# Advanced Oracle Performance Troubleshooting

# Query Transformations


## Randolf Geist

http://oracle-randolf.blogspot.com/
http://www.sqltools-plusplus.org:7676/
info@sqltools-plusplus.org

# Who am I

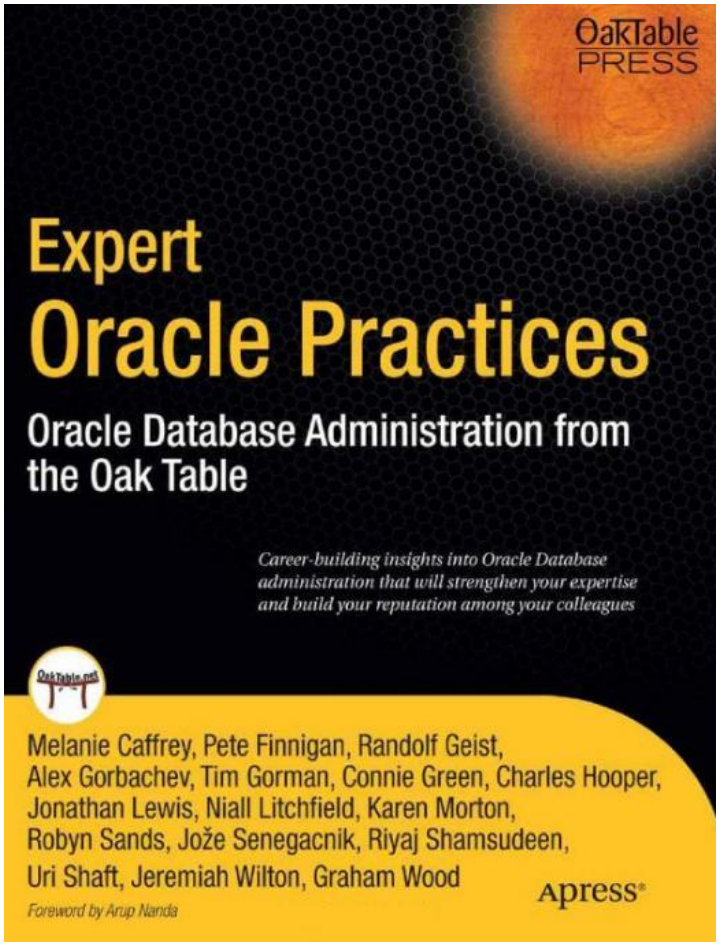
- Independent Consultant
- Located in Germany
- Oracle ACE Director
- OCP 8i, 9i, 10g
- Member of the OakTable Network
  http://www.oaktable.net

# Who am I

- Regular speaker at UKOUG, SIOUG, DOAG, MOTS etc.
- My Blog: Oracle related stuff
  http://oracle-randolf.blogspot.com
- Maintainer of SQLTools++ 
  http://www.sqltools-plusplus.org:7676/
  http://sqltpp.sourceforge.net

# Who am I

Co-author of the latest OakTable book "Expert Oracle Practices"



# Agenda

- Introduction
- Constraints
- Controlling basic transformations
- Transformations not supported (yet)

# Introduction

- The Cost Based Optimizer (CBO) applies a mathematical model to a query to arrive at an execution plan with the lowest cost found
- In order to allow the CBO to evaluate more, potentially more efficient execution plans, the CBO attempts to apply transformations to the query being optimized
- The whole purpose of Query Transformations is to find more efficient execution plans

# Introduction

- “Query Transformation” means that the CBO automatically rewrites the given query into a different one that is guaranteed to generate the same result
- The transformed query might look quite different from the original one
- Oracle by default does not reveal the transformations applied, you can however often tell from the execution plan generated that the underlying query is different from the original one

# Introduction

- With every release Oracle adds more and more possible transformations

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

# Introduction

- Until Oracle 9i the Query Transformation phase and the costing phase were separated
- First the CBO applied transformations based on heuristic rules
- The then transformed query was optimized by exploring the different possible access and join orders and choosing the plan with the lowest cost found

# Introduction

- Since Oracle 10g the two phases are interleaved: The so called “Cost Based Query Transformation” (CBQT) got introduced
- This means that the CBO now tries different transformed and untransformed variants of the query and runs them through the costing algorithm
- The one with the lowest cost will be chosen
- There are still cases where no costing takes place

# Introduction

- This has several consequences:
  - Transformations that were applied in pre-10g versions might now get rejected in 10g and later due to the costing
  - The optimization phase is more complex and potentially takes longer with CBQT
  - Due to the known limitations of a cost based optimizer the transformation variant selected based on the cost model might be actually less efficient than predicted by the cost

# Introduction

- The CBO trace file (event 10053) contains a lot of detailed information about the work performed by Query Transformation engine respectively the different attempts of the CBQT algorithm
- For most SQL statements the generated CBO trace file can become quite large and hard to read
- You might want to try the 10053 trace file viewer available from here: http://jonathanlewis.wordpress.com/2010/04/30/10053-viewer/

# Introduction

## Sample CBO trace file snippet:

```
**********************************
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.
...
```

# Agenda

- Introduction
- Constraints
- Controlling basic transformations
- Transformations not supported (yet)

# Constraints

- Several transformations are only possible if constraints are defined on the data that enable the CBO to apply a particular transformation
- The most common problem is a missing NOT NULL constraint
- Other constraints, like UNIQUE, PRIMARY, FOREIGN KEY and CHECK are also used by the CBO

# Constraints

- Therefore it is recommended to add as many constraints as possible
- The tradeoff needs to be evaluated between overhead of enforcing and maintaining the constraints and the benefit of more efficient execution plans at query time
- Since quite often data is only written once but queried many times the common belief that constraints only add overhead without any performance benefit is actually a misbelief

# Constraints

- Some common transformations that depend on constraints
  - NOT IN into NOT EXISTS: The column(s) used in a NOT IN subquery must be NOT NULL for transforming this into an anti-join (lifted in 11g)
  - Join elimination (10.2+): Tables can only be eliminated if a validated PK/FK constraint is defined
  - Outer Join elimination (11.1+): At least an UNIQUE constraint on the column(s) of the table being outer joined needs to be defined

# Constraints

## NOT IN into NOT EXISTS

- Why is NOT IN not equivalent to NOT EXISTS?
  - X IN (10, 20, 30, NULL) =>
    X = 10 OR X = 20 OR X = 30 OR X = NULL
  - X = NULL evaluates NEVER to TRUE
  - Due to the boolean OR X IN () is equivalent to EXISTS (at least one is TRUE) and can be transformed even in the presence of NULLs

# Constraints

## NOT IN into NOT EXISTS

- Why is NOT IN not equivalent to NOT EXISTS?
  - X NOT IN (10, 20, 30, NULL) =>
    X <> 10 AND X<> 20 AND X <> 30
    AND X <> NULL
  - X <> NULL never evaluates to TRUE
  - In the presence of a NULL value in the list the result of X NOT IN () is always an empty set
  - Therefore NOT IN with possible NULLs is not equivalent to NOT EXISTS

# Constraints

## NOT IN into NOT EXISTS

```
SQL>
SQL> select
  2          *
  3  from
  4          dept
  5  where
  6          not exists (
  7            select
  8                    null
  9            from
 10                    emp
 11            where
 12                    emp.deptno = dept.deptno
 13          );

    DEPTNO DNAME          LOC
---------- -------------- -------------
        40 OPERATIONS     BOSTON

SQL>
SQL> select
  2          *
  3  from
  4          dept
  5  where
  6          deptno not in (
  7            select
  8                    deptno
  9            from
 10                    emp
 11          );

    DEPTNO DNAME          LOC
---------- -------------- -------------
        40 OPERATIONS     BOSTON
```

# Constraints

## NOT IN into NOT EXISTS

```
SQL>
SQL> update emp set deptno = NULL where rownum = 1;

1 row updated.

SQL> select
  2           *
  3  from
  4           dept
  5  where
  6           not exists (
  7             select
  8                     null
  9             from
 10                     emp
 11             where
 12                     emp.deptno = dept.deptno
 13           );

    DEPTNO DNAME          LOC
---------- -------------- -------------
        40 OPERATIONS     BOSTON

SQL>
SQL> select
  2           *
  3  from
  4           dept
  5  where
  6           deptno not in (
  7             select
  8                     deptno
  9             from
 10                     emp
 11           );

no rows selected
```

# Constraints

## Join Elimination (10.2+):

```
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 (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |           |   106 |  2332 |     3   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL | EMPLOYEES |   106 |  2332 |     3   (0)| 00:00:01 |
-------------------------------------------------------------------------------

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

   1 - filter("E"."DEPARTMENT_ID" IS NOT NULL)
```

# Constraints

Check Constraints

- Oracle can use check constraints to eliminate unnecessary processing
- If predicates are applied that are known to be “not there” in the table due to check constraints Oracle will “shortcut” the execution

# Constraints

## Check Constraints

```
create table t_check (c1 number, c2 number, c3 number, c4 number);

alter table t_check add constraint t_check_c2_chk check (c2 in (10, 20, 30));

select * from t_check where c2 = 35;

-------------------------------------------------------------------------------
| Id  | Operation          | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |         |     1 |    52 |     0   (0)|          |
|*  1 |  FILTER            |         |       |       |            |          |
|*  2 |   TABLE ACCESS FULL| T_CHECK |     1 |    52 |     2   (0)| 00:00:01 |
-------------------------------------------------------------------------------

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

   1 - filter(NULL IS NOT NULL)
   2 - filter("C2"=35)
```

# Constraints

- You might want to ask why the previous two transformations are helpful?
- Probably no-one usually writes SQL that joins unnecessary tables or queries for data that is known “not to be there”?
- Although this is true – if you think about a generic “view layer” on top of data the join elimination is a very helpful feature
- Also end-users might run ad-hoc queries that can also benefit from the shortcut execution using check constraints

# Agenda

- Introduction
- Constraints
- Controlling basic transformations
- Transformations not supported (yet)

# Controlling basic transformations

- Although the CBO is quite good at choosing the most efficient transformation and gets more sophisticated with each release it still might get it wrong in some cases
- Even with the CBQT feature sometimes the cost estimates are incorrect and therefore the wrong transformation is selected

# Controlling basic transformations

- Therefore it is sometimes helpful to explicitly control transformations using hints
- Most of these hints are officially documented and it is therefore safe and supported to use them

# Controlling basic transformations

- A general note on hints
  - The usage of some hints has changed with Oracle 10g
  - Oracle 10g introduced the “QB_NAME” hint
  - The hint allows to assign user-defined names to query blocks
  - These query block names can then be used in the hint to express exactly to which query block it is supposed to be applied

# Controlling basic transformations

- Be aware of the side effects of using ANSI JOINs
  - In some releases some transformations are not supported when using ANSI JOINs
  - Sometimes it appears that hints are ignored by the CBO with ANSI JOINs (but they are never, only in case of bugs)
  - This is caused by the fact that Oracle transforms ANSI JOINs into Oracle join syntax, sometimes using so called LATERAL views
  - Due to these additional transformations hints might be moved to query parts where they are no longer valid and therefore are ignored

# Controlling basic transformations

- Be aware of the side effects of using ANSI JOINs
  - Check the different Query Blocks by using the +ALIAS format option of the DBMS_XPLAN.DISPLAY* functions
  - You can also check the hints in the OUTLINE section added by using the +OUTLINE format option
  - Use the QB_NAME hint to control the name of the Query Blocks
  - Try manual rewrite to Oracle syntax in such cases – this is not trivial

# View merging

- Oracle usually attempts to merge views in order to find more possible join and access orders
- With every release the so called “Complex View Merging” (CVM) is enhanced
- This means that Oracle can merge many views that contain complex expressions like aggregates, DISTINCT, GROUP BY etc.

# Why is it important? – No merge

```
select e.first_name, e.last_name, dept_locs_v.street_address, dept_locs_v.postal_code
from employees e,
     (select d.department_id, d.department_name, l.street_address, l.postal_code
      from departments d, locations l
      where d.location_id = l.location_id) dept_locs_v
where dept_locs_v.department_id = e.department_id
and e.last_name = 'Smith';
```

Limited join orders

```
-------------------------------------------------------------------
| Id  | Operation                    | Name        | Cost (%CPU)|
-------------------------------------------------------------------
|   0 | SELECT STATEMENT             |             |     7  (15)|
|*  1 |  HASH JOIN                   |             |     7  (15)|
|   2 |   TABLE ACCESS BY INDEX ROWID| EMPLOYEES   |     2   (0)|
|*  3 |    INDEX RANGE SCAN          | EMP_NAME_IX |     1   (0)|
|   4 |   VIEW                       |             |     5  (20)|
|*  5 |    HASH JOIN                 |             |     5  (20)|
|   6 |     TABLE ACCESS FULL        | LOCATIONS   |     2   (0)|
|   7 |     TABLE ACCESS FULL        | DEPARTMENTS |     2   (0)|
-------------------------------------------------------------------
```

Source: Optimizer Blog

# Why is it important? – Merged

```
select e.first_name, e.last_name, l.street_address, l.postal_code
from employees e, departments d, locations l
where d.location_id = l.location_id
and d.department_id = e.department_id
and e.last_name = 'Smith';
```

More join options

```
-------------------------------------------------------------------------
| Id  | Operation                     | Name         | Cost (%CPU)|
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |              |     4   (0)|
|   1 |  NESTED LOOPS                 |              |            |
|   2 |   NESTED LOOPS                |              |     4   (0)|
|   3 |    NESTED LOOPS               |              |     3   (0)|
|   4 |     TABLE ACCESS BY INDEX ROWID| EMPLOYEES    |     2   (0)|
|*  5 |      INDEX RANGE SCAN         | EMP_NAME_IX  |     1   (0)|
|   6 |     TABLE ACCESS BY INDEX ROWID| DEPARTMENTS |     1   (0)|
|*  7 |      INDEX UNIQUE SCAN        | DEPT_ID_PK   |     0   (0)|
|*  8 |    INDEX UNIQUE SCAN          | LOC_ID_PK    |     0   (0)|
|   9 |   TABLE ACCESS BY INDEX ROWID | LOCATIONS    |     1   (0)|
-------------------------------------------------------------------------
```

Source: Optimizer Blog

# View merging

- If a view is not merged it is optimized like a separate query on its own
- You will usually find a VIEW operator in the execution plan that indicates a non-merged view
- The VIEW operator also means that the view's result set will be projected / materialized at that point of the execution plan
- Inline views and stored views are the same
- The MERGE / NO_MERGE hints control view merging

# View merging

```
select
         a.object_owner, b.*
from
        (
          select
                  *
          from
                  t1
        ) a
      , (
          select
                  *
          from
                  t2
        ) b
where
        a.user_id = b.user_id
and     a.object_name = 'BLA';
```

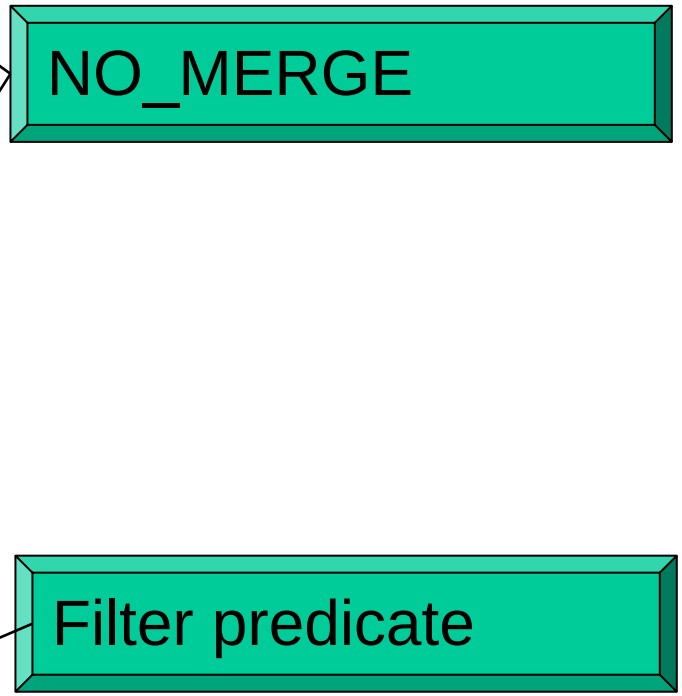
Inline views

```
-------------------------------------------
| Id  | Operation          | Name | Rows  |
-------------------------------------------
|   0 | SELECT STATEMENT   |      |     8 |
|*  1 |  HASH JOIN         |      |     8 |
|*  2 |   TABLE ACCESS FULL| T1   |     8 |
|   3 |   TABLE ACCESS FULL| T2   |    30 |
-------------------------------------------
```

Views merged

# View merging

```
select
         a.object_owner, b.*
from
        (
          select /*+ no_merge */
                 t1.*
          from
                 t1
        ) a
      , (
          select /*+ no_merge */
                 t2.*
          from
                 t2
        ) b
where
        a.user_id = b.user_id
and     a.object_name = 'BLA';
```



# View merging

```
-----------------------------------------------
| Id  | Operation            | Name | Rows  |
-----------------------------------------------
|   0 | SELECT STATEMENT     |      |     8 |
|*  1 |  HASH JOIN           |      |     8 |
|   2 |   VIEW               |      |     8 |
|*  3 |    TABLE ACCESS FULL | T1   |     8 |
|   4 |   VIEW               |      |    30 |
|   5 |    TABLE ACCESS FULL | T2   |    30 |
-----------------------------------------------

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

   1 - access("A"."USER_ID"="B"."USER_ID")
   3 - filter("T1"."OBJECT_NAME"='BLA')
```

No view merging

Pushed into view

# View merging

```
select   /*+ qb_name(main) no_merge(@v1) no_merge(@v2) */
         a.object_owner, b.*
from
        (
          select  /*+ qb_name(v1) */
                  t1.*
          from
                  t1
        ) a
      , (
           select /*+ qb_name(v2) */
                  t2.*
           from
                  t2
        ) b
where
        a.user_id = b.user_id
and     a.object_name = 'BLA';
```

# ALIAS and PROJECTION

## Use the ALIAS / PROJECTION or ADVANCED format option of DBMS_XPLAN

```
Query Block Name / Object Alias (identified by operation id):
-------------------------------------------------------------

   1 - MAIN
   2 - V1   / A@MAIN
   3 - V1   / T1@V1
   4 - V2   / B@MAIN
   5 - V2   / T2@V2

Column Projection Information (identified by operation id):
-----------------------------------------------------------

   1 - (#keys=1) "B"."USER_ID"[NUMBER,22],
       "A"."OBJECT_OWNER"[NUMBER,22], "B"."USERNAME"[VARCHAR2,46]
   2 - "A"."USER_ID"[NUMBER,22], "A"."OBJECT_OWNER"[NUMBER,22]
   3 - "T1"."USER_ID"[NUMBER,22], "T1"."OBJECT_OWNER"[NUMBER,22]
   4 - "B"."USER_ID"[NUMBER,22], "B"."USERNAME"[VARCHAR2,46]
   5 - "T2"."USER_ID"[NUMBER,22], "T2"."USERNAME"[VARCHAR2,46]
```

Query block names

Projection details

# Filter Pushdown

- Normally Oracle tries to apply filter predicates as early as possible
- In rare cases where you do not want to push filter predicates into a view you can prevent that by adding ROWNUM to the projection of that view
- Usually this also prevents automatically the merging of the view, but you might want to add a NO_MERGE hint additionally to make the intention clearer

# Filter Pushdown

```
select   /*+ qb_name(main) no_merge(@v1) no_merge(@v2) */
         a.object_owner, b.*
from
        (
          select  /*+ qb_name(v1) */
                  t1.*, rownum
          from
                  t1
        ) a
      , (
           select /*+ qb_name(v2) */
                  t2.*
           from
                  t2
        ) b
where
        a.user_id = b.user_id
and     a.object_name = 'BLA';
```

ROWNUM predicate

Filter predicate

# Filter Pushdown

```
-----------------------------------------------
| Id  | Operation             | Name | Rows  |
-----------------------------------------------
|   0 | SELECT STATEMENT      |      | 30000 |
|*  1 |  HASH JOIN            |      | 30000 |
|   2 |   VIEW                |      |    30 |
|   3 |    TABLE ACCESS FULL  | T2   |    30 |
|*  4 |   VIEW                |      | 30000 |
|   5 |    COUNT              |      |       |
|   6 |     TABLE ACCESS FULL | T1   | 30000 |
-----------------------------------------------

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

   1 - access("A"."USER_ID"="B"."USER_ID")
   4 - filter("A"."OBJECT_NAME"='BLA')
```

Late filtering

# Subquery unnesting

- Oracle can “unnest” different kinds of correlated and uncorrelated subqueries like (NOT) EXISTS, (NOT) IN, ANY, ALL etc.
- Unnesting means that Oracle transforms the subquery into a join which often can be more efficient than running the subquery as filter / access subquery

# Why is it important?

- Without this transformation Oracle needs to evaluate the filter clause potentially as many times as rows are generated by the main query
- This can be resource and time consuming
- There are cases where the untransformed query is actually more efficient than the join

# Subquery unnesting

- For particular logical constructs like EXISTS or NOT EXISTS Oracle supports join techniques that can not be expressed in SQL like semi and anti joins
- Unnesting can be controlled via the UNNEST / NO_UNNEST hint

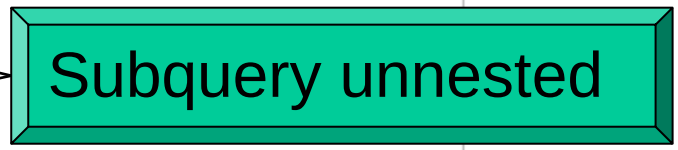
# Subquery unnesting

```
select  /*+ leading(t1) no_swap_join_inputs(t2) */
        *
from
        t1, t2
where
        t1.user_id = t2.user_id
and     not exists (select /*+ qb_name(subq) */ null from t2 where t1.user_id = t2.user_id);

-----------------------------------------
| Id  | Operation           | Name | Rows  |
-----------------------------------------
|   0 | SELECT STATEMENT    |      |  1000 |
|*  1 |  HASH JOIN          |      |  1000 |
|*  2 |   HASH JOIN ANTI    |      |  1000 |
|   3 |    TABLE ACCESS FULL| T1   | 30000 |
|   4 |    TABLE ACCESS FULL| T2   |    30 |
|   5 |   TABLE ACCESS FULL | T2   |    30 |
-----------------------------------------

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

   1 - access("T1"."USER_ID"="T2"."USER_ID")
   2 - access("T1"."USER_ID"="T2"."USER_ID")
```



# Subquery unnesting

```
select  /*+ leading(t1) no_swap_join_inputs(t2) no_unnest(@subq) */
        *
from
        t1, t2
where
        t1.user_id = t2.user_id
and     not exists (select /*+ qb_name(subq) */ null from t2 where t1.user_id = t2.user_id);

-------------------------------------------
| Id  | Operation           | Name | Rows  |
-------------------------------------------
|   0 | SELECT STATEMENT    |      | 29000 |
|*  1 |  FILTER             |      |       |
|*  2 |   HASH JOIN         |      | 30000 |
|   3 |    TABLE ACCESS FULL| T1   | 30000 |
|   4 |    TABLE ACCESS FULL| T2   |    30 |
|*  5 |   TABLE ACCESS FULL | T2   |     1 |
-------------------------------------------

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

   1 - filter( NOT EXISTS (SELECT /*+ NO_UNNEST QB_NAME ("SUBQ") */ 0
              FROM "T2" "T2" WHERE "T2"."USER_ID"=:B1))
   2 - access("T1"."USER_ID"="T2"."USER_ID")
   5 - filter("T2"."USER_ID"=:B1)
```

Filter Subquery

# Subquery unnesting

- The FILTER SUBQUERY can be efficient for several reasons:
  - It is controlled by the first operation of the FILTER operator and therefore can use additional information from these row sources for a potentially efficient access to the data (similar to an inner table of a NESTED LOOP)
  - Oracle uses a clever caching algorithm for filter and scalar subqueries that caches the result and thereby potentially limits the number of actual execution of the subquery

# Filter Subquery Caching

```
-------------------------------------------------------------------------
| Id  | Operation           | Name | Starts | E-Rows | A-Rows |
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |      |      1 |        |      0 |
|*  1 |  FILTER             |      |      1 |        |      0 |
|*  2 |   HASH JOIN         |      |      1 |  30000 |  30000 |
|   3 |    TABLE ACCESS FULL| T1   |      1 |  30000 |  30000 |
|   4 |    TABLE ACCESS FULL| T2   |      1 |     30 |     30 |
|*  5 |   TABLE ACCESS FULL | T2   |     30 |      1 |     30 |
-------------------------------------------------------------------------

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

   1 - filter( IS NULL)
   2 - access("T1"."USER_ID"="T2"."USER_ID")
   5 - filter("T2"."USER_ID"=:B1)
```

Only 30 subquery executions, but 30,000 rows in driving row source

# Filter postponement

- FILTER SUBQUERIES by default are applied “late” by Oracle
- If it is more efficient to filter “early” you can control the evaluation of the filter by using the PUSH_SUBQ / NO_PUSH_SUBQ hint

# Filter postponement

```
select  /*+ leading(t1) no_swap_join_inputs(t2) no_unnest(@subq) push_subq(@subq) */
        *
from
        t1, t2
where
        t1.user_id = t2.user_id
and     not exists (select /*+ qb_name(subq) */ null from t2 where t1.user_id = t2.user_id);

-------------------------------------------------------------------------------
| Id  | Operation           | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |      |  1500 | 39000 |    28   (4)| 00:00:01 |
|*  1 |  HASH JOIN          |      |  1500 | 39000 |    26   (4)| 00:00:01 |
|*  2 |   TABLE ACCESS FULL | T1   |  1500 | 21000 |    24   (5)| 00:00:01 |
|*  3 |   TABLE ACCESS FULL | T2   |     1 |     3 |     2   (0)| 00:00:01 |
|   4 |   TABLE ACCESS FULL | T2   |    30 |   360 |     2   (0)| 00:00:01 |
-------------------------------------------------------------------------------

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

   1 - access("T1"."USER_ID"="T2"."USER_ID")
   2 - filter( NOT EXISTS (SELECT /*+ PUSH_SUBQ NO_UNNEST QB_NAME
             ("SUBQ") */ 0 FROM "T2" "T2" WHERE "T2"."USER_ID"=:B1))
   3 - filter("T2"."USER_ID"=:B1)
```

Filter is applied early

# Filter postponement

```
-------------------------------------------------------------------------------
| Id  | Operation           | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |      |  1500 | 39000 |    28   (4)| 00:00:01 |
|*  1 |  HASH JOIN          |      |  1500 | 39000 |    26   (4)| 00:00:01 |
|*  2a|   FILTER            |      |  1500 | 21000 |    24   (5)| 00:00:01 |
|   2b|    TABLE ACCESS FULL| T1   | 30000 | 21000 |    24   (5)| 00:00:01 |
|*  3 |    TABLE ACCESS FULL| T2   |     1 |     3 |     2   (0)| 00:00:01 |
|   4 |   TABLE ACCESS FULL | T2   |    30 |   360 |     2   (0)| 00:00:01 |
-------------------------------------------------------------------------------

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

   1 - access("T1"."USER_ID"="T2"."USER_ID")
   2a- filter( NOT EXISTS (SELECT /*+ PUSH_SUBQ NO_UNNEST QB_NAME
             ("SUBQ") */ 0 FROM "T2" "T2" WHERE "T2"."USER_ID"=:B1))
   3 - filter("T2"."USER_ID"=:B1)
```

What the plan should look like

# Join Predicate Pushdown

- If Oracle can not merge views then there is another possibility: It can try to push the join predicate from one row source into the other using a NESTED LOOP join
- This can be efficient if there is an efficient access on the join expression to the inner row source
- This transformation can be controlled via the PUSH_PRED / NO_PUSH_PRED hint

# Join Predicate Pushdown

```
select   /*+ qb_name(main) no_merge(@v1) no_merge(@v2) push_pred(b) index(b) */
         a.*, b.*
from
        (
          select  /*+ qb_name(v1) */
                  t2.*
          from
                  t2
        ) a
      , (
           select /*+ qb_name(v2) */
                  t1.*
           from
                  t1
        ) b
where
        a.user_id = b.user_id (+)
and     a.username = 'BLA'
;
```

```
| Id  | Operation                     | Name    | Rows  |
-----------------------------------------------------------
|   0 | SELECT STATEMENT              |         |  1000 |
|   1 |  NESTED LOOPS OUTER           |         |  1000 |
|   2 |   VIEW                        |         |     1 |
|*  3 |    TABLE ACCESS FULL          | T2      |     1 |
|   4 |   VIEW PUSHED PREDICATE       |         |    33 |
|   5 |    TABLE ACCESS BY INDEX ROWID| T1      |  1000 |
|*  6 |     INDEX RANGE SCAN          | T1_IDX  |  1000 |
-----------------------------------------------------------
```

Join predicate pushed into inner view

# Or Expansion

- Oracle can transform multiple OR expressions into a concatenation using UNION ALL
- This can be efficient if the combined cost of the separate operations is less than the cost of the non-transformed query, for example if different indexes can be used for different expressions
- This transformation is controlled by the USE_CONCAT / NO_EXPAND hints

# Or Expansion

```
select * from t1 where user_id = 1 or object_owner = 2;

-------------------------------------------
| Id  | Operation          | Name | Rows  |
-------------------------------------------
|   0 | SELECT STATEMENT   |      |  1967 |
|*  1 |  TABLE ACCESS FULL| T1   |  1967 |
-------------------------------------------

Predicate Information (identified by e
---------------------------------------

   1 - filter("USER_ID"=1 OR "OBJECT_OWNER"=2)
```

Two separate indexes not used

# Or Expansion

```
select /*+ use_concat index(t1) */ * from t1 where user_id = 1 or object_owner = 2;

-----------------------------------------------------------
| Id  | Operation                     | Name    | Rows  |
-----------------------------------------------------------
|   0 | SELECT STATEMENT              |         |  1967 |
|   1 |  CONCATENATION                |         |       |
|   2 |   TABLE ACCESS BY INDEX ROWID | T1      |  1000 |
|*  3 |    INDEX RANGE SCAN           | T1_IDX2 |  1000 |
|*  4 |   TABLE ACCESS BY INDEX ROWID | T1      |   967 |
|*  5 |    INDEX RANGE SCAN           | T1_IDX  |  1000 |
-----------------------------------------------------------

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

   3 - access("OBJECT_OWNER"=2)
   4 - filter(LNNVL("OBJECT_OWNER"=2))
   5 - access("USER_ID"=1)
```

Separate query blocks
Indexes can be used

# Agenda

- Introduction
- Constraints
- Controlling basic transformations
- Transformations not supported (yet)

# Add a join optimization

- Oracle 11g introduced the so called Nested Loop Join Batching (controlled via the NLJ_BATCHING hint)
- It separates a Nested Loop Join into two separate Nested Loops, the first one accessing the index, the second one accessing the table

```
select /*+ use_nl(t1 t2) index(t1) qb_name(v1) */
       t2.*
     , t1.*
from
       t1
     , t2
where
       t1.user_id = t2.user_id
and    t2.username = 'BLA'
;
```

```
-----------------------------------------------
| Id  | Operation                     | Name   |
-----------------------------------------------
|   0 | SELECT STATEMENT              |        |
|   1 |  NESTED LOOPS                 |        |
|   2 |   NESTED LOOPS                |        |
|*  3 |    TABLE ACCESS FULL          | T2     |
|*  4 |    INDEX RANGE SCAN           | T1_IDX |
|   5 |   TABLE ACCESS BY INDEX ROWID| T1     |
-----------------------------------------------
```

# Add a join optimization

- Thinking this idea further: If a join was selective and could be done without performing a potentially expensive visit to the table then it would be great if Oracle could postpone the table visit after the join using only the remaining rows after join
- This is however not implemented (yet) with the exception of the inner row source of a NESTED LOOP using an index lookup
- Nothing stops us however from implementing this ourselves
- This means we actually can optimize a query by adding one (or more) join(s)!

# Add a join optimization

```
select  /*+ qb_name(main) use_nl(t1 v1) use_nl(t2 v1) no_merge(@v1) leading(v1) */
        *
from
        t1
      , t2
      , (
          select /*+ leading(t1) use_hash(t1 t2) index(t1) index(t2) qb_name(v1) */
                 t2.rowid as t2_rid
               , t1.rowid as t1_rid
          from
                 t1
               , t2
          where
                 t1.user_id = t2.user_id
        ) v1
where
        v1.t1_rid = t1.rowid
and     v1.t2_rid = t2.rowid
;
```

Inner query picks up ROWIDs, outer query joins to tables

```
-------------------------------------------------
| Id  | Operation                     | Name    |
-------------------------------------------------
|   0 | SELECT STATEMENT              |         |
|   1 |  NESTED LOOPS                 |         |
|   2 |   NESTED LOOPS                |         |
|   3 |    VIEW                       |         |
|*  4 |     HASH JOIN                 |         |
|   5 |      INDEX FULL SCAN          | T1_IDX  |
|   6 |      INDEX FULL SCAN          | T2_IDX  |
|   7 |    TABLE ACCESS BY USER ROWID | T2      |
|   8 |   TABLE ACCESS BY USER ROWID  | T1      |
-------------------------------------------------
```

# Or Subquery

- Oracle at present can not transform a query containing an OR with a subquery into a potentially more efficient join

```
select
        *
from
        t1
where
        t1.object_name = 'BLA'
or      not exists (select /*+ qb_name(subq) */ null from t2 where t1.user_id = t2.user_id);

-------------------------------------
| Id  | Operation          | Name   |
-------------------------------------
|   0 | SELECT STATEMENT   |        |
|*  1 |  FILTER            |        |
|   2 |   TABLE ACCESS FULL| T1     |
|*  3 |   INDEX RANGE SCAN | T2_IDX |
-------------------------------------

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

   1 - filter("T1"."OBJECT_NAME"='BLA' OR  NOT EXISTS (SELECT /*+
              QB_NAME ("SUBQ") */ 0 FROM "T2" "T2" WHERE "T2"."USER_ID"=:B1))
   3 - access("T2"."USER_ID"=:B1)
```

Subquery can not be unnested due to OR

# Or Subquery

- But we can do the transformation (concatenation) ourselves manually
- Be aware of the potential side effects of NULLS
- Since Oracle 10g the LNNVL function is officially documented (it was used internally since Oracle 8)
- This function comes really handy in such cases – read and understand what it does

# Or Subquery

```
select
        *
from
        t1
where
        t1.object_name = 'BLA'
union all
select
        *
from
        t1
where
        not exists (select /*+ qb_name(subq) */ null from t2 where t1.user_id = t2.user_id)
and     lnnvl(t1.object_name = 'BLA')
;
```

Manual concatenation
Subquery can be unnested

```
-----------------------------------------
| Id  | Operation              | Name   |
-----------------------------------------
|   0 | SELECT STATEMENT       |        |
|   1 |  UNION-ALL             |        |
|*  2 |   TABLE ACCESS FULL    | T1     |
|*  3 |   HASH JOIN RIGHT ANTI |        |
|   4 |    INDEX FULL SCAN     | T2_IDX |
|*  5 |    TABLE ACCESS FULL   | T1     |
-----------------------------------------
```

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

   2 - filter("T1"."OBJECT_NAME"='BLA')
   3 - access("T1"."USER_ID"="T2"."USER_ID")
   5 - filter(LNNVL("T1"."OBJECT_NAME"='BLA'))
```

# Query Transformations

Questions

&

Answers

# Reference

- Jonathan Lewis
  - Cost Based Oracle: Fundamentals, Apress
  - Oracle Scratchpad Blog
    http://jonathanlewis.wordpress.com
  - Oracle Optimizer Blog
    http://blogs.oracle.com/optimizer/
- Joze Senegacnik
  - Query Transformations, OOW 2010

# Query Transformations

Thank you!

http://oracle-randolf.blogspot.com/

http://www.sqltools-plusplus.org:7676/

info@sqltools-plusplus.org