SQL Tuning for Day-to-Day Data Warehouse Support

Nikos Karagiannidis

oradwstories.blogspot.com

Introduction

In this article, we will describe the most popular methods that DW engineers use today, in order to solve SQL performance problems, based on our real-world experience. We will point out the pros and cons of each method, as well as describe the basic tools/commands for using each one, and provide some best-practice recommendations.

The intended audience is DW support engineers, DW developers and DBAs. The methods presented are not DW-specific and pertain to SQL tuning in general, so this article will be useful to a broader audience of database professionals with interest in SQL tuning and not only to Data Warehouse professionals.

Problem Definition

In real-world large data warehouses solving SQL performance problems is a daily task. Thousands of SQL statements are executed every night during the execution of the daily ETL (Extract Transform Load) flows and -during the day- from the execution of standard reports and various data analysis tasks initiated by end-users. Therefore, DW professionals (Support Engineers, DBAs, or DW developers) are very often called, in order to solve the problem of a DW running task (an SQL statement) that is taking too long. Quite often this call takes place during "anti-social" hours, meaning too late at night, or too early in the morning, or during weekends. Finally, whenever such a call occurs, it is almost always designated (by the business users) as a "critical super-urgent" issue, and it is you, the DW professional, that must find a solution fast.

But how? How can one make a query that has been running for the last N hours, finish successfully? Especially when

- the person that is assigned to solve the problem typically knows nothing about the "logic" of the query
- the query is most probably complex and long
- the execution plan is most probably more than two pages long
- the query does not even finish (or at least finish in a reasonable time that allows us to experiment)

Limitations of most-popular Solutions

From our experience the most popular methods employed for the aforementioned task are:

(a) to try to run manually the problematic statement (i.e., from SQL Developer or any other DB client tool) with the hope that from a "different environment" the Cost Based Optimizer (CBO) will choose a better plan and the statement (e.g. a DML command) will finish. Then, we just skip this node from the ETL flow and allow the rest of the tasks to continue execution,

(b) to gather statistics for the underlying tables and try to execute the DW task again, or

(c) a combination of the two.

Of course, there are also other approaches followed, depending on the expertise of the specific DW professional. However, in the majority of the cases (again based on our real-world experience) the methods used are:

- "blind" in that they do not identify the root-cause of the performance problem
- Are not systematic but rather follow a "trial and error" approach
- Do not guarantee success in a short time (remember that the management is over your shoulder waiting for you to solve the issue fast)
- Do not guarantee success in a long time either. Success is merely a matter of luck.

Goal of this article

In this article, we will present a set of methods used for solving problems of DW *running* tasks (i.e. SQL statements). So the focus is on trying to solve an SQL performance problem of a running query, or DML command, or a DDL command - like a CTAS (Create Table As Select) command-, hereafter called a "*DW task*", on a production Data Warehouse and we view it from a DW support perspective. This means that our ultimate goal is to find (as soon as possible) a way - any way- that will make the DW task finish in a time that will not negatively impact the SLA with the business.

We assume that there is no other problem at the database level, and the DW task in question is not blocked by some other session, nor there is some other resource contention problem. So we are dealing with a performance problem that needs a clear "*SQL tuning solution*".

To this end, we distinguish between two broad categories of methods for SQL tuning: (a) the "blind" ones – called the "Black Box approach" and (b) the ones that identify the problematic operation in the plan – called the "White Box approach". We present each one and argue on the pros and cons.

This article is organized as follows. In the next section we introduce the two aforementioned approaches for SQL tuning. Then, in the following sections, we drill down to each approach and describe in more detail the various methods available for each one. In the final section, we summarize and provide major conclusions.

Two Approaches: You choose!

Whenever you have to solve the performance problem of a DW task that is running and has been running for too long, you have actually two roads to follow:

The first road is to avoid diving into the details of the execution plan and try to fix the problem with an "execution plan agnostic" manner. In this case, you view the DW task as a "*black box*", which receives as input a set of tables, along with their corresponding optimizer statistics, and then produces an execution plan that will yield a result at a specific response time.

The second road (the one less traveled by), is to understand where the problem is in the execution plan of the running DW task, and then try to fix it. In this case, you view the DW task as "*white box" (or a glass box)*, where apart from the input and output you are also exposed to the details of the execution plan.

These two roads essentially, correspond to the two basic approaches for SQL tuning during DW support. From the many years of our working experience with Data Warehouses, we can safely state that the black box approach is by far the most popular one. This is clearly because it is easier to apply and requires less SQL tuning skills. Of course, from a DW support perspective, at the end of the day, only the result matters; i.e., if we have managed, or not, to finish the delayed DW task and meet the SLA with the business. The choice is ours. Next, we outline the methods that belong to each approach and discuss on the pros and cons.

The Black Box Approach for SQLTuning a DW Task

Any SQL tuning problem, by definition, pertains to the execution plan of the SQL statement in question. The problem is definitely (somewhere) in the plan and the solution is again (somewhere) in the plan.

Admittedly, execution plans of real-world DW tasks are quite often daunting. Pages of operations, going to a depth of many levels are a typical picture of a DW task execution plan. At the heart of the black box approach lies the concept that if we omit the inspection of the plan, and try out various methods that aim at fixing the problematic plan, but without identifying what, or where, is the problem in the plan, then we will save valuable time and find a solution faster. Moreover, inspecting a complex plan requires some mental clarity, which is kind of difficult to achieve during "anti-social" hours, which is the typical time for a call to a DW support engineer, and thus the black-box approach is usually the most natural choice.

Therefore the main benefit of this approach is that it is fast and easy. The major drawback though is that it is completely blindfold. This simply means that if you don't get lucky early on, then you can get stuck in an endless trial and error procedure. Next, we outline the most popular representatives of this approach:

Execute the DW Task Manually from a DB client

This is the most popular method used by DW support engineers that have a limited experience in SQL tuning. They kill the session(s) that has been running for hours and try to run it manually from their DB client of choice (SQL*Plus, SQL Developer, TOAD etc.). The interesting thing is that sometimes this method works! I.e., the DW task completes and the ETL flow continues to the next task by skipping the task that was run manually.

There is not much that can be set for this method, apart from explaining why sometimes this naïve approach works. If we leave aside all the possible supernatural causes, a logical explanation would be the differences in the *optimizer environment* between the DB user executing the ETL flows and the DB user with which the DW support engineer is running the task by hand. For example, it is quite often the case that some important (to the optimizer) initialization parameters have different values between these two users and this triggers the generation of a different execution plan for each one. Thus, we get different execution times.

A typical such initialization parameter that we very often see changed from its default value, especially in Exadata-based data warehouses, is the *optimizer_index_cost_adj*. This parameter influences the cost calculation of index accesses and has a default value of 100. Very often in Exadata databases this is changed to a higher value, in order to make index range scans "to look" more expensive to the optimizer and thus be avoided. However, this change is not always implemented for all DB users. For example, we might choose to change the default value only for end-users, in order to achieve a higher performance for reporting, while leave the default value for the users executing the nightly ETL flows, so as to allow the use of indexes for update and delete operations.

If we want to run a statement manually and wish to imitate the optimizer environment of a different user (at least for some specific initialization parameters that we know that have different values), then we can use the OPT_PARAM hint. For example, in order to run a query with a value

(e.g., 10000) in parameter optimizer_index_cost_adj different than the default, then just use the following hint:

```
/*+ OPT_PARAM('optimizer_index_cost_adj' 10000)*/
```

Gather Table Statistics and try again

This is another popular method used by DW support engineers that sometimes works, which belongs to the black box approach. Whenever there is a performance problem for a specific statement, DW support engineers kill the running task, gather statistics for the corresponding tables and then try to run the task again.

This method is based on the fact that execution plans generated by the optimizer are based on cost calculations, which in turn are based on object statistics. Therefore, misleading statistics on the tables relevant to the problematic statement might result to inferior execution plans.

Again there is not much to be said on this method apart from emphasizing on when object statistics can be misleading. A simple way to ascertain to what degree gathered statistics are misleading, is to compare the NUM_ROWS column from ALL_TABLES for a specific table to a simple SELECT COUNT(*) from the same table. According to (Hasler, 2014) a problem exists *only* when the difference is in *orders of magnitude* and not by a factor of two or three. For example, if the actual number of rows is 1,000 and from statistics we get 3,000, then definitely this is not a problem for the optimizer. However, if the actual number of rows is 1,000 and from the gathered statistics we see 1,000,000, then this might lead to a wrong decision for the optimizer, which might, or might not, result to a performance problem. We can't really know, since according to the black-box approach we don't search for the root-cause of the problem, we just try a solution blindfold.

Call SQL Tuning Advisor

Another option when trying to deal with a performance issue for a specific SQL statement, during DW support, is to call the Oracle *SQL Tuning Advisor* for this statement and review its recommendations. Then, proceed in implementing the ones that seem most promising to you, or all of them, and rerun the statement in question to see if elapsed time has improved.

The SQL Tuning Advisor implements a special mode of the Query Optimizer, in which after the optimizer spends some time (the default is 30 minutes but this is configurable) analyzing the statement and the underlying objects (tables and indexes), it produces a set of recommendations for SQL tuning. It's up to you to decide whether you will implement them or not.

Basically, the recommendations you can get from the SQL Tuning Advisor are like the following:

1. Gather statistics for specific tables/indexes
2. Implement an SQL Profile
3. Create some new index
4. A proposal for a specific rewrite for your SQL statement
5. Finding of alternative plans from the execution history of this sql_id, and proposal for the creation of an SQL Plan Baseline, in order "to force" this alternative plan in future executions of this SQL statement.

From this list the most "painless" ones to implement -and thus the ones that are most commonly chosen during DW support sessions- are 1 and 2. This is because the creation of a new index (item 3) might impact other SQL statements as well, and thus it is not a decision that can be taken from a DW support engineer, while trying to deal with the performance incident, during "anti-social" hours, unless this is the only option for making the DW task complete on time. Even

so, it is a better idea to drop the index after the task completes and let the decision for a permanent creation of this index, for the next working day, along with the developers and DBAs.

Similarly, the rewrite of a production SQL statement (item 4 above) is not something to be decided during DW support sessions. However, the DW support engineer can try to rewrite this statement on his/her DB client of choice and then try to execute the DW task manually, without changing the production code. Of course since typically DW support engineers don't know the "logic" of the statement, in general they are very reluctant to touch the SQL code that someone else has written.

Finally, the creation of an SQL Plan Baseline (item 5) will influence all future executions of this task and force a specific execution plan. This is again an important decision that needs some testing before deciding to implement it and cannot be taken during a DW support session.

Therefore, for all these reasons the easiest recommendations to implement are the gathering of statistics (item 1) and the creation of an SQL Profile (item 2). Note that an SQL Profile does not force a specific execution plan, it just provides some more information to the optimizer (e.g. the cardinalities of specific operations in the plan) so as the optimizer can choose (hopefully) a better plan. According to the Oracle Database Performance Tuning Guide, an SQL profile is to an SQL statement what statistics are to a table or index.

In (Karagiannidis1, 2015), we provide two scripts: a) *sqltune_exec.sql* and b) *sqltune_report.sql* for calling the SQL Tuning Advisor. With the former you can create a so-called "SQL Tuning Task", which is the task that does all the aforementioned analysis for a specific sql_id, and the latter reports the list of recommendations. For more information about Automatic SQL Tuning you can read the Oracle Database Performance Tuning Guide (Oracle3, 2015).

Find an older execution plan and try this one

This is the most sophisticated method in the black-box approach. Conceptually, all we try to do is to find for our problematic DW task, a different execution plan (than the one that is currently executing) from the execution plan history of the database, which corresponds (according to our ETL execution metadata) to a "good" execution of the DW task in question (i.e., one with an acceptable execution time) and try to run the task with this older plan. Let's outline the basic steps for implementing this method:

1. **Step 1**:
   For the specific DW task, search in your ETL execution metadata, in order to find some good executions in the past.

2. **Step 2**:
   For the specific sql id corresponding to the DW task from Step 1, search the database execution plan history and try to correlate a specific execution plan (corresponding to a unique *Plan Hash Value (phv)*) to a good execution of the task that you have found from Step1.

3. **Step 3**:
   Run the DW task with the old plan, i.e., the one identified from Step 2.

The first step is very important because it will drive all the other steps. Essentially, we need to search our ETL tool's metadata in order to find acceptable executions of the DW task in question. In (Karagiannidis1, 2015), we provide script *owb_execs.sql*, which searches Oracle Warehouse Builder metadata for a specific leaf node (e.g., a mapping, or a PL/SQL procedure) of an ETL flow and returns all executions for a specific time period, with usefull information such as the duration in minutes of the execution of this node, the percentile 80 of the duration of this node in the time period of interest etc. (note: this script is basically a query over the view `all_rt_audit_executions`, which belongs to the Oracle Warehouse Builder Runtime Repository

Public Views and might need slight modifications to run in your environment. In any case, querying ETL tool metadata should be something straightforward)

The second step consists of a search in the database execution plan history for the specific sql id corresponding to the DW task in question. Note that in Oracle an execution plan is uniquely identified by the so-called *Plan Hash Value (phv*), which is a unique numeric value. When we have a specific sql id at hand, then we have a great chance that Oracle has stored the execution history of our sql id in AWR (i.e., the Automatic Workload Repository) and expose it via the DBA_HIST_SQLSTAT view. So the only thing we have to do is to query this view. Note that AWR stores hourly snapshots and therefore in order to get the rough time that the execution took place you must join to DBA_HIST_SNAPSHOT.

It is important to understand that when you review the history of executions of a specific sql id and you see the list of various execution plans that were used in the past; you must be sure which one of these are the "good" ones. This is where the results from Step 1 must be used. From Step 1 we have found the exact day and time of good executions of the DW task of interest and we just need to match these to the execution plan history from AWR, in order to get a list of candidate good plans. In (Karagiannidis6, 2015), we provide detailed information and scripts on how you can achieve this. Moreover, the script *fs_awr.sql* with which you can search the execution history of a specific sql_id can be found in (Karagiannidis1, 2015). Finally, in (Karagiannidis7, 2013), we provide additional information on how to retrieve *all available* execution plans of a query, regardless if these reside in AWR, or in the *Cursor Cache*, or both.

The last step is to run the DW task with one of the old plans that we have found in the AWR with the hope that our problematic task will now complete in a reasonable time. So, we have a specific sql id (that of the DW task) and a plan hash value (that of the good plan) and we want to “force” the optimizer to use this specific plan for this SQL statement. How do we do this? Next, we briefly describe three methods and argue which is the most suitable in the context of DW support:

*Force an execution plan with an SQL Plan Base*line

There are several ways with which you can force a specific plan in Oracle and guarantee plan stability. From 11g and onwards the recommended method for achieving this is called: *SQL Plan Management* with the use of SQL *Plan Baselines*. Links to useful material on SQL Plan Management and SQL Plan Baselines can be found in (Karagiannidis8, 2014). The method, as well as the script for forcing a specific execution plan with an SQL Plan Baseline, is described in detail in (Karagiannidis9, 2015).

In our opinion, this method is not the most appropriate to be utilized during a DW support session that, as we have mentioned many times, takes place during “anti-social” hours and has a specific goal, which is not to permanently solve the problem, but to find a way to make the DW task complete in time so as not to miss the SLA with the business. This is because by definition, SQL Plan Baselines are a permanent solution on your production Data Warehouse. Once you have defined them, then the specific task will always run with the provided execution plan, unless the SQL text changes or you decide to evolve the SQL plan baseline, or of course if you drop the SQL plan baseline. This might very well be the appropriate solution but this is not something to be decided during a DW support session; it must be decided after some testing (in normal working hours), along with the developers and DBAs. So we do not recommend this method for DW support.

Force an execution plan with a (special type of) SQL Profile

Prior to SQL Plan Baselines (e.g. in versions before 11g) one could achieve the same result with *a special type* of SQL Profile that can “fix” the execution plan of a specific SQL statement. Note that this is a different type of SQL Profile than the ones recommended from the SQL Tuning Advisor, which we discussed in the previous section. As we mentioned in that section the SQL Profiles proposed by the SQL Tuning Advisor *do not* force a specific plan but merely provide

statistical information to the optimizer for the query in question. In (Karagiannidis10, 2013), we describe in detail this method.

The same exactly arguments hold for this method as the one with the SQL Plan Baselines for forcing a plan. We do not believe that it is quite appropriate during a DW support session, because it implements a permanent solution.

```
SQL_ID  9zp6k1bfjr35r, child number 0
-------------------------------------
select /*+ ORDERED USE_NL(TLARGE) full(TLARGE) NOPARALLEL */ * from
TSMALL, TLARGE WHERE tsmall.id = tlarge.id

Plan hash value: 3757567954

-----------------------------------------------
| Id  | Operation                   | Name   |
-----------------------------------------------
|   0 | SELECT STATEMENT            |        |
|   1 |  NESTED LOOPS               |        |
|   2 |   TABLE ACCESS STORAGE FULL| TSMALL |
|*  3 |   TABLE ACCESS STORAGE FULL| TLARGE |
-----------------------------------------------
Query Block Name / Object Alias (identified by operation id):
-------------------------------------------------------------

   1 - SEL$1
   2 - SEL$1 / TSMALL@SEL$1
   3 - SEL$1 / TLARGE@SEL$1

Outline Data
-------------

  /*+
      BEGIN_OUTLINE_DATA
      IGNORE_OPTIM_EMBEDDED_HINTS
      OPTIMIZER_FEATURES_ENABLE('11.2.0.3')
      DB_VERSION('11.2.0.3')
      OPT_PARAM('_optimizer_use_cbqt_star_transformation' 'false')
      OPT_PARAM('star_transformation_enabled' 'true')
      OPT_PARAM('optimizer_index_cost_adj' 10000)
      ALL_ROWS
      OUTLINE_LEAF(@"SEL$1")
      FULL(@"SEL$1" "TSMALL"@"SEL$1")
      FULL(@"SEL$1" "TLARGE"@"SEL$1")
      LEADING(@"SEL$1" "TSMALL"@"SEL$1" "TLARGE"@"SEL$1")
      USE_NL(@"SEL$1" "TLARGE"@"SEL$1")
      END_OUTLINE_DATA
  */

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

   3 - storage("TSMALL"."ID"="TLARGE"."ID")
       filter("TSMALL"."ID"="TLARGE"."ID")
```

Listing 1. The Outline Data section from DBMS_XPLAN.DISPLAY_CURSOR output.

Use hints from the Outline Data section output from dbms_xplan.display_awr

The most suitable option for running an SQL statement and forcing a specific execution plan, in the context of DW support session is to use hints. Hints will allow you to run the problematic DW task (e.g.. manually from a DB client) and continue with the rest of the flow, without affecting the production code. Of course, in collaboration with developers and DBAs a decision should be

taken at a later time, as to what is the most suitable solution for the specific problem, and this might as well be to incorporate these hints into the production code permanently.

The problem is that we need a "set of hints" that will guarantee that the "whole" execution plan generated will be the same with one identified as a "good plan" from Step 2 above. Fortunately, Oracle provides us with exactly this set of hints. These hints are kept internally by Oracle in stored outlines and SQL plan baselines but we can access them from the *Outline Data section* of the DBMS_XPLAN.DISPLAY_AWR function when using the ADVANCED format option, like the following (check also script *xplan_awr_all.sql* from (Karagiannidis1, 2015)):

```
select *
from table(
       dbms_xplan.display_awr(    sql_id =>'&sql_id',
                                  plan_hash_value => '&plan_hash_value',
                                  format => 'ADVANCED ALLSTATS LAST')
       );
```

In Listing 1, we show an example of the Outline Data section generated by the DBMS_XPLAN.DISPLAY* functions for a specific sql id and plan hash value. If you just take the the /*+ */ hint and paste it into your statement at any point where a hint is legal, the optimizer will generate the specific plan and thus you can easily test the "good" plan from Step 2, from a DB client without affecting the production data warehouse.

The White Box Approach for SQL Tuning a DW Task

This approach urges you to fearlessly look at the execution plan of the problematic DW task in order to identify "the problem". "The problem" in the execution plan is nothing more than the operations that are delaying the most. Note that wall-clock time is what we are interested here, and not DB time. What are the operations that Oracle has been executing for the most of the time? That is what matters the most.

At the heart of this approach lies the concept that the solution to the SQL tuning problem is merely to find the operations in the plan that consume most of the time, then understand why is this happening, and then try to change this part of the plan (typically with hints). Of course, this is easily said but not always so easily done. Don't forget that a typical execution plan, of a real-world DW task, might consist of hundreds of operations. Moreover, usually the DW professional assigned to solve the performance issue, is not the developer of the running task and knows nothing about the "logic" behind it.

Therefore, this approach definitely has a steeper start, but once you manage to identify the problem, then you are half way to the solution. And, more importantly, this will be the "real" solution, since you have identified the root-cause of the delay, i.e., the real problem in the execution plan. Compare this with the black-box approach, where the "solutions" are somewhat coincidental and interim.

Thankfully, Oracle provides us with the tools to make the identification part easier. Next we will discuss how one can easily identify the most time consuming operations in the plan, then discuss about the most common reasons for such delays, and finally show how to fix the problem with the use of optimizer hints.

Find the operation that causes the delay

If I had to answer the question, what is the coolest performance management feature implemented in the Oracle DBMS up to now, then, with no hesitation, my answer would definitely be, the *Active Session History* (ASH). In (Wood, 2005), one can find a great introduction to ASH and the fantastic capabilities that it has offered for managing Oracle performance. ASH has been first introduced in 10g version and it has been constantly enhanced with each new version. Unfortunately, it is licensed as part of the Diagnostic pack.

Due to lack of space, we won't get into the details of ASH in this article (see (Wood, 2005) and the Oracle Performance Tuning Guide), but for the sake of the discussion, keep in mind that ASH provides us with second-level snapshots of all active sessions, and thus it makes it very easy to understand the activity history of an SQL statement (i.e., of the problematic DW task). It is so easy, which is merely a simple group-by query:

```
SELECT SQL_PLAN_LINE_ID, SQL_PLAN_OPERATION, SQL_PLAN_OPTIONS,
       count(distinct sample_time) wctime,
       round(count(distinct sample_time)/
              sum(count(distinct sample_time)) over() *100) "WC Time (%)"
FROM V$ACTIVE_SESSION_HISTORY
WHERE
       SQL_ID = '&sql_id'
GROUP BY SQL_PLAN_LINE_ID, SQL_PLAN_OPERATION, SQL_PLAN_OPTIONS
ORDER BY wctime desc;
```

The above query returns the "most active" operations of the execution plan, for an SQL statement corresponding to a specific sql_id. "Most active" means that the session (or the sessions for a parallel query) has consumed the most (wall-clock) time executing these operations. The operation is uniquely identified within the plan by its id, which is returned in the column `SQL_PLAN_LINE_ID`. Note that all three columns (`sql_plan_line_id`, `sql_plan_operation` and `sql_plan_options`), which describe an operation in the execution plan (similarly to the output of `dbms_xplan.display_cursor()`), are available from 11g version.

In Listing 2, we can see an example of the output of the above query over ASH data. We can see the execution plan of a MERGE statement and then, the output of the query. Clearly, operation with id 1 is taking most of the time. Undoubtedly, finding the exact operations that delay the most, especially for big complex plans, is a huge step towards the identification of the problem. Nevertheless, for most pragmatic cases it is not enough. We have to understand what is the reason for the delay, so as to be able to find a fix for the execution plan.

In (Karagiannidis1, 2015) you can find an SQL*plus script (*ash_ops.sql*) for returning the most time-consuming operations of an SQL statement from ASH, which can be used for your daily DW support tasks. Also there, you can find the "sibling" script *ash_events.sql*, which returns the prevailing wait events of an SQL statement, again from ASH data. Normally, the picture you get from the most waited wait-events must be compatible to the most time-consuming operations. For example, if the most time-consuming operation is an index range scan, then the most waited event will be a "db file sequential read". If for example, the dominating wait event is not in the "User I/O" class, or "on CPU", then most probably the problem cannot be solved with SQL tuning, i.e., by impacting the operations in the execution plan, and is something else, irrelevant to SQL tuning (e.g. a resource contention issue, or some other DB problem). These two scripts can be invaluable tools for identifying the problem in an execution plan.

```
-------------------------------------------------------------------------------
| Id  | Operation                      | Name                  | E-Rows |
-------------------------------------------------------------------------------
|   0 | MERGE STATEMENT                |                       |        |
|   1 |  MERGE                         | SOC_CUSTOMER_VIEW_FCT |        |
|   2 |   PX COORDINATOR               |                       |        |
|   3 |    PX SEND QC (RANDOM)         | :TQ10002              |    998K|
|   4 |     VIEW                       |                       |        |
|*  5 |      HASH JOIN                 |                       |    998K|
|   6 |       PX RECEIVE               |                       |    578K|
|   7 |        PX SEND HASH            | :TQ10001              |    578K|
|   8 |         PX BLOCK ITERATOR      |                       |    578K|
|*  9 |          TABLE ACCESS STORAGE FULL| CV_ACCOUNT_TOTALS  |    578K|
|  10 |       BUFFER SORT              |                       |        |
|  11 |        PX RECEIVE              |                       |   8302K|
|  12 |         PX SEND HASH           | :TQ10000              |   8302K|
|  13 |          TABLE ACCESS STORAGE FULL| SOC_CUSTOMER_VIEW_FCT |   8302K|
-------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------

   5 - access("A"."UCM_CUSTOMER_ID"="UCM_CUSTOMER_ID")
   9 - storage(:Z>=:Z AND :Z<=:Z AND "ACTIVE_ACCOUNT_IND"<>1)
       filter("ACTIVE_ACCOUNT_IND"<>1)

SQL_PLAN_LINE_ID SQL_PLAN_OPERATION             SQL_PLAN_OPTIONS                 WCTIME WC Time (%)
---------------- ------------------------------ ------------------------------ ---------- ----------
               1 MERGE                                                            901         73
              10 BUFFER                         SORT                             233         19
              13 TABLE ACCESS                   STORAGE FULL                      64          5
              12 PX SEND                        HASH                              19          2
              11 PX RECEIVE                                                        7          1
               2 PX COORDINATOR                                                    3          0
               3 PX SEND                        QC (RANDOM)                       2          0
               5 HASH JOIN                                                         1          0
               0 MERGE STATEMENT                                                   1          0

9 rows selected.
```

Listing 2. Simple query to find the most time-consuming operations of an execution plan.

Another way to achieve the same thing, i.e., to find the most time consuming operations in the plan, as well as the dominating wait events, is to use the *Real-Time SQL Monitoring* feature (Oracle1, 2015). Real-time SQL monitoring is an 11g feature and provides us with the ability to monitor an SQL statement on-line as it is executing. In (Karagiannidis2, 2015), you can find a real-world example of identifying the problematic operations in a complex execution plan, for a query that it does not even finish, with the use of ASH, or Real-Time SQL Monitoring.

What is needed next, is to understand why is this happening? This is the most difficult part and is the part that requires: (a) experience and (b) the capability to read and understand execution plans. We will discuss next some of the most typical reasons for operations that take too much time to complete.

Find the reason behind the delay

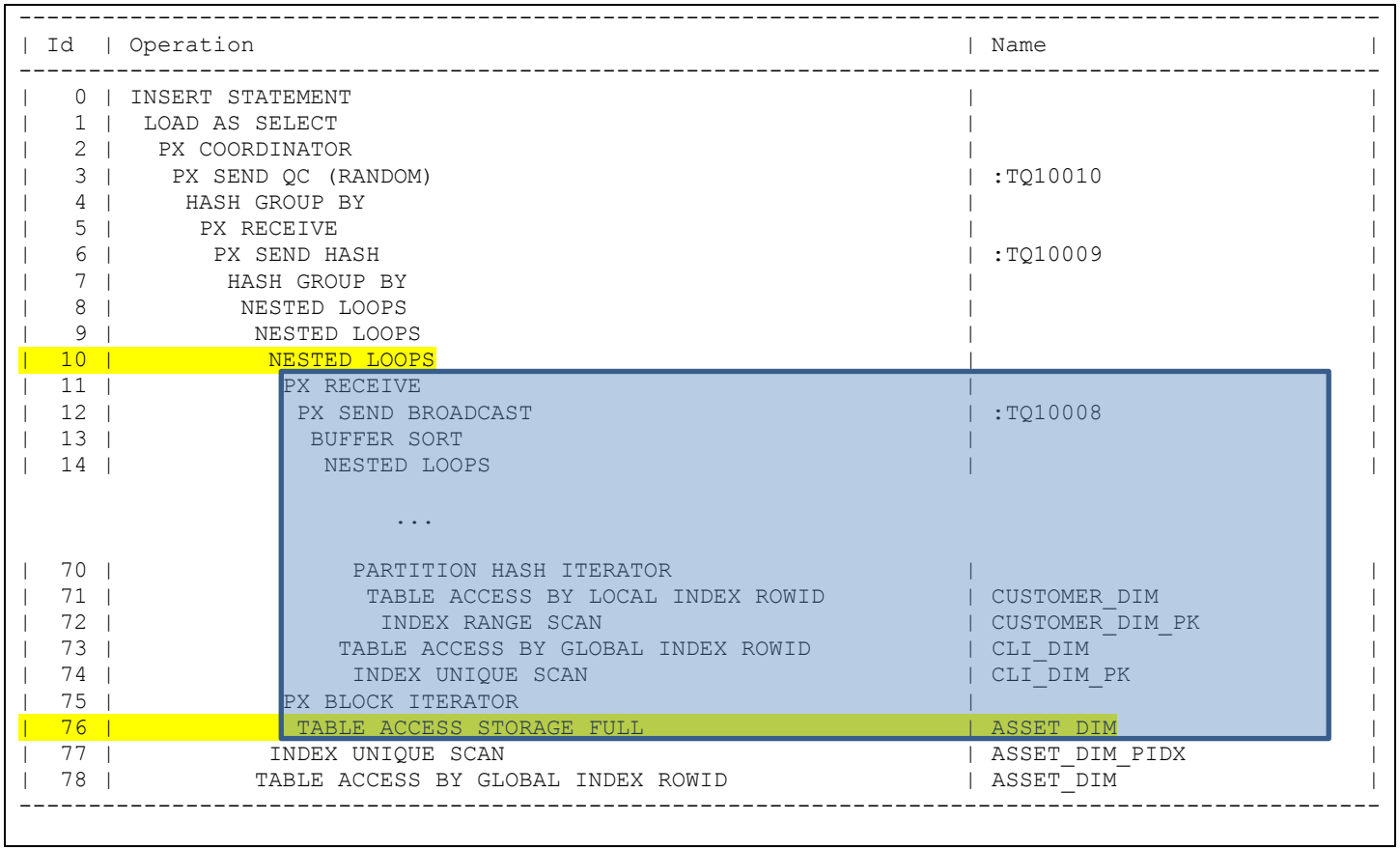
The following example is from a real incident on a production Data Warehouse. We have a bulk insert operation that is experiencing a performance issue. The execution plan is depicted in Listing 3. It is a moderately large plan consisting of 79 operations. We have deliberately omitted some of the lines in the plan so as not to clutter the page with unnecessary information. When the DW Support engineer was called, this DML had been running already for five hours. In Listing 4, we show the output of the ash_ops.sql script, which returns the most time-consuming operations from the ASH.

ASH clearly shows operation with id 76, a full table scan on a table called ASSET_DIM, as the operation that consumed almost 100% of the elapsed time (this is a rounded result of course). Bear in mind that there is no filter predicate on this table, and thus we want to access all the rows. So a full table scan seems like a better option than an index range scan and justifies the decision of the optimizer for accessing ASSET_DIM table. Moreover, since the specific case was on an Exadata machine, where it is known that full table scans are preferable (at least for typical DW queries), since they are a prerequisite for smart scans (an Exadata performance feature), this gives us one more reason to believe that a full table scan is the right choice. But then, why is there a delay? What is wrong with this operation?

The answer to this question is generally the most difficult part in the SQL tuning process. Here is where experience, as well as familiarization with reading execution plans, is a must-have. When reading execution plans is very important to always have in mind (among other things) the parent-child relationship between operations. This relationship can unveil very interesting results when you are investigating the root-cause of a delay.

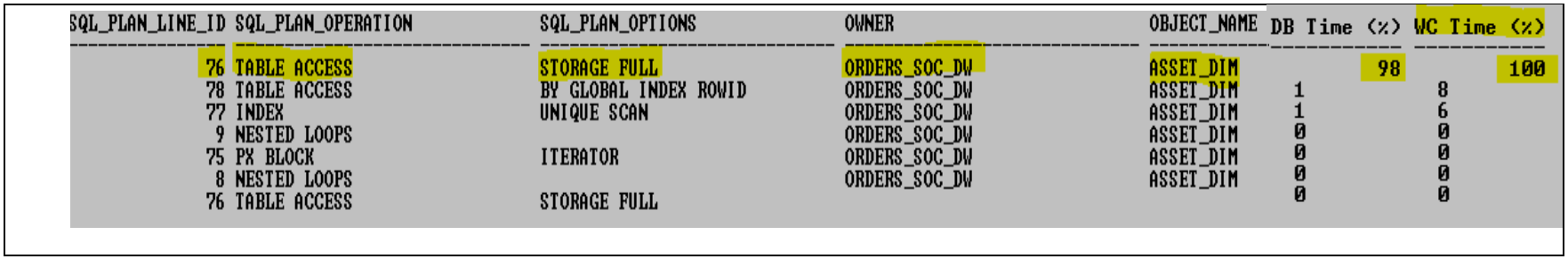
If we look carefully again at Listing 3, we can see that the problematic operation (id = 76) is the second child of a NESTED LOOPS operation (the one with id 10). This is depicted by the shaded rectangle. We know from theory, that the second child operation of a NESTED LOOPS join is executed as many times as are the rows of the driving table (or better: the driving “row source”), which is the first child operation. So the full table scan of table ASSET_DIM, in our example, is executed thousands of times, as many as are the output rows from the first child row-source. This means that the problem is not in the method by which we access the data (i.e., the full table scan operation), but that we do this repeatedly (thousands of times), which is a consequence of the NESTED LOOPS join method, or maybe of the join order chosen by the optimizer, which did not resulted in an early filtering of the intermediate results rows, or of both of these issues.

The mere reason for discussing the above example is to point out that once we identify the most time-consuming operations, then we need to understand what is wrong in the plan. This is the only way for finding a fix in the white-box method. In the specific case, the fix was a simple hint that forced a HASH JOIN instead of a NESTED LOOPS for ASSET_DIM but also the determination of the first table in the join order with a LEADING hint. This allowed the DW task to complete in 23 secs (remember when DW support was called, the task had been running for 5 hours without any evidence of when it will finish).



```
-----------------------------------------------------------------------------------------------
| Id  | Operation                                                | Name                         |
-----------------------------------------------------------------------------------------------
|   0 | INSERT STATEMENT                                         |                              |
|   1 |  LOAD AS SELECT                                          |                              |
|   2 |   PX COORDINATOR                                         |                              |
|   3 |    PX SEND QC (RANDOM)                                   | :TQ10010                     |
|   4 |     HASH GROUP BY                                        |                              |
|   5 |      PX RECEIVE                                          |                              |
|   6 |       PX SEND HASH                                       | :TQ10009                     |
|   7 |        HASH GROUP BY                                     |                              |
|   8 |         NESTED LOOPS                                     |                              |
|   9 |          NESTED LOOPS                                    |                              |
|  10 |           NESTED LOOPS                                   |                              |
|  11 |            PX RECEIVE                                    |                              |
|  12 |             PX SEND BROADCAST                            | :TQ10008                     |
|  13 |              BUFFER SORT                                 |                              |
|  14 |               NESTED LOOPS                               |                              |
                                                                                              
                                    ...                                                       
                                                                                              
|  70 |                 PARTITION HASH ITERATOR                  |                              |
|  71 |                  TABLE ACCESS BY LOCAL INDEX ROWID       | CUSTOMER_DIM                 |
|  72 |                   INDEX RANGE SCAN                       | CUSTOMER_DIM_PK              |
|  73 |                TABLE ACCESS BY GLOBAL INDEX ROWID        | CLI_DIM                      |
|  74 |                 INDEX UNIQUE SCAN                        | CLI_DIM_PK                   |
|  75 |            PX BLOCK ITERATOR                             |                              |
|  76 |             TABLE ACCESS STORAGE FULL                    | ASSET_DIM                    |
|  77 |           INDEX UNIQUE SCAN                              | ASSET_DIM_PIDX               |
|  78 |          TABLE ACCESS BY GLOBAL INDEX ROWID              | ASSET_DIM                    |
-----------------------------------------------------------------------------------------------
```

Listing 3. The execution plan of the problematic INSERT statement.



SQL_PLAN_LINE_ID	SQL_PLAN_OPERATION	SQL_PLAN_OPTIONS	OWNER	OBJECT_NAME	DB Time (%)	WC Time (%)
76	TABLE ACCESS	STORAGE FULL	ORDERS_SOC_DW	ASSET_DIM	98	100
78	TABLE ACCESS	BY GLOBAL INDEX ROWID	ORDERS_SOC_DW	ASSET_DIM	1	8
77	INDEX	UNIQUE SCAN	ORDERS_SOC_DW	ASSET_DIM	1	6
9	NESTED LOOPS		ORDERS_SOC_DW	ASSET_DIM	0	0
75	PX BLOCK	ITERATOR	ORDERS_SOC_DW	ASSET_DIM	0	0
8	NESTED LOOPS		ORDERS_SOC_DW	ASSET_DIM	0	0
76	TABLE ACCESS	STORAGE FULL			0	0

Listing 4. Output of ash_ops.sql script, showing the most time-consuming operations.

The good news is that, in the majority of the cases, the reasons that cause delays in operations of an execution plan, at least for Data Warehouse type of workloads, are quite common, and that these reasons keep showing up again and again in real-world cases. Actually, they are so common that we can group them in two large categories: a) *Repetition related* and b) *Filtering related* (this categorization of SQL performance problems can be found in (Savvinov, 2015)). We provide an outline of each category next.

Repetition-Related Performance Problems

A repetition-related performance problem is one where an operation is executed repeatedly many more times than necessary. This repetition is caused by the driving operation, which typically is a sibling operation (the 1st child of the common parent). This case appears in the so-called *related-combined operations.* In (Antognini, 2014), related-combine operations are defined as all the operations having multiple children, where one of the children controls the execution of all other children. The following operations are of this type: NESTED LOOPS, FILTER, UPDATE, CONNECT BY WITH FILTERING, UNION ALL (RECURSIVE WITH), and BITMAP KEY ITERATION. Of course, the most usual case is the NESTED LOOPS operation. Let's see a simple example of the problem:

In Listing 5, we show the execution plan of a simple join between TSMALL and TLARGE. Table TSMALL has 99 rows while TLARGE 10,000. TSMALL is the driving table and thus controls the execution of the Full Table Scan (FTS) over TLARGE (operation 3). As we can see from the "Starts" column, this FTS operation is executed 99 times, as many as are the rows of TSMALL. This repetition causes a delay of 15 seconds (as we can see from the actual time – A-Time column). Note that the very same join executed with a hash join, takes only 1.63 seconds (almost 1/10 of the time). The execution plan with the hash join is depicted in Listing 6. In this case, the FTS over TLARGE (operation with id 3) is executed only once due to the hash join method.

```
select /*+ ORDERED USE_NL(TLARGE) full(TLARGE) NOPARALLEL */ *
from TSMALL, TLARGE
WHERE tsmall.id = tlarge.id
-------------------------------------------------------------------------------------------------------------
| Id  | Operation                   | Name   | Starts | E-Rows | A-Rows |   A-Time   | Buffers | Reads  |
-------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |        |      1 |        |     99 |00:00:15.20 |    6146K|   6145K|
|   1 |  NESTED LOOPS               |        |      1 |     99 |     99 |00:00:15.20 |    6146K|   6145K|
|   2 |   TABLE ACCESS STORAGE FULL| TSMALL |      1 |     99 |     99 |00:00:00.01 |      19 |      0 |
|*  3 |   TABLE ACCESS STORAGE FULL| TLARGE |     99 |      1 |     99 |00:00:15.20 |    6146K|   6145K|
-------------------------------------------------------------------------------------------------------------

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

   3 - storage("TSMALL"."ID"="TLARGE"."ID")
```

Listing 5. The FTS on TLARGE is executed 99 times. This is a typical example of a repetition-related performance problem.

```
select /*+ ORDERED USE_HASH(TLARGE) full(TLARGE) NOPARALLEL gather_plan_statistics*/ *
from TSMALL, TLARGE
WHERE tsmall.id = tlarge.id
-------------------------------------------------------------------------------------------------------------
| Id  | Operation                   | Name   | Starts | E-Rows | A-Rows |   A-Time   | Buffers | Reads  |
-------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |        |      1 |        |     99 |00:00:01.63 |   62095 |  62072 |
|*  1 |  HASH JOIN                  |        |      1 |     99 |     99 |00:00:01.63 |   62095 |  62072 |
|   2 |   TABLE ACCESS STORAGE FULL| TSMALL |      1 |     99 |     99 |00:00:00.01 |      12 |      0 |
|   3 |   TABLE ACCESS STORAGE FULL| TLARGE |      1 |  10000 |  10000 |00:00:01.63 |   62083 |  62072 |
-------------------------------------------------------------------------------------------------------------

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

   1 - access("TSMALL"."ID"="TLARGE"."ID")
```

Listing 6. The FTS on TLARGE is executed only once due to the HASH JOIN join method.

```
select /*+ index(TLARGE) gather_plan_statistics  */ *
from TSMALL,TLARGE
WHERE tsmall.id = tlarge.id
-----------------------------------------------------------------------------------------------------
| Id  | Operation                     | Name       | Starts | E-Rows | A-Rows |   A-Time   | Buffers |
-----------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |            |      1 |        |     99 |00:00:00.01 |      59 |
|   1 |  NESTED LOOPS                 |            |      1 |        |     99 |00:00:00.01 |      59 |
|   2 |   NESTED LOOPS                |            |      1 |     99 |     99 |00:00:00.01 |      45 |
|   3 |    TABLE ACCESS STORAGE FULL  | TSMALL     |      1 |     99 |     99 |00:00:00.01 |      19 |
|*  4 |    INDEX RANGE SCAN           | TLARGE_IDX |     99 |      1 |     99 |00:00:00.01 |      26 |
|   5 |   TABLE ACCESS BY INDEX ROWID| TLARGE     |     99 |      1 |     99 |00:00:00.01 |      14 |
-----------------------------------------------------------------------------------------------------

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

   4 - access("TSMALL"."ID"="TLARGE"."ID")
```

Listing 7. The use of an index access path to TLARGE reduces dramatically the LIOs per row ratio.

Note that the real problem with the nested loops plan, in our example, is that it results into too much more work than it is actually necessary. The result of the join of the two tables is 99 rows. Now, in order to get these 99 rows, in the nested loops plan we have to "pay" the cost of doing 99x[TLARGE number of blocks] logical reads, while in the hash join plan we have to pay the cost of doing only 1x[TLARGE number of blocks]. So in terms of number of logical reads (which according to (Antognini, 2014) is an excellent metric to tell you the amount of work done by the database engine) we are paying a very high price (number of logical reads per returned row). Actually, the LIOs per row ratio is 6,146K/99 = 63,570 for the NESTED LOOP plan, and 62,095/99 = 627 for the HASH JOIN plan. This can be computed by dividing the "Buffers"/"A-Rows" at operation 1 in Listing 5 and Listing 6 respectively.

In our case, both plans have too large LIOs per row ratios. If we create an index on the second table (TLARGE), and access the rows through the index, then we get a below-second elapsed time and the LIOs per row ratio drops down to 59/99 = 0.6. This is depicted in Listing 7. So you see that the problem caused by the excessive repetition in this example, could be fixed either by removing the repetition altogether (via a HASH JOIN join method), or by minimizing the blocks that we have to read at each iteration.

A similar repetitive behavior can be found with the FILTER operation when it has more than one children operations. This operation typically appears when a subquery fails to be unnested by the optimizer. FILTER is executed in a "nested-loops fashion", i.e., for each row of the driving row-source the second child operation is executed. In Listing 8, we show an example of a FILTER operation and its repetitive behavior. In this case the presence of an OR predicate blocks the subquery un-nesting and thus the FILTER operation appears in the execution plan. Note from the "Starts" column, that operation with id 3 (fast full index scan) is executed 99 times, as many as the rows of the driving table (TSMALL)

```
select /*+ noparallel gather_plan_statistics */ *
from tsmall s where
  exists (select 1 from tlarge where id = s.id ) OR id > 100

-----------------------------------------------------------------------------------------------------------------------
| Id  | Operation                              | Name       | Starts | E-Rows | A-Rows |   A-Time    | Buffers | Reads  |
-----------------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                       |            |      1 |        |     99 |00:00:00.01 | 4771 |          1 |
|*  1 |  FILTER                                |            |      1 |        |     99 |00:00:00.01 | 4771 |          1 |
|   2 |   TABLE ACCESS STORAGE FULL            | TSMALL     |      1 |     99 |     99 |00:00:00.01 |   19 |          0 |
|*  3 |   INDEX STORAGE FAST FULL SCAN FIRST R| TLARGE_IDX |     99 |      1 |     99 |00:00:00.01 | 4752 |          1 |
-----------------------------------------------------------------------------------------------------------------------

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

   1 - filter(("ID">100 OR  IS NOT NULL))
   3 - filter("ID"=:B1)
```

Listing 8. The FILTER operation with two children has a "repetitive behavior" just like a NESTED LOOPS operation

The main point of this session is that repetition can be a performance problem when the number of repetitions is large and/or the repeated operation is large (e.g., a full table scan of a big table). We should be aware of this and always watch out for repetition-related bottlenecks during the investigation of why some operations are taking too long to complete. In a following session, we will provide some tips on how to remove this repetition from an execution plan but next we will describe the filtering-related category of performance problems.

Filtering-Related Performance Problems

A filtering-related performance problem is one where we read more rows than we actually need to (see the LIOs per row ratio discussion above). As a result, we are wasting a lot I/O due to the lack of a more efficient access path to the data. Typical examples of this problem are the lack of an appropriate index, or the lack of partition pruning (i.e., we access all the partitions of the underlying table when the relevant data are stored in only a few) in the presence of a selective filter predicate. Typical operations in an execution plan that might denote this problem are the FULL TABLE SCAN, and a PARTITION operation with the "ALL" option (e.g. PARTITION RANGE ALL).

Again the logical reads per row returned is the most appropriate metric in order to measure the problem. According to (Antognini, 2014) the following rule of thumb applies:

- Access paths that lead to less than about 5 logical reads per returned row are probably good.
- Access paths that lead to up to about 10–15 logical reads per returned row are probably acceptable.

- Access paths that lead to more than about 20 logical reads per returned row are probably inefficient. In other words, there's probably room for improvement.

In typical Data Warehouse workloads, Full Table Scans access paths are much more frequent than index accesses. This is due to the fact that analytical queries need to access a lot of data in order to group and aggregate them in various ways. So a typical DW query accesses millions of rows and returns only a few. Also typical DML and DDL statements embedded in the daily ETL flows do massive (bulk) inserts/updates on the data and not selective updates on a few rows, as in OLTP databases. So Full Table Scans (FTS) are "not bad" in DWs. On the contrary, one can say that they are the norm. Moreover, on Exadata DWs, FTS are a prerequisite for the smart scan optimization, which offloads some of the processing to the storage nodes.

Based on our experience we have seen the following cases as the most usual problems in this category:

1. An FTS is the right choice for accessing the data but is too slow because of:
    a. Inappropriate use of parallelism, or not use of parallelism at all
        i. Parallel DML is not enabled (it must be enabled explicitly), or blocked by some other reason (e.g. the existence of a trigger).
        ii. No parallel query due to no parallel degree on the underlying tables, or inappropriate parallel hints on the query.
    b. Too many chained rows in the table that cause single block reads, instead of multi-block reads. Note that the presence of chained rows on a table is also a blocker for smart scans on Exadata.
2. Partition pruning is blocked because of:
    a. Wrong partitioning scheme on the underlying tables. I.e., the chosen partitioning key does not match the most common filtering predicates of the SQL statements. This is clearly a design problem.
    b. A filtering condition that blocks pruning (e.g., the presence of a function)
3. In a distributed DML statement, where you typically read data from a remote site (a data source) via a database link and insert them into a DW staging table, processing is executed at the wrong site. The problem usually is that a large table travels over the network in order to be processed (e.g. joined or filtered) locally (at the DW site), where the processing should take place at the remote site before the data travel.
4. An update or delete of a small percentage of the rows of a large table (typically) less than 0.01% and an FTS is used instead of an index access.

In this section, we have tried to emphasize on the importance of understanding why some operations of an execution plan are taking too much time to complete. This is essential in order to fix the problem in the white box approach. To this end, we have categorized performance problems into two broad categories that -based on our experience – correspond to the majority of cases showing up in Data Warehouses. In the next section, we will provide some guidelines on the possible fixes for these problem categories.

Fix the Problem

The aim of this section is to provide some practical advice on how to remedy repetition-related problems and filtering-related problems. Let us repeat that the scope of this article is to help the day-to-day DW support. Therefore, the primary goal of the proposed fixes is to help the

problematic SQL statement to complete in a timely fashion, as soon as possible. Therefore, we propose the use of hints in order to influence the choices of the optimizer and impact the execution plan. This might not be the final solution for a production DW, and might be considered merely as an interim patch for dealing with the problem and closing the delay incident. The final solution might require a redesign of the code (e.g., change in the logic of the statement), and/or of the underlying structures (e.g., change of the partitioning scheme of a table), or a revision of the ETL flow logic, or some combination of all these, etc.

For solving a repetition-related problem there are actually only two options:

A. Remove the repetition altogether

B. Minimize the number of iterations and/or minimize the work per iteration

When the repetition problem is related to a join operation then we have a "problematic" NESTED LOOPS operation. Of course, the first solution that comes into mind is to replace the NESTED LOOPS join method with a HASH JOIN method and can be easily accomplished with the USE_HASH hint.

Indeed, a HASH JOIN is an "unrelated combine" operation (according to (Antognini, 2014) terminology), which means that the access to the second table (the probe table) of the join is not dependent on the first table (the driving table) and thus the repetition will be completely removed, i.e., the probe table will be accessed only once (as in Listing 6). The problem is that the join method might not be the real problem and even if you use the USE_HASH hint, there might be a case where the optimizer completely ignores your hint (see (Karagiannidis3, 2015) for the details). This is true especially for SQL statements that include n joins (where n > 1), an "n-way join" as it is called. Remember that a join operation joins only two tables at a time, and thus when we have n tables to be joined, the optimizer must decide on a specific order by which these tables will be joined. It is very important to remember that the first and most important decision when we have to join n tables is to find "a good" join order. It is so important that once we find a good join order and "hint it" to the optimizer, then usually this is enough and we don't need to get into the details of which join method (i.e., a NESTED LOOPS, or a HASH JOIN, or a MERGE JOIN) must be used for each join. In fact, the optimizer first decides on the join order and then on the join method (Nyffenegger, 2008). A good join order can certainly fix a repetition-related performance problem caused by a NESTED LOOPS join and can help towards A or B mentioned above.

Correct the Join order

When we have to join n tables and we want to find a "good" join order then there is one and only strategy to follow: *eliminate as much data as possible, as early as possible*.

Each time that we join two tables, an intermediate result (a row-source) is generated, which is joined with the next table in the join order and so on. This forms a so-called *join-tree.* In Figure 1, we depict an example of a *left-deep join tree* formed by the join of 5 tables (t1,t2,t3,t4,t5). (Note that the optimizer is only generating left-deep join trees, instead of *right-deep*, or *bushy* join-trees but these can be forced with the use of appropriate hints. See (Karagiannidis5, 2015) for a discussion of how right-deep join trees can be very useful in typical DW star queries.)

When dealing with large tables, it is of utmost importance that the very first joins executed, filter out the majority of the rows, otherwise it is certain that you will end up with a performance issue. The optimizer knows that, and does its best to find such an order, basically based on object statistics, but there are occasions that it fails to get such a good join order. In these cases, you can influence the join order decision with the use of the LEADING, or ORDERED hints (Oracle2, 2015).

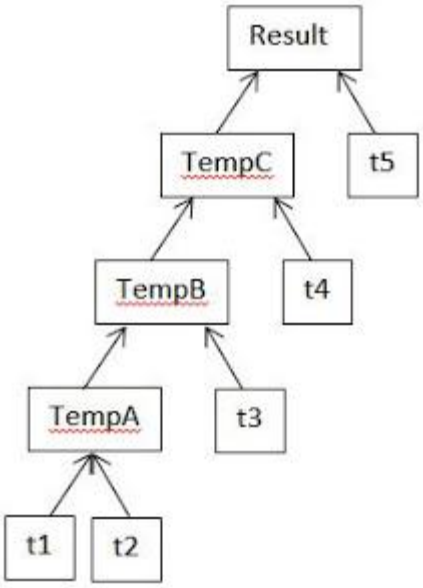


Figure 1. A left-deep join tree

In (Tow, 2003) an excellent method is introduced for finding a very good join order. This method has been also presented in (Hailey, 2013) by Kyle Hailey. It is actually quite simple:

1. Start with the table having the most selective filter.
2. Move on to the table with the next most selective filter moving along a join path and (if possible) going from a child to parent table (and not the opposite).

Let's see the method applied to a specific example. In this example we will not use an ad hoc query with multiple joins but rather we will use a *star query*. A star query is a query over a star schema and it constitutes the most frequent query pattern in a Data Warehouse database. It consists of a join of the Fact Table (the central table) to the surrounding Dimension tables, with some filters applied (usually on the Dimension tables, e.g. time, product, customer etc.) and finally a group by some dimension columns (e.g., month, product type, customer segment, etc.) and aggregations over some fact tables fields (e.g., sum of revenue, or count of orders etc.).

Such a star query is listed in Listing 9. A star query includes an n-way join but with a very important characteristic. The most significant characteristic of a star query n-way join is that all tables are joined only via the central table of the star schema (aka the fact table). Note that usually the fact table is orders of magnitude larger than the surrounding tables (aka the Dimension tables). This is a real headache for finding an optimal join order (i.e., one that eliminates as many rows as possible early on), simply because, even from the very first join, the fact table must be one of the two tables joined, and thus from the very first intermediate result, the majority of the rows are included in the join processing. This is why Oracle introduced the so-called *Star Transformation*, even from version 8, which results in a complete rewriting of the star query, in such a way so that the join of the fact table to the dimension tables is eliminated altogether (although implemented indirectly) and the qualifying rows of the fact table are accessed through bitmap indexes. Although, the star transformation is a brilliant idea in theory, in practice its stability has always been a problem. Too many bugs in all major versions of Oracle, that resulted to ORA-00600 errors or even wrong results. Moreover, it requires a "pure" star query, which is kind of rare in a real-world data warehouses and so, at least from our experience, its usefulness is kind of limited. For more details on the star transformation, you can read (Antognini, 2014), or the Oracle Database Data Warehousing Guide.

In Listing 9, we have noted the join conditions as well as the filter conditions. Note that not all tables have filter predicates. Once we have identified on the SQL text the filter predicates and the join conditions, then we can draw the *join graph*. This is depicted in Figure 2. Each node represents a table from the FROM clause and each arrow corresponds to a join (expressed via a join condition in the SQL text), pointing from the child to the parent table. As you can see, all joins are through the fact table in the middle. The filter conditions are expressed in the form of the *Filter Ratio (FR),* which is the ratio of the rows returned from each table when the filter condition is applied, to the total number of rows in the table. As you can see, there is an FR only for the three tables which include a filter predicate.

In Figure 3, we apply the aforementioned method for finding the best join order. According to the method, we select as the first table for our join order the table with the most selective filter (i.e., the smallest FR). This is the dimension table DWH2_WCRM_ORDER_TYPE_DIM, which has an FR = 0.003. Note that as we select each table for our join order, we can gradually add a table in a LEADING hint. This is depicted in the bottom of Figure 3.

Next, since there is no parent table to follow, we can only move towards the child table, which is the fact table CONSTR_WCRM_FCT with an FR = 0.02. Thus we can add one more table in the join order, which now becomes: DWH2_WCRM_ORDER_TYPE_DIM → CONSTR_WCRM_FCT. Next we have to select a parent table to visit and, since we are at the fact table, there are a lot of parent tables to choose. Essentially, all the dimension tables that are joined to the fact table, are eligible for the next table in the join order. However, according to the method, we have to choose the parent table with the most restrictive filter. In our case this is dimension table DWH2_WCRM_CONST_STATUS_DIM with an FR = 0,07.

So now the join order becomes: DWH2_WCRM_ORDER_TYPE_DIM → CONSTR_WCRM_FCT → DWH2_WCRM_CONST_STATUS_DIM. So far our leading hint has been formed as following:

/*+ LEADING (DWH2_WCRM_ORDER_TYPE_DIM, CONSTR_WCRM_FCT, DWH2_WCRM_CONST_STATUS_DIM) */

Since, there is no filter predicate on the rest of the tables, and also since we have already included in our join order the fact table (which is the only child table in a star schema), then all the joins that will follow will not change the number of rows produced from each join.

I.e., there is no further elimination of rows via a join to pursue, and vice –versa, there is no multiplication of rows via a join to delay to incorporate in the join order. Therefore, it does not really matter which tables will be included in the join order. Only the three that we have selected are important. And thus we are finished and have found an optimal join order for our star query, which can be expressed via a LEADING hint and applied as a patch for our problematic query, in order to fix the performance issue.

In the case of queries including views, or inline views, or subquery factored queries (WITH clause), then we can find the best join-order for each subquery individually and for the outer query block, handle the subqueries as tables, therefore applying the above method similarly. This can be accomplished in conjunction with the NO_MERGE hint which forces the optimizer to process each subquery separately. This is depicted next.

```
SELECT /*+ LEADING(v,t5, t1) */ …
FROM   t1, (  SELECT /*+ NO_MERGE LEADING(t3, t4, t2) */ …
              FROM t2,t3,t4
              WHERE <join conditions>
       )v,
       t5
WHERE <join conditions>
```

Also, you can find a more detailed description of such tuning techniques in the presence of subqueries in this (Lewis, 2015) great post by Jonathan Lewis.

```
SELECT ...
FROM
  PRESENT_PERIF.CONSTR_WCRM_FCT,
  TARGET_DW.PROVIDER_DIM  DWH2_WCRM_DEKTIS_PROVIDER_DIM,
  TARGET_DW.PROVIDER_DIM  DWH2_WCRM_DOTIS_PROVIDER_DIM,
  PRESENT_PERIF.ORDER_ORDER_STATUS_DIM  DWH2_WCRM_CONST_STATUS_DIM,
  PRESENT_PERIF.CONSTR_ORDER_TYPE_DIM  DWH2_WCRM_ORDER_TYPE_DIM,
  PRESENT_PERIF.CONSTR_ORDER_TYPE_DIM  DWH2_WCRM_ORIG_ORDER_TYPE_DIM,
  PRESENT_PERIF.SOURCE_SYSTEMS_DIM
WHERE
  ( DWH2_WCRM_DEKTIS_PROVIDER_DIM.PROVIDER_SK=PRESENT_PERIF.CONSTR_WCRM_FCT.PROVIDER_SK  )
  AND  ( DWH2_WCRM_DOTIS_PROVIDER_DIM.PROVIDER_SK=PRESENT_PERIF.CONSTR_WCRM_FCT.PROVIDER_SK_OWN  )
  AND  ( PRESENT_PERIF.CONSTR_WCRM_FCT.SOURCESYSTEM_SK=PRESENT_PERIF.SOURCE_SYSTEMS_DIM.SOURCESYSTEM_SK  )
  AND  ( PRESENT_PERIF.CONSTR_WCRM_FCT.ORDER_STATUS_SK=DWH2_WCRM_CONST_STATUS_DIM.STATUS_SK  )
  AND  ( PRESENT_PERIF.CONSTR_WCRM_FCT.CONSTR_ORDER_SK_ORIG=DWH2_WCRM_ORIG_ORDER_TYPE_DIM.CONSTR_ORDER_SK  )
  AND  ( PRESENT_PERIF.CONSTR_WCRM_FCT.CONSTR_ORDER_SK=DWH2_WCRM_ORDER_TYPE_DIM.CONSTR_ORDER_SK  )
  AND  ( PRESENT_PERIF.CONSTR_WCRM_FCT.BUSINESS_SOURCE =  'WNP'  )
  AND
  (
   case
    when ( PRESENT_PERIF.CONSTR_WCRM_FCT.NP_ID_NUMBER ) like 'P%' then 'Portability'
    when ( PRESENT_PERIF.CONSTR_WCRM_FCT.NP_ID_NUMBER ) like 'D%' then 'Disconnection'
    when ( PRESENT_PERIF.CONSTR_WCRM_FCT.NP_ID_NUMBER ) like 'U%' then 'Update'
    else 'Undefined'
end  IN  ( 'Portability'  )
   AND
   PRESENT_PERIF.CONSTR_WCRM_FCT.CONSTR_DATE_TRUNC  >=  '01-01-2015 00:00:00'
   AND
   DWH2_WCRM_CONST_STATUS_DIM.STATUS_CUSTGROUP_DESCR  IN  ( 'Ολοκληρωμένη'  )
  )
   AND DWH2_WCRM_ORDER_TYPE_DIM.CONSTR_ORDER_CODE in ('PRM_301', 'PRM_303')
GROUP BY ...
```

Joins

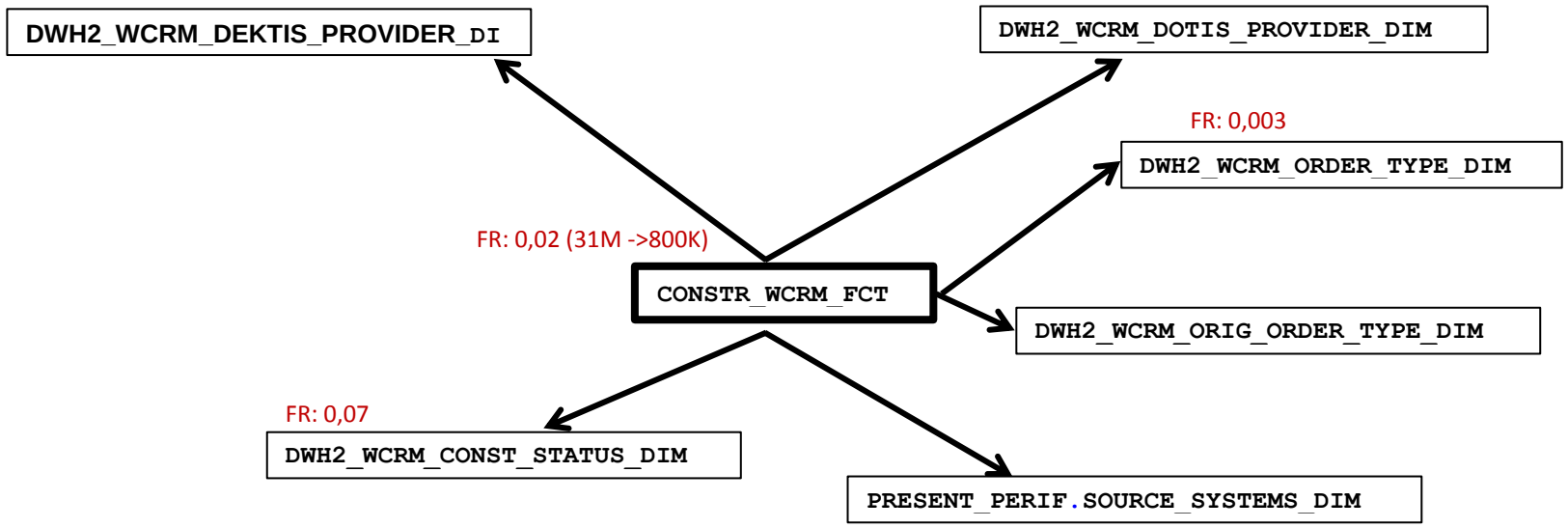
Filters

Listing 9. An example star query. All the tables are joined though the fact table.

Eliminating a FILTER operation

A FILTER operation, as we have mentioned previously, can be also the root of repetition. This is true when it appears in an execution plan having two, or more, child operations. In this case, the rows of the first child operation drive the execution of the other children operations, exactly as in the case of a NESTED LOOPS operation. Usually, the presence of this operation in an execution plan denotes the inability of the optimizer to *unnest* a subquery in the WHERE clause (i.e., a subquery, correlated or not, within a: IN, NOT IN, EXISTS, NOT EXISTS construct). Indeed, the

presence of an OR condition, or if a subquery contains some types of aggregation, or if it contains the rownum pseudocolumn, can block subquery unnesting. Also, semi-join and anti-join subqueries containing set operators can only be unnested as of version 11.2 (Antognini, 2014).

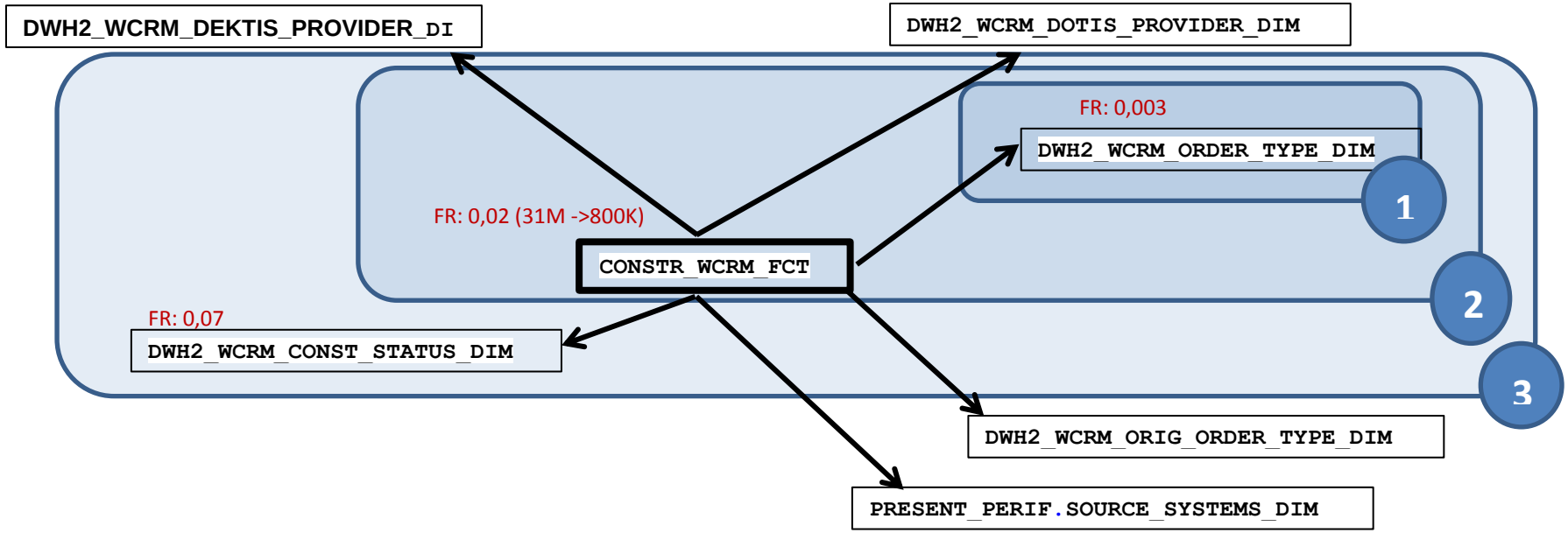


FR: Filter Ratio = (select count(*) from table where <condition>)/ (select count(*) from table)

Figure 2. The join graph of the example star query.

Note that the optimizer will always try to unnest a subquery, which is basically a *heuristic query transformation*, unless it finds some "blocking factor" as the ones mentioned previously. Note that in some cases, such as when you have a construct like WHERE col = (<subquery>), then subquery unnesting is a *cost-based transformation*, which means that the optimizer will decide based on the calculated cost, if it's worth to proceed with this transformation (see (Hasler, 2014) for more details).

Whenever you have a repletion-related performance problem and you identify the FILTER operation as the root-cause, then you can try to use the *UNNEST hint* within the subquery (or subqueries) that are relevant, in order to influence the optimizer who has rejected it, due to a cost-based decision. If this does not work, then there is no other way than to try to rewrite the SQL statement, in such a way, that unnesting will be allowed.



/*+ LEADING (DWH2_WCRM_ORDER_TYPE_DIM, CONSTR_WCRM_FCT, DWH2_WCRM_CONST_STATUS_DIM) */

Figure 3. Finding the best join order for the example star query.

For example, in (Karagiannidis2, 2015) we deal with a problematic FILTER operation, which causes a large Full Table Scan to be executed thousands of times. The inability of the optimizer to unnest the corresponding subquery and thus come up with a FILTER operation, was due to a (redundant as it resulted eventually) OR predicate of the form OR IN (<subquery>). Once this was removed, the query finished in a few seconds.

OR conditions with subqueries are a very common reason for problematic FILTER operations. The most usual method for eliminating such an OR condition is to rewrite the statement using a UNION ALL operation. In fact, this is also a standard (cost-based) query transformation performed by the optimizer when dealing with OR predicates and it is called *"OR-expansion"* . Moreover, there exists a special hint (*/*+ USE_CONCAT */)* in order to force this transformation.

In Listing 10, we show an example of a subquery that failed to be unnested by the optimizer due to the presence of an OR condition. In Listing 11, we have rewritten the query using a UNION ALL operation. Note that a UNION ALL does not remove duplicate rows. In order, to ensure that both queries return the same result (even in the presence of duplicate rows), we use function LNNVL(), which returns TRUE when the condition passed as a parameter is FALSE or NULL.

```
select *
from tsmall
where
    id > 10
    or
    id not in (select id from tlarge);
-------------------------------------------------------------------------------------
| Id  | Operation                                | Name   | Starts | E-Rows | A-Rows |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                         |        |      1 |        |     90 |
|*  1 |  FILTER                                  |        |      1 |        |     90 |
|   2 |   TABLE ACCESS STORAGE FULL              | TSMALL |      1 |     99 |     99 |
|*  3 |   TABLE ACCESS STORAGE FULL FIRST ROWS   | TLARGE |     10 |      1 |      9 |
-------------------------------------------------------------------------------------

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

   1 - filter(("ID">10 OR  IS NULL))
   3 - filter(LNNVL("ID"<>:B1))
```

Listing 10. A FILTER operation appears due to the OR condition

```
select *
    from tsmall
    where
        id > 10
UNION ALL
    select *
    from tsmall
    where
        id not in (select id from tlarge) and lnnvl(id > 10);
-----------------------------------------------------------------------------
| Id  | Operation                    | Name   | Starts | E-Rows | A-Rows |
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |        |      1 |        |     90 |
|   1 |  UNION-ALL                   |        |      1 |        |     90 |
|*  2 |   TABLE ACCESS STORAGE FULL  | TSMALL |      1 |     90 |     89 |
|*  3 |   HASH JOIN ANTI NA          |        |      1 |      1 |      1 |
|*  4 |    TABLE ACCESS STORAGE FULL | TSMALL |      1 |      9 |     10 |
|   5 |    TABLE ACCESS STORAGE FULL | TLARGE |      1 |  10000 |   9997 |
-----------------------------------------------------------------------------

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

   2 - storage("ID">10)
       filter("ID">10)
   3 - access("ID"="ID")
   4 - storage(LNNVL("ID">10))
       filter(LNNVL("ID">10))
```

Listing 11. After rewriting of the query the subquery is successfully unnested.

Moving on to the filtering-related performance problem we will discuss next some common fixes for the most typical cases.

Efficient Full Table Scans

A Full Table Scan (FTS) is the right access path to the data when we need to access all the rows of the table, or when we have a not very selective filter. FTS operations are very common in DW workloads, in contrast to OLTP environments, where index range scans are the norm. Moreover, an FTS is one of the two prerequisites for the Exadata smart scan optimization.

Very often we see an FTS being the cause of a delay in an execution plan. In the previous sections we covered the case where an FTS is repeated multiple times due to a repetition-based parent operation and discussed possible fixes. Now we are going to discuss how an FTS can be run efficiently.

Use PARALLEL hint correctly

An FTS on a large table is without a doubt a perfect candidate for parallelization. Indeed, dividing the scanning of the blocks to multiple parallel slaves is essentially the only feature we have to make an FTS go faster. Very often we see developers/dw support engineers trying to enforce parallelism to an FTS operation with the use of the PARALLEL hint, with no success.

It is important to understand that the PARALLEL hint only works when the optimizer has decided to access the underlying table via an FTS operation. If it has chosen an index access path , then the PARALLEL hint is simply ignored. Therefore, it is suggested that the PARALLEL hint is used in conjunction to the FULL hint, which forces an FTS access path. So instead of writing: `/*+ PARALLEL(t <degree>) */`. It is better to write: `/*+ FULL(t) PARALLEL(t <degree>) */`

Moreover, on a busy system a very large degree of parallelism might cause a bigger problem than the one it is trying to solve. Very often, developers use the PARALLEL hint without a degree specified, having the impression that Oracle will choose the DOP (Degree of Parallelism) of the table (i.e., the parallel degree declared for the corresponding table at table creation time). This is not true. When you don't specify a degree in a parallel hint, then Oracle will request a DOP equal to the number of CPUs multiplied by the parameter parallel_threads_per_cpu. Bear in mind that the number of parallel slaves allocated is at least twice the number of the requested DOP (due to intra-parallelism between the operations of a plan, the parallel slaves are divided to producers and consumers), and thus you can easily see your parallelism being skyrocketed. In (Karagiannidis4, 2013), we show an example of this case.

Move tables to fix chained rows

Row chaining is when a row is too big to fit into a single data block and must be stored in "a chain of blocks". Initially the row is stored in a single block but as time passes, due to updates that take place, or due to add column operations, there comes a time that it no longer fits in a single block. In this case, Oracle allocates a second block to store the part of the row that does fit and stores a pointer in the initial block that point to it. Thus a chain of blocks is created.

The main problem with full table scans and chained rows is that an FTS operation over a table with a large percentage of chained rows is totally inefficient. This due to the fact that multi-block read, (i.e., the main performance feature of FTS (and Fast Full Index Scans)), is turned into single block reads, as Oracle has to read through the pointers and follow the chain of blocks for reading the chained rows. Moreover, chained rows are a blocker for smart scan on Exadata DWs.

This problem can be identified quite easily with the method described in section "Find the operation that causes the delay". Essentially, if we spot an FTS on a table as the bottleneck, and this is not due to a repetition problem, and also the prevailing wait event indicates a single block read (i.e., a "db file sequential read", or a "cell single block physical read" on Exadata), then most probably the underlying table has a significant percent of chained rows. To fix the problem you must issue a MOVE command, like the following:

```
ALTER TABLE t MOVE PARALLEL <d>;
```

The PARALLEL clause has been added in order to make the MOVE operation execute in parallel and thus finish faster. Note that indexes of the table must be rebuilt after the MOVE. For partitioned/sub-partitioned tables in particular this is easy because of the UPDATE INDEXES option in the MOVE command:

```
ALTER TABLE t MOVE partition p PARALLEL <d> UPDATE INDEXES;
```

Don't block partition pruning

Whenever you have an FTS operation on a partitioned table and at the execution plan you see the PARTITION operation with the "ALL" option (e.g. PARTITION RANGE ALL), although there is a restriction on the partitioning key of the table, then you have to remove, or rewrite, the condition that prevents Oracle from eliminating irrelevant partitions.

According to (Antognini, 2014), the partitioning key conditions that block partition pruning are the following:

1. Inequality conditions (!= or <>)
2. NOT IN conditions
3. IS NOT NULL conditions
4. Conditions based on expressions and functions
5. BETWEEN, >, >=, <, <= conditions, block partition pruning in *hash* partitioned tables
6. IS NULL conditions, block partition pruning in *hash* partitioned tables

To the degree that is feasible, according to the logic of the statement in question, a rewrite must be attempted in order to remove the blocking conditions.

Use parallel DML and APPEND for bulk inserts

Whenever, there is a large INSERT or MERGE operation that takes too long, then direct path loading, invoked via the APPEND hint, as well as parallel DML are imperative. Direct path loading is the method used by Oracle for achieving fast bulk inserts. Basically, it bypasses the buffer cache and loads the data directly from the server process PGA to disk. In addition, it inserts rows above the HWM (High Water Mark) of the table (i.e., it moves the HWM). This means that it spends more space than a conventional insert but it is faster because it does not try to find empty spots into blocks to insert the data.

Note that whenever you insert data via a direct path load you require a table-level lock (instead of a row-level lock), therefore concurrent DML operations on the same table will not be allowed, and that you cannot query or modify direct-path inserted data immediately after the insert is complete. If you attempt to do so, an ORA-12838 error is generated. You must first issue a COMMIT statement before attempting to read or modify the newly-inserted data.

Similarly, it is very important for large DML operations to exploit parallel DML. Note that direct path writes are the default mode for parallel inserts. However, parallel DML is not by default enabled. You have to explicitly enable it by issuing the command:

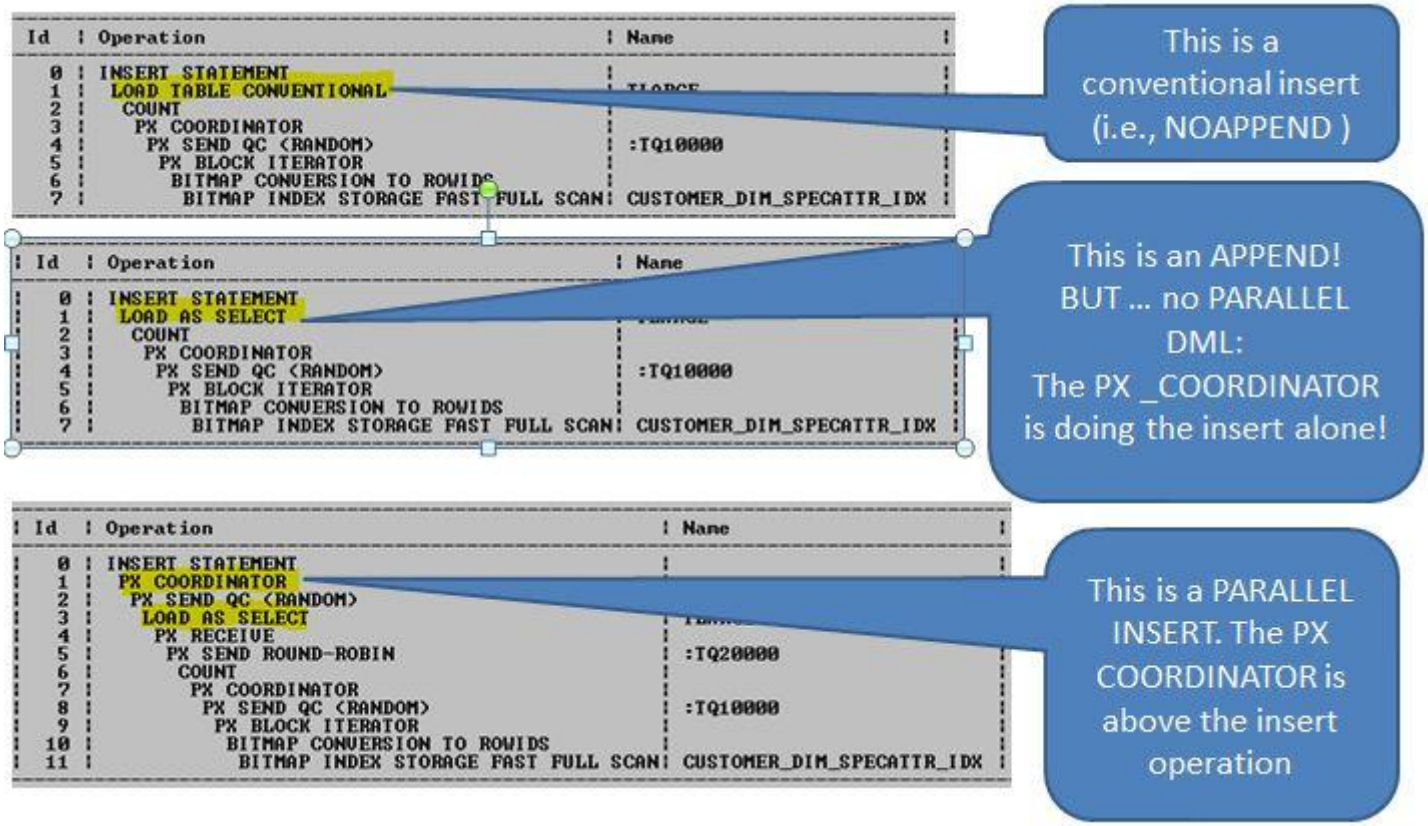


Figure 4. The differences in the execution plan between a conventional insert, a direct path insert and a parallel insert.

```
ALTER SESSION { ENABLE | FORCE } PARALLEL DML;
```

Note that very often we see execution plans of INSERT INTO SELECT statements, where the SELECT part is run in parallel (parallel query) but the INSERT is executed serially. This results in a performance problem for the INSERT operation. It is very important for the DW support engineer to be able to identify from the execution plan, whether a direct path write is happening or a conventional insert, as well as whether parallel DML has kicked in. In Figure 4, we depict three different execution plans of the same INSERT INTO SELECT statement. In the first one, from the LOAD TABLE CONVENTIONAL operation we can infer that a conventional insert takes place. In the second plan, we see the LOAD AS SELECT operation, which denotes a direct path insert. Note however that a serial insert is taking place. This can be easily detected from the fact that the

PX COORDINATOR line is below the LOAD AS SELECT OPERATION. This is a clear indication that the insert operation is executed by a single session, that of the parallel query coordinator. In the third plan, we have enabled parallel DML and thus we see the PX COORDINATOR line above the LOAD AS SELECT. This is the picture of a parallel insert and is important for us to remember it.

Efficient Distributed Queries

One of the most typical query patterns for a data warehouse is to "select" data from a remote site (i.e., a data source) and insert them into a local staging table. Also, in the majority of the cases, this statement will include some joins to local tables as well as some filter predicates on both local and remote tables. Let's say that a typical query pattern, in its most basic form will be something like the following:

```
INSERT INTO <local staging table>
SELECT <columns of interest>
FROM   <remote_table>@<remote_site> t_remote
       <local_table> t_local
WHERE
       <join condition between t_remote and t_local>
       <filter predicates on t_local>
       <filter predicates on t_remote>
```

Very often, the table at the remote site has a significant number of rows. For example, a very common scenario is that the remote table is a large table containing transaction data (e.g., order lines, Call Detail Records, invoice lines etc.) with a timestamp. The goal of the distributed statement is to select only the rows that correspond to a specific time period (e.g., the transactions of the last day, a.k.a. "the daily delta") and insert them into a local staging table. In order to achieve this, we maintain locally (i.e., at the data warehouse site) a very small "control table" that holds the current data warehouse "running date" (i.e., the date of interest for transaction data).

Therefore conceptually, we have a join of the very small (local) control table to the very large (remote) transactional table, and this join yields a relatively small amount of rows (i.e., the daily delta of transactional data, which is small, compared to the whole transactional table). Now, what is the most efficient method to execute this distributed query?

Let's examine first, what is the *wrong* way to execute it. To be honest, this is the method that we see so often happening in real life, and which is a very common reason for delays in the extraction phase of ETL flows. In order to understand the concept let's leave aside for a moment the INSERT part and just focus on the SELECT, which consists of a join of the (small) local table to the (large) remote table in order to get only the delta as an output.

Since the remote table is the large one, is not very difficult to infer that the “wrong way” is to execute this join *locally*, i.e., at the data warehouse site. No matter what join method will be chosen by the optimizer (e.g., NESTED LOOPS, or HASH JOIN), the end result of a locally processed join will be a very inefficient execution plan. This is simply because a local execution will either result into the travel over the network of a large data set, or into a significant amount of roundtrips caused by the consecutive probing of the large table (one probe for each row of the small table).

In Listing 12, we can see the execution of this distributed join locally, with a HASH JOIN method. Observe the “Remote SQL Information” section, which denotes the query executed at the remote site. We can see that the whole table at the remote site travels through the network *unrestricted*.

In Listing 13, we can see the same join executed locally, with a NESTED LOOPS method. In this case, for each row of the driving table (at the local site) we execute at the remote site the statement seen in the “Remote SQL Information” section. In other words, for each row of the local table we probe the large table at the remote site with a specific predicate. The bind variable denotes the predicate that we send for each such probe. This might lead to a significant amount of roundtrips.

If the (local) driving table returns only a single row (e.g. the reference date, in order to get the corresponding delta of transactions from the remote side), then this execution plan could be acceptable; since there will be only one row from the driving table (i.e. a single roundtrip) and the data traveled over the network will be only the daily delta of transactions. However, it is very important that this date restriction is imposed on the remote site, otherwise the whole table (and not just the delta) will travel over the network. This “unfortunate event” can happen easily (and we have seen this in practice many times), if we use a function to get the reference date.

In Listing 14, we have simulated this scenario by using a function to express the local column in the join predicate. Observe how different is now the plan from the one in Listing 13. Note the FILTER operation that has shown up as operation with id 3. This operation applies locally the filter that should have been executed remotely. If you check the “Remote SQL Information” section, you will see that no restriction is imposed on the large table at the remote site. The consequence of the function in the join predicate, has resulted in the worst-case scenario, since now we are probing the remote large table many times (due to the NESTED LOOPS way of working) and with each such probe, the whole large table is send over the network. This case comes from a true story, where some developer has decided to use a function in the join predicate, in order to get the DW reference date, instead of using a simple column.

We have discussed the consequences of executing the distributed join locally (i.e., at the DW side). Clearly, the most efficient plan is the one, where the small local table is sent over to the remote site and the join is executed remotely (i.e., at the data source and not at the data warehouse). This can be easily achieved with the use of the *DRIVING_SITE hint*, with which we instruct the optimizer at what site we want the execution to take place. This is depicted in Listing 15.

From the “Remote SQL Information” section we can see that the remote site is the data warehouse and the remote query is a SELECT on the small table. This is the most efficient execution plan for our distributed query.

Unfortunately, if the same distributed query is included inside an INSERT INTO SELECT statement, or a CTAS statement (Create Table As Select) then the optimizer, for no obvious reason, chooses to ignore the DRIVING_SITE hint and execute the SELECT part locally. This limits the usefulness of the DRIVING_SITE hint only to distributed queries. However, there is a workaround to this limitation (according to (Lewis, 2015)), by wrapping the SELECT part inside a pipelined table function. Other options are, to execute the staging of the delta at the remote site and then send that over to the DW site. Nevertheless, even if the join is executed locally, you

have to guarantee that a filter predicate is applied at the remote site, as in Listing 13, so as a restricted set of rows travels over the network.

```
select /*+ leading(lcl) use_hash(rmt)  */ *
from tsmall_local lcl, tlarge@exadwhprd rmt
where
       lcl.id = rmt.id

Plan hash value: 1246216434

-------------------------------------------------------------------------------------------------------------------------------
| Id  | Operation          | Name         | E-Rows |E-Bytes| Cost (%CPU)| E-Time   | Inst   |IN-OUT|  OMem |  1Mem | Used-Mem |
-------------------------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |              |        |       | 29296 (100)|          |        |      |    |          |          |
|*  1 |  HASH JOIN         |              |     99 |   389K| 29296   (1)| 00:05:52 |        |      |   732K|   732K| 1226K (0)|
|   2 |   TABLE ACCESS FULL| TSMALL_LOCAL |     99 |   194K|    16   (0)| 00:00:01 |        |      |    |          |          |
|   3 |   REMOTE           | TLARGE       |  10000 |    19M| 29279   (1)| 00:05:52 | EXADW~ | R->S |    |          |          |
-------------------------------------------------------------------------------------------------------------------------------

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

   1 - access("LCL"."ID"="RMT"."ID")

Remote SQL Information (identified by operation id):
----------------------------------------------------

   3 - SELECT /*+ USE_HASH ("RMT") */ "ID","DESCR" FROM "TLARGE" "RMT" (accessing 'EXADWHPRD' )
```

Listing 12. The execution of the distributed join locally causes the whole table at the remote site to travel through the network *unrestricted*.

```
select *
from tsmall_local lcl, tlarge@exadwhprd rmt
where
       lcl.id = rmt.id

-------------------------------------------------------------------------------------------------------
| Id  | Operation          | Name         | E-Rows |E-Bytes| Cost (%CPU)| E-Time   | Inst   |IN-OUT|
-------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |              |        |       |   920 (100)|          |        |      |
|   1 |  NESTED LOOPS      |              |     99 |   389K|   920   (2)| 00:00:12 |        |      |
|   2 |   TABLE ACCESS FULL| TSMALL_LOCAL |     99 |   194K|    16   (0)| 00:00:01 |        |      |
|   3 |   REMOTE           | TLARGE       |      1 |  2015 |     9   (0)| 00:00:01 | EXADW~ | R->S |
-------------------------------------------------------------------------------------------------------

Remote SQL Information (identified by operation id):
----------------------------------------------------

   3 - SELECT "ID","DESCR" FROM "TLARGE" "RMT" WHERE :1="ID" (accessing 'EXADWHPRD' )
```

Listing 13. The execution of the distributed join locally with a NESTED LOOPS might result into a significant amount of roundtrips.

```
select /*+ leading(lcl) use_nl(rmt) */ *
from tsmall_local lcl, tlarge@loopback rmt
where
       myfunc(lcl.id) = rmt.id

-------------------------------------------------------------------------------------------------------
| Id  | Operation          | Name         | Rows  | Bytes | Cost (%CPU)| Time     | Inst   |IN-OUT|
-------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |              |  4171 |    16M| 89666   (2)| 00:17:56 |        |      |
|   1 |  NESTED LOOPS      |              |  4171 |    16M| 89666   (2)| 00:17:56 |        |      |
|   2 |   TABLE ACCESS FULL| TSMALL_LOCAL |    99 |   195K|    17   (0)| 00:00:01 |        |      |
|*  3 |   FILTER           |              |    42 | 84630 |   900   (1)| 00:00:11 |        |      |
|   4 |    REMOTE          | TLARGE       |       |       |            |          | LOOPB~ | R->S |
-------------------------------------------------------------------------------------------------------

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

   3 - filter("RMT"."ID"="MYFUNC"("LCL"."ID"))

Remote SQL Information (identified by operation id):
----------------------------------------------------

   4 - SELECT /*+ USE_NL ("RMT") */ "ID","DESCR" FROM "TLARGE" "RMT" (accessing 'LOOPBACK')
```

Listing 14. The presence of a function in the join predicate has resulted into a remote query with no restrictions.

```
select /*+ driving_site(rmt)    */ *
from tsmall_local lcl, tlarge@exadwhprd rmt
where
       lcl.id = rmt.id

-------------------------------------------------------------------------------------------------------------------------
| Id  | Operation                          | Name         | Rows  | Bytes | Cost (%CPU)| Time     | TQ/Ins |IN-OUT| PQ Distrib |
-------------------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT REMOTE            |              |  4411 |    17M|  9055   (1)| 00:00:01 |        |      |            |
|   1 |  PX COORDINATOR                    |              |       |       |            |          |        |      |            |
|   2 |   PX SEND QC (RANDOM)              | :TQ10002     |  4411 |    17M|  9055   (1)| 00:00:01 |  Q1,02 | P->S | QC (RAND)  |
|*  3 |    HASH JOIN BUFFERED              |              |  4411 |    17M|  9055   (1)| 00:00:01 |  Q1,02 | PCWP |            |
|   4 |     BUFFER SORT                    |              |       |       |            |          |  Q1,02 | PCWC |            |
|   5 |      PX RECEIVE                    |              |  4411 |  8679K|    11   (0)| 00:00:01 |  Q1,02 | PCWP |            |
|   6 |       PX SEND HASH                 | :TQ10000     |  4411 |  8679K|    11   (0)| 00:00:01 | DWHPRD | S->P | HASH       |
|   7 |        REMOTE                      | TSMALL_LOCAL |  4411 |  8679K|    11   (0)| 00:00:01 |      ! | R->S |            |
|   8 |     PX RECEIVE                     |              | 10000 |    19M|  9043   (0)| 00:00:01 |  Q1,02 | PCWP |            |
|   9 |      PX SEND HASH                  | :TQ10001     | 10000 |    19M|  9043   (0)| 00:00:01 |  Q1,01 | P->P | HASH       |
|  10 |       PX BLOCK ITERATOR            |              | 10000 |    19M|  9043   (0)| 00:00:01 |  Q1,01 | PCWC |            |
|  11 |        TABLE ACCESS STORAGE FULL   | TLARGE       | 10000 |    19M|  9043   (0)| 00:00:01 |  Q1,01 | PCWP |            |
-------------------------------------------------------------------------------------------------------------------------

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

   3 - access("A2"."ID"="A1"."ID")

Remote SQL Information (identified by operation id):
----------------------------------------------------

   7 - SELECT "ID","DESCR" FROM "TSMALL_LOCAL" "A2" (accessing '!' )
```

Listing 15. With the use of the DRIVING_SITE hint the join is executed at the remote site.

Conclusion

In this article, we have provided a description of the most noteworthy methods for SQL tuning a DW running task with a performance problem. Our perspective was that of a DW support engineer, who is called at night, in order to solve a performance issue in a production Data Warehouse. Therefore, our primary goal was to help the DW support engineer solve the performance problem of a specific DW task and manage to keep the SLA with the business.

To this end we have distinguished between two approaches for SQL Tuning a problematic DW task: a) the "blindfold" one, called the "Black Box approach" because it is trying to solve the problem without losing too much time identifying the root-cause, and b) the "White-Box approach", which is based on the identification of the problem in the execution plan and attacking this.

The methods presented are not DW-specific and thus can used for SQL tuning any type of workload (DW-like or OLTP-like). Moreover, the methods presented, especially for the White Box approach can be very useful, apart from DW support engineers, also to developers and DBAs in order to implement more efficient SQL statements.

In the future, we are planning to update this article with new features of the Oracle database available in 12c version, such as the adaptive query optimization and the in-memory option.

References

(Antognini, 2014)	Troubleshooting Oracle Performance, 2nd Edition, Christian Antognini, Apress, 2014.
(Hailey, 2013)	SQL Tuning Methodology, Kyle Hailey, Kscope 2013. Available at: http://www.slideshare.net/khailey/kscope-2013-vst
(Hasler, 2014)	Expert Oracle SQL, Tony Hasler, Apress 2014.
(Lewis, 2015)	Five Hints, Jonathan Lewis 2015. Available at: https://jonathanlewis.wordpress.com/2015/12/03/five-hints/
(Karagiannidis1, 2015)	Github repository with SQL tuning scripts. Nikos Karagiannidis. Available at: https://github.com/nkarag/sqltuning4DWsupport
(Karagiannidis2, 2015)	How to effectively tune a query that does not even finish (SQL Monitoring and ASH in action). Nikos Karagiannidis. Available at: http://oradwstories.blogspot.gr/2015/01/how-to-effectively-tune-query-that-does.html
(Karagiannidis3, 2015)	Join hints and join ordering, or, why is the optimizer ignoring my join hint? Nikos Karagiannidis. Available at: http://oradwstories.blogspot.gr/2015/03/join-hints-and-join ordering-or-why-is.html
(Karagiannidis4, 2013)	Parallel hint with no degree specified. Nikos Karagiannidis. Available at: http://oradwstories.blogspot.gr/2013/04/parallel-hint-with-no-degree-specified.html
(Karagiannidis5, 2015)	Right-Deep Join Trees and Star Schema Queries, Nikos Karagiannidis. Available at: http://oradwstories.blogspot.gr/2015/07/right-deep-join-trees-and-star-schema.html
(Karagiannidis6, 2015)	Detecting a change in the execution plan of a query, Nikos Karagiannidis. Available at: http://oradwstories.blogspot.gr/2015/02/detecting-change-in-execution-plan-of.html
(Karagiannidis7, 2013)	Find all available Execution Plans for a specific SQL statement, Nikos Karagiannidis. Available at: http://oradwstories.blogspot.gr/2013/08/find-all-available-execution-plans-for.html
(Karagiannidis8, 2014)	SQL Plan Management / SQL Plan Baselines Material, Nikos Karagiannidis. Available at: http://oradwstories.blogspot.gr/2014/07/sql-plan-management-sql-plan-baselines.html
(Karagiannidis9, 2015)	Using SQL Plan Baselines to Revert to an Older Plan, Nikos Karagiannidis. Available at: http://oradwstories.blogspot.gr/2015/03/dealing-fast-with-change-of-execution.html
(Karagiannidis10, 2013)	Using SQL Profiles for Forcing a Specific Execution Plan, Nikos Karagiannidis. Available at: http://oradwstories.blogspot.gr/2013/08/using-sql-profiles-for-forcing-specific.html

(Nyffenegger, 2008)	On forcing a nested loop join instead of a hash join, René Nyffenegger, 2008.Available at: http://www.adp-gmbh.ch/blog/2008/01/17.php
(Oracle1, 2015)	Real Time SQL Monitoring, Oracle Database Performance Tuning Guide (11g), Oracle Corporation, http://docs.oracle.com/cd/B28359_01/server.111/b28274/instance_tune.htm#CACGEEIF
(Oracle2, 2015)	Hints, Oracle Database SQL Language Reference (11g), Oracle Corporation, http://docs.oracle.com/cd/E11882_01/server.112/e41084/sql_elements006.htm#SQLRF51101
(Oracle3, 2015)	Automatic SQL Tuning, Oracle Database Performance Tuning Guide (11g), Oracle Corporation, https://docs.oracle.com/cd/E11882_01/server.112/e41573/sql_tune.htm#PFGRF028
(Savvinov, 2015)	Efficiency-based SQL tuning, Nikolay Savvinov. Available at: http://savvinov.com/2013/01/28/efficiency-based-sql-tuning/
(Tow, 2003)	SQL Tuning, Dan Tow, O'Reilly, 2003
(Wood, 2005)	"Sifting through the ASHes". Graham Wood. Oracle Coorporation, 2005. http://www.oracle.com/technetwork/database/manageability/ppt-active-session-history-129612.pdf