

About the Speaker

Jože Senegačnik

- Located in Slovenia
- Registered private researcher
- First experience with Oracle Version 4 in 1988
- 21+ years of experience with Oracle RDBMS.
- Proud member of the OakTable Network www.oaktable.net
- Oracle ACE Director
- Co-author of the OakTable book “Expert Oracle Practices” by Apress (Jan 2010)
- VP of Slovenian OUG (SIOUG) board
- CISA – Certified IS auditor
- Blog about Oracle: http://joze-senegacnik.blogspot.com

- PPL(A) – private pilot license / night qualified
- Blog about flying: http://jsenegacnik.blogspot.com
- Blog about Building Ovens, Baking and Cooking: http://senegacnik.blogspot.com



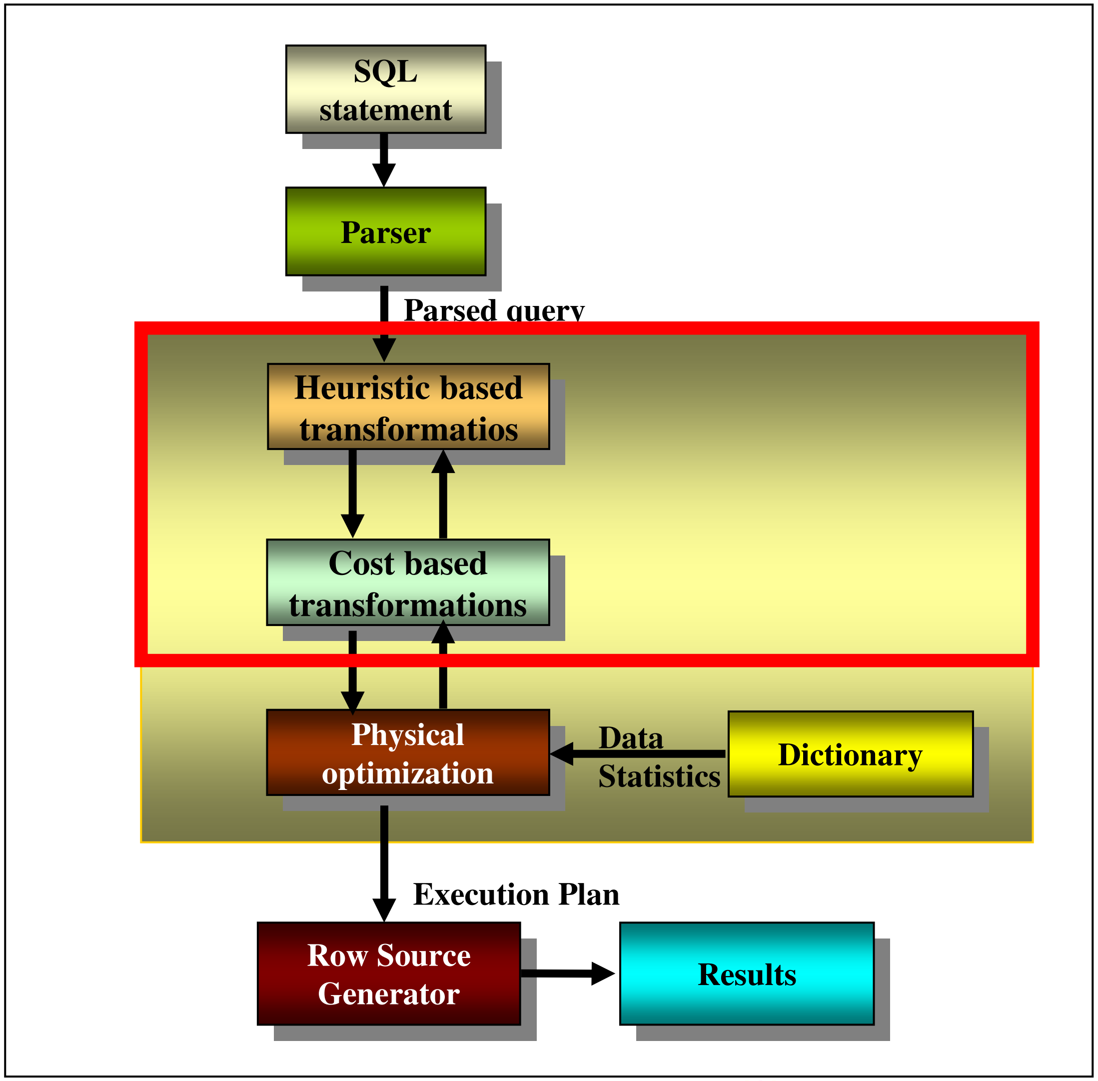
Agenda

- Introduction
- Query Transformations
- Conclusions

Cost Based Optimizer Trace (event 10053)

- The following abbreviations are used in the optimizer trace:
  - JPPD - join predicate push-down
  - OJPPD - old-style (non-cost-based) JPPD
  - FPD - filter push-down
  - PM - predicate move-around
  - CVM - complex view merging
  - SPJ - select-project-join
  - SJC - set join conversion
  - SU - subquery unnesting
  - OBYE - order by elimination
  - OST - old style star transformation
  - ST - new (cbqt) star transformation
  - CNT - count(col) to count(*) transformation
  - JE - Join Elimination
  - JF - join factorization
  - SLP - select list pruning
  - DP - distinct placement

SQL Statement Processing



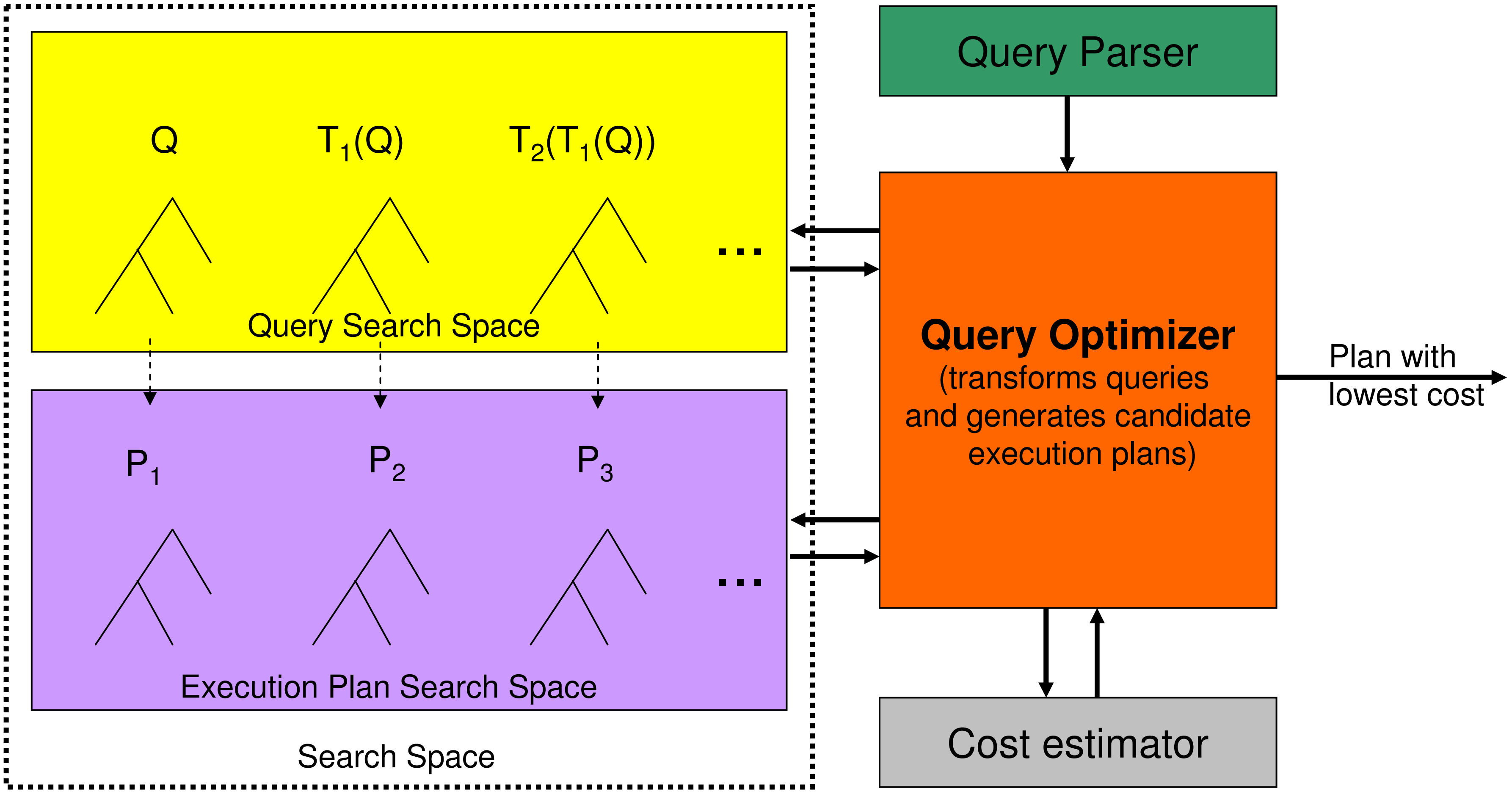
Query Optimization

- Query optimization is performed in two phases

1. **Logical optimization** (query transformation)

2. **Physical optimization** – finds information
    - Possible access method to every table (full scan, index lookup,...)
    - Possible join method for every join (HJ, SM, NL)
    - Join order for the query tables (join(join(A,B), C)

Query Optimization



Why Query Transformations?

- The goal of transformation is to enhance the query performance.
- Transformation generates semantically equivalent form of statement, which produces the same results, but significantly differs in performance.
- Transformation rely on algebraic properties that are not always expressible in SQL, e.g, anti-join and semi-join.

Transformations

- CBO supports different approaches:
  - Automatic – which always produce a faster plan
  - Heuristic-based
    - Prior to 10gR1
    - Assumption – produce faster plan most of the time
    - User has to set parameters or use hints
  - Cost-based
    - Since 10gR1
    - Transformation does not always produce a faster query
    - Query optimizer costs non transformed query and transformed query and picks the cheapest form
    - No need to set parameters or use hints
- Transformation may span more than one query block

Query Transformations

SU - Subquery Unnesting

SU - subquery unnesting

- Original may be sub-optimal because of multiple, redundant re-evaluation of the sub-query
- Un-nesting
  - sub-query converted into an inline view connected using a join, then merged into the outer query
  - Enables new access paths, join orders, join methods (anti-/semi-join)
- A wide variety of un-nesting
  - Any (IN), All (NOT IN), [NOT] EXISTS, correlated, uncorrelated, aggregated, group by
- Some are automatic; what used to be heuristic-based is cost-based since Oracle10g
- Related optimizer hints: UNNEST, NO_UNNEST

SU - unnesting NOT EXISTS

```
SELECT c.cust_id, c.cust_first_name, c.cust_last_name
FROM customers c
WHERE NOT EXISTS
   (SELECT 1
    FROM orders o
    WHERE o.cust_id = c.cust_id);
```

⬇

```
SELECT c.cust_id, c.cust_first_name, c.cust_last_name
FROM customers c, orders o
WHERE c.cust_id A= o.cust_id;
```

Execution Plan for NOT EXISTS

```
SQL> select cust_id,cust_first_name,cust_last_name
       from customers c
       where not exists ( select 1 from sales s where s.cust_id = c.cust_id);
```

```
---------------------------------------------------------
| Id  | Operation                      | Name           |
---------------------------------------------------------
|   0 | SELECT STATEMENT               |                |
|*  1 |  HASH JOIN ANTI                |                |
|   2 |   TABLE ACCESS FULL            | CUSTOMERS      |
|   3 |   PARTITION RANGE ALL          |                |
|   4 |    BITMAP CONVERSION TO ROWIDS |                |
|   5 |     BITMAP INDEX FAST FULL SCAN| SALES_CUST_BIX |
---------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - access("S"."CUST_ID"="C"."CUST_ID")
```

Tables are from SH demo schema.

Excerpt from CBO Trace

```
...
*****************************
Cost-Based Subquery Unnesting
*****************************
SU: Unnesting query blocks in query block SEL$1 (#1)
    that are valid to unnest.
    Subquery Unnesting on query block SEL$1
(#1)SU: Performing unnesting that does not require costing.
SU: Considering subquery unnest on query block SEL$1 (#1).
SU: Checking validity of unnesting subquery SEL$2 (#2)
SU: Passed validity checks.
SU: Unnesting subquery query block SEL$2
(#2)SU: Transform ALL/NOTEXISTS subquery into a regular
   antijoin.
Registered qb: SEL$5DA710D3 0x211bdab0 (SUBQUERY UNNEST
   SEL$1; SEL$2)
...
```

SU - unnesting EXISTS

```
SELECT c.cust_id, c.cust_first_name, c.cust_last_name
FROM customers c
WHERE EXISTS
   (SELECT 1
    FROM orders o
    WHERE o.cust_id = c.cust_id);
```

⬇

```
SELECT c.cust_id, c.cust_first_name, c.cust_last_name
FROM customers c, orders o
WHERE c.cust_id S= o.cust_id;
```

Execution Plan for EXISTS

```
SQL> select cust_id,cust_first_name,cust_last_name
       from customers c
       where exists ( select 1 from sales s where s.cust_id = c.cust_id);
```

```
---------------------------------------------------------
| Id  | Operation                     | Name            |
---------------------------------------------------------
|   0 | SELECT STATEMENT              |                 |
|*  1 |  HASH JOIN SEMI               |                 |
|   2 |   TABLE ACCESS FULL           | CUSTOMERS       |
|   3 |   PARTITION RANGE ALL         |                 |
|   4 |    BITMAP CONVERSION TO ROWIDS |                |
|   5 |     BITMAP INDEX FAST FULL SCAN| SALES_CUST_BIX |
---------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - access("S"."CUST_ID"="C"."CUST_ID")
```

Tables are from SH demo schema.

Excerpt from CBO Trace

```
...
*****************************
Cost-Based Subquery Unnesting
*****************************
SU: Unnesting query blocks in query block SEL$1 (#1) that
    are valid to unnest.
Subquery Unnesting on query block SEL$1
(#1)SU: Performing unnesting that does not require costing.
SU: Considering subquery unnest on query block SEL$1 (#1).
SU: Checking validity of unnesting subquery SEL$2 (#2)
SU: Passed validity checks.
SU: Transforming EXISTS subquery to a join.
...
```

SU - unnesting aggregated sub-query

```
SELECT distinct p.prod_id, p.prod_name
FROM products p, sales s
WHERE p.prod_id = s.prod_id
AND s.quantity_sold < (SELECT AVG (quantity_sold)
                       FROM sales
                       WHERE prod_id = p.prod_id);
```

```
SELECT distinct p.prod_id, p.prod_name
FROM products p, sales s,
    (SELECT AVG (quantity_sold) as avgqnt, prod_id
     FROM sales
     GROUP BY prod_id) v
WHERE p.prod_id = s.prod_id
AND s.quantity_sold < v.avgqnt
AND v.prod_id = s.prod_id;
```

Execution Plan for Un-transformed Query

```
--------------------------------------------------------------+
| Id  | Operation                                    | Name          |
--------------------------------------------------------------+
| 0   | SELECT STATEMENT                      |               |
| 1   |  FILTER                               |               |
| 2   |   HASH JOIN                           |               |
| 3   |    TABLE ACCESS FULL                  | PRODUCTS      |
| 4   |    PARTITION RANGE ALL                |               |
| 5   |     TABLE ACCESS FULL                 | SALES         |
| 6   |   SORT AGGREGATE                      |               |
| 7   |    PARTITION RANGE ALL                |               |
| 8   |     TABLE ACCESS BY LOCAL INDEX ROWID | SALES         |
| 9   |      BITMAP CONVERSION TO ROWIDS      |               |
| 10  |       BITMAP INDEX SINGLE VALUE       | SALES_PROD_BIX|
--------------------------------------------------------------+
Predicate Information:
----------------------
1 - filter("S"."QUANTITY_SOLD"<)
2 - access("P"."PROD_ID"="S"."PROD_ID")
10 - access("PROD_ID"=:B1)
```

Execution Plan for Transformed Query

```
-----------------------------------------------------
| Id  | Operation                | Name     | Rows  |
-----------------------------------------------------
|   0 | SELECT STATEMENT         |          |  3615 |
|   1 |  HASH UNIQUE             |          |  3615 |
|*  2 |   HASH JOIN              |          | 56104 |
|   3 |    TABLE ACCESS FULL     | PRODUCTS |    72 |
|*  4 |    HASH JOIN             |          | 56104 |
|   5 |     VIEW                 |          |    72 |
|   6 |      HASH GROUP BY       |          |    72 |
|   7 |       PARTITION RANGE ALL|          |   918K|
|   8 |        TABLE ACCESS FULL | SALES    |   918K|
|   9 |     PARTITION RANGE ALL  |          |   918K|
|  10 |      TABLE ACCESS FULL   | SALES    |   918K|
-----------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("P"."PROD_ID"="S"."PROD_ID")
   4 - access("V"."PROD_ID"="S"."PROD_ID")
       filter("S"."QUANTITY_SOLD"<"V"."AVGQNT")
```

What CBO Really Does is ...

```
SELECT DISTINCT P.PROD_ID ITEM_1,
       P.PROD_NAME ITEM_2,
       CASE WHEN S.QUANTITY_SOLD <
            AVG(S.QUANTITY_SOLD) OVER ( PARTITION BY S.PROD_ID)
            THEN S.ROWID END VW_COL_3
FROM   SH.SALES S,SH.PRODUCTS P
WHERE  P.PROD_ID=S.PROD_ID
```

```
--------------------------------------------
| Id  | Operation                 | Name     |
--------------------------------------------
|   0 | SELECT STATEMENT          |          |
|   1 |  HASH UNIQUE              |          |
|*  2 |   VIEW                    | VW_WIF_1 |
|   3 |    WINDOW SORT            |          |
|*  4 |     HASH JOIN             |          |
|   5 |      TABLE ACCESS FULL    | PRODUCTS |
|   6 |      PARTITION RANGE ALL|          |
|   7 |       TABLE ACCESS FULL | SALES    |
--------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - filter("VW_COL_3" IS NOT NULL)
   4 - access("P"."PROD_ID"="S"."PROD_ID")
```

FPD – Filter Push Down

FPD – Filter Push Down (1)

```
select distinct c4
from

   (select /*+ no_merge */ c4, count(*) cnt
    from t1 group by c4) a

where a.cnt > 100
```

```
----------------------------------------+------------------------------------+
| Id  | Operation             | Name    | Rows  | Bytes | Cost  | Time       |
----------------------------------------+------------------------------------+
| 0   | SELECT STATEMENT      |         |       |       |     4 |            |
| 1   |  VIEW                 |         |     1 |    13 |     4 |  00:00:01  |
|*2   |   FILTER              |         |       |       |       |            |
| 3   |    HASH GROUP BY      |         |     1 |     3 |     4 |  00:00:01  |
| 4   |     TABLE ACCESS FULL | T1      |  1000 |  3000 |     3 |  00:00:01  |
----------------------------------------+------------------------------------+
Predicate Information:
----------------------
2 - filter(COUNT(*)>100)
```

- a.cnt > 100 is pushed inside subquery

FPD – Filter Push Down (2)

- Excerpt from CBO trace

```
********************************
COST-BASED QUERY TRANSFORMATIONS
********************************
FPD: Considering simple filter push (pre rewrite) in query block SEL$1 (#0)
FPD:  Current where clause predicates "A"."CNT">100

try to generate transitive predicate from check constraints for query block
   SEL$1 (#0)
finally: "A"."CNT">100

FPD:   Following are pushed to having clause of query block SEL$2 (#0)
COUNT(*)>100
FPD: Considering simple filter push (pre rewrite) in query block SEL$2 (#0)
FPD:  Current where clause predicates  ??
```

View Merging

View Merging

- Views are created for several reasons
  - Security
  - Abstraction (factorize same work performed by many queries)
  - Describe business logic
- However, they are used in different contexts
  - Filter on a view column
  - Join to tables or other views
  - Order by or group by on view column(s)
- View *merging*
  - Allows optimizer to explore more plans, e.g, enabled access paths or consider more join orders

View Merging

- *Simple view*
  - Select-Project-Join
  - Merged automatically as it is always better.
- *Complex view*
  - Aggregation / group by, distinct, or outer-join
  - Complex view merging was heuristic-based;
  - It is cost-based in 10g
- In the following examples, in-line views are used to make it easy to see the view definition.
- All optimizations related to views apply to both inline views and user-defined views.

Select-Project-Join View Merging

```
SELECT t1.x, v.z
  FROM t1, t2, (SELECT t3.z, t4.m
                  FROM t3, t4
                 WHERE t3.k = t4.k AND t4.q = 5) v
 WHERE t2.p = t1.p AND t2.m = v.m;
```

⬇

```
SELECT t1.x, t3.z
  FROM t1, t2, t3, t4
 WHERE t2.p = t1.p AND t2.m = t4.m AND t3.k = t4.k AND t4.q = 5;
```

CVM - complex view merging

```
SELECT e1.last_name, e1.salary, v.avg_salary
   FROM employees e1,
   (SELECT department_id, avg(salary) avg_salary
      FROM employees e2
      GROUP BY department_id) v
WHERE e1.department_id = v.department_id
AND e1.salary > v.avg_salary;
```

↓

```
SELECT e1.last_name last_name,
       e1.salary salary, avg(e2.salary) avg_salary
FROM hr.employees e1, hr.employees e2
WHERE e1.department_id = e2.department_id
GROUP BY e2.department_id,e1.rowid,e1.salary,e1.last_name
HAVING e1.salary > avg(e2.salary)
```

PM - predicate move-around

PM - predicate move-around (1)

- Moves inexpensive predicates into view query blocks in order to perform earlier filtering.
- Generates filter predicates based on
  - transitivity or
  - functional dependencies.
- Filter predicates are moved through SPJ, GROUP BY, DISTINCT views and views with OLAP constructs
- Copies of filter predicates can be moved up, down, and across query blocks.
- Enables new access paths and reduce the size of data that is processed later in more costly operations like joins or aggregations.
- It is performed automatically

PM - predicate move-around (2)

```
SELECT v1.k1, v2.q, max1
  FROM (SELECT   t1.k AS k1, MAX (t1.a) AS max1
            FROM t1, t2
           WHERE t1.k = 6 AND t1.z = t2.z
        GROUP BY t1.k) v1,
       (SELECT t1.k AS k2, t3.q AS q
          FROM t1, t3
         WHERE t1.y = t3.y AND t3.z > 4) v2
 WHERE v1.k1 = v2.k2 AND max1 > 50;
```

```
SELECT v1.x, v2.q, max1
  FROM (SELECT   t1.k AS k1, MAX (t1.a) AS max1
            FROM t1, t2
           WHERE t1.k = 6 AND t1.z = t2.z AND t1.a > 50
        GROUP BY t1.k) v1,
       (SELECT t1.k AS k2, t3.q AS q
          FROM t1, t3
         WHERE t1.y = t3.y AND t3.z > 4 AND t1.k = 6) v2
 WHERE v1.k1 = v2.k2;
```

JPPD - join predicate push-down

JPPD - join predicate push-down (1)

- Many types of views can not be merged; e.g., views containing UNION ALL/UNION; anti-/semi-joined views; some outer-joined views
- As an alternative, join predicates can be pushed inside unmerged views
- A pushed-down join predicate acts as a correlating condition inside the view and opens up new access paths e.g., index based nested-loop join
- Decision to do JPPD is cost-based

JPPD - join predicate push-down (2)

```
SELECT t1.c, t2.x
  FROM t1, t2, (SELECT t4.x, t3.y
                  FROM t4, t3
                 WHERE t3.p = t4.q AND t4.k > 4) v
 WHERE t1.c = t2.d AND t1.x = v.x(+) AND t2.d = v.y(+);
```

⬇

```
SELECT t1.c, t2.x
  FROM t1,
       t2,
    (SELECT t4.x, t3.y
       FROM t4, t3
      WHERE t3.p = t4.q AND t4.k > 4 AND t1.x = t4.x AND t2.d = t3.y) v
 WHERE t1.c = t2.d;
```

JF – Join Factorization

JF – Join Factorization

- Purpose:
  - Branches of UNION / UNION ALL that join a common table are combined to reduce the number of accesses to this common table.
- If this transformation is applied then the VW_JF* in the execution plan is a result of the join factorization.

JF – Join Factorization

```
(SELECT A1.C1 C1, A2.C2 C2
   FROM JOC.A1 A1, JOC.A2 A2
  WHERE A1.C1 = A2.C3 AND A1.C1 > 1)
UNION ALL
(SELECT A1.C1 C1, A2.C2 C2
   FROM JOC.A1 A1, JOC.A2 A2
  WHERE A1.C1 = A2.C3 AND A1.C1 > 20)
```

↓

```
SELECT VW_JF_SEL$906F71F0.C1 C1, VW_JF_SEL$906F71F0.C2 C2
  FROM (SELECT VW_JF_SET$48F2D741.ITEM_2 C1, A2.C2 C2
          FROM ( (SELECT A1.C1 ITEM_1, A1.C1 ITEM_2
                     FROM JOC.A1 A1
                    WHERE A1.C1 > 1)
                 UNION ALL
                 (SELECT A1.C1 ITEM_1, A1.C1 ITEM_2
                    FROM JOC.A1 A1
                   WHERE A1.C1 > 20)) VW_JF_SET$48F2D741,
               JOC.A2 A2
         WHERE VW_JF_SET$48F2D741.ITEM_1 = A2.C3) VW_JF_SEL$906F71F0
```

union all
operation

JF – Join Factorization

```
(SELECT A1.C1 C1, A2.C2 C2
   FROM JOC.A1 A1, JOC.A2 A2
  WHERE A1.C1 = A2.C3 AND A1.C1 > 1)
UNION ALL
(SELECT A1.C1 C1, A2.C2 C2
   FROM JOC.A1 A1, JOC.A2 A2
  WHERE A1.C1 = A2.C3 AND A1.C1 > 20)
```

```
---------------------------------------------------------------------------------------------
| Id  | Operation             | Name                 | Rows  | Bytes | Cost (%CPU)| Time     |
---------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |                      |   10M|   300M|  7568  (11)| 00:00:22 |
|*  1 |  HASH JOIN            |                      |   10M|   300M|  7568  (11)| 00:00:22 |
|   2 |   VIEW                | VW_JF_SET$48F2D741   |     2 |    52 |  5008   (8)| 00:00:15 |
|   3 |    UNION-ALL          |                      |       |       |            |          |
|*  4 |     TABLE ACCESS FULL| A1                   |     1 |     2 |  2504   (8)| 00:00:08 |
|*  5 |     TABLE ACCESS FULL| A1                   |     1 |     2 |  2504   (8)| 00:00:08 |
|   6 |   TABLE ACCESS FULL  | A2                   | 5242K|    20M|  2377   (8)| 00:00:07 |
---------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - access("ITEM_1"="A2"."C3")
   4 - filter("A1"."C1">1)
   5 - filter("A1"."C1">20)
```

JE - Join Elimination

JE - Join Elimination (1)

- Eliminate unnecessary joins if there are constraints defined on join columns. If join has no impact on query results it can be eliminated.
  - e.departmens_id is foreign key and joined to primary key d.department_id
- Eliminate unnecessary outer joins – doesn't even require primary key – foreign key relationship to be defined.

```
SQL> select e.first_name, e.last_name, e.salary
     from employees e,
          departments d
     where e.department_id = d.department_id;
```

```
--------------------------------------+-----------------------------------+
| Id  | Operation          | Name     | Rows  | Bytes | Cost  | Time        |
--------------------------------------+-----------------------------------+
| 0   | SELECT STATEMENT   |          |       |       |     3 |             |
| 1   |  TABLE ACCESS FULL | EMPLOYEES|   106 |  2332 |     3 |  00:00:01 |
--------------------------------------+-----------------------------------+
Predicate Information:
----------------------
1 - filter("E"."DEPARTMENT_ID" IS NOT NULL)
```

JE - Join Elimination (2)

- Excerpt from CBO trace

```
...
JE:   Considering Join Elimination on query block SEL$1 (#0)
*************************
Join Elimination (JE)
*************************
JE:   cfro: EMPLOYEES objn:70291 col#:11 dfro:DEPARTMENTS
   dcol#:11
Query block (26649C50) before join elimination:
SQL:******* UNPARSED QUERY IS *******
SELECT "E"."FIRST_NAME" "FIRST_NAME","E"."LAST_NAME"
   "LAST_NAME","E"."SALARY" "SALARY" FROM "HR"."EMPLOYEES"
   "E","HR"."DEPARTMENTS" "D" WHERE
   "E"."DEPARTMENT_ID"="D"."DEPARTMENT_ID"
JE:   eliminate table: DEPARTMENTS
Registered qb: SEL$F7859CDE 0x26649c50 (JOIN REMOVED FROM QUERY
   BLOCK SEL$1; SEL$1; "D"@"SEL$1")
…
```

JE - Join Elimination (3)

- **Purpose of join elimination**
  - Usually people don’t write such “stupid” statements directly
  - Such situations are very common when a view is used which contains a join and only a subset of columns is used and therefor a join operation is really not required at all.
- **Known Limitations** (Source: Optimizer group blog)
  - Multi-column primary key-foreign key constraints are not supported.
  - Referring to the join key elsewhere in the query will prevent table elimination. For an inner join, the join keys on each side of the join are equivalent, but if the query contains other references to the join key from the table that could otherwise be eliminated, this prevents elimination. A workaround is to rewrite the query to refer to the join key from the other table.

SJC – Set Join Conversion

SJC - Set-Join Conversion

- Conversion of a set operator to a join operator.
- Disabled by default in 11gR2

- To enable it there are three options:
  - `alter session set "_convert_set_to_join"=true;`
  - `/*+ OPT_PARAM('_convert_set_to_join‘,'true') */`
  - `/*+ SET_TO_JOIN */`

No SJC By Default

```
select c4  from t1 minus select c2 from t2 ;
```

```
----------------------------------------------------------------------------
| Id  | Operation            | Name | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |      |  1000 |  6000 |     8  (63)| 00:00:01 |
|   1 |  MINUS               |      |       |       |            |          |
|   2 |   SORT UNIQUE        |      |  1000 |  3000 |     4  (25)| 00:00:01 |
|   3 |    TABLE ACCESS FULL| T1   |  1000 |  3000 |     3   (0)| 00:00:01 |
|   4 |   SORT UNIQUE        |      |  1000 |  3000 |     4  (25)| 00:00:01 |
|   5 |    TABLE ACCESS FULL| T2   |  1000 |  3000 |     3   (0)| 00:00:01 |
----------------------------------------------------------------------------
```

SJC with OPT_PARAM hint

```
select /*+ opt_param('_convert_set_to_join','true') */ x.c4
from t1 x
minus
select y.c4
from t1 y;
```

```
---------------------------------------------------------------------------
| Id  | Operation            | Name | Rows  | Bytes | Cost (%CPU)| Time     |
---------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |      |     3 |    18 |     8  (25)| 00:00:01 |
|   1 |  HASH UNIQUE         |      |     3 |    18 |     8  (25)| 00:00:01 |
|*  2 |   HASH JOIN ANTI     |      |    10 |    60 |     7  (15)| 00:00:01 |
|   3 |    TABLE ACCESS FULL| T1   |  1000 |  3000 |     3   (0)| 00:00:01 |
|   4 |    TABLE ACCESS FULL| T1   |  1000 |  3000 |     3   (0)| 00:00:01 |
---------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access(SYS_OP_MAP_NONNULL("X"."C4")=SYS_OP_MAP_NONNULL("Y"."C4"))
```

SJC with SET_TO_JOIN Hint

```
select /*+ SET_TO_JOIN */ x.c4
from t1 x
minus
select y.c4
from t1 y;
```

```
----------------------------------------------------------------------------
| Id  | Operation           | Name | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |      |     3 |    18 |     8  (25)| 00:00:01 |
|   1 |  HASH UNIQUE        |      |     3 |    18 |     8  (25)| 00:00:01 |
|*  2 |   HASH JOIN ANTI    |      |    10 |    60 |     7  (15)| 00:00:01 |
|   3 |    TABLE ACCESS FULL| T1   |  1000 |  3000 |     3   (0)| 00:00:01 |
|   4 |    TABLE ACCESS FULL| T1   |  1000 |  3000 |     3   (0)| 00:00:01 |
----------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access(SYS_OP_MAP_NONNULL("X"."C4")=SYS_OP_MAP_NONNULL("Y"."C4"))
```

SJC in CBO Trace

- Excerpt from CBO trace

```
SJC: Considering set-join conversion in query block SET$1 (#0)
*************************
Set-Join Conversion (SJC)
*************************
SJC:   Checking validity of SJC on query block SET$1 (#0)
SJC:   Passed validity checks.
SJC: SJC: Applying SJC on query block SET$1 (#0)
Registered qb: SEL$09AAA538 0x99f85c60 (SET QUERY BLOCK SET$1; SET$1)
---------------------
QUERY BLOCK SIGNATURE
---------------------
  signature (): qb_name=SEL$09AAA538 nbfros=2 flg=0
    fro(0): flg=0 objn=247624 hint_alias="X"@"SEL$1"
    fro(1): flg=0 objn=247624 hint_alias="Y"@"SEL$2"

SJC: performed
```

OBYE - Order BY Elimination

OBYE - order by elimination (1)

- OBYE operation eliminates unnecessary order by opertion from the SQL statement

```
select /*+ qb_name( main ) */ count(*) from (
   select /*+ qb_name( q1 ) */ p.prod_name
   from products p
   order by p.prod_name
);
```

```
-------------------------------------------------------------------
| Id  | Operation                    | Name                   | Rows  |
-------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                        |     1 |
|   1 |  SORT AGGREGATE              |                        |     1 |
|   2 |   BITMAP CONVERSION COUNT    |                        |    72 |
|   3 |    BITMAP INDEX FAST FULL SCAN| PRODUCTS_PROD_STATUS_BIX |       |
-------------------------------------------------------------------
```

OBYE - order by elimination (2)

- From CBO Trace in 11g; 10gR2 has similar output

```
....
***************************
Order-by elimination (OBYE)
***************************
OBYE: Removing order by from query block Q1 (#0) (order not used)
Registered qb: SEL$7AB500E1 0x464f6080 (ORDER BY REMOVED FROM QUERY BLOCK
   Q1; Q1)
---------------------
QUERY BLOCK SIGNATURE
---------------------
  signature (): qb_name=SEL$7AB500E1 nbfros=1 flg=0
    fro(0): flg=0 objn=70488 hint_alias="P"@"Q1"

OBYE:     OBYE performed.
...
```

CNT - count(col) to count(*) transformation

CNT - count(col) to count(*) transformation

```
SQL> create table t1 (c1 number not null);
SQL> select count(c1) from t1;
```

```
CNT:   Considering count(col) to count(*) on query block
   SEL$1 (#0)
*************************
Count(col) to Count(*) (CNT)
*************************
CNT:     Converting COUNT(C1) to COUNT(*).
CNT:     COUNT() to COUNT(*) done.
```

- All rows should have a value and therefore Oracle can simply count the number of rows
- There is no need to actually retrieve the column value.

CNT - count(col) to count(*) transformation

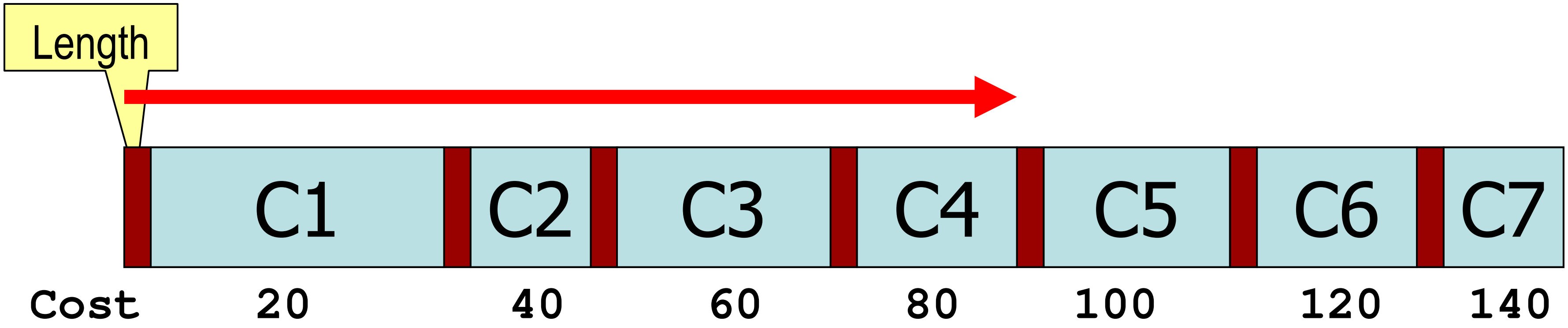
```
SQL> alter table t1 add (c2 varchar2(10)); /* nullable col */

SQL> select count(c2) from t1;
```

From CBO trace:

```
CNT:   Considering count(col) to count(*) on query block SEL$1 (#0)
*************************
Count(col) to Count(*) (CNT)
*************************
CNT:     COUNT() to COUNT(*) not done.
query block SEL$1 (#0) unchanged
```

CBO’s Column Retrieval Cost

- Oracle stores columns in variable length format
- Each row is parsed in order to retrieve one or several columns.
- Each parsed column introduces cost of 20 CPU cycles regardless if it will be extracted or not.



CNT - count(col) to count(*) transformation

- Comparing the calculated cost from CBO trace file
  - –Without CNT Transformation

```
Cost: 34.4695  Degree: 1  Card: 56229.0000  Bytes: 224916
Resc: 34.4695  Resc_io: 34.0000  Resc_cpu: 10399260
```

  - –With CNT transformation <u>the CPU cost is reduced</u>

```
Cost: 34.4187  Degree: 1  Card: 56229.0000  Bytes: 0
Resc: 34.4187  Resc_io: 34.0000  Resc_cpu: 9274680
```

- The cost is reduced for 20 CPU cycles per row – Oracle has less work to do – accesses only the row directory and the row header in database block and doesn't need to parse the row data.

Conclusions

Conclusions

1. Help CBO by defining all possible constraints. CBO uses them extensively during the SQL statement transformations. Telling more “truth” to CBO usually helps.
2. Feed the CBO with accurate statistics, only for complex expressions use dynamic sampling.
3. Misestimated cardinality in Cost Based Transformation leads to sub-optimal plan.
4. Use transformation techniques when rewriting the statement to obtain optimal plan. One can even use **NO_QUERY_TRANSFORMATION** hint to disable all transformations.

References

- http://blogs.oracle.com/optimizer or former http://optimizermagic.blogspot.com/
- For more detailed study:
  - **Enhanced Subquery Optimizations in Oracle, VLDB'09**, *http://www.vldb.org/pvldb/2/vldb09-423.pdf*
  - **Patent registration: Join Factorization of UNION/UNION ALL Queries**, *http://www.freepatentsonline.com/7644062.pdf*
  - **Cost-Based Query Transformation in Oracle**, VLDB'06, September 2006, Seoul, Korea, *http://delivery.acm.org/10.1145/1170000/1164215/p1026-ahmed.pdf?key1=1164215&key2=7529733711&coll=&dl=ACM&CFID=15151515&CFTOKEN=6184618*
  - **Query Optimization in Oracle Database10g Release 2**, An Oracle White Paper, June 2005, *http://www.oracle.com/technology/products/bi/db/10g/pdf/twp_general_query_optimization_10gr2_0605.pdf*
  - Mohamed Zait**, Oracle10g SQL Optimization, Trivadis** CBO days, June 2006, Zurich, Switzerland
  - Jonathan Lewis, **Cost Based Oracle**, Apress

Thank you for your interest!

Q&A