THE EXPERT'S VOICE® IN ORACLE

Oracle SQL Tuning with Oracle SQLTXPLAIN

THE FAST AND EASY WAY TO TUNE SQL LIKE A PRO

Stelios Charalambides

Apress®

Contents at a Glance

Introduction

This book is intended as a practical guide to an invaluable tool called SQLTXPLAIN, commonly known simply as SQLT. You may never have heard of it, but if you have anything to do with Oracle tuning, SQLT is one of the most useful tools you'll find. Best of all, it's freely available from Oracle. All you need to do is learn how to use it.

How This Book Came to Be Written

I've been a DBA for over twenty years. In that time, I dealt with many, many tuning problems yet it was only when I began to work for Oracle that I learned about SQLT. As a part of the tuning team at Oracle Support I used SQLT every day to solve customers' most complex tuning problems. I soon realized that my experience was not unique. Outside Oracle, few DBAs knew that SQLT existed. An even smaller number knew how to use it. Hence the need for this book.

Don't Buy This Book

If you're looking for a text on abstract tuning theory or on how to tune "raw" SQL. This book is about how to use SQLT to do Oracle SQL tuning. The approach used is entirely practical and uses numerous examples to show the SQLT tool in action.

Do Buy This Book

If you're a developer or a DBA and are involved with Oracle SQL tuning problems. No matter how complex your system or how many layers of technology there are between you and your data, getting your query to run efficiently is where the rubber meets the road. Whether you're a junior DBA, just starting your career, or an old hand who's seen it all before, this book is designed to show you something completely practical that will be useful in your day-to-day work.

An understanding of SQLT will radically improve your ability to solve tuning problems and will also give you an effective checklist to use against new code and old.

Tuning problems are among the most complex technical problems around. SQLTXPLAIN is a fantastic tool that will help you solve them. Prepare to be smitten.

CHAPTER 1

Introduction to SQLTXPLAIN

Welcome to the world of fast Oracle SQL tuning with SQLTXPLAIN, or SQLT as it is typically called. Never heard of SQLT? You're not alone. I'd never heard of it before I joined ORACLE, and I had been a DBA for more years than I care to mention. That's why I'm writing this book. SQLT is a fantastic tool because it helps you diagnose tuning problems quickly. What do I mean by that? I mean that in half a day, maximum, you can go from a slow SQL to having an understanding of why SQL is malfunctioning, and finally, to knowing how to fix the SQL.

Will SQLT fix your SQL? No. Fixing the SQL takes longer. Some tables are so large that it can take days to gather statistics. It may take a long time to set up the test environment and roll the fix to production. The important point is that in half a day working with SQLT will give you an explanation. You'll know why the SQL was slow, or you'll be able to explain why it can't go any faster.

You need to know about SQLT because it will make your life easier. But let me back up a little and tell you more about what SQLT is, how it came into existence, why you probably haven't heard of it, and why you should use it for your Oracle SQL tuning.

What Is SQLT?

SQLT is a set of packages and scripts that produces HTML-formatted reports, some SQL scripts and some text files. The entire collection of information is packaged in a zip file and often sent to Oracle Support, but you can look at these files yourself. There are just over a dozen packages and procedures (called "methods") in SQLT. These packages and procedures collect different information based on your circumstances. We'll talk about the packages suitable for a number of situations later.

What's the Story of SQLT?

They say that necessity is the mother of invention, and that was certainly the case with SQLT. Oracle support engineers handle a huge number of tuning problems on a daily basis; problem is, the old methods of linear analysis are just too slow. You need to see the big picture fast so you can zoom in on the detail and tell the customer what's wrong. As a result, Carlos Sierra, a support engineer at the time (now a member of the Oracle Center of Expertise—a team of experts within Oracle) created SQLT. The routines evolved over many visits to customer sites to a point where they can gather *all* the information required quickly and effectively. He then provided easy-to-use procedures for reporting on those problems.

Carlos Sierra, the genius of SQLT, now spends much of his time improving SQLT code and adapting the SQLT code to new versions of the RDBMS. He also assists Oracle Tuning Performance engineers with SQL tuning through the medium of SQLT.

Why Haven't You Heard of SQLT?

If it's so useful, why haven't you heard about SQLT? Oracle has tried to publicize SQLT to the DBA community, but still I get support calls and talk to DBAs who have never heard of SQLT—or if they have, they've never used it. This amazing tool is free to supported customers, so there's no cost involved. DBAs need to look at problematic SQL often, and SQLT is hands down the fastest way to fix a problem. The learning curve may be high, but it's nowhere near as high as the alternatives: interpreting raw 10046 trace files or 10053 trace files. Looking through tables of statistics to find the needle in the haystack, guessing about what might fix the problem and trying it out? No thanks. SQLT is like a cruise missile that travels across the world right to its target.

Perhaps DBAs are too busy to learn a tool, which is not even mentioned in the release notes for Oracle. It's not in the documentation set, it's not officially part of the product set either. It's just a tool, written by a talented support engineer, and it happens to be better than any other tool out there. Let me repeat. It's free.

It's also possible that some DBAs are so busy focusing on the obscure minutiae of tuning that they forget the real world of fixing SQL. Why talk about a package that's easy to use when you could be talking about esoteric hidden parameters for situations you'll never come across? SQLT is a very practical tool.

Whatever the reason, if you haven't used SQLT before, my mission in this book is to get you up and running as fast and with as little effort from you as possible. I promise you installing and using SQLT is easy. Just a few simple concepts, and you'll be ready to go in 30 minutes.

How Did I Learn About SQLT?

Like the rest of the DBA world (I've been a DBA for many years), I hadn't heard of SQLT until I joined Oracle. It was a revelation to me. Here was this tool that's existed for years, which was exactly what I needed many times in the past, although I'd never used it. Of course I had read many books on tuning in years past: for example, Cary Millsaps's classic *Optimizing Oracle Performance*, and of course *Cost-Based Oracle Fundamentals* by Jonathan Lewis.

The training course (which was two weeks in total) was so intense that it was described by at least two engineers as trying to drink water from a fire hydrant. Fear not! This book will make the job of learning to use SQLT much easier.

Now that I've used SQLT extensively in day-to-day tuning problems, I can't imagine managing without it. I want you to have the same ability. It won't take long. Stick with me until the end of the book, understand the examples, and then try and relate them to your own situation. You'll need a few basic concepts (which I'll cover later), and then you'll be ready to tackle your own tuning problems. Remember to use SQLT regularly even when you don't have a problem; this way you can learn to move around the main HTML file quickly to find what you need. Run a SQLT report against SQL that isn't a problem. You'll learn a lot. Stick with me on this amazing journey.

Getting Started with SQLT

Getting started with SQLT couldn't be easier. I've broken the process down into three easy steps.

1. Downloading SQLT
2. Installing SQLT
3. Running your first SQLT report

SQLT will work on many different platforms. Many of my examples will be based on Microsoft Windows, but Linux or Unix is just as easy to use, and there are almost no differences in the use of SQLT between the platforms. If there are, I'll make a note in the text.

How Do You Get a Copy of SQLT?

How do you download SQLT? It's simple and easy. I just did it to time myself. It took two minutes. Here are the steps to get the SQLT packages ready to go on your target machine:

1. Find a web browser and log in to My Oracle Support (`http://support.oracle.com`)
2. Go to the knowledge section and type "SQLT" in the search box. Note 215187.1 entitled "SQLT (SQLTXPLAIN) – Tool that helps to diagnose a SQL statement performing poorly [ID 215187.1]" should be at the top of the list.
3. Scroll to the bottom of the note and choose the version of SQLT suitable for your environment. There are currently versions suitable from 9i to 11 g.
4. Download the zip file (the version I downloaded was 2Mbytes).
5. Unzip the zip file.

You now have the SQLT programs available to you for installation onto any suitable database. You can download the zip file to a PC and then copy it to a server if needed.

How Do You Install SQLT?

So without further ado, let's install SQLT so we can do some tuning:

1. Download the SQLT zip file appropriate for your environment (see steps above).
2. Unzip the zip file to a suitable location.
3. Navigate to your "install" directory under the unzipped area (in my case it is `C:\Document and Settings\Stelios\Desktop\SQLT\sqlt\install`, your locations will be different).
4. Connect as `sys`, e.g., `sqlplus / as sysdba`
5. Make sure your database is running
6. Run the `sqcreate.sql` script.
7. Select the default for the first option. (We'll cover more details of the installation in Appendix A.)
8. Enter and confirm the password for SQLTXPLAIN (the owner of the SQLT packages).
9. Select the tablespace where the SQLTXPLAIN will keep its packages and data (in my case, USERS).
10. Select the temporary tablespace for the SQLTXPLAIN user (in my case, TEMP).
11. Then enter the `username` of the user in the database who will use SQLT packages to fix tuning problems. Typically this is the schema that runs the problematic SQL (in my case this is STELIOS).
12. Then enter "T", "D" or "N." This reflects your license level for the tuning and diagnostics packs. Most sites have both so you would enter "T", (this is also the default). My test system is on my PC (an evaluation platform with no production capability) so I would also enter "T". If you have the diagnostics pack, only enter "D"; and if you do not have these licenses, enter "N".

The last message you see is "SQCREATE completed. Installation completed successfully."

Running Your First SQLT Report

Now that SQLT is installed, it is ready to be used. Remember that installing the package is done as sys and that running the reports is done as the target user. Please also bear in mind that although I have used many examples from standard schemas available from the Oracle installation files, your platform and exact version of Oracle may well be different, so please don't expect your results to be exactly the same as mine. However, your results will be similar to mine, and the results you see in your environment should still make sense.

1. Now exit SQL and change your directory to `...\SQLT\run`. In my case this is `C:\Documents and Settings\Stelios\Desktop\SQLT\sqlt\run`. From here log in to SQLPLUS as the target user.

2. Then enter the following SQL (this is going to be the statement we will tune):

   ```
   SQL>select count(*) from dba_objects;
   ```

3. Then get the SQL_ID value from the following SQL

   ```
   SQL>select sql_id from v$sqlarea where sql_text like 'select count(*) from
   dba_objects%';
   ```

   In my case the SQL_ID was g4pkmrqrgxg3b.

4. Now we execute our first SQLT tool `sqltxtract` from the target schema (in this case STELIOS) with the following command:

   ```
   SQL>@sqltxtract g4pkmrqrgxg3b
   ```

5. Enter the password for SQLTXPLAIN (which you entered during the installation). The last message you will see if all goes well is "SQLTXTRACT completed".

6. Now create a `zip` directory under the `run` directory and copy the zip file created into the `zip` directory. Unzip it.

7. Finally from your favorite browser navigate to and open the file named `sqlt_s<nnnnn>_main.html`. The symbols "nnnnn" represent numbers created to make all SQLT reports unique on your machine. In my case the file is called `sqlt_s89906_main.html`

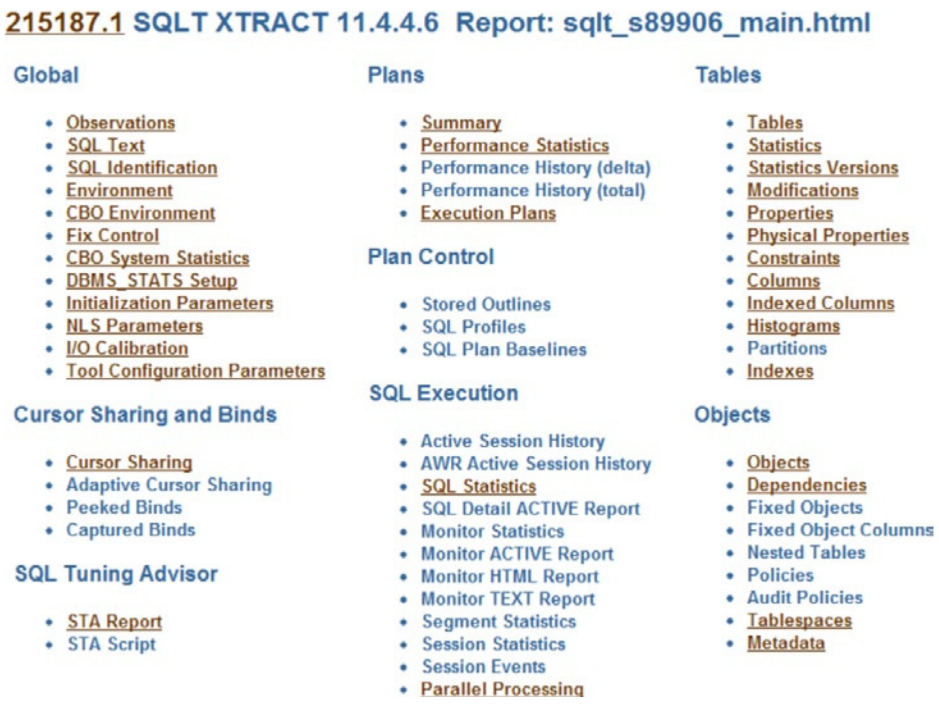
Congratulations! You have your first SQLT XTRACT report to look at.

When to Use SQLTXTRACT and When to Use SQLTXECUTE

SQLT XTRACT is the easiest report to create because it does not require the execution of the SQL at the time of the report generation. The report can be collected after the statement has been executed. SQLTXECUTE, on the other hand, executes the SQL statement and thus has better run-time information and access to the actual rows returned. This means it can assess the accuracy of the estimated cardinality of the steps in the execution plan (see "Cardinality and Selectivity" later in this chapter). SQLTXECUTE will get you more information, but it is not always possible to use this method, perhaps because you are in a production environment or perhaps the SQL statement is currently taking three days to run, which is why you are investigating this in the first place. We will look at both SQLTXECUTE and SQLTXTRACT report (and other SQLT options also). For now we will concentrate on one simple SQLTXTRACT report on a very simple SQL statement. So let's dive in.

Your First SQLT Report

Before we get too carried away with all the details of using the SQLT main report, just look at Figure 1-1. It's the beginning of a whole new SQLT tuning world. Are you excited? You should be. This header page is just the beginning. From here we will look at some basic navigation, just so you get an idea of what is available and how SQLT works, in terms of its navigation. Then we'll look at what SQLT is actually reporting about the SQL.



215187.1 SQLT XTRACT 11.4.4.6 Report: sqlt_s89906_main.html

Global

- Observations
- SQL Text
- SQL Identification
- Environment
- CBO Environment
- Fix Control
- CBO System Statistics
- DBMS_STATS Setup
- Initialization Parameters
- NLS Parameters
- I/O Calibration
- Tool Configuration Parameters

Cursor Sharing and Binds

- Cursor Sharing
- Adaptive Cursor Sharing
- Peeked Binds
- Captured Binds

SQL Tuning Advisor

- STA Report
- STA Script

Plans

- Summary
- Performance Statistics
- Performance History (delta)
- Performance History (total)
- Execution Plans

Plan Control

- Stored Outlines
- SQL Profiles
- SQL Plan Baselines

SQL Execution

- Active Session History
- AWR Active Session History
- SQL Statistics
- SQL Detail ACTIVE Report
- Monitor Statistics
- Monitor ACTIVE Report
- Monitor HTML Report
- Monitor TEXT Report
- Segment Statistics
- Session Statistics
- Session Events
- Parallel Processing

Tables

- Tables
- Statistics
- Statistics Versions
- Modifications
- Properties
- Physical Properties
- Constraints
- Columns
- Indexed Columns
- Histograms
- Partitions
- Indexes

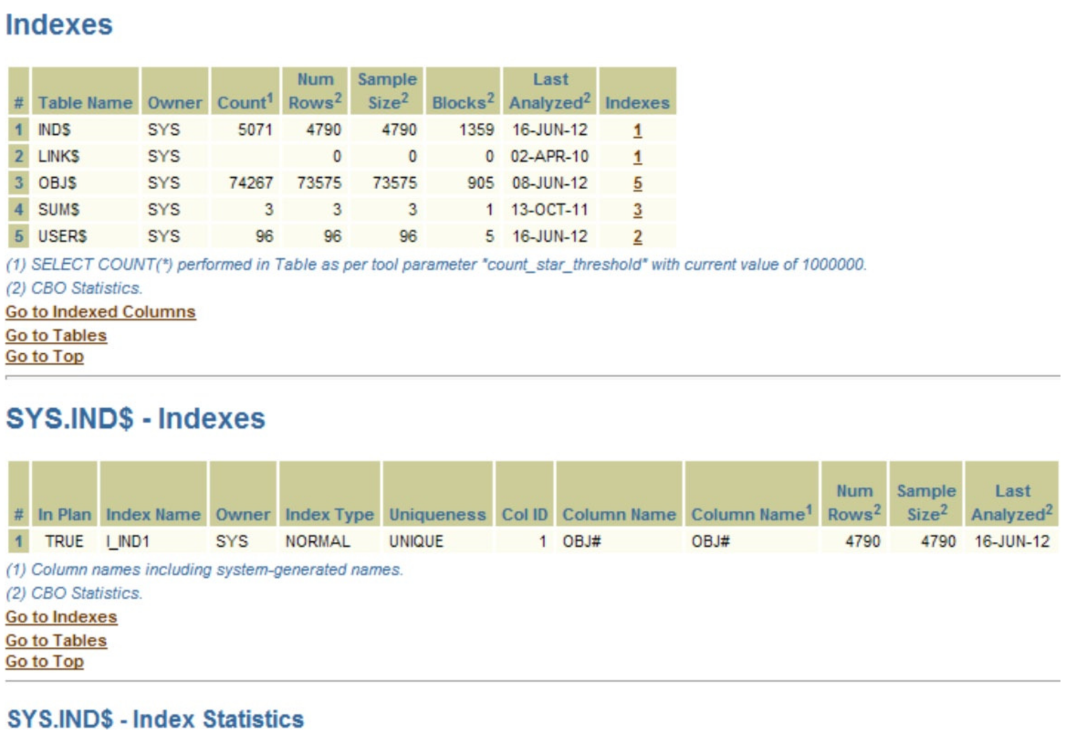
Objects

- Objects
- Dependencies
- Fixed Objects
- Fixed Object Columns
- Nested Tables
- Policies
- Audit Policies
- Tablespaces
- Metadata

Figure 1-1. *The top part of the SQLT report shows the links to many areas*

Some Simple Navigation

Let's start with the basics. Each hyperlinked section has a `Go to Top` hyperlink to get you back to the top. There's a lot of information in the various sections, and you can get lost. Other related hyperlinks will be grouped together above the `Go to Top` hyperlink. For example, if I clicked on `Indexes` (the last link under the `Tables` heading), I would see the page shown in Figure 1-2.



Indexes

#	Table Name	Owner	Count[1]	Num Rows[2]	Sample Size[2]	Blocks[2]	Last Analyzed[2]	Indexes
1	IND$	SYS	5071	4790	4790	1359	16-JUN-12	1
2	LINK$	SYS		0	0	0	02-APR-10	1
3	OBJ$	SYS	74267	73575	73575	905	08-JUN-12	5
4	SUM$	SYS	3	3	3	1	13-OCT-11	3
5	USER$	SYS	96	96	96	5	16-JUN-12	2

(1) SELECT COUNT(*) performed in Table as per tool parameter "count_star_threshold" with current value of 1000000.
(2) CBO Statistics.
Go to Indexed Columns
Go to Tables
Go to Top

SYS.IND$ - Indexes

#	In Plan	Index Name	Owner	Index Type	Uniqueness	Col ID	Column Name	Column Name[1]	Num Rows[2]	Sample Size[2]	Last Analyzed[2]
1	TRUE	I_IND1	SYS	NORMAL	UNIQUE	1	OBJ#	OBJ#	4790	4790	16-JUN-12

(1) Column names including system-generated names.
(2) CBO Statistics.
Go to Indexes
Go to Tables
Go to Top

SYS.IND$ - Index Statistics

Figure 1-2. *The Indexes section of the report*

Before we get lost in the SQLT report let's again look at the header page (Figure 1-1). The main sections cover all sorts of aspects of the system.

- CBO environment
- Cursor sharing
- Adaptive cursor sharing
- SQL Tuning Advisor (STA) report
- Execution plan(s) (there will be more than one plan if the plan changed)
- SQL*Profiles
- Outlines
- Execution statistics
- Table metadata
- Index metadata
- Column definitions
- Foreign keys

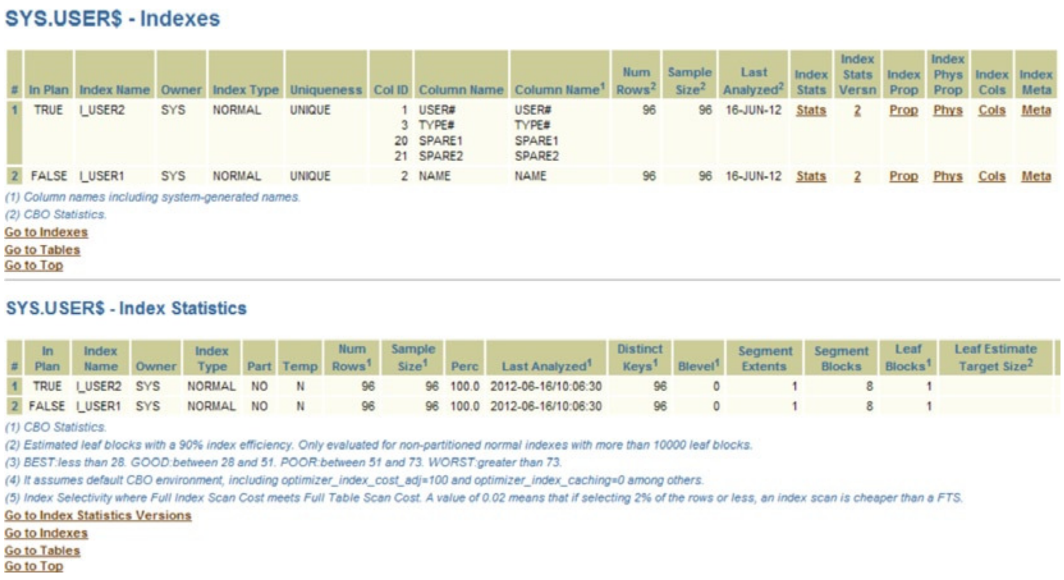
Take a minute and browse through the report.

Did you notice the hyperlinks on some of the data within the tables? SQLT collected all the information it could find and cross-referenced it all.

So for example, continuing as before from the main report at the top (Figure 1-1)

1. Click on `Indexes`, the last heading under `Tables`.
2. Under the `Indexes` column of the `Indexes` heading, the numbers are hyperlinked (see Figure 1-2). I clicked on 2 of the `USERS$` record.

   Now you can see the details of the columns in that table (see Figure 1-3). As an example here we see that the index `I_USER2` was used in the execution of my query (the `In Plan` column value is set to `TRUE`).



SYS.USER$ - Indexes

#	In Plan	Index Name	Owner	Index Type	Uniqueness	Col ID	Column Name	Column Name[1]	Num Rows[2]	Sample Size[2]	Last Analyzed[2]	Index Stats	Index Stats Versn	Index Prop	Index Phys Prop	Index Cols	Index Meta
1	TRUE	I_USER2	SYS	NORMAL	UNIQUE	1 3 20 21	USER# TYPE# SPARE1 SPARE2	USER# TYPE# SPARE1 SPARE2	96	96	16-JUN-12	Stats	2	Prop	Phys	Cols	Meta
2	FALSE	I_USER1	SYS	NORMAL	UNIQUE	2	NAME	NAME	96	96	16-JUN-12	Stats	2	Prop	Phys	Cols	Meta

(1) Column names including system-generated names.
(2) CBO Statistics.
Go to Indexes
Go to Tables
Go to Top

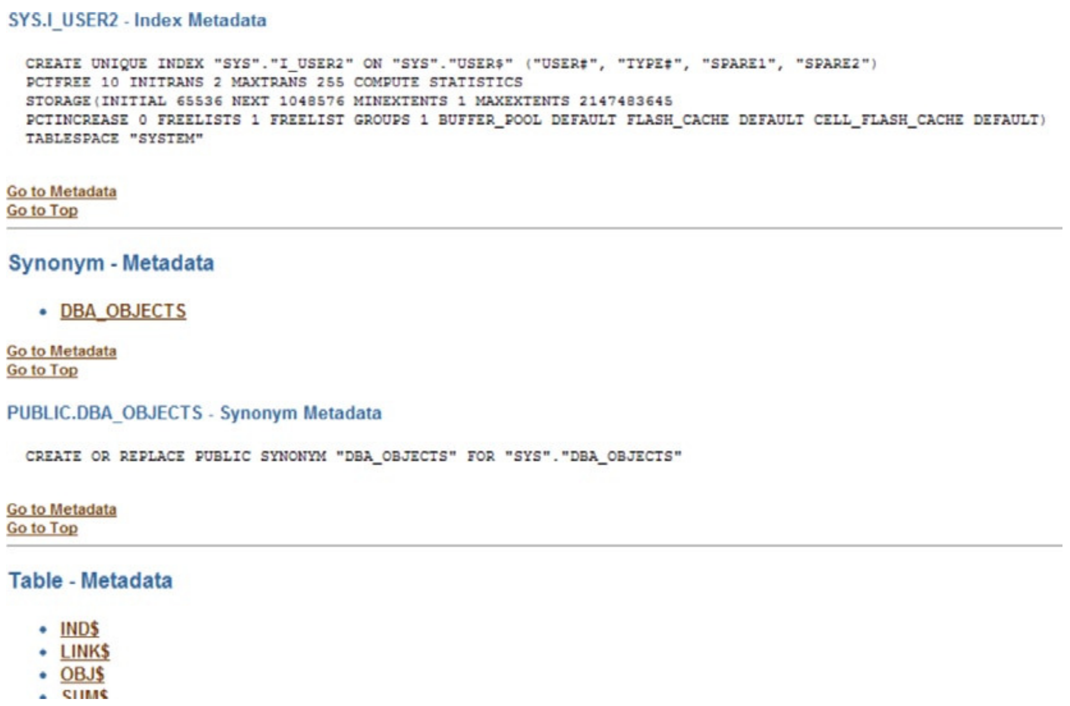
SYS.USER$ - Index Statistics

#	In Plan	Index Name	Owner	Index Type	Part	Temp	Num Rows[1]	Sample Size[1]	Perc	Last Analyzed[1]	Distinct Keys[1]	Blevel[1]	Segment Extents	Segment Blocks	Leaf Blocks[1]	Leaf Estimate Target Size[2]
1	TRUE	I_USER2	SYS	NORMAL	NO	N	96	96	100.0	2012-06-16/10:06:30	96	0	1	8	1	
2	FALSE	I_USER1	SYS	NORMAL	NO	N	96	96	100.0	2012-06-16/10:06:30	96	0	1	8	1	

(1) CBO Statistics.
(2) Estimated leaf blocks with a 90% index efficiency. Only evaluated for non-partitioned normal indexes with more than 10000 leaf blocks.
(3) BEST:less than 28. GOOD:between 28 and 51. POOR:between 51 and 73. WORST:greater than 73.
(4) It assumes default CBO environment, including optimizer_index_cost_adj=100 and optimizer_index_caching=0 among others.
(5) Index Selectivity where Full Index Scan Cost meets Full Table Scan Cost. A value of 0.02 means that if selecting 2% of the rows or less, an index scan is cheaper than a FTS.
Go to Index Statistics Versions
Go to Indexes
Go to Tables
Go to Top

***Figure 1-3.** An Index's detailed information about statistics*

3. Now, in the `Index Meta` column (far right in Figure 1-3), click on the `Meta` hyperlink for the `I_USER2` index to display the index metadata shown in Figure 1-4.


SYS.I_USER2 - Index Metadata

```
CREATE UNIQUE INDEX "SYS"."I_USER2" ON "SYS"."USER$" ("USER#", "TYPE#", "SPARE1", "SPARE2")
PCTFREE 10 INITRANS 2 MAXTRANS 255 COMPUTE STATISTICS
STORAGE(INITIAL 65536 NEXT 1048576 MINEXTENTS 1 MAXEXTENTS 2147483645
PCTINCREASE 0 FREELISTS 1 FREELIST GROUPS 1 BUFFER_POOL DEFAULT FLASH_CACHE DEFAULT CELL_FLASH_CACHE DEFAULT)
TABLESPACE "SYSTEM"
```

Go to Metadata
Go to Top

Synonym - Metadata

- DBA_OBJECTS

Go to Metadata
Go to Top

PUBLIC.DBA_OBJECTS - Synonym Metadata

```
CREATE OR REPLACE PUBLIC SYNONYM "DBA_OBJECTS" FOR "SYS"."DBA_OBJECTS"
```

Go to Metadata
Go to Top

Table - Metadata

- IND$
- LINK$
- OBJ$

Figure 1-4. *Metadata about an index can be seen from the "Meta" hyperlink*

Here we see the statement we would need to create this index. Do you have a script to do that? Well SQLT can get it better and faster. So now that you've seen a SQLT report, how do you approach a problem? You've opened the report, and you have one second to decide. Where do you go?

Well, that all depends.

How to Approach a SQLT Report

As with any methodology, different approaches are considered for different circumstances. Once you've decided there is something wrong with your SQL, you could use a SQLT report. Once you have the SQLT report, you are presented with a header page, which can take you to many different places (no one reads a SQLT report from start to finish in order). So where do you go from the main page?

If you're absolutely convinced that the execution plan is wrong, you might go straight to "Execution Plans" and look at the history of the execution plans. We'll deal with looking at those in detail later.

Suppose you think there is a general slowdown on the system. Then you might want to look at the "Observations" section of the report.

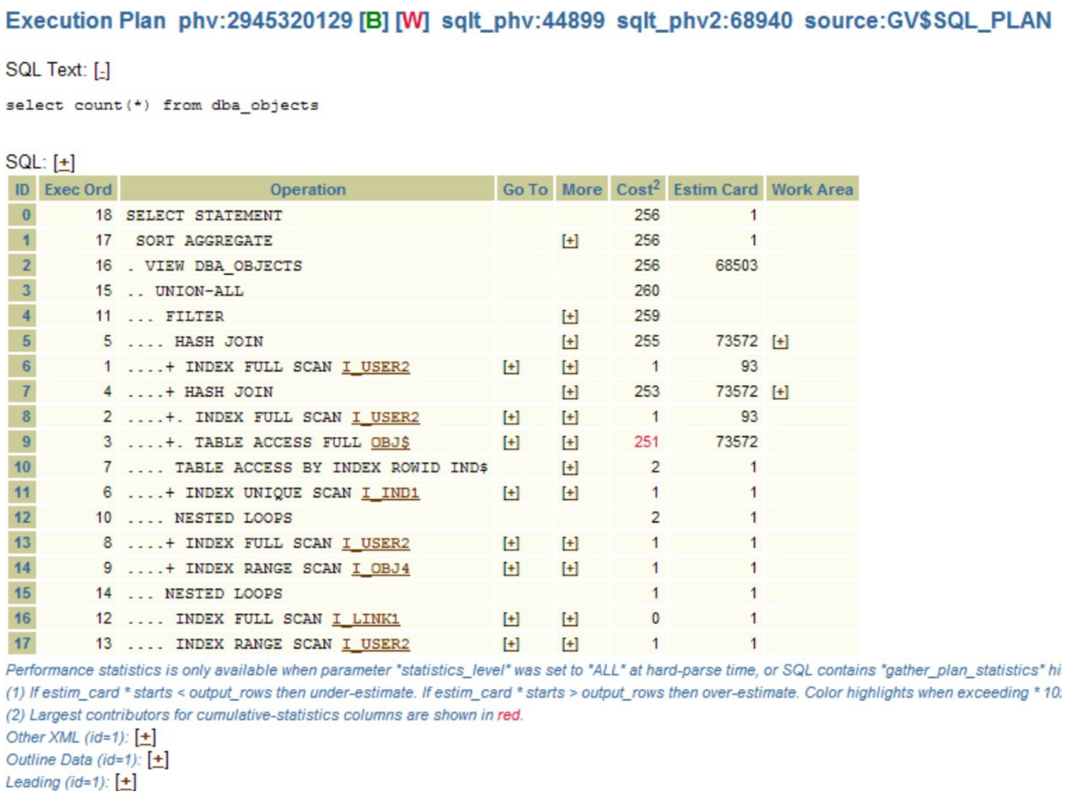
Maybe something happened to your statistics, so you'll certainly need to look at the "Statistics" section of the report under "Tables."

All of the sections I've mentioned above are sections you will probably refer to for every problem. The idea is to build up a picture of your SQL statement, understand the statistics related to the query, understand the cost-based optimizer (CBO) environment and try and get into its "head." Why did it do what it did? Why does it not relate to what you think it ought to do? The SQLT report is the explanation from the optimizer telling you why it decided to do what it did. Barring the odd bug, the CBO usually has a good reason for doing what it did. Your job is to set up the environment so that the CBO agrees with your worldview and run the SQL faster!

Cardinality and Selectivity

My objective throughout this book, apart from making you a super SQL tuner, is to avoid as much jargon as possible and explain tuning concepts as simply as possible. After all we're DBAs, not astrophysicists or rocket scientists.

So before explaining some of these terms it is important to understand why these concepts are key to the CBO operation and to your understanding of the SQL running on your system. Let's first look at cardinality. It is defined as the number of rows expected to be returned for a particular column if a predicate selects it. If there are no statistics for the table, then the number is pretty much based on heuristics about the number of rows, the minimum and maximum values, and the number of nulls. If you collect statistics then these statistics help to inform the guess, but it's still a guess. If you look at every single row of a table (collecting 100 percent statistics), it might still be a guess because the data might have changed, or the data may be skewed (we'll cover skewness later). That dry definition doesn't really relate to real life, so let's look at an example. Click on the "Execution Plans" hyperlink at the top of the SQLT report to display an execution plan like the one shown in Figure 1-5.


Execution Plan phv:2945320129 [B] [W] sqlt_phv:44899 sqlt_phv2:68940 source:GV$SQL_PLAN

SQL Text: [-]

```
select count(*) from dba_objects
```

SQL: [+]

ID	Exec Ord	Operation	Go To	More	Cost^2	Estim Card	Work Area
0	18	SELECT STATEMENT			256	1	
1	17	SORT AGGREGATE		[+]	256	1	
2	16	. VIEW DBA_OBJECTS			256	68503	
3	15	.. UNION-ALL			260		
4	11	... FILTER		[+]	259		
5	5	.... HASH JOIN		[+]	255	73572	[+]
6	1	....+ INDEX FULL SCAN I_USER2	[+]	[+]	1	93	
7	4	....+ HASH JOIN		[+]	253	73572	[+]
8	2	....+. INDEX FULL SCAN I_USER2	[+]	[+]	1	93	
9	3	....+. TABLE ACCESS FULL OBJ$	[+]	[+]	251	73572	
10	7	.... TABLE ACCESS BY INDEX ROWID IND$		[+]	2	1	
11	6	....+ INDEX UNIQUE SCAN I_IND1	[+]	[+]	1	1	
12	10	.... NESTED LOOPS			2	1	
13	8	....+ INDEX FULL SCAN I_USER2	[+]	[+]	1	1	
14	9	....+ INDEX RANGE SCAN I_OBJ4	[+]	[+]	1	1	
15	14	... NESTED LOOPS			1	1	
16	12	.... INDEX FULL SCAN I_LINK1	[+]	[+]	0	1	
17	13	.... INDEX RANGE SCAN I_USER2	[+]	[+]	1	1	

Performance statistics is only available when parameter "statistics_level" was set to "ALL" at hard-parse time, or SQL contains "gather_plan_statistics" hi
*(1) If estim_card * starts < output_rows then under-estimate. If estim_card * starts > output_rows then over-estimate. Color highlights when exceeding * 10:*
(2) Largest contributors for cumulative-statistics columns are shown in red.
Other XML (id=1): [+]
Outline Data (id=1): [+]
Leading (id=1): [+]

Figure 1-5. *An execution plan in the "Execution Plan" section*

In the "Execution Plan" section, you'll see the "Estim Card" column. In my example, look at the `TABLE ACCESS FULL OBJ$` step. Under the "Estim Card" column the value is 73,572. Remember cardinality is the number of rows returned from a step in an execution plan. The CBO (based on the table's statistics) will have an estimate for the cardinality. The "Estim Card" column then shows what the CBO expected to get from the step in the query. The 73,572 shows that the CBO *expected* to get 73,572 records from this step, but in fact got 73,235. So how good was the CBO's estimate for the cardinality (the number of rows returned for a step in an execution plan)? In our simple example we can do a very simple direct comparison by executing the query show below.

```
SQL> select count(*) from dba_objects;
  COUNT(*)
----------
     73235
SQL>
```

So cardinality is the actual number of rows that will be returned, but of course the optimizer can't know the answers in advance. It has to guess. This guess can be good or bad, based on statistics and skewness. Of course, histograms can help here.

For an example of selectivity, let's look at the page (see Figure 1-6) we get by selecting `Columns` from the `Tables` options on the main page (refer to Figure 1-1).

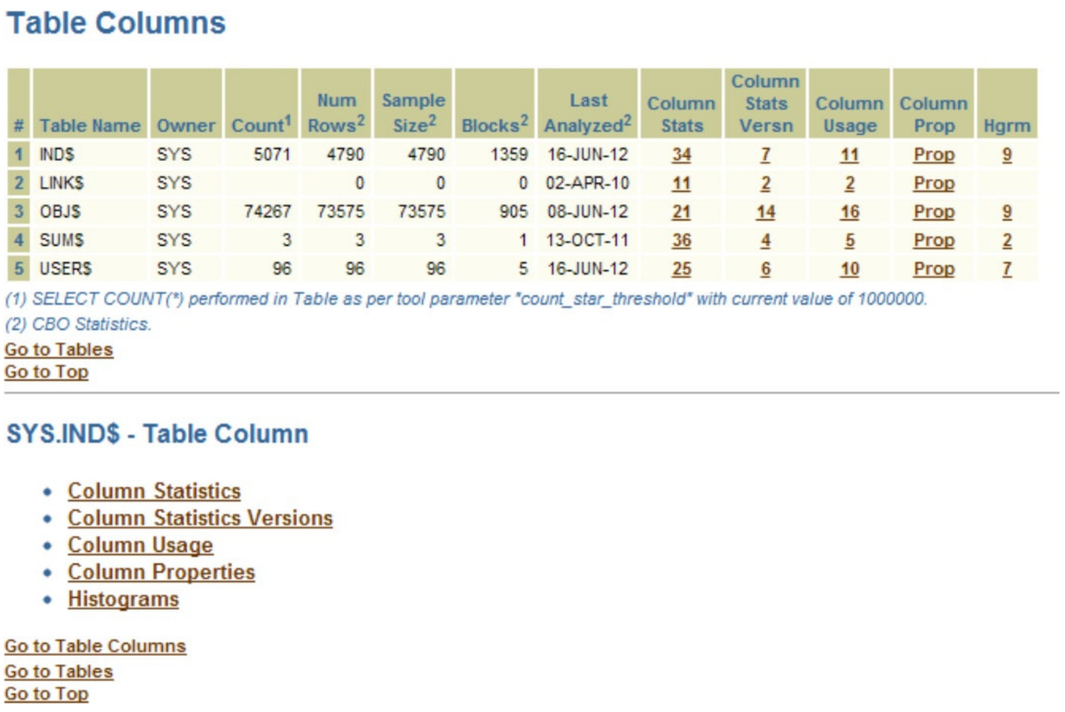


Table Columns

#	Table Name	Owner	Count[1]	Num Rows[2]	Sample Size[2]	Blocks[2]	Last Analyzed[2]	Column Stats	Column Stats Versn	Column Usage	Column Prop	Hgrm
1	IND$	SYS	5071	4790	4790	1359	16-JUN-12	34	7	11	Prop	9
2	LINK$	SYS		0	0	0	02-APR-10	11	2	2	Prop	
3	OBJ$	SYS	74267	73575	73575	905	08-JUN-12	21	14	16	Prop	9
4	SUM$	SYS	3	3	3	1	13-OCT-11	36	4	5	Prop	2
5	USER$	SYS	96	96	96	5	16-JUN-12	25	6	10	Prop	7

(1) SELECT COUNT(*) performed in Table as per tool parameter "count_star_threshold" with current value of 1000000.
(2) CBO Statistics.
Go to Tables
Go to Top

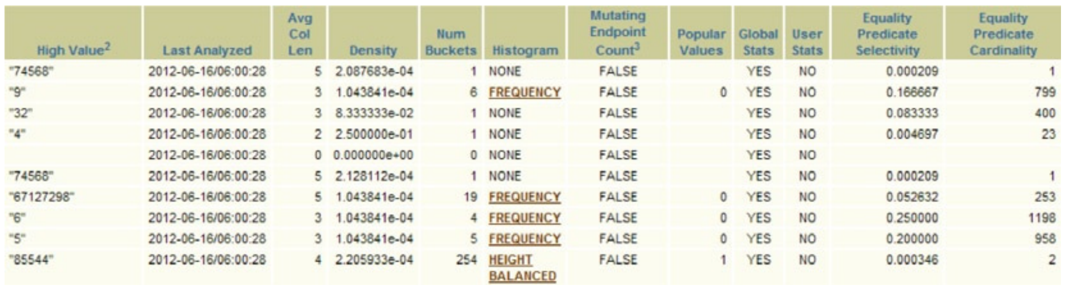
SYS.IND$ - Table Column

- Column Statistics
- Column Statistics Versions
- Column Usage
- Column Properties
- Histograms

Go to Table Columns
Go to Tables
Go to Top

Figure 1-6. *The "Table Column" section of the SQLT report*

Look at the "SYS.IND$ - Table Column" section. From the "Table Columns" page, if we click on the "34" under the "Column Stats" column, we will see the column statistics for the `SYS.IND$` index. Figure 1-7 shows a subset of the page from the "High Value" column to the "Equality Predicate Cardinality" column. Look at the "Equality Predicate Selectivity" and "Equality Predicate Cardinality" columns (the last two columns). Look at the values in the first row for `OBJ#`.



High Value[2]	Last Analyzed	Avg Col Len	Density	Num Buckets	Histogram	Mutating Endpoint Count[3]	Popular Values	Global Stats	User Stats	Equality Predicate Selectivity	Equality Predicate Cardinality
"74568"	2012-06-16/06:00:28	5	2.087683e-04	1	NONE	FALSE		YES	NO	0.000209	1
"9"	2012-06-16/06:00:28	3	1.043841e-04	6	FREQUENCY	FALSE	0	YES	NO	0.166667	799
"32"	2012-06-16/06:00:28	3	8.333333e-02	1	NONE	FALSE		YES	NO	0.083333	400
"4"	2012-06-16/06:00:28	2	2.500000e-01	1	NONE	FALSE		YES	NO	0.004697	23
	2012-06-16/06:00:28	0	0.000000e+00	0	NONE	FALSE		YES	NO		
"74568"	2012-06-16/06:00:28	5	2.128112e-04	1	NONE	FALSE		YES	NO	0.000209	1
"67127298"	2012-06-16/06:00:28	5	1.043841e-04	19	FREQUENCY	FALSE	0	YES	NO	0.052632	253
"6"	2012-06-16/06:00:28	3	1.043841e-04	4	FREQUENCY	FALSE	0	YES	NO	0.250000	1198
"5"	2012-06-16/06:00:28	3	1.043841e-04	5	FREQUENCY	FALSE	0	YES	NO	0.200000	958
"85544"	2012-06-16/06:00:28	4	2.205933e-04	254	HEIGHT BALANCED	FALSE	1	YES	NO	0.000346	2

Figure 1-7. *Selectivity is found in the "Equality Predicate Selectivity" column*

Selectivity is 0.000209, and cardinality is 1.

This translates to "I expect to get 1 row back for this equality predicate, which is equivalent to a 0.000209 chance (1 is certainty 0 is impossible) or in percentage terms I'll get 0.0209 percent of the entire table if I get the matching rows back."

Notice that as the cardinality increases the selectivity also increases. The selectivity only varies between 0 and 1 (or if you prefer 0 percent and 100 percent) and cardinality *should* only vary between 0 and the total number of rows in the table (excluding nulls). I say *should* because these values are based on statistics. What would happen if you gathered statistics on a partition (say it had 10 million rows) and then you truncate that partition, but don't tell the optimizer (i.e., you don't gather new statistics, or clear the old ones). If you ask the CBO to develop an execution plan in this case it might expect to get 10 million rows from a predicate against that partition. It might "think" that a full table scan would be a good plan. It might try to do the wrong thing because it had poor information.

To summarize, cardinality is the count of expected rows, and selectivity is the same thing but on a 0–1 scale. So why is all this important to the CBO and to the development of good execution plans? The short answer is that the CBO is working hard for you to develop the quickest and simplest way to get your results. If the CBO has some idea about how many rows will be returned for steps in the execution plan, then it can try variations in the execution plan and choose the plan with the least work and the fastest results. This leads into the concept of "cost," which we will cover in the next section.

What Is Cost?

Now that we have cardinality for an object we can work with other information derived from the system to calculate a cost for any operation. Other information from the system includes the following:

- Speed of the disks
- Speed of the CPU
- Number of CPUs
- Database block size

These metrics can be easily extracted from the system and are shown in the SQLT report also (under the "Environment" section). The amount of I/O and CPU resource used on the system for any particular step can now be calculated and thus used to derive a definite cost. This is the key concept for all tuning. The optimizer is always trying to reduce the cost for an operation. I won't go into details about how these costs are calculated because the exact values are not important. All you need to know is this: higher is worse, and worse can be based on higher cardinality (possibly based on out-of-date statistics), and if your disk I/O speeds are wrong (perhaps optimistically low) then full table scans might be favored when indexes are available. Cost can also be directly translated into elapsed time (on a quiet system), but that probably isn't what you need most of the time because you're almost always trying to get an execution time to be reduced, i.e., lower cost. As we'll see in the next section, you can get that information from SQLT. SQLT will also produce a 10053 trace file in some cases, so you can look at the details of how the cost calculations are made.

Reading the Execution Plan Section

We saw the execution plan section previously. It looks interesting, and it has a wobbly left edge and lots of hyperlinks. What does it all mean? This is a fairly simple execution plan, as it doesn't go on for pages and pages (like SIEBEL or PeopleSoft execution plans).

There are a number of simple steps to reading an execution plan. I'm sure there's more than one way of reading an execution plan, but this is the way I approach the problem. Bear in mind in these examples that if you are familiar with the pieces of SQL being examined, you may go directly to the section you think is wrong; but in general if you are seeing the execution plan for the first time, you will start by looking at a few key costs.

The first and most important cost is the overall cost of the entire query. This is always shown as "ID 0" and is always the first row in the execution plan. In our example shown in Figure 1-5, this is a cost of 256. So to get the cost for the entire query just look at the first row. This is also the last step to be executed ("Exec Ord" is 18). The execution order is not top to bottom, the Oracle engine will carry out the steps in the order shown by the value in the "Exec Ord" column. So if we followed the execution through, the Oracle engine would do the execution in this order:

1. `INDEX FULL SCAN I_USER2`
2. `INDEX FULL SCAN I_USER2`
3. `TABLE ACCESS FULL OBJ$`
4. `HASH JOIN`
5. `HASH JOIN`
6. `INDEX UNIQUE SCAN I_IND1`
7. `TABLE ACCESS BY INDEX ROWID IND$`
8. `INDEX FULL SCAN I_USERS2`
9. `INDEX RANGE SCAN I_OBJ4`
10. `NESTED LOOP`
11. `FILTER`
12. `INDEX FULL SCAN I_LINK1`
13. `INDEX RANGE SCAN I_USERS2`
14. `NESTED LOOPS`
15. `UNION-ALL`
16. `VIEW DBA_OBJECTS`
17. `SORT AGGREGATE`
18. `SELECT STATEMENT`

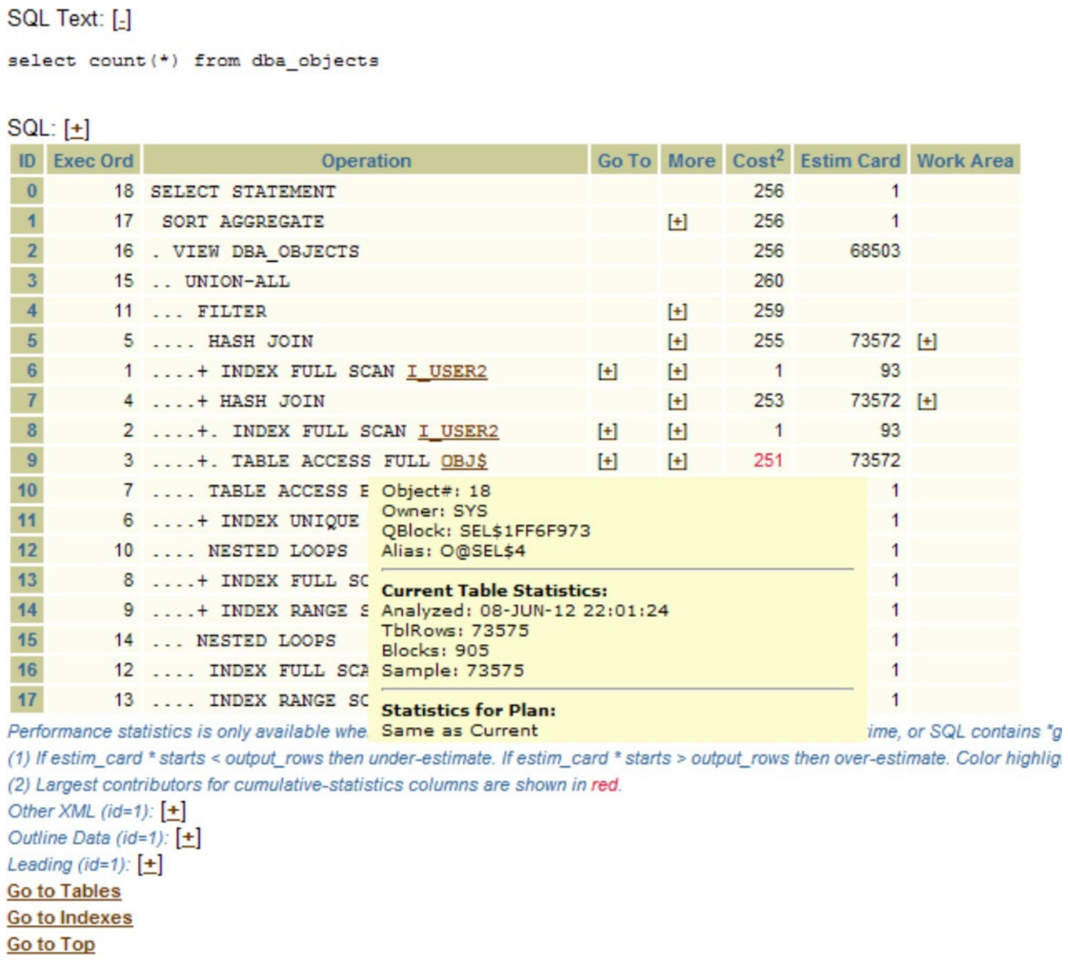
However, nobody ever represents the plan of a SQL statement like this. What is important to realize is that the wobbly left edge gives information about how the steps are carried out. The less-indented operations indicate outer operations that are being carried out on inner (more indented) operations. So for example steps 2, 3, and 4 would be read as "An index full scan is carried out using `I_USERS2`, then a full table scan of `OBJ$` and the results of these are `HASH JOINED` to produce a result set." Each operation produces results for a less-indented section until the final result is presented to the `SELECT (ID=0)`.

The “Operation” column is also marked with “+” and “–” to indicate sections of equal indentation. This is helpful in lining up operations to see which result sets an operation is working on. So, for example, it is important to realize that the HASH JOIN at step 5 is using results from steps 1, 4, 2, and 3. We’ll see more complex examples of these later. It is also important to realize that the costs shown are aggregate costs for each operation as well. So the cost shown on the first line is the cost for the entire operation, and we can also see that most of the cost of the entire operation came from step 3. (SQLT helpfully shows the highest cost operation in red). So let’s look at step 1 (as shown in Figure 1-5) in more detail. In our simple case this is

```
"INDEX FULL SCAN I_USER2"
```

Let’s translate the full line into English: “First get me a full index scan of index I_USERS2. I estimate 93 rows will be returned which, based on your current system statistics (Single block read time and multi-block read times and CPU speed), will be a cost of 1.”

The second and third steps are another INDEX FULL SCAN and a TABLE ACCESS FULL of OBJ$. This third step has a cost of 251. The total cost of the entire SQL statement is 256 (top row). So if were looking to tune this statement we know that the benefit must come from this third step (it is a cost of 251 out of a total cost of 256). Now place your cursor over the word “TABLE” on step 3 (see Figure 1-8).



SQL Text: [-]

```
select count(*) from dba_objects
```

SQL: [+]

ID	Exec Ord	Operation	Go To	More	Cost²	Estim Card	Work Area
0	18	SELECT STATEMENT			256	1	
1	17	SORT AGGREGATE		[+]	256	1	
2	16	. VIEW DBA_OBJECTS			256	68503	
3	15	.. UNION-ALL			260		
4	11	... FILTER		[+]	259		
5	5	.... HASH JOIN		[+]	255	73572	[+]
6	1	....+ INDEX FULL SCAN I_USER2	[+]	[+]	1	93	
7	4	....+ HASH JOIN		[+]	253	73572	[+]
8	2	....+. INDEX FULL SCAN I_USER2	[+]	[+]	1	93	
9	3	....+. TABLE ACCESS FULL OBJ$	[+]	[+]	251	73572	
10	7	.... TABLE ACCESS E				1	
11	6	....+ INDEX UNIQUE				1	
12	10	.... NESTED LOOPS				1	
13	8	....+ INDEX FULL SC				1	
14	9	....+ INDEX RANGE S				1	
15	14	... NESTED LOOPS				1	
16	12	.... INDEX FULL SCA				1	
17	13	.... INDEX RANGE SC				1	

```
Object#: 18
Owner: SYS
QBlock: SEL$1FF6F973
Alias: O@SEL$4

Current Table Statistics:
Analyzed: 08-JUN-12 22:01:24
TblRows: 73575
Blocks: 905
Sample: 73575

Statistics for Plan:
Same as Current
```

Performance statistics is only available whe ime, or SQL contains "g

(1) If estim_card * starts < output_rows then under-estimate. If estim_card * starts > output_rows then over-estimate. Color highlig

(2) Largest contributors for cumulative-statistics columns are shown in red.

Other XML (id=1): [+]

Outline Data (id=1): [+]

Leading (id=1): [+]

Go to Tables

Go to Indexes

Go to Top

Figure 1-8. *More details can be obtained by ‘hovering’ over links*

Notice how information is displayed about the object.

Object#: 18

Owner: SYS

Qblock: SEL$1FF6F973

Alias: O@SEL$4

Current Table Statistics*:*

Analyzed: 08-JUN-12 22:01:24

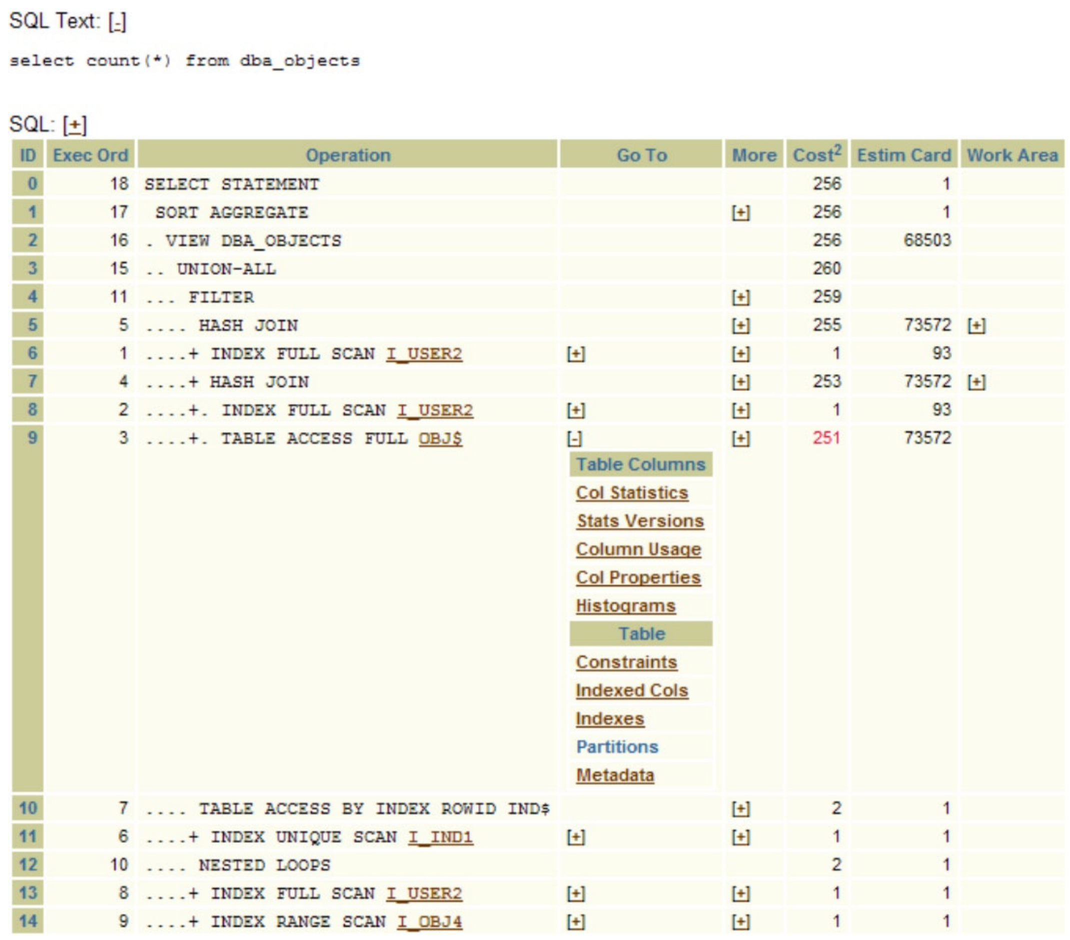
TblRows: 73575

Blocks: 905

Sample 73575

Just by hovering your mouse over the object you get its owner, the query block name, when it was last analyzed, and how big the object is.

Now let's look at the "Go To" column. Notice the "+" under that column? Click on the one for step 3, and you'll get a result like the one in Figure 1-9.



SQL Text: [-]

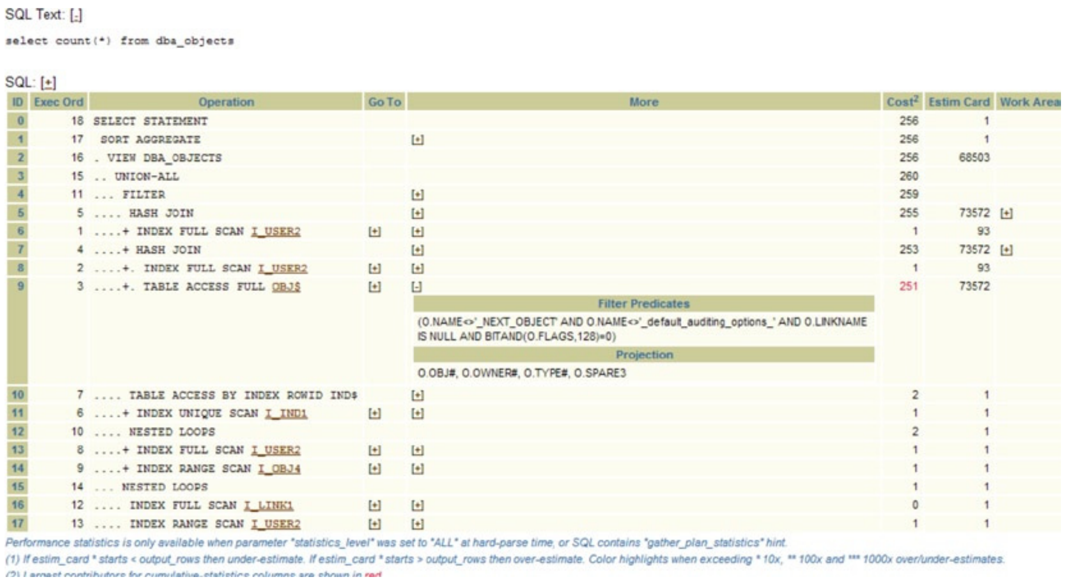
```
select count(*) from dba_objects
```

SQL: [+]

ID	Exec Ord	Operation	Go To	More	Cost^2	Estim Card	Work Area
0	18	SELECT STATEMENT			256	1	
1	17	SORT AGGREGATE		[+]	256	1	
2	16	. VIEW DBA_OBJECTS			256	68503	
3	15	.. UNION-ALL			260		
4	11	... FILTER		[+]	259		
5	5	.... HASH JOIN		[+]	255	73572	[+]
6	1	....+ INDEX FULL SCAN I_USER2	[+]	[+]	1	93	
7	4	....+ HASH JOIN		[+]	253	73572	[+]
8	2	....+. INDEX FULL SCAN I_USER2	[+]	[+]	1	93	
9	3	....+. TABLE ACCESS FULL OBJ$	[-] Table Columns Col Statistics Stats Versions Column Usage Col Properties Histograms Table Constraints Indexed Cols Indexes Partitions Metadata	[+]	251	73572	
10	7	.... TABLE ACCESS BY INDEX ROWID IND$		[+]	2	1	
11	6	....+ INDEX UNIQUE SCAN I_IND1	[+]	[+]	1	1	
12	10	.... NESTED LOOPS			2	1	
13	8	....+ INDEX FULL SCAN I_USER2	[+]	[+]	1	1	
14	9	....+ INDEX RANGE SCAN I_OBJ4	[+]	[+]	1	1	

***Figure 1-9.** More hyperlinks can be revealed by expanding sections on the execution plan*

So right from the execution plan you can go to the "Col Statistics" or the "Stats Versions" or many other things. You decide where you want to go next, based on what you've understood so far and on what you think is wrong with your execution plan. Now close that expanded area and click on the "+" under the "More" column for step 3 (see Figure 1-10)



SQL Text: [-]

```
select count(*) from dba_objects
```

SQL: [+]

ID	Exec Ord	Operation	Go To	More	Cost^2	Estim Card	Work Area
0	18	SELECT STATEMENT			256	1	
1	17	SORT AGGREGATE		[+]	256	1	
2	16	. VIEW DBA_OBJECTS			256	68503	
3	15	.. UNION-ALL			260		
4	11	... FILTER		[+]	259		
5	5	.... HASH JOIN		[+]	255	73572	[+]
6	1	....+ INDEX FULL SCAN I_USER2	[+]	[+]	1	93	
7	4	....+ HASH JOIN		[+]	253	73572	[+]
8	2	....+. INDEX FULL SCAN I_USER2	[+]	[+]	1	93	
9	3	....+. TABLE ACCESS FULL OBJ$	[+]	[-] Filter Predicates (O.NAME<>'_NEXT_OBJECT' AND O.NAME<>'_default_auditing_options_' AND O.LINKNAME IS NULL AND BITAND(O.FLAGS,128)=0) Projection O.OBJ#, O.OWNER#, O.TYPE#, O.SPARE3	251	73572	
10	7	.... TABLE ACCESS BY INDEX ROWID IND$		[+]	2	1	
11	6	....+ INDEX UNIQUE SCAN I_IND1	[+]	[+]	1	1	
12	10	.... NESTED LOOPS			2	1	
13	8	....+ INDEX FULL SCAN I_USER2	[+]	[+]	1	1	
14	9	....+ INDEX RANGE SCAN I_OBJ4	[+]	[+]	1	1	
15	14	... NESTED LOOPS			1	1	
16	12	.... INDEX FULL SCAN I_LINK1	[+]	[+]	0	1	
17	13	.... INDEX RANGE SCAN I_USER2	[+]	[+]	1	1	

Performance statistics is only available when parameter "statistics_level" was set to "ALL" at hard-parse time, or SQL contains "gather_plan_statistics" hint.
(1) If estim_card * starts < output_rows then under-estimate. If estim_card * starts > output_rows then over-estimate. Color highlights when exceeding * 10x, ** 100x and *** 1000x over/under-estimates.
(2) Largest contributors for cumulative-statistics columns are shown in red

Figure 1-10. *Here we see an expansion under the "More" heading*

Now we see the filter predicates and the projections. These can help you understand which line in the execution plan the optimizer is considering predicates for and which values are in play for filters.

Just above the first execution plan is a section called "Execution Plans." This lists all the different execution plans the Oracle engine has seen for this SQL. Because execution plans can be stored in multiple places in the system, you could well have multiple entries in the "Execution Plans" section of the report. Its source will be noted (under the "Source" column). Here is a list of sources I've come across:

- `GV$SQL_PLAN`
- `GV$SQLAREA_PLAN_HASH`
- `PLAN_TABLE`
- `DBA_SQLTUNE_PLANS`
- `DBA_HIST_SQL_PLAN`

SQLT will look for plans in as many places as possible so that it can you give you a full range of options. When SQLT gathers this information, it will look at the cost associated with each of these plans and label them with "W" in red (worst) and "B" in green (best). In my simple test case, the "Best" and "Worst" are the same, as there is only one execution plan in play. However you'll notice there are three records: one came from mining the memory `GV$SQL_PLAN`, one came from `PLAN_TABLE` (i.e., an `EXPLAIN PLAN`) and one came from `DBA_SQLTUNE_PLANS`, (SQL Tuning Analyzer) whose source is `DBA_SQLTUNE_PLANS`.

When you have many records here, perhaps a long history, you can go back and see which plans were best and try to see why they changed. Noting the timing of a change can sometimes be crucial, as it can help you zoom in on the change that made things worse.

Before we launch into even more detailed use of the "Execution Plans" section, we'll need more complex examples.

Join Methods

This book is focused on very practical tuning with SQLT. I try to avoid unnecessary concepts and tuning minutiae. For this reason I will not cover every join method available or every DBA table that might have some interesting information about performance or every hint. These are well documented in multiple sources, not least of which is the Oracle Performance guide (which I recommend you read). However, we need to cover some basic concepts to ensure we get the maximum benefit from using SQLT. So, for example, here are some simple joins. As its name implies, a join is a way of "joining" two data sets together: one might contain a person's name and age and another table might contain the person's name and income level. In which case you could "join" these tables to get the names of people of a particular age and income level. As the name of the operation implies, there must be something to join the two data sets together: in our case, it's the person's name. So what are some simple joins? (i.e., ones we'll see in out SQLT reports).

> HASH JOINS (HJ) – The *smaller* table is hashed and placed into memory. The *larger* table is then scanned for rows that match the hash value in memory. If the larger and smaller tables are the wrong way around this is inefficient. If the tables are not large, this is inefficient. If the smaller table does not fit in memory, then this is more than inefficient: it's really bad!
>
> NESTED LOOP (NL) – Nested Loop joins are better if the tables are smaller. Notice how in the execution plan examples above there is a HASH JOIN and a NESTED LOOP. Why was each chosen for the task? The details of each join method and its associated cost can be determined from the 10053 trace file. It is a common practice to promote the indexes and NL by adjusting the optimizer parameters `Optimizer_index_cost_adj` and `optimizer_index_caching` parameters. This is not generally a winning strategy. These parameters should be set to the defaults of 100 and 0. Work on getting the object and system statistics right first.
>
> CARTESIAN JOINS – Usually bad. Every row of the first table is used as a key to access every row of the second table. If you have a very few number of rows in the joining tables this join is OK. In most production environments, if you see this occurring then something is wrong, usually statistics.
>
> SORT MERGE JOINS (SMJ) – Generally joined in memory if memory allows. If the cardinality is high then you would expect to see SMJs and HJs.

Summary

In this chapter we covered the basics of using SQLTXTRACT. This is a simple method of SQLT that does not execute the SQL statement in question. It extracts the information required from all possible sources and presents this in a report.

In this chapter we looked at a simple download and install of SQLT. You've seen that installing SQLT on a local database can take very little time, and its use is very simple. The report produced was easy to unzip and can be used to investigate the SQL performance. In this first example we briefly mentioned cardinality and selectivity and how these affect the cost-based optimizer's plans. Now let's investigate more of SQLT's features and look at more complex examples.

CHAPTER 2

The Cost-Based Optimizer Environment

When I'm solving tricky tuning problems, I'm often reminded of the story of the alien who came to earth to try his hand at driving. He'd read all about it and knew the physics involved in the engine. It sounded like fun. He sat down in the driver's seat and turned the ignition; the engine ticked over nicely, and the electrics were on. He put his seatbelt on and tentatively pressed the accelerator pedal. Nothing happened. Ah! Maybe the handbrake was on. He released the handbrake and pressed the accelerator again. Nothing happened. Later, standing back from the car and wondering why he couldn't get it to go anywhere, he wondered why the roof was in contact with the road.

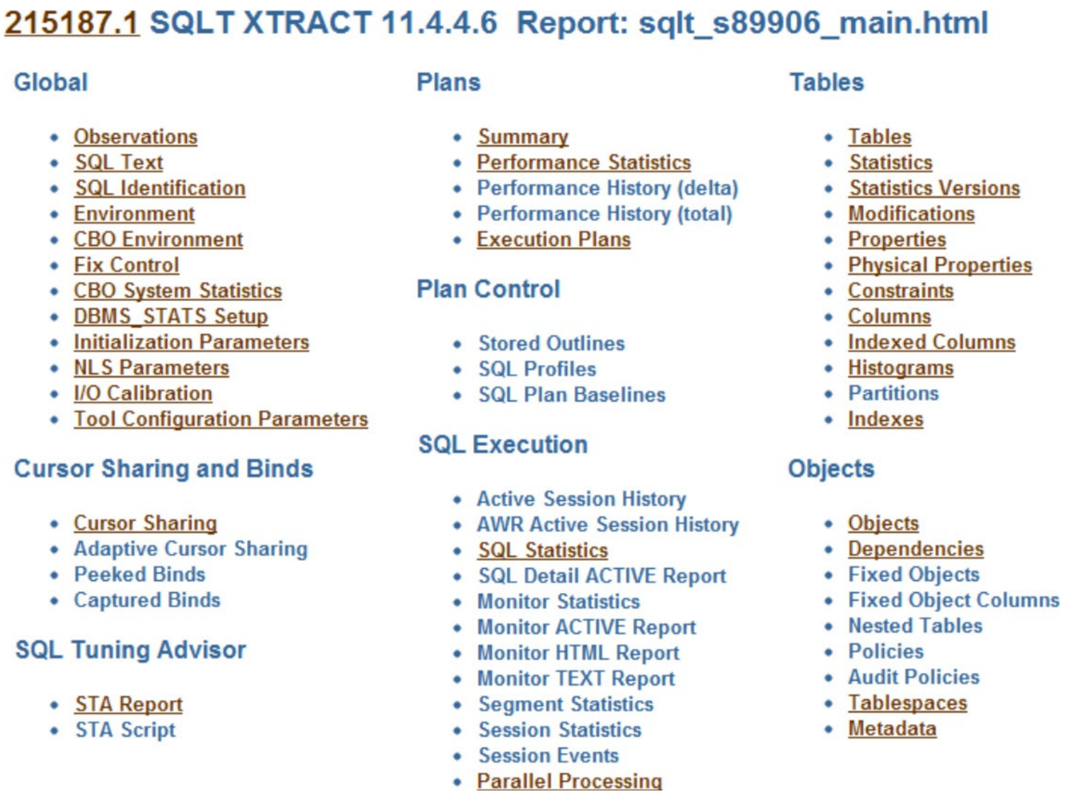
My rather strange analogy is trying to help point out that before you can tune something, you need to know what it should look like in broad terms. Is 200ms reasonable for a single block read time? Should system statistics be collected over a period of 1 minute? Should we be using hash joins for large table joins? There are 1,001 things that to the practiced eye look wrong, but to the optimizer it's just the truth.

Just like the alien, the Cost Based Optimizer (CBO) is working out how to get the best performance from your system. It knows some basic rules and guestimates (heuristics) but doesn't know about your particular system or data. You have to tell it about what you have. You have to tell the alien that the black round rubbery things need to be in contact with the road. If you 'lie' to the optimizer, then it could get the execution plan wrong, and by wrong I mean the plan will perform badly. There are rare cases where heuristics are used inappropriately or there are bugs in the code that lead the CBO to take shortcuts (Query transformations) that are inappropriate and give the wrong results. Apart from these, the optimizer generally delivers poor performance because it has poor information to start with. Give it good information, and you'll generally get good performance.

So how do you tell if the "environment" is right for your system? In this chapter we'll look at a number of aspects of this environment. We'll start with (often neglected) system statistics and then look at the database parameters that affect the CBO. We'll briefly touch on Siebel environments and the have a brief look at histograms (these are covered in more detail in the next chapter). Finally, we'll look at both overestimates and underestimates (one of SQLT's best features is highlighting these), and then we'll dive into a real life example, where you can play detective and look at examples to hone your tuning skills (no peeking at the answer). Without further ado let's start with system statistics.

System Statistics

System statistics are an often-neglected part of the cost-based optimizer environment. If no system statistics have been collected for a system then the SQLT section "Current System Statistics" will show nothing for a number of important parameters for the system. An example is shown in Figure 2-3. It will guess these values. But why should you care if these values are not supplied to the optimizer? Without these values the optimizer will apply its best guess for scaling the timings of a number of crucial operations. This will result in inappropriate indexes being used when a full table scan would do or vice versa. These settings are so important that in some dynamic environments where the workload is changing, for example from the daytime OLTP to a nighttime DW (Data Warehouse) environment, that different sets of system statistics should be loaded. In this section we'll look at why these settings affect the optimizer, how and when they should be collected, and what to look for in a SQLT report.


215187.1 SQLT XTRACT 11.4.4.6 Report: sqlt_s89906_main.html

Global

- Observations
- SQL Text
- SQL Identification
- Environment
- CBO Environment
- Fix Control
- CBO System Statistics
- DBMS_STATS Setup
- Initialization Parameters
- NLS Parameters
- I/O Calibration
- Tool Configuration Parameters

Cursor Sharing and Binds

- Cursor Sharing
- Adaptive Cursor Sharing
- Peeked Binds
- Captured Binds

SQL Tuning Advisor

- STA Report
- STA Script

Plans

- Summary
- Performance Statistics
- Performance History (delta)
- Performance History (total)
- Execution Plans

Plan Control

- Stored Outlines
- SQL Profiles
- SQL Plan Baselines

SQL Execution

- Active Session History
- AWR Active Session History
- SQL Statistics
- SQL Detail ACTIVE Report
- Monitor Statistics
- Monitor ACTIVE Report
- Monitor HTML Report
- Monitor TEXT Report
- Segment Statistics
- Session Statistics
- Session Events
- Parallel Processing

Tables

- Tables
- Statistics
- Statistics Versions
- Modifications
- Properties
- Physical Properties
- Constraints
- Columns
- Indexed Columns
- Histograms
- Partitions
- Indexes

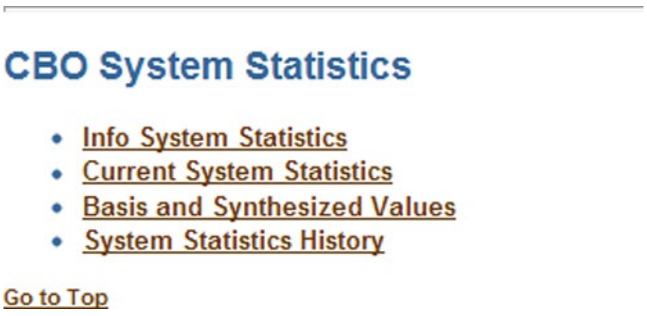
Objects

- Objects
- Dependencies
- Fixed Objects
- Fixed Object Columns
- Nested Tables
- Policies
- Audit Policies
- Tablespaces
- Metadata

Figure 2-1. *The top section of the SQLT report*

Let's remind ourselves what the first part of the HTML report looks like (see Figure 2-1). Remember this is one huge HTML page with many sections.

From the main screen, in the Global section, select "CBO System Statistics". This brings you to the section where you will see a heading "CBO System Statistics" (See Figure 2-2).

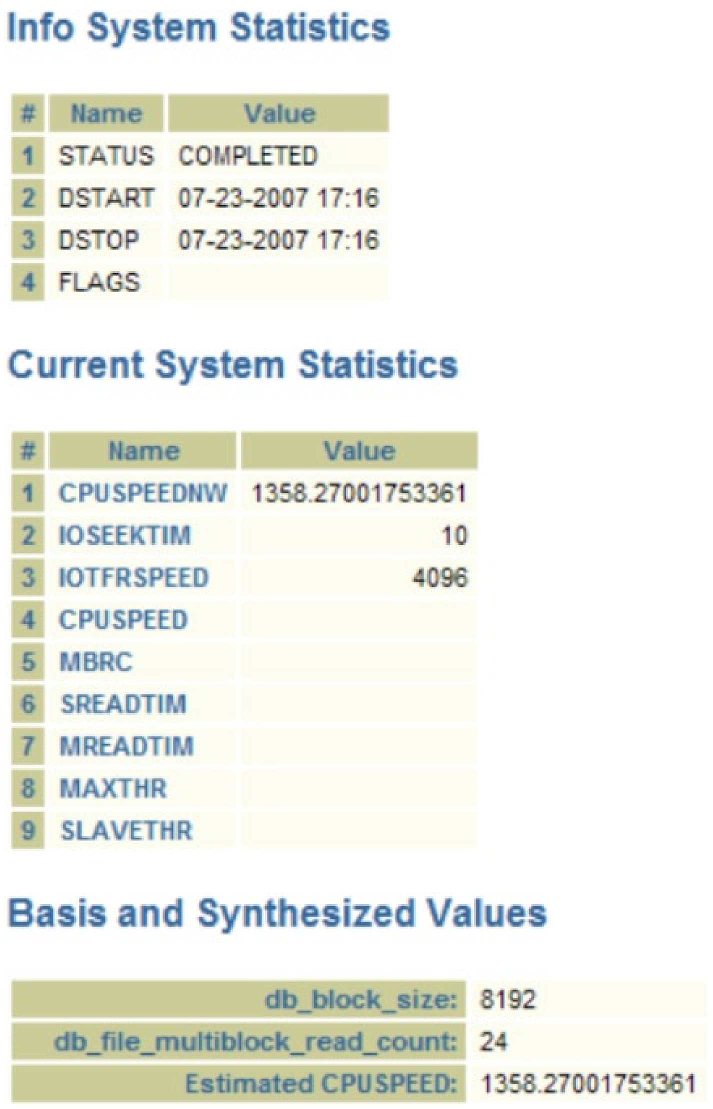

CBO System Statistics

- Info System Statistics
- Current System Statistics
- Basis and Synthesized Values
- System Statistics History

Go to Top

Figure 2-2. *The "CBO System Statistics" section*

Now click on "Info System Statistics." Figure 2-3 shows what you will see.



Info System Statistics

#	Name	Value
1	STATUS	COMPLETED
2	DSTART	07-23-2007 17:16
3	DSTOP	07-23-2007 17:16
4	FLAGS	

Current System Statistics

#	Name	Value
1	CPUSPEEDNW	1358.27001753361
2	IOSEEKTIM	10
3	IOTFRSPEED	4096
4	CPUSPEED	
5	MBRC	
6	SREADTIM	
7	MREADTIM	
8	MAXTHR	
9	SLAVETHR	

Basis and Synthesized Values

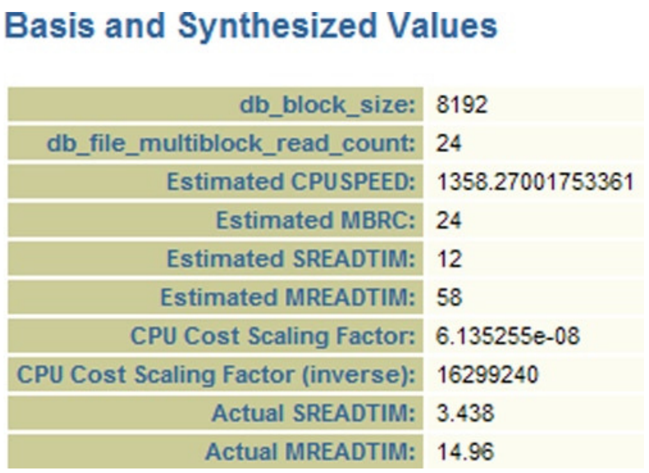
db_block_size:	8192
db_file_multiblock_read_count:	24
Estimated CPUSPEED:	1358.27001753361

Figure 2-3. *The "Info System Statistics" section*

The "Info System Statistics" section shows many pieces of important information about your environment. This screenshot also shows the "Current System Statistics" and the top of the "Basis and Synthesized Values" section.

Notice, when the System Statistics collection was started. It was begun on 23rd of July 2007 (quite a while ago). Has the workload changed much since then? Has any new piece of equipment been added? New SAN drives? Faster disks? All of these could affect the performance characteristics of the system. You could even have a system that needs a different set of system statistics for different times of the day.

Notice anything else strange about the system statistics? The start and end times are almost identical. The start and end time **should** be scheduled to collect information about the system characteristics at the *start* and *end* times of the representative workload. These values mean that they where set at database creation time and never changed. Look at the "Basis and Synthesized Values" sections shown in Figure 2-4.



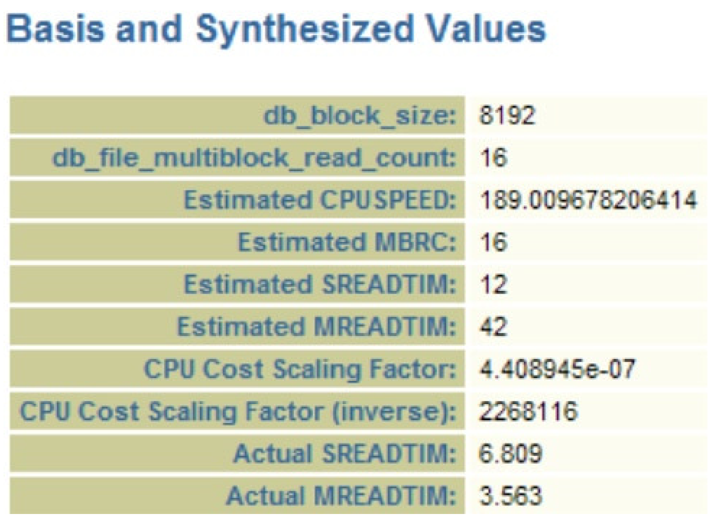
Basis and Synthesized Values

db_block_size:	8192
db_file_multiblock_read_count:	24
Estimated CPUSPEED:	1358.2700175336 1
Estimated MBRC:	24
Estimated SREADTIM:	12
Estimated MREADTIM:	58
CPU Cost Scaling Factor:	6.135255e-08
CPU Cost Scaling Factor (inverse):	16299240
Actual SREADTIM:	3.438
Actual MREADTIM:	14.96

Figure 2-4. *From the "Basis and Synthesized Values" section just under "Info System Statistics" section*

The estimated SREADTIM (single block read time in ms.) and MREADTIM (multi-block read time in ms) are 12ms and 58ms, whereas the actual values (just below) are 3.4ms and 15ms. Are these good values? It can be hard to tell because modern SAN systems can deliver blistering I/O read rates. For traditional non-SAN systems you would expect multi-block read times to be higher than single block read times and the normally around 9ms and 22ms. In this case they are in a reasonable range. The Single block read time is less than the multi-block read time (you would expect that, right?).

Now look in Figure 2-5 at a screen shot from a different system.



Basis and Synthesized Values

db_block_size:	8192
db_file_multiblock_read_count:	16
Estimated CPUSPEED:	189.009678206414
Estimated MBRC:	16
Estimated SREADTIM:	12
Estimated MREADTIM:	42
CPU Cost Scaling Factor:	4.408945e-07
CPU Cost Scaling Factor (inverse):	2268116
Actual SREADTIM:	6.809
Actual MREADTIM:	3.563

Figure 2-5. *Basis and synthezed values section under Info System Statistics*

Notice anything unusual about the Actual SREADTIM and Actual MREADTIM?

Apart from the fact that the Actual SREADTIM is 6.809ms (a low value) and the Actual MREADTIM is 3.563ms (also a low value). The problem here is that the Actual MREADTIM is less than the SREADTIM. If you see values like these, you should be alert to the possibility that full table scans are going to be costed lower than operations that require single block reads.

What does it mean to the optimizer for MREADTIM to be less than SREADTIM? This is about equivalent to you telling the optimizer that it's OK to drive the car upside down with the roof sliding on the road. It's the wrong way round. If the optimizer takes the values in Figure 2-5 as the truth, it will favor steps that involve multi-block reads. For example, the optimizer will favor full table scans. That could be very bad for your run-time execution. If on the other hand you have a fast SAN system you may well have a low Actual MREADTIM.

The foregoing is just one example how a bad number can lead the optimizer astray. In this specific case you would be better off having no Actual values and relying on the optimizer's guesses, which are shown as the estimated SREADTIM and MREADTIM values. Those guesses would be better than the actual values.

How do you correct a situation like I've just described? It's much easier that you would think. The steps to fix this kind of problem are shown in the list below:

1. Choose a time period that is representative of your workload. For example, you could have a daytime workload called WORKLOAD.
2. Create a table to contain the statistics information. In the example below we have called the table SYSTEM_STATISTICS.
3. Collect the statistics by running the GATHER_SYSTEM_STATS procedure during the chosen time period.
4. Import those statistics using DBMS_STATS.IMPORT_SYSTEM_STATS.

Let's look at the steps for collecting the system statistics for a 2-hour interval in more detail. In the first step we create a table to hold the values we will collect. In the second step we call the routine DBMS_STATS.GATHER_SYSTEM_STATS, with an INTERVAL parameter of 120 minutes. Bear in mind that the interval parameter should be chosen to reflect the period of your representative workload.

```
exec DBMS_STATS.CREATE_STAT_TABLE ('SYS','SYSTEM_STATISTICS');
BEGIN
   DBMS_STATS.GATHER_SYSTEM_STATS ('interval',interval => 120, stattab => 'SYSTEM_STATISTICS',
statid => 'WORKLOAD');
END;
/
execute DBMS_STATS.IMPORT_SYSTEM_STATS(stattab => 'SYSTEM_STATISTICS', statid => 'WORKLOAD',
statown => 'SYS');
Once you have done this you can view the values from the SQLT report or from a SELECT statement.

SQL> select * from sys.aux_stats$;

SNAME                          PNAME                               PVAL1 PVAL2
------------------------------ ------------------------------ ---------- ------------------
SYSSTATS_INFO                  STATUS                                    COMPLETED
SYSSTATS_INFO                  DSTART                                    09-29-2012 11:01
SYSSTATS_INFO                  DSTOP                                     09-29-2012 11:02
SYSSTATS_INFO                  FLAGS                                   0
SYSSTATS_MAIN                  CPUSPEEDNW                        972.327
SYSSTATS_MAIN                  IOSEEKTIM                              10
SYSSTATS_MAIN                  IOTFRSPEED                           4096
SYSSTATS_MAIN                  SREADTIM                            8.185
SYSSTATS_MAIN                  MREADTIM                           55.901
SYSSTATS_MAIN                  CPUSPEED                              972
SYSSTATS_MAIN                  MBRC
SYSSTATS_MAIN                  MAXTHR
SYSSTATS_MAIN                  SLAVETHR

13 rows selected.
```

If you get adept at doing this you can even set up different statistics tables for different workloads and import them and delete the old statistics when not needed. To delete the existing statistics you would use

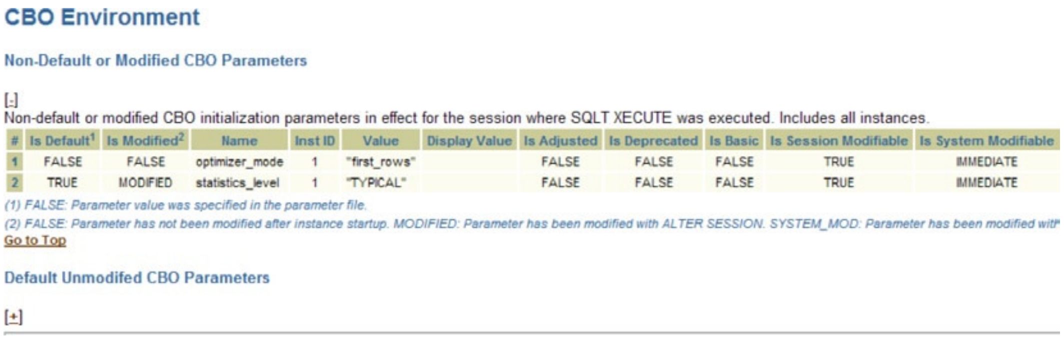
```
SQL> execute DBMS_STATS.DELETE_SYSTEM_STATS;
```

One word of caution, however, with setting and deleting system stats. This kind of operation will influence the behavior of the CBO for every SQL on the system. It follows therefore that any changes to these parameters should be made carefully and tested thoroughly on a suitable test environment.

Cost-Based Optimizer Parameters

Another input into the CBO's decision-making process (for developing your execution plan) would be the CBO parameters. These parameters control various aspects of the cost based optimizer's behavior. For example, `optimizer_dynamic_sampling` controls the level of dynamic sampling to be done for SQL execution. Wouldn't it be nice to have a quick look at every system and see the list of parameters that have been changed from the defaults? Well with SQLT that list is right there under "CBO Environment".

Figure 2-6 is an example where almost nothing has been changed. It's simple to tell this because there are only 2 rows in this section of the SQLT HTML report. The `optimizer_mode` has been changed from the default. If you see hundreds of entries here then you should look at the entries carefully and assess if any of the parameters that have been changed are causing you a problem. This example represents a DBA who likes to leave things alone.



CBO Environment

Non-Default or Modified CBO Parameters

[-]
Non-default or modified CBO initialization parameters in effect for the session where SQLT XECUTE was executed. Includes all instances.

#	Is Default[1]	Is Modified[2]	Name	Inst ID	Value	Display Value	Is Adjusted	Is Deprecated	Is Basic	Is Session Modifiable	Is System Modifiable
1	FALSE	FALSE	optimizer_mode	1	"first_rows"		FALSE	FALSE	FALSE	TRUE	IMMEDIATE
2	TRUE	MODIFIED	statistics_level	1	"TYPICAL"		FALSE	FALSE	FALSE	TRUE	IMMEDIATE

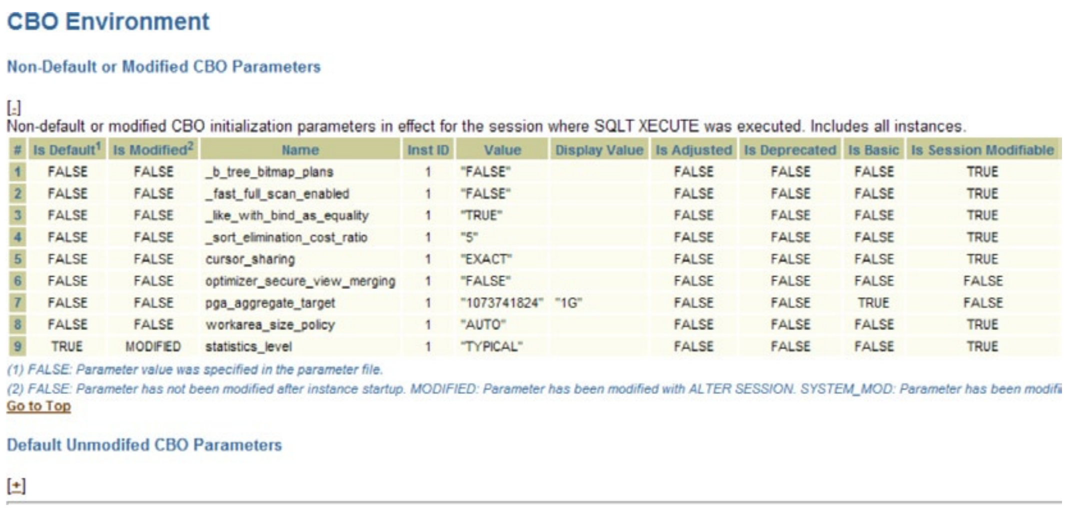
(1) FALSE: Parameter value was specified in the parameter file.
(2) FALSE: Parameter has not been modified after instance startup. MODIFIED: Parameter has been modified with ALTER SESSION. SYSTEM_MOD: Parameter has been modified wit
Go to Top

Default Unmodifed CBO Parameters

[+]

Figure 2-6. *The CBO environment section. Only 2 records indicates a system very close to the default settings*

Figure 2-7 shows an example where more than just two parameters have been changed from their default setting. Now instead of 2 rows of the previous example we have 8 non-default values. Each one of these parameters needs to be justified.


CBO Environment

Non-Default or Modified CBO Parameters

[-]
Non-default or modified CBO initialization parameters in effect for the session where SQLT XECUTE was executed. Includes all instances.

#	Is Default[1]	Is Modified[2]	Name	Inst ID	Value	Display Value	Is Adjusted	Is Deprecated	Is Basic	Is Session Modifiable
1	FALSE	FALSE	_b_tree_bitmap_plans	1	"FALSE"		FALSE	FALSE	FALSE	TRUE
2	FALSE	FALSE	_fast_full_scan_enabled	1	"FALSE"		FALSE	FALSE	FALSE	TRUE
3	FALSE	FALSE	_like_with_bind_as_equality	1	"TRUE"		FALSE	FALSE	FALSE	TRUE
4	FALSE	FALSE	_sort_elimination_cost_ratio	1	"5"		FALSE	FALSE	FALSE	TRUE
5	FALSE	FALSE	cursor_sharing	1	"EXACT"		FALSE	FALSE	FALSE	TRUE
6	FALSE	FALSE	optimizer_secure_view_merging	1	"FALSE"		FALSE	FALSE	FALSE	FALSE
7	FALSE	FALSE	pga_aggregate_target	1	"1073741824"	"1G"	FALSE	FALSE	TRUE	FALSE
8	FALSE	FALSE	workarea_size_policy	1	"AUTO"		FALSE	FALSE	FALSE	TRUE
9	TRUE	MODIFIED	statistics_level	1	"TYPICAL"		FALSE	FALSE	FALSE	TRUE

(1) FALSE: Parameter value was specified in the parameter file.
(2) FALSE: Parameter has not been modified after instance startup. MODIFIED: Parameter has been modified with ALTER SESSION. SYSTEM_MOD: Parameter has been modifi
Go to Top

Default Unmodifed CBO Parameters

[+]

Figure 2-7. *The CBO environment with many non-standard parameter settings*

We also have 4 hidden parameters set (they are preceded by underscores). In this example each of the hidden parameters should be carefully researched to see if it can be explained. If you have kept careful records or commented on your changes you may know why `_b_tree_bitmap_plans` has been set to FALSE. Often, however, parameters like these can stay set in a system for years with no explanation.

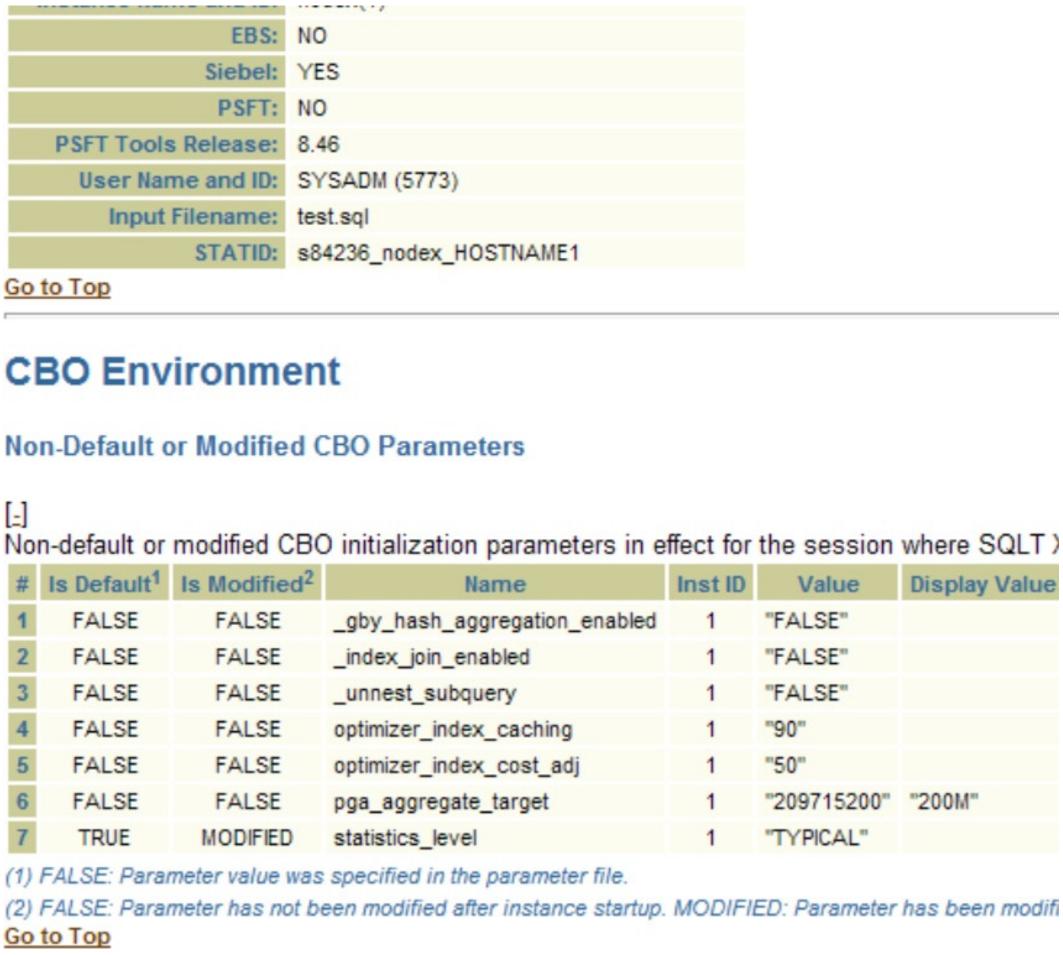
The following are common explanations:

- Somebody changed it a while ago, we don't know why, and he/she has left now.
- We don't want to change it in case it breaks something.

This section is useful and can often give you a clue as to what has been changed in the past (perhaps you're new to the current site). Take special note of hidden parameters. Oracle support will take a good look at these and decide if their reason for being still holds true. It is generally true that hidden parameters are not likely doing you any favors, especially if you don't know what they're for. Naturally, you can't just remove them from a production system. You have to execute key SQL statements on a test system and then remove those parameters on that test system to see what happens to the overall optimizer cost.

Siebel Environment Considerations

Some environments are special just because Oracle engineering have decided that a special set of parameters are better for them. This is the case with Siebel Systems Customer Relationship Management (CRM) application. There are hidden parameters that Oracle engineering has determined get the best performance from your system. If your system is Siebel, then in the "Environment" section you will see something like that shown in Figure 2-8.



EBS:	NO
Siebel:	YES
PSFT:	NO
PSFT Tools Release:	8.46
User Name and ID:	SYSADM (5773)
Input Filename:	test.sql
STATID:	s84236_nodex_HOSTNAME1

Go to Top

CBO Environment

Non-Default or Modified CBO Parameters

[-]
Non-default or modified CBO initialization parameters in effect for the session where SQLT

#	Is Default[1]	Is Modified[2]	Name	Inst ID	Value	Display Value
1	FALSE	FALSE	_gby_hash_aggregation_enabled	1	"FALSE"	
2	FALSE	FALSE	_index_join_enabled	1	"FALSE"	
3	FALSE	FALSE	_unnest_subquery	1	"FALSE"	
4	FALSE	FALSE	optimizer_index_caching	1	"90"	
5	FALSE	FALSE	optimizer_index_cost_adj	1	"50"	
6	FALSE	FALSE	pga_aggregate_target	1	"209715200"	"200M"
7	TRUE	MODIFIED	statistics_level	1	"TYPICAL"	

(1) FALSE: Parameter value was specified in the parameter file.
(2) FALSE: Parameter has not been modified after instance startup. MODIFIED: Parameter has been modifi
Go to Top

Figure 2-8. *Example of parameter settings for a Siebel CRM environment*

The "go to" place for Siebel tuning is the "Performance Tuning Guidelines for Siebel CRM Applications on Oracle Database" white paper. This can be found in Note 781927.1. There are many useful pieces of information in this document, but with regard to optimizer parameters, Page 9 lists the non-default parameters that should be set for good performance. For example, `optimizer_index_caching` should be set to 0.

Do not even attempt to tune your SQL on a Siebel system until these values are all correct. For example, on the above Siebel system if there was a performance problem, the first steps would be to fix all the parameters that are wrong according to Note 781927.1. You cannot hope to get stable performance from a Siebel CRM system unless this foundation is in place.

Hints

Hints were created to give the DBA and developer some control over what choices the optimizer is allowed to make. An example hint is USE_NL. In the example below I have created two minimal tables called `test` and `test2`, each with a single row. Not surprisingly if I let the optimizer choose a plan it will use a `MERGE JOIN CARTESIAN` as there are only single rows in each of these tables.

```
SQL> select
  test.col1,
  test2.col1
from
  test,
  test2
where
  test.col1=test2.col1;
      COL1       COL1
---------- ----------
         1          1
         1          1
SQL> set autotrace traceonly explain;
SQL> /
Execution Plan
Plan hash value: 1571755046
------------------------------------------------------------------------------
| Id  | Operation             | Name  | Rows  | Bytes | Cost (%CPU)| Time     |
------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |       |     2 |    52 |     6   (0)| 00:00:01 |
|   1 |  MERGE JOIN CARTESIAN|       |     2 |    52 |     6   (0)| 00:00:01 |
|   2 |   TABLE ACCESS FULL   | TEST2 |     1 |    13 |     3   (0)| 00:00:01 |
|   3 |   BUFFER SORT         |       |     2 |    26 |     3   (0)| 00:00:01 |
|*  4 |    TABLE ACCESS FULL  | TEST  |     2 |    26 |     3   (0)| 00:00:01 |
------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   4 - filter("TEST"."COL1" = "TEST2"."COL1")
Note
-----
   - dynamic sampling used for this statement (level=2)
SQL> list
  1* select
  test.col1,
  test2.col1
from
  test,
  test2
where
  test.col1=test2.col1
```

The command `list` above shows the previous DML (Data Manipulation Language). I then amended the SQL to contain a single hint. The syntax for all hints begins with `/*+` and ends with `*/`. In the case of `USE_NL` the portion inside the bracket can take multiple entries representing tables (either a table name, as in our case, or an alias if used). Here is the modified query.

```
SQL> select /*+ USE_NL(test) */ * from test, test2 where test.col1=test2.col1;

Execution Plan
----------------------------------------------------------
Plan hash value: 74026472
----------------------------------------------------------------------------
| Id  | Operation          | Name  | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |       |     2 |    52 |     6   (0)| 00:00:01 |
|   1 |  NESTED LOOPS      |       |     2 |    52 |     6   (0)| 00:00:01 |
|   2 |   TABLE ACCESS FULL| TEST2 |     1 |    13 |     3   (0)| 00:00:01 |
|*  3 |   TABLE ACCESS FULL| TEST  |     2 |    26 |     3   (0)| 00:00:01 |
----------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   3 - filter("TEST"."COL1"="TEST2"."COL1")
Note
-----
   -   dynamic sampling used for this statement (level=2)
```

Notice how in the second execution plan a NESTED LOOP was used.

What we're saying to the optimizer is: "we know you're not as clever as we are, so ignore the rules and just do what we think at this point."

Sometimes using hints is right, but sometimes it's wrong. Occasionally hints are inherited from old code, and it is a brave developer who removes them in the hope that performance will improve. Hints are also a form of data input to the CBOs process of developing an execution plan. Hints are often needed because the other information fed to the optimizer is wrong. So, for example, if the object statistics are wrong you may *need* to give the optimizer a hint because its statistics are wrong.

Is this the correct way to use a hint? No. The problem with using a hint like this is that it may have been right when it was applied, but it could be wrong later, and in fact it probably is. If you want to tune code, first remove the hints, let the optimizer run free, while feeling the blades of data between its toes, free of encumbrances. Make sure it has good recent, statistics, and see what it comes up with.

You can always get the SQL Text that you are evaluating by clicking on the "SQL Text" link from the top section of the SQLT report. Here's another example of a query. This time we're using a USE_NL hint with two aliases

```
SQL> set autotrace traceonly explain;
SQL> select cust_first_name, amount_sold
  2  from customers C, sales S
  3  where c.cust_id=s.cust_id and amount_sold>100;
SQL> REM
SQL> REM Here is the execution plan
SQL> REM
```

```
Execution Plan
----------------------------------------------------------
Plan hash value: 3549450340
------------------------------------------------------------------------------------------
| Id  | Operation             | Name      | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |           |   144K|  3107K|       |  1118   (2)| 00:00:14 |
|*  1 |  HASH JOIN            |           |   144K|  3107K|  1304K|  1118   (2)| 00:00:14 |
|   2 |   TABLE ACCESS FULL   | CUSTOMERS | 55500 |   650K|       |   405   (1)| 00:00:05 |
|   3 |   PARTITION RANGE ALL|           |   144K|  1412K|       |   496   (4)| 00:00:06 |
|*  4 |    TABLE ACCESS FULL  | SALES     |   144K|  1412K|       |   496   (4)| 00:00:06 |
------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - access("C"."CUST_ID"="S"."CUST_ID")
   4 - filter("AMOUNT_SOLD">100)
```

I've only shown part of the execution plan because we're only interested in the operations (under the "Operation" column). Id 1 shows an operation of hash join. This is probably reasonable as there are over 900,000 rows in sales. The object with the most rows (in this case SALES) is read first and then CUSTOMERS (with only 55,500 rows). So if we decided to use a nested loop instead we would add a hint to the code like this:

```
SQL> select /*+ USE_NL(C S) */
  cust_first_name,
  amount_sold
from
  customers C,
  sales S
where
  c.cust_id=s.cust_id
  and amount_sold>100;
Execution Plan
----------------------------------------------------------
Plan hash value: 4237376444
---------------------------------------------------------------------------------------------
| Id  | Operation                     | Name         | Rows  | Bytes | Cost (%CPU)| Time     |
---------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |              |   144K|  3107K|   145K  (1)| 00:29:03 |
|   1 |  NESTED LOOPS                 |              |       |       |            |
|   2 |   NESTED LOOPS                |              |   144K|  3107K|   145K  (1)| 00:29:03 |
|   3 |    PARTITION RANGE ALL        |              |   144K|  1412K|   496   (4)| 00:00:06 |
|*  4 |     TABLE ACCESS FULL         | SALES        |   144K|  1412K|   496   (4)| 00:00:06 |
|*  5 |    INDEX UNIQUE SCAN          | CUSTOMERS_PK |     1 |       |     0   (0)| 00:00:01 |
|   6 |   TABLE ACCESS BY INDEX ROWID| CUSTOMERS    |     1 |    12 |     1   (0)| 00:00:01 |
---------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   4 - filter("AMOUNT_SOLD">100)
   5 - access("C"."CUST_ID"="S"."CUST_ID")
```

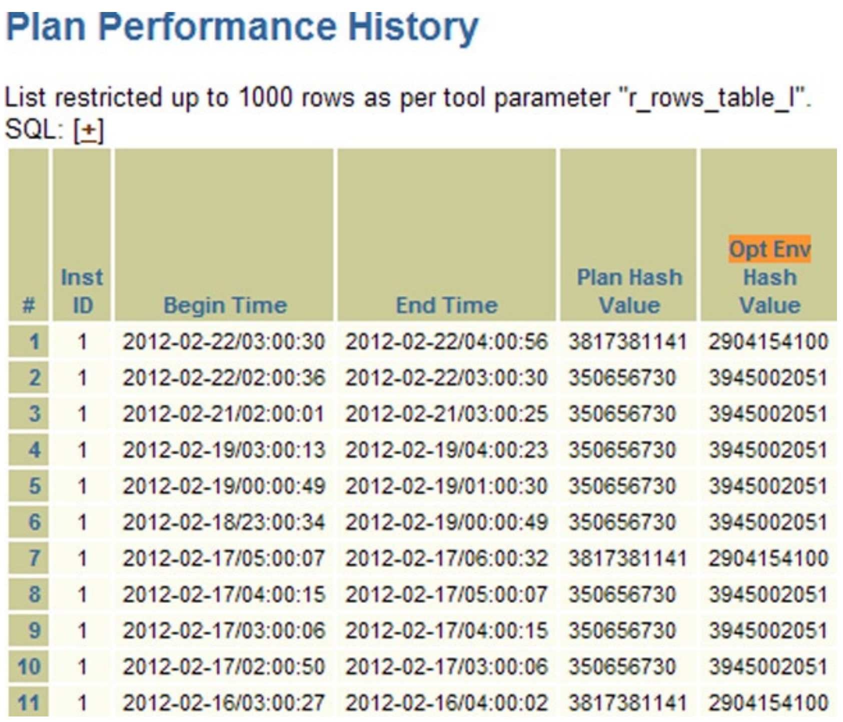
The hint is enclosed inside “/*+” and “*/” as before. Anything inside these hint brackets will then be considered by the optimizer. But it is important to realize that the hint (in this case “USE_NL(C S)”) must be valid. Notice that now the operations have changed. Now the plan is to use nested loops instead of a hash join. The cost is much higher than the hash join plan at 145,000, but the optimizer has to obey the hint because it is valid. If it is not valid, no error is generated and the optimizer carries on as if there was no hint. Look at what happens.

```
SQL> select /*+ MY_HINT(C S) */ cust_first_name, amount_sold
from
  customers C, sales Swhere c.cust_id=s.cust_id and amount_sold>100;

Execution Plan
----------------------------------------------------------
Plan hash value: 3549450340
-----------------------------------------------------------------------------------------
| Id  | Operation            | Name      | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |           |   144K|  3107K|       |  1118   (2)| 00:00:14 |
|*  1 |  HASH JOIN           |           |   144K|  3107K|  1304K|  1118   (2)| 00:00:14 |
|   2 |   TABLE ACCESS FULL  | CUSTOMERS | 55500 |   650K|       |   405   (1)| 00:00:05 |
|   3 |   PARTITION RANGE ALL|           |   144K|  1412K|       |   496   (4)| 00:00:06 |
|*  4 |    TABLE ACCESS FULL | SALES     |   144K|  1412K|       |   496   (4)| 00:00:06 |
-----------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - access("C"."CUST_ID"="S"."CUST_ID")
   4 - filter("AMOUNT_SOLD">100)
```

The optimizer saw my hint, didn’t recognize it as a valid hint, ignored it and did its own thing, which in this case was to go back to the hash join.

History of Changes

Often when investigating a performance problem, it is crucial to get an idea when SQL performance changed. Vague reports like “it was fine yesterday” are not precise enough, although they may help you by directing your analysis to a particular time period in your SQLT report. Every change in the optimizer environment (parameter changes) is timestamped. Every time an execution changes a new execution plan is created that is also timestamped, every time an SQL statement is executed its metrics are gathered and stored in the AWR repository. This is the same source of data that the automatic facilities in Oracle use to suggest improvements in your SQL. SQLT uses these sources to build a history of the executions of your chosen SQL ID. This mine of information is only one click away from the top of the SQLT report. Click on the “Performance History” link under the “Plans” heading from the top of the SQLT HTML report. Depending on how many executions of your SQL there are in the system you may see something like the screen shot in Figure 2-9. There are other columns in the full HTML report but we’ll limit our discussion to the “Opt Env Hash Value” for now.



Plan Performance History

List restricted up to 1000 rows as per tool parameter "r_rows_table_l".
SQL: [+]

#	Inst ID	Begin Time	End Time	Plan Hash Value	Opt Env Hash Value
1	1	2012-02-22/03:00:30	2012-02-22/04:00:56	3817381141	2904154100
2	1	2012-02-22/02:00:36	2012-02-22/03:00:30	350656730	3945002051
3	1	2012-02-21/02:00:01	2012-02-21/03:00:25	350656730	3945002051
4	1	2012-02-19/03:00:13	2012-02-19/04:00:23	350656730	3945002051
5	1	2012-02-19/00:00:49	2012-02-19/01:00:30	350656730	3945002051
6	1	2012-02-18/23:00:34	2012-02-19/00:00:49	350656730	3945002051
7	1	2012-02-17/05:00:07	2012-02-17/06:00:32	3817381141	2904154100
8	1	2012-02-17/04:00:15	2012-02-17/05:00:07	350656730	3945002051
9	1	2012-02-17/03:00:06	2012-02-17/04:00:15	350656730	3945002051
10	1	2012-02-17/02:00:50	2012-02-17/03:00:06	350656730	3945002051
11	1	2012-02-16/03:00:27	2012-02-16/04:00:02	3817381141	2904154100

Figure 2-9. *The optimizer Env Hash Value changed on the 22nd of February from 3945002051 to 2904154100. This means something in the CBO enviroment changed on that date*

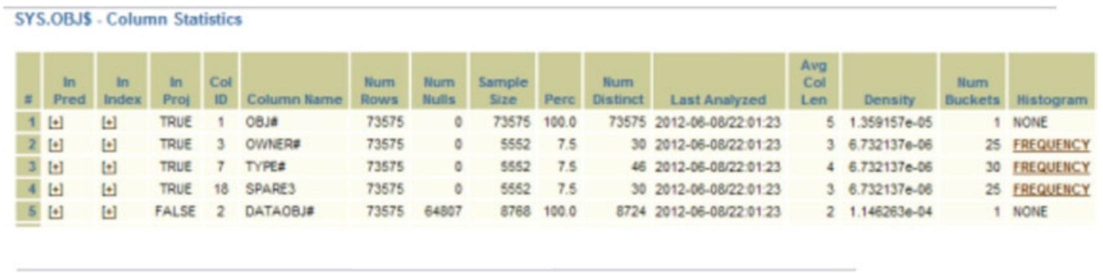
Look at the "Opt Env Hash Value" for the statement history in Figure 2-9. For the one SQL statement that we are analyzing with its list og "Begin Time" and "End Times" we see other changes taking place. For example, the plan hash value changed (for the same SQL statement) and so did the "Opt Env Hash Value". Its value is 2904154100 until the 17th of February 2012. Then it changes to 3945002051 then back to 2904154100 and then back to 3945002051 (on the 18th of February). Something about the optimizer's environment changed on those dates. Did the change improve the situation or make it worse? Notice that every time the optimizer hash value changes, the hash value of the plan changes also. Somebody or something is changing the optimizer's environment and affecting the execution plan.

Column Statistics

One of the example SQLT reports (the first one we looked at) had the following line in the observation:

TABLE	*SYS.OBJ$*	*Table contains 4 column(s) referenced in predicates where the number of distinct values does not match the number of buckets.*

If I follow the link to the column statistics (from the top section of the main HTML report, click on "Columns" then "Column Statistics"), I can see the results in Figure 2-10.



SYS.OBJ$ - Column Statistics

#	In Pred	In Index	In Proj	Col ID	Column Name	Num Rows	Num Nulls	Sample Size	Perc	Num Distinct	Last Analyzed	Avg Col Len	Density	Num Buckets	Histogram
1	[+]	[+]	TRUE	1	OBJ#	73575	0	73575	100.0	73575	2012-06-08/22:01:23	5	1.359157e-05	1	NONE
2	[+]	[+]	TRUE	3	OWNER#	73575	0	5552	7.5	30	2012-06-08/22:01:23	3	6.732137e-06	25	FREQUENCY
3	[+]	[+]	TRUE	7	TYPE#	73575	0	5552	7.5	46	2012-06-08/22:01:23	4	6.732137e-06	30	FREQUENCY
4	[+]	[+]	TRUE	18	SPARE3	73575	0	5552	7.5	30	2012-06-08/22:01:23	3	6.732137e-06	25	FREQUENCY
5	[+]	[+]	FALSE	2	DATAOBJ#	73575	64807	8768	100.0	8724	2012-06-08/22:01:23	2	1.146263e-04	1	NONE

Figure 2-10. *Column statistics*

The second line of statistics is for the OWNER# table column. Look toward the right at the Num Distinct value. You'll see that it is 30, representing that number of estimated distinct values. At the time the statistics were collected there probably were 30 or so distinct values. But notice the "Histogram" column! It shows a FREQUENCY type histogram with 25 buckets. In a FREQUENCY type histogram each possible value has a bucket in which is kept the number of values for that value. So for example if every value from 1 to 255 had only one value then each bucket would contain a "1". If the first bucket (labeled "1" had 300 in it, this would mean that the value 1 had been found 300 times in the data. These numbers kept in the buckets are then used by the optimizer to calculate costs for retrieving data related to that value. So in our example above retrieving a "1" would be more costly than retrieving a "2" from the same table (because there are more values that are "1"s). If we had 10 buckets (each with their FREQUENCY value) and then we attempt to retrieve some data and find an "11", then the optimizer has to guess that this new value was never collected and makes certain assumptions about the values likelihood (such as for higher and lower than the maximum values) the likelihood drops off dramatically as we go above the maximum value or below the minimum value. Values that have no buckets in the middle of the range are interpolated. So if you have less than 255 buckets, there is no good reason to have less than the actual number of buckets in a FREQUENCY histogram. That doesn't make any sense. The result of this kind of anomaly is that the histogram information kept for the OWNER# column will have five distinct values missing. If the distinct values are popular, and if the query you are troubleshooting happens to use them in a predicate, the optimizer will have to guess their cardinality.

Imagine for example, the following situation:

1. Imagine a two bucket histogram with "STELIOS" and "STEVEN". Now we add a third bucket "STEPHAN", who happens to be the biggest owner of objects in OBJ$.
2. Let's say the histogram values for STELIOS and STEVEN as follows:
   - STELIOS – 100 objects
   - STEVEN – 110 objects
   - This means that for this particular table STELIOS has 100 records for objects he owns and STEVEN has 110 objects that he owns. So when the histogram was created the buckets for STELIOS and STEVEN were filled with 100 and 110.
3. Further say that STEPHAN really owns 500 objects, so he has 500 records in the table. The optimizer doesn't know that though, because STEPHAN was created at some point after statistics were collected.

The optimizer now guesses the cardinality of STEPHAN as 105, when in fact STEPHAN has 500 objects. Because STEPHAN falls between STELIOS and STEVEN (alphabetically speaking), the optimizer presumes the Num Distinct value for STEPHAN falls between the values for the other two users (105 is the average of the two adjacent, alphabetically speaking, buckets. The result is that the CBO's guess for cardinality will be a long way out. We would see this in the execution plan (as an under estimate, if we ran a SQLT XECUTE), and we would drill into the OWNER# column and see that the column statistics were wrong. To fix the problem, we would gather statistics for SYS.OBJ$. In this case, of course, since the example we used was a SYS object there are special procedures for gathering statistics, but generally this kind of problem will occur on a user table and normal `DBMS_STATS` gathering procedures should be used.

Out-of-Range Values

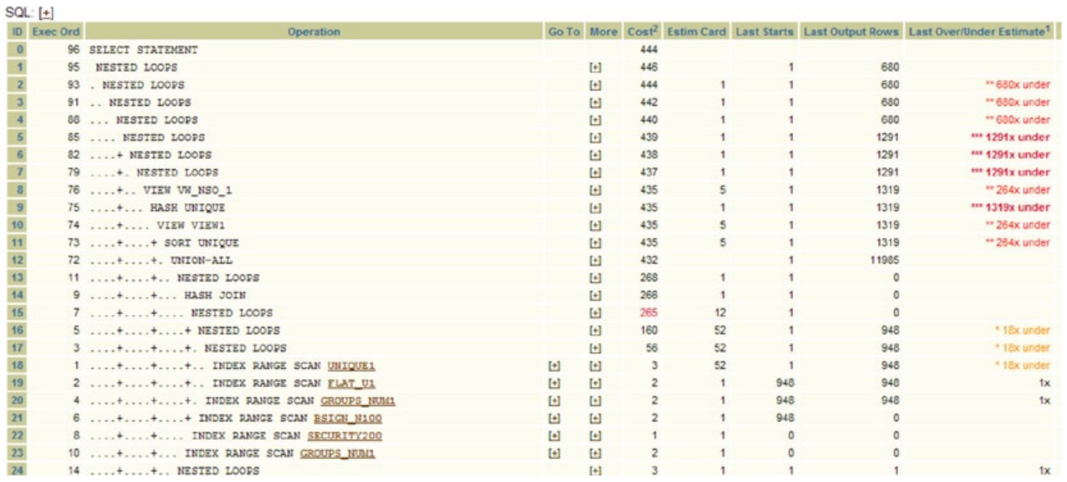
The situation the CBO is left with when it has to guess the cardinality between two values is bad enough but is not as bad as the situation when the value in a predicate is out of range: either larger than the largest value seen by the statistics or smaller than the smallest value seen by the statistics. In these cases the optimizer assumes that the estimated value for the out of range value tails off towards zero. If the value is well above the highest value, the optimizer may estimate a very low cardinality, say 1. A cardinality of 1 might persuade the optimizer to try a Cartesian join, which would result in very poor performance if the actual cardinality was 10,000. The method of solving such a problem would be the same.

1. Get the execution plan with XECUTE.
2. Look at the execution plan in the Execution Plan section and look at the predicates under "more" as described earlier.
3. Look at the statistics for the columns in the predicates and see if there is something wrong. Examples of signs of trouble would be
    a. A missing bucket (as described in the previous section)
    b. No histograms but highly skewed data

Out-of-range values can be particularly troublesome when data is being inserted at the higher or lower ends of the current data set. In any case by studying the histograms that appear in SQLT you have a good chance of understanding your data and how it changes with time. This is invaluable information for designing a strategy to tune your SQL or collecting good statistics.

Over Estimates and Under Estimates

Now let's look at a sample of a piece of SQL having so many joins that the number of operations is up to 96. See Figure 2-11, which shows a small portion of the resulting execution plan.



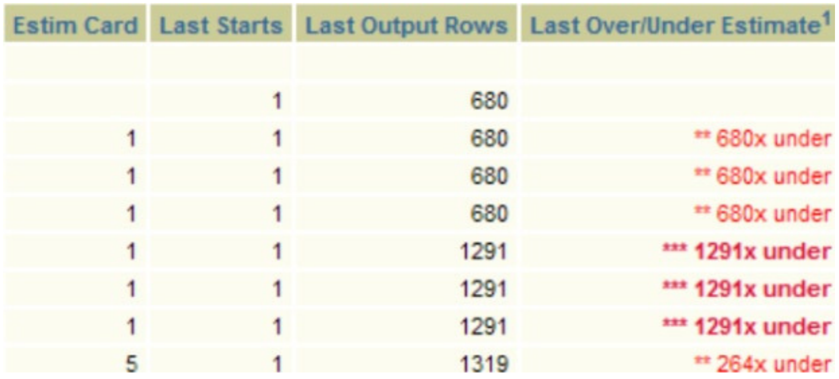
SQL: [+]

ID	Exec Ord	Operation	Go To	More	Cost[2]	Estim Card	Last Starts	Last Output Rows	Last Over/Under Estimate[1]
0	96	SELECT STATEMENT			444				
1	95	NESTED LOOPS		[+]	446		1	680	
2	93	. NESTED LOOPS		[+]	444	1	1	680	** 680x under
3	91	.. NESTED LOOPS		[+]	442	1	1	680	** 680x under
4	88	... NESTED LOOPS		[+]	440	1	1	680	** 680x under
5	85	.... NESTED LOOPS		[+]	439	1	1	1291	*** 1291x under
6	82	....+ NESTED LOOPS		[+]	438	1	1	1291	*** 1291x under
7	79	....+. NESTED LOOPS		[+]	437	1	1	1291	*** 1291x under
8	76	....+.. VIEW VW_NSO_1		[+]	435	5	1	1319	** 264x under
9	75	....+... HASH UNIQUE		[+]	435	1	1	1319	*** 1319x under
10	74	....+.... VIEW VIEW1		[+]	435	5	1	1319	** 264x under
11	73	....+....+ SORT UNIQUE		[+]	435	5	1	1319	** 264x under
12	72	....+....+. UNION-ALL		[+]	432		1	11985	
13	11	....+....+.. NESTED LOOPS		[+]	268	1	1	0	
14	9	....+....+... HASH JOIN		[+]	266	1	1	0	
15	7	....+....+.... NESTED LOOPS		[+]	265	12	1	0	
16	5	....+....+....+ NESTED LOOPS		[+]	160	52	1	948	* 18x under
17	3	....+....+....+. NESTED LOOPS		[+]	56	52	1	948	* 18x under
18	1	....+....+....+.. INDEX RANGE SCAN UNIQUE1	[+]	[+]	3	52	1	948	* 18x under
19	2	....+....+....+.. INDEX RANGE SCAN FLAT_U1	[+]	[+]	2	1	948	948	1x
20	4	....+....+....+. INDEX RANGE SCAN GROUPS_NUM1	[+]	[+]	2	1	948	948	1x
21	6	....+....+....+ INDEX RANGE SCAN BSIGN_N100	[+]	[+]	2	1	948	0	
22	8	....+....+.... INDEX RANGE SCAN SECURITY200	[+]	[+]	1	1	0	0	
23	10	....+....+... INDEX RANGE SCAN GROUPS_NUM1	[+]	[+]	2	1	0	0	
24	14	....+....+.. NESTED LOOPS		[+]	3	1	1	1	1x

Figure 2-11. *A small part of the execution plan, with 96 steps*

How do we handle an execution plan that has 96 steps, or more? Do we hand that plan over to development and tell them to tune it? With SQLT you don't need to do this.

Let's look at this page in more detail, by zooming in on the top right hand portion and look at the over and under estimates part of the screen (see Figure 2-12).



Estim Card	Last Starts	Last Output Rows	Last Over/Under Estimate[1]
	1	680	
1	1	680	** 680x under
1	1	680	** 680x under
1	1	680	** 680x under
1	1	1291	*** 1291x under
1	1	1291	*** 1291x under
1	1	1291	*** 1291x under
5	1	1319	** 264x under

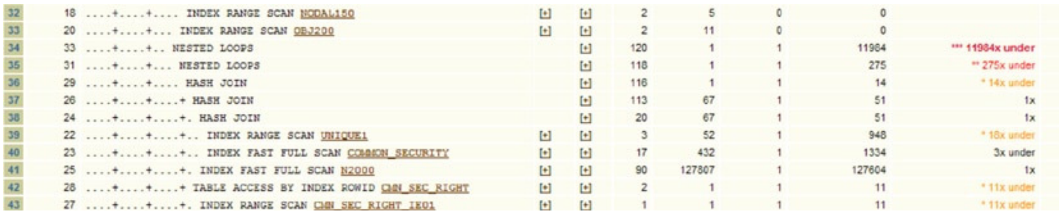
Figure 2-12. *The top right hand portion of the section showing the execution plan's over and under estimates*

We know that Figure 2-12 is a SQLT XECUTE report, (we know this because we have over and under estimate values in the report). But what are these over and under estimates? The numbers in the "Last Over/Under Estimate" column, represent by how many factors the actual number of rows expected by the optimizer for that operation is wrong. The rows returned is also dependent on the rows returned from the previous operation. So, for example, if we followed the operation count step by step from "Exec Ord" (see Figure 2-11), we would have these steps:

1. INDEX RANGE SCAN actually returned 948 rows
2. INDEX RANGE SCAN actually returned 948 rows
3. The result of step 1 and 2 was fed into a NESTED LOOP which actually returned 948 rows
4. INDEX RANGE SCAN actually returned 948 rows
5. NESTED LOOP (of the previous step with result of step 3)

And so on. The best way to approach this kind of problem is to read the steps in the execution plan, understand them, look at the over and under estimates and from there determine where to focus your attention.

Now look at Figure 2-13. Step ID 34 (which is the third line in Figure 2-13 and the 33rd step in the execution plan. Remember the execution order is shown by the numbers immediately to the left of the operation names e.g. INDEX RANGE SCAN) shows an under estimate of 11,984. This NESTED LOOP Is a result of the sections below it. We can drill into why the estimates are as they are by clicking on the "+" in the "More" column. From the "More" column we can look at the access predicates and see why the estimated cardinality and the actual rows returned diverged.



32	18	....+....+.... INDEX RANGE SCAN NODAL150	[+]	[+]	2	5	0	0	
33	20	....+....+... INDEX RANGE SCAN OBJ200	[+]	[+]	2	11	0	0	
34	33	....+....+.. NESTED LOOPS		[+]	120	1	1	11984	*** 11984x under
35	31	....+....+... NESTED LOOPS		[+]	118	1	1	275	** 275x under
36	29	....+....+.... HASH JOIN		[+]	116	1	1	14	* 14x under
37	26	....+....+....+ HASH JOIN		[+]	113	67	1	51	1x
38	24	....+....+....+. HASH JOIN		[+]	20	67	1	51	1x
39	22	....+....+....+.. INDEX RANGE SCAN UNIQUE1	[+]	[+]	3	52	1	948	* 18x under
40	23	....+....+....+.. INDEX FAST FULL SCAN COMMON_SECURITY	[+]	[+]	17	432	1	1334	3x under
41	25	....+....+....+. INDEX FAST FULL SCAN N2000	[+]	[+]	90	127807	1	127604	1x
42	28	....+....+....+ TABLE ACCESS BY INDEX ROWID CMN_SEC_RIGHT	[+]	[+]	2	1	1	11	* 11x under
43	27	....+....+....+. INDEX RANGE SCAN CMN_SEC_RIGHT_IE01	[+]	[+]	1	1	1	11	* 11x under

Figure 2-13. *Over and under estimated values can be good clues*

So for large statements like this, we work on each access predicate, each under and over estimate, working from the biggest estimation error to the smallest until we know the reason for each. In some cases, the cause will be stale statistics. In other cases, it will be skewed data. With SQLT, looking at a 200-line execution plan is no longer a thing to be feared. If you address each error as far as the optimizer is concerned (it expected 10 rows and got 1000) you can, step by step, fix the execution plan. You don't need hints to twist the CBO into the shape you *guess* might be right. You just need to make sure it has good statistics for system performance, single and multiblock read times, CPU speed and object statistics. Once you have all the right statistics in place, the optimizer will generate a good plan. If the execution plan is sometimes right and sometimes wrong, then you could be dealing with skewed data, in which case you'll need to consider the use of histograms. We discuss skewness as a special topic in much more detail in Chapter 4.

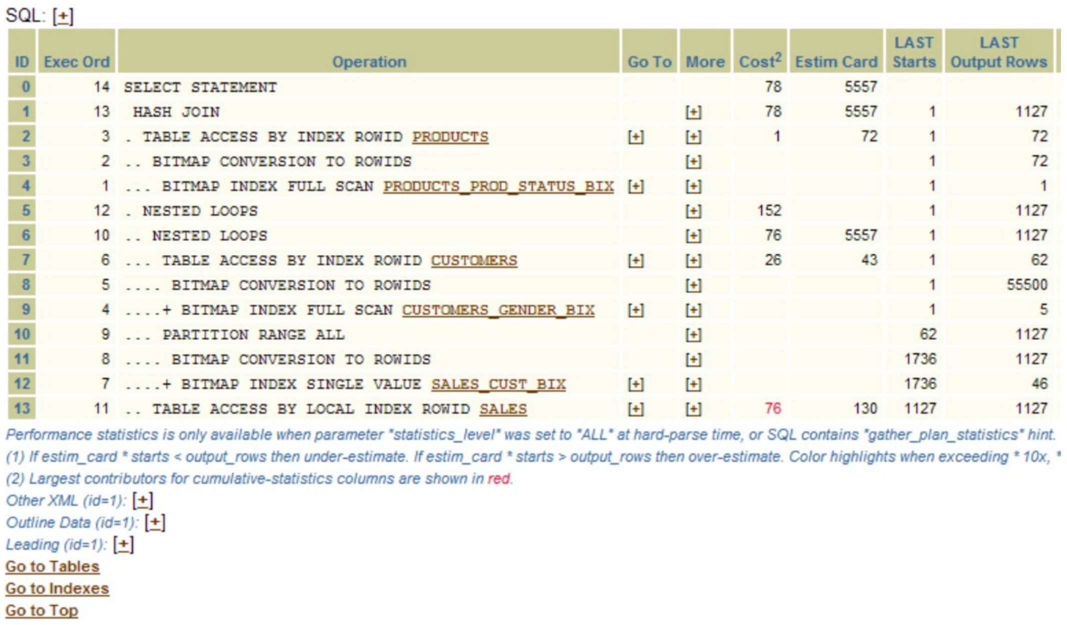
The Case of the Mysterious Change

Now that you've learned a little bit about SQLT and how to use it, we can look at an example without any prior knowledge of what the problem might be. Here is the scenario:

> *A developer comes to you and says his SQL was fine up until 3pm the previous day. It was doing hash joins as he wanted and expected, so he went to lunch. When he came back the plan had changed completely. All sorts of weird bit map indexes are being used. His data hasn't changed, and he hasn't collected any new statistics. What happened? He ends by saying "Honest, I didn't change anything."*

Once you've confirmed that the data has not changed, and no-one has added any new indexes (or dropped any), you ask for a SQLT XECUTE report (as the SQL is fairly short running and this is a development system).

Once you have that you look at the execution plans. The plan in Figure 2-14 happens to be the one you view first.



SQL: [+]

ID	Exec Ord	Operation	Go To	More	Cost^2	Estim Card	LAST Starts	LAST Output Rows
0	14	SELECT STATEMENT			78	5557		
1	13	HASH JOIN		[+]	78	5557	1	1127
2	3	. TABLE ACCESS BY INDEX ROWID PRODUCTS	[+]	[+]	1	72	1	72
3	2	.. BITMAP CONVERSION TO ROWIDS		[+]			1	72
4	1	... BITMAP INDEX FULL SCAN PRODUCTS_PROD_STATUS_BIX	[+]	[+]			1	1
5	12	. NESTED LOOPS		[+]	152		1	1127
6	10	.. NESTED LOOPS		[+]	76	5557	1	1127
7	6	... TABLE ACCESS BY INDEX ROWID CUSTOMERS	[+]	[+]	26	43	1	62
8	5	.... BITMAP CONVERSION TO ROWIDS		[+]			1	55500
9	4	....+ BITMAP INDEX FULL SCAN CUSTOMERS_GENDER_BIX	[+]	[+]			1	5
10	9	... PARTITION RANGE ALL		[+]			62	1127
11	8	.... BITMAP CONVERSION TO ROWIDS		[+]			1736	1127
12	7	....+ BITMAP INDEX SINGLE VALUE SALES_CUST_BIX	[+]	[+]			1736	46
13	11	.. TABLE ACCESS BY LOCAL INDEX ROWID SALES	[+]	[+]	76	130	1127	1127

Performance statistics is only available when parameter "statistics_level" was set to "ALL" at hard-parse time, or SQL contains "gather_plan_statistics" hint.
*(1) If estim_card * starts < output_rows then under-estimate. If estim_card * starts > output_rows then over-estimate. Color highlights when exceeding * 10x, **
(2) Largest contributors for cumulative-statistics columns are shown in red.
Other XML (id=1): [+]
Outline Data (id=1): [+]
Leading (id=1): [+]
Go to Tables
Go to Indexes
Go to Top

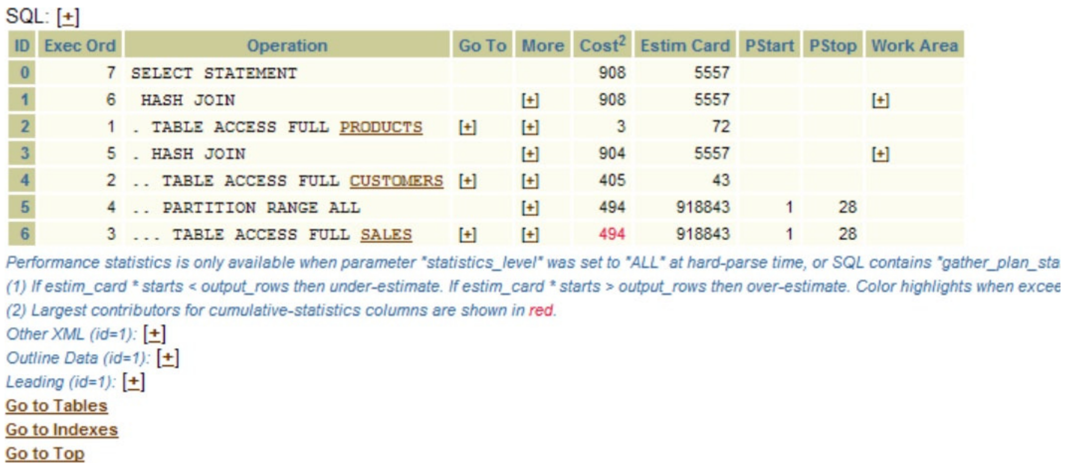
Figure 2-14. *Execution plan being investigated*

Looking at the plan, you can confirm what the developer said about a “weird” execution plan with strange bit map indexes. In fact though, there is nothing strange about this plan. It’s just that the first step is:

```
... BITMAP INDEX FULL SCAN PRODUCTS_PROD_STATUS_BIX
```

This step was not in the developer’s original plan. Hence the reason the developer perceives it as strange. For the one SQL statement the developer was working with we suspect that there are at least 2 execution plans (there can be dozens of execution plans for the one SQL statement, and SQLT captures them all).

Further down in the list of execution plans, we see that there are indeed plans using hash joins and full table scans. See Figure 2-15, which shows a different execution plan for the same SQL that the developer is working with. In this execution plan, which returns the same rows as the previous execution plan, the overall cost is 908.


SQL: [+]

ID	Exec Ord	Operation	Go To	More	Cost^2	Estim Card	PStart	PStop	Work Area
0	7	SELECT STATEMENT			908	5557			
1	6	HASH JOIN		[+]	908	5557			[+]
2	1	. TABLE ACCESS FULL PRODUCTS	[+]	[+]	3	72			
3	5	. HASH JOIN		[+]	904	5557			[+]
4	2	.. TABLE ACCESS FULL CUSTOMERS	[+]	[+]	405	43			
5	4	.. PARTITION RANGE ALL		[+]	494	918843	1	28	
6	3	... TABLE ACCESS FULL SALES	[+]	[+]	494	918843	1	28	

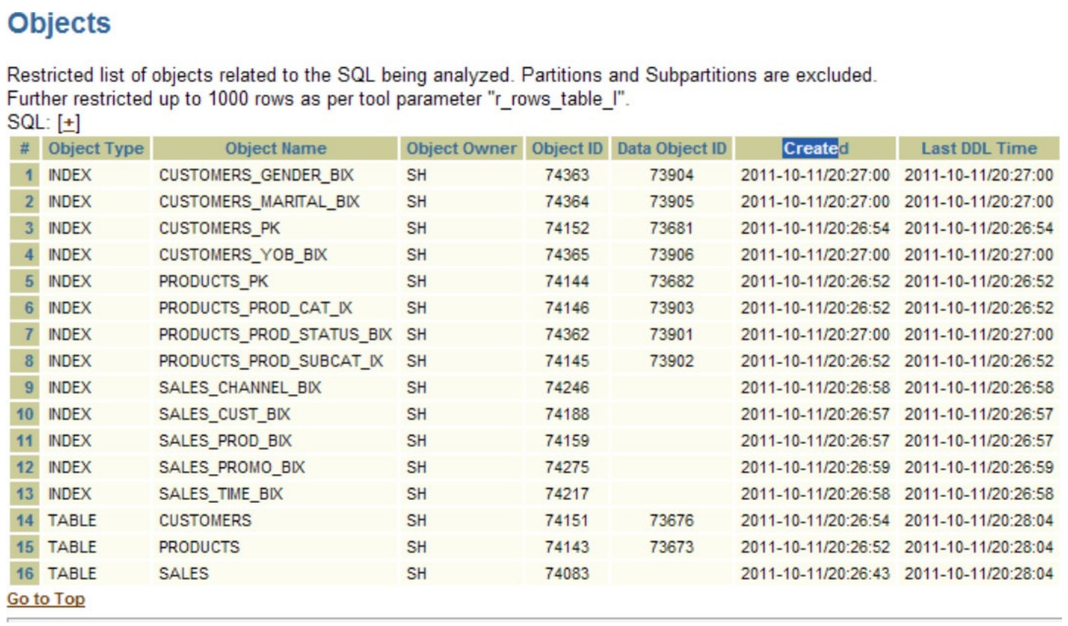
Performance statistics is only available when parameter "statistics_level" was set to "ALL" at hard-parse time, or SQL contains "gather_plan_sta
(1) If estim_card * starts < output_rows then under-estimate. If estim_card * starts > output_rows then over-estimate. Color highlights when excee
(2) Largest contributors for cumulative-statistics columns are shown in red.
Other XML (id=1): [+]
Outline Data (id=1): [+]
Leading (id=1): [+]
Go to Tables
Go to Indexes
Go to Top

Figure 2-15. *An execution plan showing a hash join*

So far we know there was a plan involving a hash join and bitmap indexes and that earlier there were plans with full table scans. If we look at the times of the statistics collection we see that indeed the statistics were gathered before the execution of these queries. This is a good thing, as statistics should be collected before the execution of a query!

■ **Note** As an aside, the ability of SQLT to present all relevant information quickly and easily is its greatest strength. It takes the guesswork out of detective work. Without SQLT, you would probably have to dig out a query to show you the time that the statistics were collected. With SQLT, the time of the collection of the statistics is right there in the report. You can check it while still thinking about the question!

So, the statistics didn’t change and the SQL text didn’t change. It’s possible that an index was added sometime over lunchtime. You can check that by looking at the objects section of the report, as shown in Figure 2-16.



Objects

Restricted list of objects related to the SQL being analyzed. Partitions and Subpartitions are excluded.
Further restricted up to 1000 rows as per tool parameter "r_rows_table_l".
SQL: [+]

#	Object Type	Object Name	Object Owner	Object ID	Data Object ID	Created	Last DDL Time
1	INDEX	CUSTOMERS_GENDER_BIX	SH	74363	73904	2011-10-11/20:27:00	2011-10-11/20:27:00
2	INDEX	CUSTOMERS_MARITAL_BIX	SH	74364	73905	2011-10-11/20:27:00	2011-10-11/20:27:00
3	INDEX	CUSTOMERS_PK	SH	74152	73681	2011-10-11/20:26:54	2011-10-11/20:26:54
4	INDEX	CUSTOMERS_YOB_BIX	SH	74365	73906	2011-10-11/20:27:00	2011-10-11/20:27:00
5	INDEX	PRODUCTS_PK	SH	74144	73682	2011-10-11/20:26:52	2011-10-11/20:26:52
6	INDEX	PRODUCTS_PROD_CAT_IX	SH	74146	73903	2011-10-11/20:26:52	2011-10-11/20:26:52
7	INDEX	PRODUCTS_PROD_STATUS_BIX	SH	74362	73901	2011-10-11/20:27:00	2011-10-11/20:27:00
8	INDEX	PRODUCTS_PROD_SUBCAT_IX	SH	74145	73902	2011-10-11/20:26:52	2011-10-11/20:26:52
9	INDEX	SALES_CHANNEL_BIX	SH	74246		2011-10-11/20:26:58	2011-10-11/20:26:58
10	INDEX	SALES_CUST_BIX	SH	74188		2011-10-11/20:26:57	2011-10-11/20:26:57
11	INDEX	SALES_PROD_BIX	SH	74159		2011-10-11/20:26:57	2011-10-11/20:26:57
12	INDEX	SALES_PROMO_BIX	SH	74275		2011-10-11/20:26:59	2011-10-11/20:26:59
13	INDEX	SALES_TIME_BIX	SH	74217		2011-10-11/20:26:58	2011-10-11/20:26:58
14	TABLE	CUSTOMERS	SH	74151	73676	2011-10-11/20:26:54	2011-10-11/20:28:04
15	TABLE	PRODUCTS	SH	74143	73673	2011-10-11/20:26:52	2011-10-11/20:28:04
16	TABLE	SALES	SH	74083		2011-10-11/20:26:43	2011-10-11/20:28:04

Go to Top

Figure 2-16. *Object information and creation times*

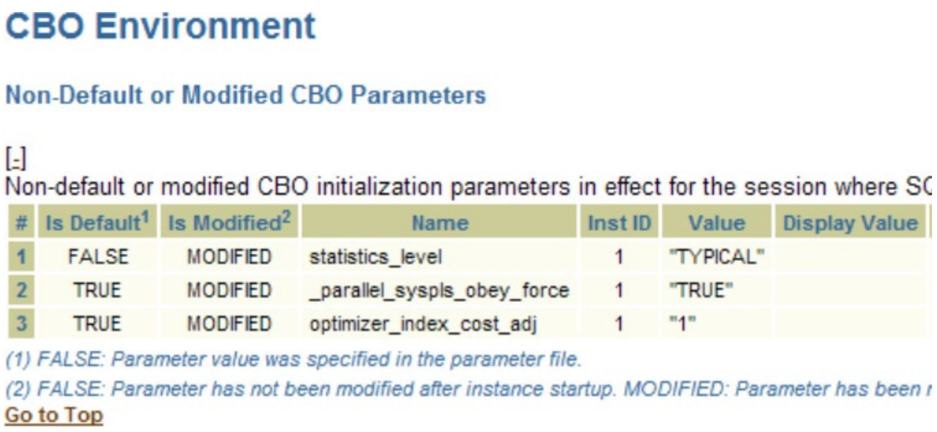
The objects section in Figure 2-16 will confirm the creation date of the index PRODUCTION_PROD_STATUS_BIX. As you can see, the index used in the BITMAP INDEX FULL SCAN was created long ago. So where are we now?

Let's review the facts:

- No new indexes have been added.
- The plan has changed—it uses more indexes.
- The statistics haven't changed.

Now you need to consider what else can change an execution plan. Here are some possibilities:

- System statistics. We check those, and they seem OK. Knowing what normal looks like helps here.
- Hints. We look to be sure. There are no hints in the SQL text.
- CBO parameters. We look and see the values in Figure 2-17.



CBO Environment

Non-Default or Modified CBO Parameters

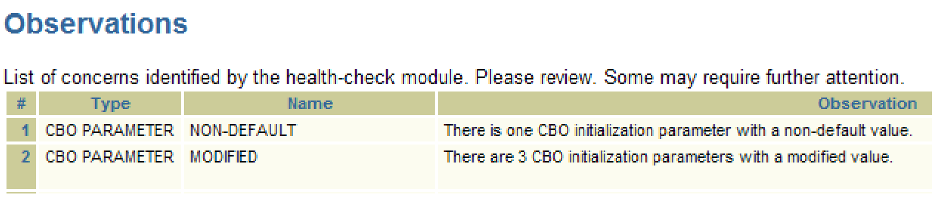
[-]
Non-default or modified CBO initialization parameters in effect for the session where SC

#	Is Default[1]	Is Modified[2]	Name	Inst ID	Value	Display Value
1	FALSE	MODIFIED	statistics_level	1	"TYPICAL"	
2	TRUE	MODIFIED	_parallel_syspls_obey_force	1	"TRUE"	
3	TRUE	MODIFIED	optimizer_index_cost_adj	1	"1"	

(1) FALSE: Parameter value was specified in the parameter file.
(2) FALSE: Parameter has not been modified after instance startup. MODIFIED: Parameter has been r
Go to Top

Figure 2-17. *The CBO environment section*

Figure 2-17 shows statistics_level, *_parallel*_syspls_obey_force, and optimizer_index_cost_adj. This is in the section “Non-Default CBO Parameters”, so you know they are not normal values. As optimizer_index_cost_adj is a parameter for adjusting the cost used by the optimizer for indexes this may have something to do with our change in execution plan. Then notice that the “Observations” section (see Figure 2-18) highlights that there are non-standard parameters.



Observations

List of concerns identified by the health-check module. Please review. Some may require further attention.

#	Type	Name	Observation
1	CBO PARAMETER	NON-DEFAULT	There is one CBO initialization parameter with a non-default value.
2	CBO PARAMETER	MODIFIED	There are 3 CBO initialization parameters with a modified value.

Figure 2-18. *The “Observations” section of the HTML report shows non-default parameter observations, in this case 1 non-default parameter*

If you look up optimizer_index_cost_adj, you will see that its default is 100 not 1. So now you have a working theory: The problem could lie with that parameter.

Now you can go to the users terminal, run his query, set the session value for optimizer_index_cost_adj to 100, re-run the query and see the different execution plan. We see the results below.

```
Execution Plan
----------------------------------------------------------
Plan hash value: 725901306

------------------------------------------------------------------------------------
| Id  | Operation               | Name       | Rows  | Bytes | Cost (%CPU)| Time     ||
------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT        |            |  5557 |   303K|   908   (3)| 00:00:11 ||
|*  1 |  HASH JOIN              |            |  5557 |   303K|   908   (3)| 00:00:11 ||
|   2 |   TABLE ACCESS FULL     | PRODUCTS   |    72 |  2160 |     3   (0)| 00:00:01 ||
|*  3 |   HASH JOIN             |            |  5557 |   141K|   904   (3)| 00:00:11 ||
|*  4 |    TABLE ACCESS FULL    | CUSTOMERS  |    43 |   516 |   405   (1)| 00:00:05 ||
|   5 |    PARTITION RANGE ALL|            |   918K|    12M|   494   (3)| 00:00:06 ||
|   6 |     TABLE ACCESS FULL | SALES      |   918K|    12M|   494   (3)| 00:00:06 ||
------------------------------------------------------------------------------------
```

```
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - access("S"."PROD_ID"="P"."PROD_ID")
   3 - access("C"."CUST_ID"="S"."CUST_ID")
   4 - filter("C"."CUST_FIRST_NAME"='Theodorick')
SQL> alter session set optimizer_index_cost_adj=1;
Session altered.
SQL> @q2
Execution Plan
Plan hash value: 665279032
-----------------------------------------------------------------------------------------------
| Id  | Operation                           | Name                       |Rows |Bytes|Cost(%CPU)|
|   0 | SELECT STATEMENT                    |                            | 5557| 303K|  78   (7)|
|*  1 |  HASH JOIN                          |                            | 5557| 303K|  78   (7)|
|   2 |   TABLE ACCESS BY INDEX ROWID       | PRODUCTS                   |   72|2160 |   1   (0)|
|   3 |    BITMAP CONVERSION TO ROWIDS      |                            |     |     |          |
|   4 |     BITMAP INDEX FULL SCAN          | PRODUCTS_PROD_STATUS_BIX |     |     |          |
|   5 |   NESTED LOOPS                      |                            |     |     |          |
|   6 |    NESTED LOOPS                     |                            | 5557| 141K|  76   (6)|
|*  7 |     TABLE ACCESS BY INDEX ROWID     | CUSTOMERS                  |   43| 516 |  26   (4)|
|   8 |      BITMAP CONVERSION TO ROWIDS    |                            |     |     |          |
|   9 |       BITMAP INDEX FULL SCAN        | CUSTOMERS_GENDER_BIX       |     |     |          |
|  10 |     PARTITION RANGE ALL             |                            |     |     |          |
|  11 |      BITMAP CONVERSION TO ROWIDS    |                            |     |     |          |
|* 12 |       BITMAP INDEX SINGLE VALUE     | SALES_CUST_BIX             |     |     |          |
|  13 |    TABLE ACCESS BY LOCAL INDEX ROWID| SALES                      |  130|1820 |  76   (6)|
Predicate Information (identified by operation id):
   1 - access("S"."PROD_ID"="P"."PROD_ID")
   7 - filter("C"."CUST_FIRST_NAME"='Theodorick')
  12 - access("C"."CUST_ID"="S"."CUST_ID")
```

Now we have a cause and a fix, which can be applied to just the one session or to the entire system.

The next step of course is to find out why `optimizer_index_cost_adj` was set, but that's a different story, involving the junior DBA (or at least hopefully not you!) who set the parameter at what he thought was session level but turned out to be system level.

Summary

In the chapter we learned about the inputs to the cost-based optimizer's algorithm and how these inputs affect the optimizer. For the optimizer to work effectively and efficiently these inputs need to be considered and set up in a way that reflects the usage of your environment and business requirements. SQLTXPLAIN helps with all of the environmental considerations by collecting information and displaying it in a way that is easy to understand. SQLTXPLAIN also helpfully highlights those elements that are out of the ordinary so that you can consider them more critically and decide if those elements are needed and are what you intended. In the next chapter we consider one of the most important aspects of the CBO environment, the object statistics. These take the most time to collect by far and are very frequently the cause of performance problems. We'll look at the effect of lack of statistics, poor timing of statistics, and other elements of this important maintenance job.

CHAPTER 3

How Object Statistics Can Make Your Execution Plan Wrong

In this chapter we'll discuss what is considered a very important subject if you are tuning SQL. Gathering object statistics is crucial; this is why Oracle has spent so much time and effort making the statistics collection process as easy and painless as possible. They know that under most circumstances, DBAs who are under pressure to get their work done as quickly as possible, in the tiny maintenance windows they are allowed, will opt for the easiest and simplest way forward. If there is a check box that says "click me, all your statistics worries will be over," they'll click it and move on to the next problem.

The automated procedure has, of course, improved over the years, and the latest algorithms for automatically collecting statistics on objects (and on columns especially) are very sophisticated. However, this does not mean you can ignore them. You need to pay attention to what's being collected and make sure it's appropriate for your data structures and queries. In this chapter we'll cover how partitions affect your plans and statistics capture. We'll also look at how to deal with sampling errors and how to lock statistics and when this should be done. If this sounds boring, then that's where SQLT steps in and makes the whole process simpler and quicker. Let's start with object statistics.

What Are Statistics?

When SQL performs badly, poor-quality statistics are the most common cause. Poor-quality statistics cover a wide range of possible deficiencies:

- Inadequate sample sizes.
- Infrequently collected samples.
- No samples on some objects.
- Collecting histograms when not needed.
- Not collecting histograms when needed.
- Collecting statistics at the wrong time.
- Collecting very small sample sizes on histograms.
- Not using more advanced options like extended statistics to set up correlation between related columns.
- Relying on auto sample collections and not checking what has been collected.

It is crucial to realize that the mission statement of the cost-based optimizer (CBO) is to develop an execution plan that runs fast and to develop it quickly. Let's break that down a little:

- "Develop quickly." The optimizer has very little time to parse or to get the statistics for the object, or to try quite a few variations in join methods, not to mention to check for SQL optimizations and develop what it considers a good plan. It can't spend a long time doing this, otherwise, working out the plan could take longer than doing the work.
- "Runs fast." Here, the key idea is that "wall clock time" is what's important. The CBO is not trying to minimize I/Os or CPU cycles, it's just trying to reduce the elapsed time. If you have multiple CPUs, and the CBO can use them effectively, it will choose a parallel execution plan.

Chapter 1 discussed cardinality, which is the number of rows that satisfy a predicate. This means that the cost of any operation is made up of three operation classes:

- Cost of single block reads
- Cost of multi-block reads
- Cost of the CPU used to do everything

When you see a cost of 1,000, what does that actually mean? An operation in the execution plan with a cost of 1,000 means that the time taken will be approximately the cost of doing 1,000 single-block reads. So in this case 1,000 x 12 ms, which gives 12 seconds (12 ms is a typical single- block read time).

So what steps does the CBO take to determine the best execution plan? In very broad terms the query is transformed (put in any shortcuts that are applicable), then plans are generated by looking at the size of the tables and deciding which table will be the inner and which will be the outer table in joins. Different join orders are tried and different access methods are tried. By "tried" I mean that the optimizer will go through a limited number of steps (its aim is to develop a plan quickly, remember) to calculate a cost for each of them and by a process of elimination get to the best plan. Sometimes the options the optimizer tries are not a complete list of plans, and this means it could miss the best plan; but this is extremely rare.

This is the estimation phase of the operation. If the operation being evaluated is a full table scan, this will be estimated based on the number of rows, the average length of the rows, the speed of the disk sub-system and so on.

Now that we know what the optimizer is doing to try and get you the right plan, we can look at what can go wrong when the object statistics are misleading.

Object Statistics

The main components comprising object statistics are tables and indexes. To simplify the discussion, we will mainly look at table statistics, but the same principles will apply to all objects. In the estimation phase of the hard parsing mentioned above, where the size of tables is estimated and joins are chosen, the number of rows in the table is crucial. A simple example would be a choice between a nested loop or a hash join. If the number of rows is wrong, then the join method may be wrong. Other ways in which statistics can be wrong is by being out of date. Let's look at the example in Figure 3-1. In this example we see "Table Statistics" for TABLE_A, a non-partitioned table with a Num Rows value of 87,116, which was 100 percent analyzed.

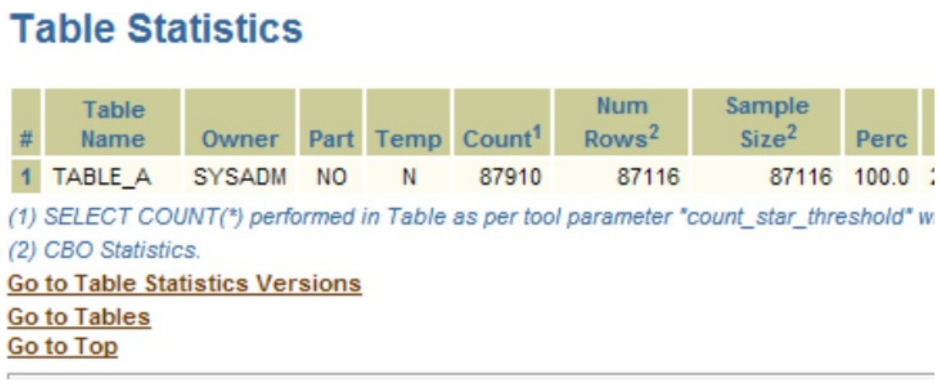


Table Statistics

#	Table Name	Owner	Part	Temp	Count[1]	Num Rows[2]	Sample Size[2]	Perc
1	TABLE_A	SYSADM	NO	N	87910	87116	87116	100.0

(1) SELECT COUNT(*) performed in Table as per tool parameter "count_star_threshold" w

(2) CBO Statistics.

Go to Table Statistics Versions

Go to Tables

Go to Top

Figure 3-1. *In the "Table Statistics" section you can see the number of rows in a table, the sample size, and the percentage of the total data set that this sample represents*

Object statistics are a vital input to the CBO, but even these can lead the optimizer astray when the statistics are out of date. The CBO also uses past execution history to determine if it needs better sampling (cardinality feedback) or makes use of bind peeking to determine which of many potential execution plans to use. Many people rely on setting everything on AUTO but this is not a panacea. If you don't understand and monitor what the auto settings are doing for you, you may not spot errors when they happen.

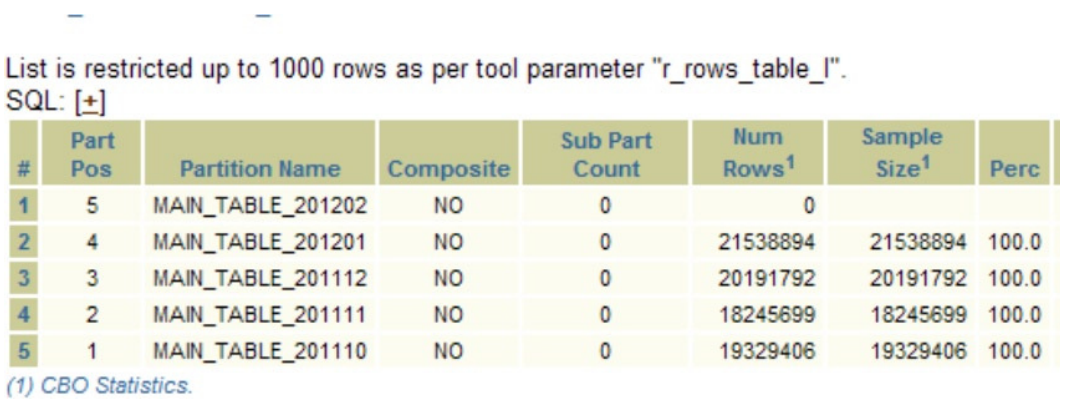
Just to clarify, in the very simplest terms, why does the optimizer get it wrong when the statistics are out of date? After all once you've collected all that statistical information about your tables, why collect it again? Let's do a thought experiment just like Einstein sitting in the trolley bus in Vienna.

Imagine you've been told there are few rows in a partition (<1,000 rows). You're probably going to do a full table scan and not use the index, but if your statistics are out of date and you've had a massive data load (say 2.5 million rows) since the last time they ran and all of them match your predicate, then your plan is going to be sub-optimal. That's tuning-speak for "regressed," which is also tuning-speak for "too slow for your manager." This underlines the importance of collecting statistics at the right time; after the data load, not before.

So far we've mentioned table statistics and how these statistics need to be of the right quality and of a timely nature. As data sets grew larger and larger over the years, so too did tables grow larger. Some individual tables became very large (Terabytes in size). This made handling these tables more difficult, purely because operations on these tables took longer. Oracle Corporation saw this trend early on and introduced table partitioning. These mini tables split a large table into smaller pieces partitioned by different keys. A common key is a date. So one partition of a table might cover 2012. This limited the size of these partitions and allowed operations to be carried out on individual partitions. This was a great innovation (that has a license cost associated with it) that simplified many day-to-day tasks for the DBA and allowed some great optimizer opportunities for SQL improvement. For example, if a partition is partitioned by date and you use a date in your predicate you might be able to use partition pruning, which only looks at the matching partitions. With this feature comes great opportunities for improvement in response times but also a greater possibility you'll get it wrong. Just like tables, partitions need to have good statistics gathered for them to work effectively.

Partitions

Partitions are a great way to deal with large data sets, especially ones that are growing constantly. Use a range partition, or even better, an interval partition. These tools allow "old" data and "new" data to be separated and treated differently, perhaps archiving old data or compressing it. Whatever the reason, many institutions use partitioning to organize their data. Especially with partitions based on date, it is common to have a new partition created which has zero or very few rows just after it is created (see Figure 3-2 for an example of this situation). The new partition MAIN_TABE_201202 has zero rows (it's only just been created), but the other partitions have millions.


List is restricted up to 1000 rows as per tool parameter "r_rows_table_l".
SQL: [+]

#	Part Pos	Partition Name	Composite	Sub Part Count	Num Rows[1]	Sample Size[1]	Perc
1	5	MAIN_TABLE_201202	NO	0	0		
2	4	MAIN_TABLE_201201	NO	0	21538894	21538894	100.0
3	3	MAIN_TABLE_201112	NO	0	20191792	20191792	100.0
4	2	MAIN_TABLE_201111	NO	0	18245699	18245699	100.0
5	1	MAIN_TABLE_201110	NO	0	19329406	19329406	100.0

(1) CBO Statistics.

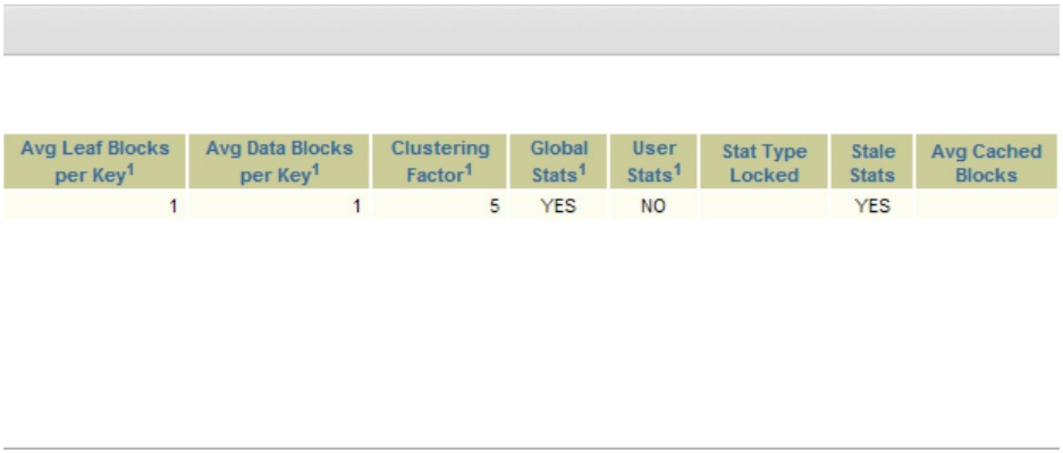
Figure 3-2. *A newly created partition will have different characteristics than an old partition. In this case, the number of partitions in MAIN_TABLE_201202 is zero*

For the recently finished partition, any new data added will not result in a huge change in the actual data versus the statistics, but for the newly created partition the optimizer currently thinks there are zero rows in this partition. How will this affect the execution plan if there are now 10,000 rows just a few hours after the statistics were collected? This kind of "initialization" of partitions can have a huge impact on execution plans, the kind of impact that comes and goes with the time of the month. Bad on the first of the month, then gradually better.

These types of situations require careful timing of statistics collections and good samples. During the time just after the creation of a new partition, there may be periods of time when a higher sample collection percentage of a partition is beneficial in order to collect any unusually skewed data in that time partition. Just being aware of what is (or could) be happening, is half the battle. If more statistics collections are not an option soon after the load, then you may have to resort to SQL Profiles or even hints.

Stale Statistics

From the point of view of SQLT if more than 10 percent of the data in a table has changed then the statistics are marked with a "YES" in the "Stale Stats" column. In the example below (Figure 3-3) you can see an index with stale statistics.



Avg Leaf Blocks per Key[1]	Avg Data Blocks per Key[1]	Clustering Factor[1]	Global Stats[1]	User Stats[1]	Stat Type Locked	Stale Stats	Avg Cached Blocks
1	1	5	YES	NO		YES	

Figure 3-3. *An index row is shown with "Stale Stats" as "YES" in this example, indicating stale statistics for this index*

What should you do with this information? That depends. Sometimes stale statistics are not a problem; it may be that even though the table has changed by more than 10 percent, the data could still be the same statistically. Ten percent is a pretty arbitrary number, and in some situations you may decide that it's too low or too high. If so, you can always change it

```
SQL>  exec dbms_stats.set_table_prefs(null,'USER2','STALE_PERCENT',5)
```

Having said that, what should you do if you see "YES" in the Stale Stats column for a particular table or index? The simple answer is (once again) "it depends." It's like a murder mystery. If you already suspect that the butler did it, and then you see he has a gun in his pocket, that might be a good lead.

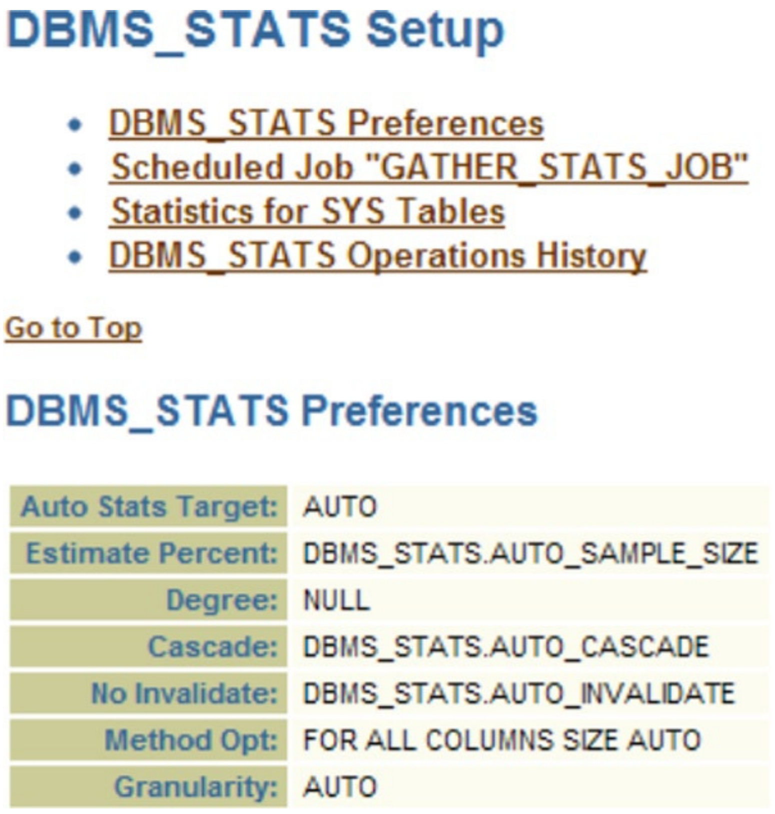
The point here is that the column would be a clue that something was wrong, and that it might be materially important in your current investigation. For example, suppose you have a table with an interval partition created and loaded with data every day. Its statistics are going to be STALE unless you run statistics collection every day after you load data in a new partition. The key is to realize that statistics collection is a tool provided by Oracle that you can use in many different ways. That control must be informed by your knowledge of the architecture, the data model, and the queries you expect on your database.

Sampling Size

This is one of the most hotly argued points of statistics collection. Many sites depend on the `DBMS_STATS.AUTO_SAMPLE_SIZE`, and this is a good choice for some if not most situations, but it's not perfect. It's just a good starting point. If you have no problems with your queries, and your statistics run in a reasonable time, then leave the defaults and go do something more useful.

If you struggle to collect the level of statistics you need to get your online day working efficiently, you may want to look closely at the statistics you are gathering. If 100 percent is not possible, then work out how much time you do have, and try to aim for that percentage. Look at what you are collecting and see if all of it is needed. If you are gathering schema stats, then ask yourself: Are all the tables in the schema being used? Are you collecting statistics on columns with no skewness? In such situations you are wasting resources collecting information that will make no difference to the optimizer. It stands to reason that changing statistics collection sample sizes should be done carefully and in small steps, with enough time between changes to evaluate the effect. This is what test systems are for.

You can see what you have set up on the system by clicking on the "DBMS_STATS" setup hyperlink from the SQLT report (Figure 3-4).


DBMS_STATS Setup

- DBMS_STATS Preferences
- Scheduled Job "GATHER_STATS_JOB"
- Statistics for SYS Tables
- DBMS_STATS Operations History

Go to Top

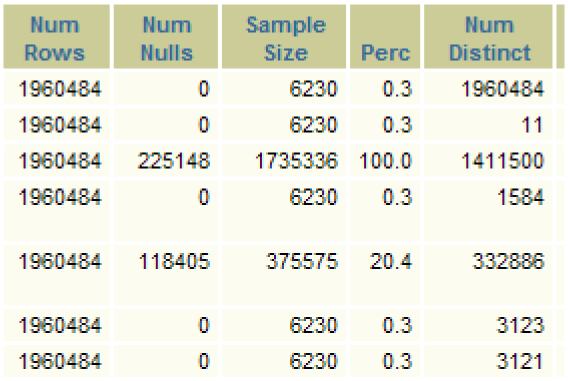
DBMS_STATS Preferences

Auto Stats Target:	AUTO
Estimate Percent:	DBMS_STATS.AUTO_SAMPLE_SIZE
Degree:	NULL
Cascade:	DBMS_STATS.AUTO_CASCADE
No Invalidate:	DBMS_STATS.AUTO_INVALIDATE
Method Opt:	FOR ALL COLUMNS SIZE AUTO
Granularity:	AUTO

Figure 3-4. *It's always good to check what DBMS_STATS preferences you have. In this example everything is set to the defaults*

Here you can see the DBMS_STATS preferences. Everything is set to AUTO. The sample size is DBMS_STATS.AUTO_SAMPLE_SIZE; Cascade is set to DBMS_STATS.AUTO_CASCADE; and Method Opt is set to FOR ALL COLUMNS SIZE AUTO.

Do any of these settings cause problems? Generally not. Sometimes, however, the sample sizes can be very small. This is a deliberate attempt by the algorithm designers to reduce the amount of time spent gathering statistics on these columns. If too many samples are similar, then the algorithm will decide that there is no point in looking any further and will finish collecting for the column. If from the top of the SQLT report we click on "columns" we are able to see the "Table Columns" area of the report. Here we can see column statistics. Of special interest are the sample sizes and the percentage that these represent (about halfway across the report). See Figure 3-5 for an example of this area (I've only shown columns 8 to 12 of the report) Notice that some sample sizes are extremely small.



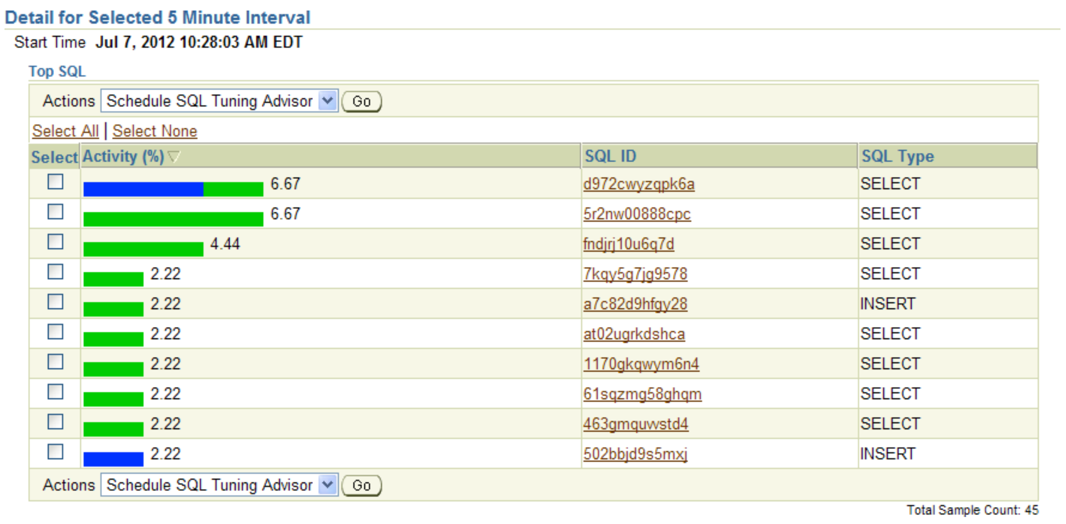
Num Rows	Num Nulls	Sample Size	Perc	Num Distinct
1960484	0	6230	0.3	1960484
1960484	0	6230	0.3	11
1960484	225148	1735336	100.0	1411500
1960484	0	6230	0.3	1584
1960484	118405	375575	20.4	332886
1960484	0	6230	0.3	3123
1960484	0	6230	0.3	3121

Figure 3-5. *In this section of the "Column Statistics" example (found by clicking on the "Columns" hyperlink from the top of the report) the auto sampling size has collected very small sample sizes of 0.3 percent*

So how do we get such small sample sizes? Imagine organizing your socks. You're a real fan of socks, so you have ten drawers full of socks. You want to know how many types of socks you have. You look in the first drawer and pick a pair of socks at random. Black. The second pair is also black, and the third. Do you keep going? Or do you now assume all the socks are black? You see the problem. The algorithm is trying to be efficient and will sample randomly, but if it is unlucky it may get too many similar samples and give up. What if there are millions of socks and they are all black except one pair (gold with silver stripes). You probably will not find that pair in your random sample. But suppose further that you love those gold-and-silvery striped socks and you want to wear them every day. In this case you will always do a full table scan of your sock drawer (as you think most socks are black and you think that all colored socks you look for will have the same distribution). This sounds counterintuitive, but you need to see that the optimizer has no idea about your actual data distribution in that column, it only has a sample (and a very small one). It then applies the rules it has worked out for that drawer for all sock searches. In fact your sock color distribution is highly skewed, and the one rare sock pair is the one you want to query all the time. Data is exactly the same. Random samples may not pick up the rare skewed values and if these are popular in queries, you may need to adjust your column sample size.

How to Gather Statistics

SQLT can help you come up with a methodology to use when creating a plan for gathering statistics. For pre-production environments with a realistic workload you'll find it sensible to run a SQLT report on key SQL statements (the ones performing badly) and look at the overall performance. Don't rely on the overall runtime alone to decide if your workload is going to be OK. Choose the top SQLs, as identified by AWR or SQL Tuning Advisor. You can even pick them off Enterprise Manager (see Figure 3-6).



Detail for Selected 5 Minute Interval

Start Time **Jul 7, 2012 10:28:03 AM EDT**

Top SQL

Actions: Schedule SQL Tuning Advisor — Go

Select All | Select None

Select	Activity (%)	SQL ID	SQL Type
☐	6.67	d972cwyzqpk6a	SELECT
☐	6.67	5r2nw00888cpc	SELECT
☐	4.44	fndjrj10u6q7d	SELECT
☐	2.22	7kqy5g7jg9578	SELECT
☐	2.22	a7c82d9hfgy28	INSERT
☐	2.22	at02ugrkdshca	SELECT
☐	2.22	1170gkqwym6n4	SELECT
☐	2.22	61sqzmg58ghqm	SELECT
☐	2.22	463gmquwstd4	SELECT
☐	2.22	502bbjd9s5mxj	INSERT

Actions: Schedule SQL Tuning Advisor — Go

Total Sample Count: 45

Figure 3-6. *You can even select the Top SQL to investigate from OEM*

Then run a SQLT XTRACT report and look at the statistics for these SQLs. Remember, you've picked the heavy hitters on the system so you need to tune these SQLs' statistics gathering to improve their performance (if possible) while having no adverse effect on other queries. If you can see small sample sizes, *which are adversely affecting performance*, then you have a candidate for statistics collection amendments.

In 11g you can set preferences for statistics gathering at many granular levels; at the database level (`dbms_stats.set_database_prefs`) the schema level (`dbms_stats.set_schema_prefs`), or the table level (`dbms_stats.set_table_prefs`). Don't just increase the percentage blindly because performance is bad. Look at the SQL that performs badly and tune those SQL statements. Turning up all statistics to 100 percent will just eat up resources and in some cases will not collect enough statistics on columns. This is why it is important to look at what is actually happening.

Saving and Restoring and Locking Statistics

Saving and restoring statistics can be extremely useful. In an earlier example, you saw that collecting statistics at the wrong time could break your execution plan. If you collected statistics when a table was empty you need to make another statistics collection when the table has a representative data set in it. On the other hand, if you collect statistics when the data in the table are representative, you can save that collection and later restore it for that table. This should allow you to represent the newly loaded table correctly, while collecting statistics at a convenient time.

You can also lock statistics from a convenient time.

```
begin
dbms_stats.lock_table_stats(ownname=> 'STELIOS', tabname=> 'TEST2' );
end;
```

Figure 3-7. *If your table statistics are very volatile, you may be better off locking them*

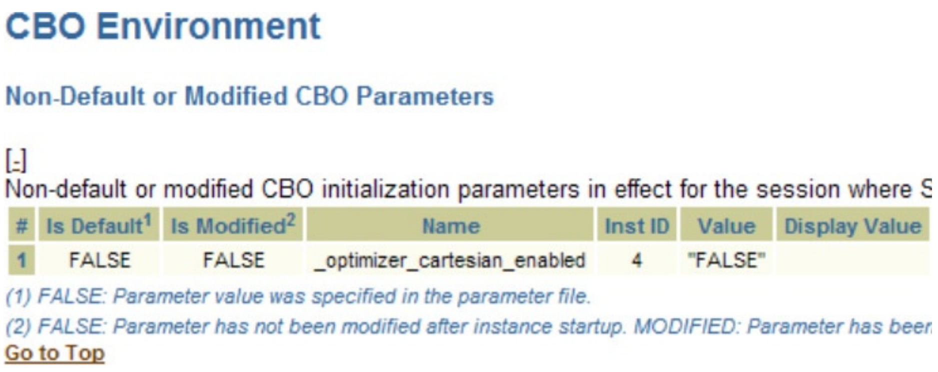
Here is an example sequence to collect statistics on an object and save them to a table called MYSTATS2.

```
SQL> create table test2 as select object_id from dba_objects;

Table created.

SQL> exec dbms_stats.gather_table_stats('STELIOS','TEST2');

PL/SQL procedure successfully completed.

SQL> exec dbms_stats.create_stat_table('STELIOS','MYSTATS2');

PL/SQL procedure successfully completed.

SQL> delete from test2;

73419 rows deleted.

SQL> commit;

Commit complete.

SQL> select count(*) from test2;

  COUNT(*)
----------
         0

SQL> exec dbms_stats.import_table_stats('STELIOS','TEST2',null,'MYSTATS2');

PL/SQL procedure successfully completed.

SQL>
```

These simple steps are all that is required to save statistics for later use. This is what SQLT does for you when you collect information about an SQL statement. The statistics for the objects are collected together and placed into the ZIP file so you can build test cases where a query will work as if it were on production but requires no data. We'll look at creating a test case in Chapter 11. It is one of the most useful tools you get with SQLT.

The Case of the Midnight Truncate

Time to look at the second case in our detective series. You're new on site and see a query with lots of hints. When you ask about it and why it has so many hints, you're told that the execution plan is wrong if the hints are removed. You also notice that cartesian joins have been disabled with `_optimizer_cartesian_enabled=FALSE` (see Figure 3-8).



CBO Environment

Non-Default or Modified CBO Parameters

[-]
Non-default or modified CBO initialization parameters in effect for the session where S

#	Is Default[1]	Is Modified[2]	Name	Inst ID	Value	Display Value
1	FALSE	FALSE	_optimizer_cartesian_enabled	4	"FALSE"	

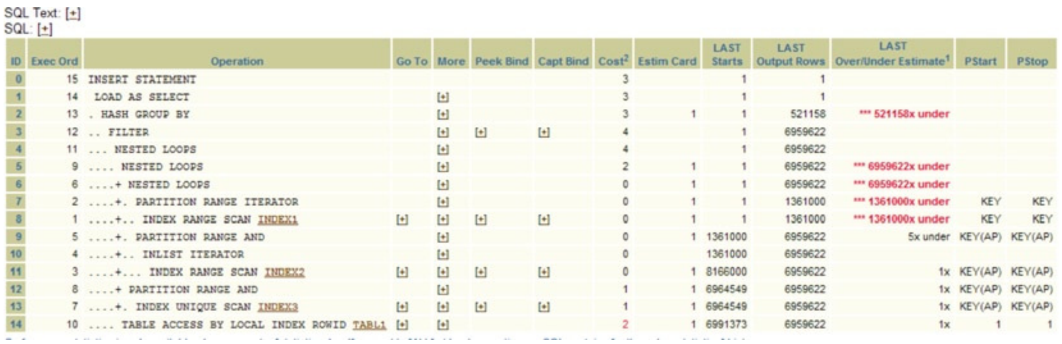
(1) FALSE: Parameter value was specified in the parameter file.
(2) FALSE: Parameter has not been modified after instance startup. MODIFIED: Parameter has beer

Go to Top

Figure 3-8. *The CBO Environment section can often reveal "odd" parameter settings*

Now I'm not suggesting that you leap into every situation and remove hints without thinking, but SQLs with hints are sometimes a sign of something else that is going wrong. Usually the culprit is Moriarty, I mean statistics, of course! Don't be prejudiced, always go with the evidence, so the first thing to do is collect some. In a duplicate environment you run the query without hints.

First look at the execution plan, specifically the one in memory (see Figure 3-9).



SQL Text: [+]
SQL: [+]

ID	Exec Ord	Operation	Go To	More	Peek Bind	Capt Bind	Cost[2]	Estim Card	LAST Starts	LAST Output Rows	LAST Over/Under Estimate[1]	PStart	PStop
0	15	INSERT STATEMENT					3		1	1			
1	14	LOAD AS SELECT		[+]			3		1	1			
2	13	. HASH GROUP BY		[+]			3	1	1	521158	*** 521158x under		
3	12	.. FILTER		[+]	[+]	[+]	4		1	6959622			
4	11	... NESTED LOOPS		[+]			4		1	6959622			
5	9	.... NESTED LOOPS		[+]			2	1	1	6959622	*** 6959622x under		
6	6	....+ NESTED LOOPS		[+]			0	1	1	6959622	*** 6959622x under		
7	2	....+. PARTITION RANGE ITERATOR		[+]			0	1	1	1361000	*** 1361000x under	KEY	KEY
8	1	....+.. INDEX RANGE SCAN INDEX1	[+]	[+]	[+]	[+]	0	1	1	1361000	*** 1361000x under	KEY	KEY
9	5	....+. PARTITION RANGE AND		[+]			0	1	1361000	6959622	5x under	KEY(AP)	KEY(AP)
10	4	....+.. INLIST ITERATOR		[+]			0		1361000	6959622			
11	3	....+... INDEX RANGE SCAN INDEX2	[+]	[+]	[+]	[+]	0	1	8166000	6959622	1x	KEY(AP)	KEY(AP)
12	8	....+ PARTITION RANGE AND		[+]			1	1	6964549	6959622	1x	KEY(AP)	KEY(AP)
13	7	....+. INDEX UNIQUE SCAN INDEX3	[+]	[+]	[+]	[+]	1	1	6964549	6959622	1x	KEY(AP)	KEY(AP)
14	10	.... TABLE ACCESS BY LOCAL INDEX ROWID TABL1	[+]	[+]			2	1	6991373	6959622	1x	1	1

Figure 3-9. *This execution plan has a glaring problem (highlighted in red no less)*

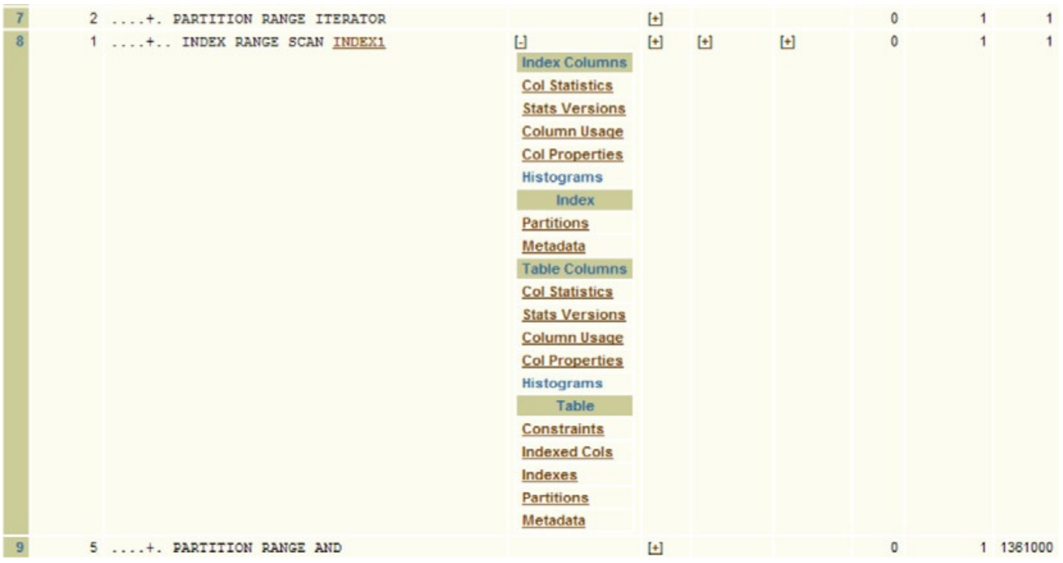
Luckily SQLT has done all the work for you. Following a SQLT XECUTE you saw that in the current execution plan, on Execution Step 1, the `INDEX RANGE SCAN`, the optimizer expected a cardinality of 1. (To see this look at the Exec Ord column, go down until you find "1". This is the first line to be executed.)

Then read across until you get to the Estim Card column. Here you'll see a value of "1". But as this was a SQLT XECUTE the SQL was executed, and the actual number of rows returned was greater than a million. The 1 row was expected, but there were actually 1 million rows. This is a big clue that something is wrong. The question at this point is "Why does the optimizer think the cardinality is going to be 1"?

Look at the Cost column. You will see that the cost is expected to be 0. The optimizer thinks there is no cost in retrieving this data.

To continue following the evidence-based trail you have to know the situation with the index statistics.

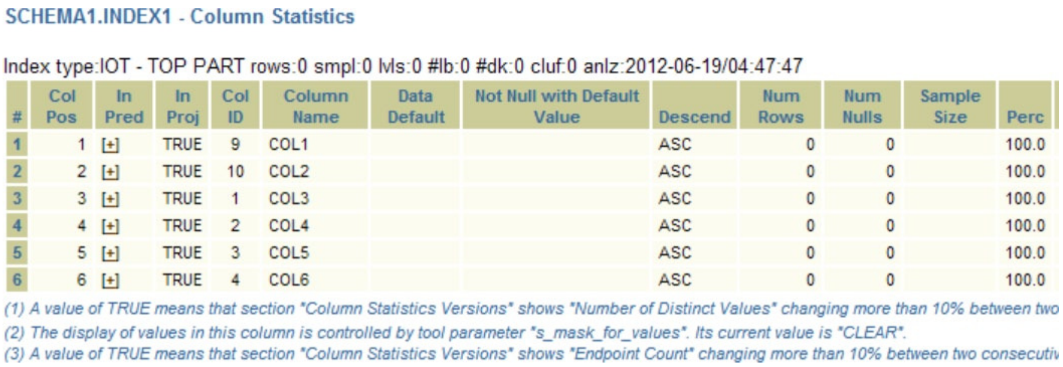
Take a look at the index. You can expand the button in the Go To column to display more links (see Figure 3-10).



7	2	....+. PARTITION RANGE ITERATOR		[+]				0	1	1
8	1	....+.. INDEX RANGE SCAN INDEX1	[-]	[+]	[+]		[+]	0	1	1
			Index Columns							
			Col Statistics							
			Stats Versions							
			Column Usage							
			Col Properties							
			Histograms							
			Index							
			Partitions							
			Metadata							
			Table Columns							
			Col Statistics							
			Stats Versions							
			Column Usage							
			Col Properties							
			Histograms							
			Table							
			Constraints							
			Indexed Cols							
			Indexes							
			Partitions							
			Metadata							
9	5	....+. PARTITION RANGE AND		[+]				0	1	1361000

Figure 3-10. *An expansion under the "more" column in an execution plan can show more links to other information*

Then click on Col Statistics under either the "Index Columns" or "Table Columns" heading to display the Column Statistics (see Figure 3-11).



SCHEMA1.INDEX1 - Column Statistics

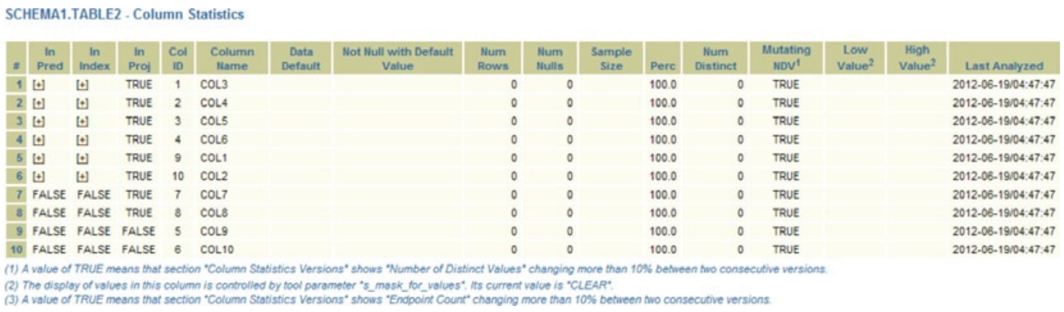
Index type:IOT - TOP PART rows:0 smpl:0 lvls:0 #lb:0 #dk:0 cluf:0 anlz:2012-06-19/04:47:47

#	Col Pos	In Pred	In Proj	Col ID	Column Name	Data Default	Not Null with Default Value	Descend	Num Rows	Num Nulls	Sample Size	Perc
1	1	[+]	TRUE	9	COL1			ASC	0	0		100.0
2	2	[+]	TRUE	10	COL2			ASC	0	0		100.0
3	3	[+]	TRUE	1	COL3			ASC	0	0		100.0
4	4	[+]	TRUE	2	COL4			ASC	0	0		100.0
5	5	[+]	TRUE	3	COL5			ASC	0	0		100.0
6	6	[+]	TRUE	4	COL6			ASC	0	0		100.0

(1) A value of TRUE means that section "Column Statistics Versions" shows "Number of Distinct Values" changing more than 10% between two
(2) The display of values in this column is controlled by tool parameter "s_mask_for_values". Its current value is "CLEAR".
(3) A value of TRUE means that section "Column Statistics Versions" shows "Endpoint Count" changing more than 10% between two consecutiv

Figure 3-11. *Column statistics reveal no rows*

Here you can see an interesting situation. The number of rows is 0 for all the columns. This means the table was empty when the statistics were collected. So, when were the statistics collected on the parent table? For that you need the parent table name. Click the back button to get to the execution plan again, and this time click on Col Statistics under the Table Columns section to display the table statistics (see Figure 3-12).



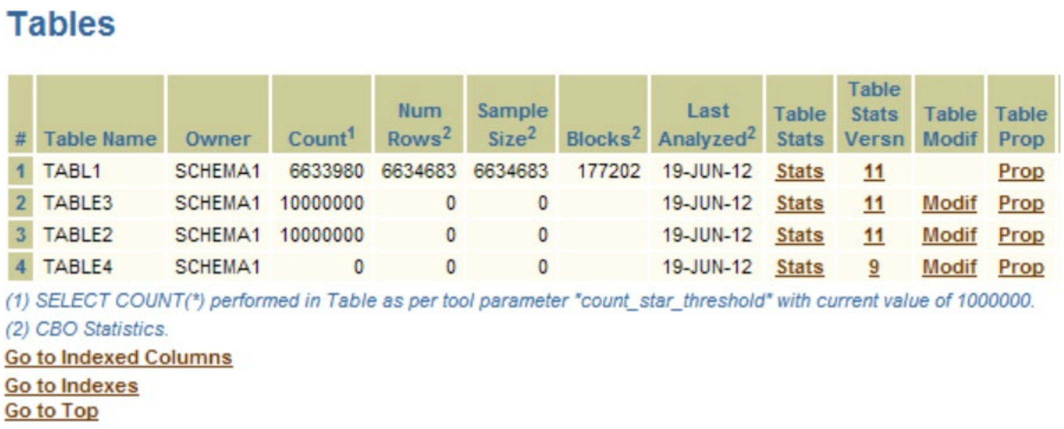
SCHEMA1.TABLE2 - Column Statistics

#	In Pred	In Index	In Proj	Col ID	Column Name	Data Default	Not Null with Default Value	Num Rows	Num Nulls	Sample Size	Perc	Num Distinct	Mutating NDV[1]	Low Value[2]	High Value[2]	Last Analyzed
1	[+]	[+]	TRUE	1	COL3			0	0		100.0	0	TRUE			2012-06-19/04:47:47
2	[+]	[+]	TRUE	2	COL4			0	0		100.0	0	TRUE			2012-06-19/04:47:47
3	[+]	[+]	TRUE	3	COL5			0	0		100.0	0	TRUE			2012-06-19/04:47:47
4	[+]	[+]	TRUE	4	COL6			0	0		100.0	0	TRUE			2012-06-19/04:47:47
5	[+]	[+]	TRUE	9	COL1			0	0		100.0	0	TRUE			2012-06-19/04:47:47
6	[+]	[+]	TRUE	10	COL2			0	0		100.0	0	TRUE			2012-06-19/04:47:47
7	FALSE	FALSE	TRUE	7	COL7			0	0		100.0	0	TRUE			2012-06-19/04:47:47
8	FALSE	FALSE	TRUE	8	COL8			0	0		100.0	0	TRUE			2012-06-19/04:47:47
9	FALSE	FALSE	FALSE	5	COL9			0	0		100.0	0	TRUE			2012-06-19/04:47:47
10	FALSE	FALSE	FALSE	6	COL10			0	0		100.0	0	TRUE			2012-06-19/04:47:47

(1) A value of TRUE means that section "Column Statistics Versions" shows "Number of Distinct Values" changing more than 10% between two consecutive versions.
(2) The display of values in this column is controlled by tool parameter "s_mask_for_values". Its current value is "CLEAR".
(3) A value of TRUE means that section "Column Statistics Versions" shows "Endpoint Count" changing more than 10% between two consecutive versions.

Figure 3-12. *Table statistics reveal no rows at the time of statistics collection*

You will see that 100 percent statistics were collected on TABLE 2, that there were no rows in the table, and that the collection happened at 04:47 in the morning. This is a fairly typical time for statistics collection to happen. Now you have to ask where the data came from and if there were 0 rows in the table at 4am. For this, click on the "Table" hyperlink from the main page to display the table details (see Figure 3-13).



Tables

#	Table Name	Owner	Count[1]	Num Rows[2]	Sample Size[2]	Blocks[2]	Last Analyzed[2]	Table Stats	Table Stats Versn	Table Modif	Table Prop
1	TABL1	SCHEMA1	6633980	6634683	6634683	177202	19-JUN-12	Stats	11		Prop
2	TABLE3	SCHEMA1	10000000	0	0		19-JUN-12	Stats	11	Modif	Prop
3	TABLE2	SCHEMA1	10000000	0	0		19-JUN-12	Stats	11	Modif	Prop
4	TABLE4	SCHEMA1	0	0	0		19-JUN-12	Stats	9	Modif	Prop

(1) SELECT COUNT(*) performed in Table as per tool parameter "count_star_threshold" with current value of 1000000.
(2) CBO Statistics.
Go to Indexed Columns
Go to Indexes
Go to Top

Figure 3-13. *The table details section shows where we can go for modification information*

This shows a new column of interest, the Table Modif column. Click on the "Modif" hyperlink for TABLE 2 to see the modification history (see Figure 3-14).

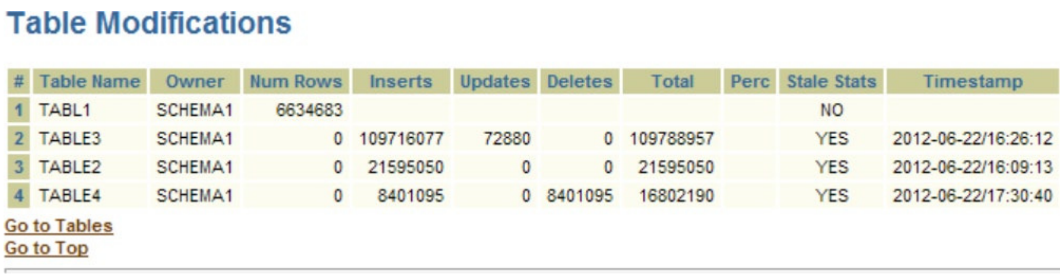


Table Modifications

#	Table Name	Owner	Num Rows	Inserts	Updates	Deletes	Total	Perc	Stale Stats	Timestamp
1	TABL1	SCHEMA1	6634683						NO	
2	TABLE3	SCHEMA1	0	109716077	72880	0	109788957		YES	2012-06-22/16:26:12
3	TABLE2	SCHEMA1	0	21595050	0	0	21595050		YES	2012-06-22/16:09:13
4	TABLE4	SCHEMA1	0	8401095	0	8401095	16802190		YES	2012-06-22/17:30:40

Go to Tables
Go to Top

Figure 3-14. *The table modifications section shows rows being inserted, deleted, and updated*

■ **Note** I know what you're thinking. SQLT is the utility that just keeps on giving. I know all this information is in the database and can be obtained with suitable queries, but SQLT has done that for you already. You don't have to go back to the database to find it. SQLT collected it, just in case.

Now you can see the modifications to the tables, with time-stamps. Read across the row for the TABLE 2 entry. There were zero rows in the table, then roughly 2 million rows were inserted, there were no updates and no deletes, which resulted in roughly 2 million rows in the table. The statistics became stale because of this (more than a 10 percent change), and this happened at 4:09 pm.

The steps are now clear. The table was empty. Statistics were collected. Two million rows were added, the query was run, the optimizer estimated a cardinality of 1 (remember it rounds up to 1 if the value would be 0), and hence calculated that NESTED LOOPS (or even worse CARTESIAN JOINS) would be fine and got it completely wrong.

Now that you know what happened you can fix the problem (if it seems appropriate), in a few different ways:

- You could collect statistics at a different time.
- You could export and import statistics as appropriate.
- You could freeze the statistics for when the table is populated.

There are probably even more ways to do this. The solution is not the issue here. The point is that once you know what happened you can design a solution that suits your circumstances. You can also see why in this case the `_optimizer_cartesian_enabled=FALSE` was needed. It was a possibility that the optimizer may have chosen a cartesian join because one of the key steps would have a cardinality of 1 (one of the few cases where a CARTESIAN JOIN makes sense). You can test your theory very simply. With TABLE 2 fully loaded, collect statistics. Then retry the query, or even better, just check what the execution plan would be.

The optimizer had bad statistics and "decided" to do something that didn't seem to make sense. SQLT collects enough information to let you see why the optimizer did what it did, and then you can set up statistics collection so that the optimizer doesn't get it wrong. The optimizer is your friend, an awesome piece of code that is nearly always right, as long as it gets the information it needs.

Summary

In this chapter we saw clearly from practical examples that statistics make a real difference and that just leaving everything on automatic pilot is not always the best strategy. You need to pay attention to what statistics you are collecting, the quality of those statistics, and how often you are collecting them. Take special care with column histograms. SQLT helps you to do this quickly and efficiently and gives you more information than a pile of scripts ever could. We touched on skewness in this chapter; but in the next chapter we'll dive right into the detail and look at what skewness is and how it affects your execution plan.

CHAPTER 4

How Skewness Can Make Your Execution Times Variable

Skewness is one of those topics that for some reason is often ignored, or misunderstood. Skewness is an important part of data handling for the cost-based optimizer. It makes a difference as to what the optimizer decides to do. It is so important that Oracle created new features such as Adaptive Cursor Sharing just to handle the consequences of skewness.

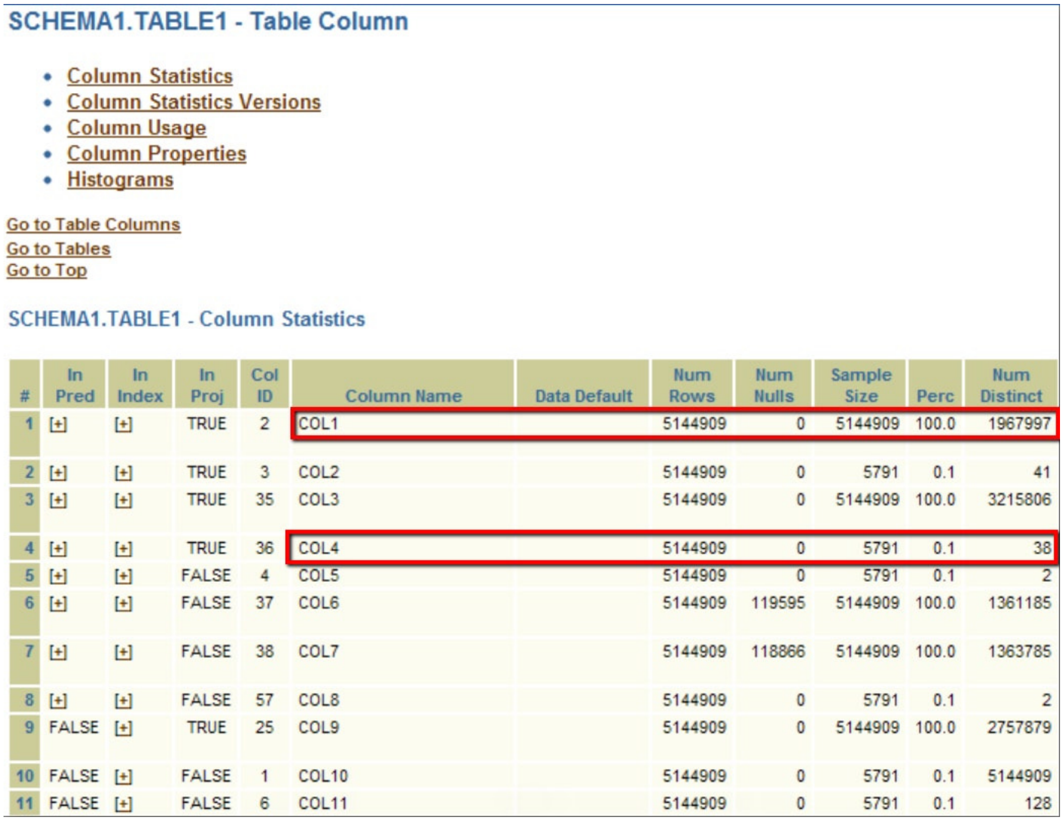
Skewness

In this section we are going to look at what skewness is, how it comes about, and how it affects your execution plan. If you handle skewness every day, you're likely to know immediately what kind of situations are likely to produce skewed data. Sometimes skewed data can arise unexpectedly as well. We'll also look at how to detect skewness, both from SQL and from SQLT.

What Is Skewness?

In the simplest terms possible a table is a representation of data and skewness is a property of a data set that results in an unexpectedly large or unexpectedly low number of matches for a particular column predicate. For example, if we plotted the amount in dollars spent at a hardware store for June, July and August we might find that the amount spent varied by only 10 percent. The variability per week would be very low. If we now extend our timeline to include Thanksgiving and Christmas, we would find that the amount spent peaks around those dates, so the amount could be twice or three times the "normal," June–August amount. If we only had June–August to predict the future, we might expect a much lower level of spending. This unexpectedly large spending around Thanksgiving and Christmas would give a skewed data set. If we now instead plotted the amount each person spent per visit, in groupings of $10 we would find another distribution. A few people spend in the $0– $10 range, perhaps more in the $10.01 to $20 range. The trend would continue up to some peak and then fall off. Most people would spend in some specific range, and then we would have the "high rollers," who spend thousands. The graph of the amount spent vs. the number of people who spent in this range will be skewed. If everybody came to the mall and spent exactly the same amount, then everybody would fall into the same range. All the other ranges would have a value of zero. This would be highly skewed data set.

This highlights a crucial point that DBAs and developer should constantly be on the alert for. Do you really truly understand your data. Sometimes it can be hard because the columns are not named in a way that is obvious. This is where database designers, developers, and DBAs must work together to get good structures, relevant indexes, and primary and foreign keys. In some databases you'll find that if sampling is set to automatic, you'll find column statistics are collected (sometimes with very small sample sizes) but that do no good because there is no skewness. In Figure 4-1 we see the information about what statistics were collected for each of the columns in the table TABLE 1.



SCHEMA1.TABLE1 - Table Column

- Column Statistics
- Column Statistics Versions
- Column Usage
- Column Properties
- Histograms

Go to Table Columns
Go to Tables
Go to Top

SCHEMA1.TABLE1 - Column Statistics

#	In Pred	In Index	In Proj	Col ID	Column Name	Data Default	Num Rows	Num Nulls	Sample Size	Perc	Num Distinct
1	[+]	[+]	TRUE	2	COL1		5144909	0	5144909	100.0	1967997
2	[+]	[+]	TRUE	3	COL2		5144909	0	5791	0.1	41
3	[+]	[+]	TRUE	35	COL3		5144909	0	5144909	100.0	3215806
4	[+]	[+]	TRUE	36	COL4		5144909	0	5791	0.1	38
5	[+]	[+]	FALSE	4	COL5		5144909	0	5791	0.1	2
6	[+]	[+]	FALSE	37	COL6		5144909	119595	5144909	100.0	1361185
7	[+]	[+]	FALSE	38	COL7		5144909	118866	5144909	100.0	1363785
8	[+]	[+]	FALSE	57	COL8		5144909	0	5791	0.1	2
9	FALSE	[+]	TRUE	25	COL9		5144909	0	5144909	100.0	2757879
10	FALSE	[+]	FALSE	1	COL10		5144909	0	5791	0.1	5144909
11	FALSE	[+]	FALSE	6	COL11		5144909	0	5791	0.1	128

Figure 4-1. *Table column statistics showing the sample size and number of distinct values*

In the figure for the SQLT report collected, we see, under the "Table Column" section, for the particular table TABLE 1, that a number of columns have had column statistics collected. Look at the highlighted section for COL1. It had 100 percent statistics collected. COL4 had 0.1 percent collected. Why is there such a vast difference between the COL1 and COL4? This is because the automatic sampling algorithm, chose only 5,791 rows for its COL4 sample size. Let me explain what the optimizer does when it samples these columns.

In this case, for table TABLE 1, for **each** column COL1, COL2, COL3, COL4, etc., the statistics gathering algorithm is sampling the values from the table. The sample size is shown in the "Sample Size" column. So if COL1 was the amount spent, then the statistics sampling algorithm would sample the columns and find there was some distribution of data (as we discussed above). If COL4 was the store name it would sample the data and find there were a certain number of distinct values (38 in this case). In each case, however, there are 5,144,909 rows of data. It's not documented what the algorithm does exactly to decide on the sample size, but it's pretty clear that if the first sample was "Joe's Hardware" and the second sample was "Joe's Hardware" and the third sample was "Joe's Hardware" the statistics-gathering algorithm might begin to "guess" after the one-thousandth sample that all the values it samples are statistically very close together. If your data is highly skewed and there was only one value in the entire column population of, say, "Sally's Hardware" inserted in COL4, then by random sampling the statistics collection algorithm might miss this value. Suppose further that despite the fact that the "Sally's Hardware" value is very rare, it is the value used in the predicate used for this query. In Oracle 10g, you might well end up with a sub-optimal plan depending on the first value used for the predicate. In 11g you would have Adaptive Cursor Sharing to help you (we discuss this more in Chapter 7). If on the other hand, however, you knew that your data was highly skewed in this way, and you saw that your auto sample size was tiny, you might very well decide that choosing a larger sample size might be a good idea.

How to Tell If Data Is Skewed

Skewness is so important that as a DBA or developer you should be on the alert for columns that might contain skewed data. When data sets are as large as they are today, it is very difficult to immediately tell if a data set is skewed. SQLTXPLAIN makes the job much easier for you by showing the statistics for the histograms, if there are any. So how do you find out if data is skewed? There's more than one way of seeing the collected data, but all methods rely on sampling. The simplest way to sample is with a query (as shown below). In this simple example we create a test table `test3` with a CTAS (Create Table As Select) statement from a known (at least to me) skewed data set. The `object_type` column of the `dba_objects` view. The `dba_objects` view lists attributes of the objects in the database, including the object type. There are many different object types: for example, `table` (this is a very common object type) and `dimension` (this is a very rare object type). Once we've created a test table called `test3`, which will contain 73,583 rows, we can sample the number of each type of object: this will give us an idea of the data set's skewness.

```
SQL> create table test3 (object_type) as select object_type from dba_objects;
Table created.
SQL> select object_type, count(object_type)
  from test3 group by object_type order by count(object_type);

OBJECT_TYPE         COUNT(OBJECT_TYPE)
------------------- ------------------
RULE                                 1
DATABASE LINK                        1
LOB PARTITION                        1
EDITION                              1
DESTINATION                          2
JAVA SOURCE                          2
SCHEDULE                             3
MATERIALIZED VIEW                    3
SCHEDULER GROUP                      4
DIMENSION                            5
CONTEXT                              7
UNDEFINED                            9
INDEXTYPE                            9
WINDOW                               9
CLUSTER                             10
RESOURCE PLAN                       10
JOB CLASS                           13
JOB                                 14
EVALUATION CONTEXT                  14
DIRECTORY                           14
PROGRAM                             19
RULE SET                            23
CONSUMER GROUP                      25
QUEUE                               39
XML SCHEMA                          52
OPERATOR                            55
PROCEDURE                          159
LIBRARY                            183
TYPE BODY                          241
SEQUENCE                           242
TABLE PARTITION                    258
FUNCTION                           304
```

```
JAVA DATA                            328
INDEX PARTITION                      420
TRIGGER                              620
JAVA RESOURCE                        834
LOB                                  995
PACKAGE BODY                        1269
PACKAGE                             1329
TYPE                                2827
TABLE                               3113
INDEX                               4134
VIEW                                5263
JAVA CLASS                         22917
SYNONYM                            27802
```

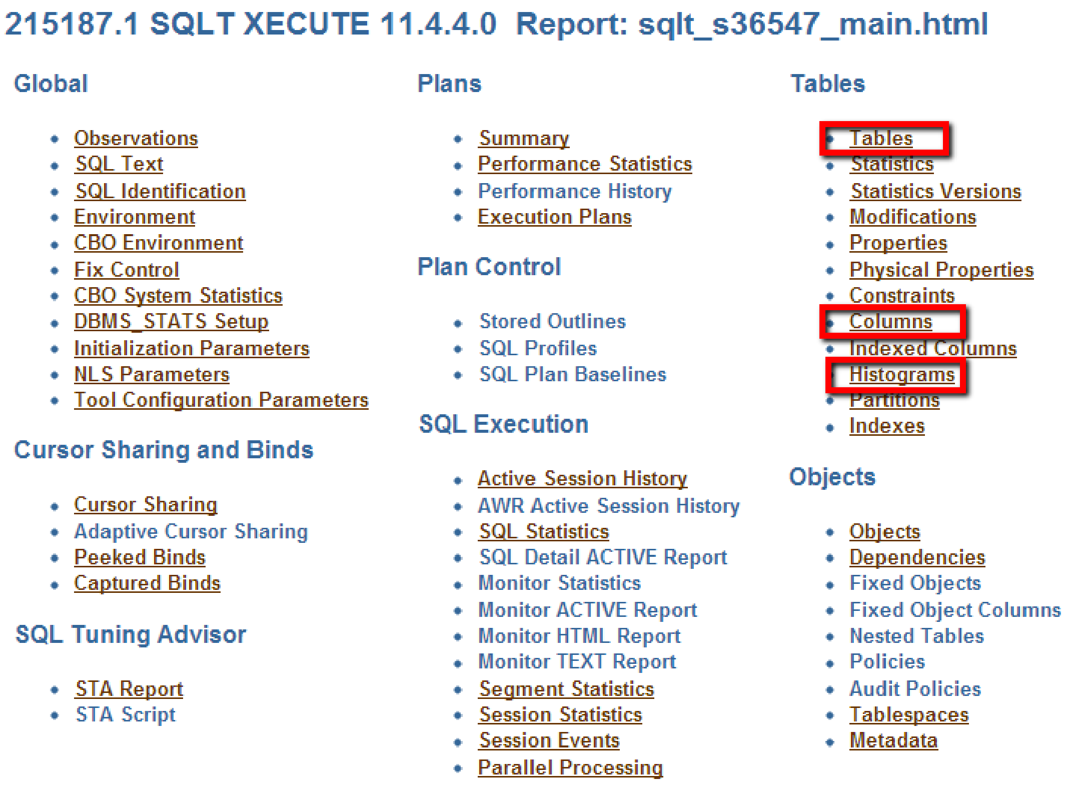
The example above shows the object type in the database vs. the number of those object types. Here the "bucket" is the `object_type` value ("bucket" has come to mean the range of values covered in a data set). So, for example, the object type of `SYNONYM` (bottom line in the example) has 27,802 values. Sometimes buckets are days of the week such as Saturday or Sunday, sometimes primary colors such as red or blue, and sometimes buckets are ranges such as "dates in 2011". In each case these are the values you plot on your X-axis if you were plotting a graph. From here on in, we will use the term "bucket" to represent the range of values in question.

Once you have issued the query you will see information that will help you decide how skewed the data is. Please note, however, that almost all data is skewed to some extent. The data will be skewed even in our example in the previous section regarding the spending of money in a superstore. The question you need to answer is this: is it skewed enough to cause a problem with my queries? A 10 percent variability doesn't usually cause a problem, but if you have one bucket with 90 percent of the samples, then the data is highly skewed. So decisions on skewness are somewhat subjective. In the example query above if I am querying my table `test3` and my predicate value is `SYNONYM`, then I am likely to get many more rows returned than if I issued my query against the value "`RESOURCE PLAN`". See the example code below. If we sample our test table for the rare value we only get 10 values.

```
SQL> select count(*) from test3 where object_type='SYNONYM';

  COUNT(*)
----------
     27802

SQL> select count(*) from test3 where object_type='RESOURCE PLAN';

  COUNT(*)
----------
        10
```

Do we care about this skewness, and is this data skewed a lot? Technically speaking even the slightest deviation from the "norm" can be classed as skewness, but the term has come to mean an amount of skewness that makes my execution plan unstable. From a more objective point of view, however, if you see as in the example above that some predicates return more than 10 times the values of other predicates, then this is very skewed. If you see a ratio of the most common to the least common value of 1.5 or higher, you can start to consider skewness as a factor.

SQLT makes the job of collecting queries like this much easier. It shows the distribution of data for a column histogram. First from the top of the report, (see Figure 4-2) we can click on "Tables", "Columns", or we could click on "Histograms". In this example we'll click on "Tables" since that's the route you'll most likely follow in an investigation.



215187.1 SQLT XECUTE 11.4.4.0 Report: sqlt_s36547_main.html

Global

- Observations
- SQL Text
- SQL Identification
- Environment
- CBO Environment
- Fix Control
- CBO System Statistics
- DBMS_STATS Setup
- Initialization Parameters
- NLS Parameters
- Tool Configuration Parameters

Cursor Sharing and Binds

- Cursor Sharing
- Adaptive Cursor Sharing
- Peeked Binds
- Captured Binds

SQL Tuning Advisor

- STA Report
- STA Script

Plans

- Summary
- Performance Statistics
- Performance History
- Execution Plans

Plan Control

- Stored Outlines
- SQL Profiles
- SQL Plan Baselines

SQL Execution

- Active Session History
- AWR Active Session History
- SQL Statistics
- SQL Detail ACTIVE Report
- Monitor Statistics
- Monitor ACTIVE Report
- Monitor HTML Report
- Monitor TEXT Report
- Segment Statistics
- Session Statistics
- Session Events
- Parallel Processing

Tables

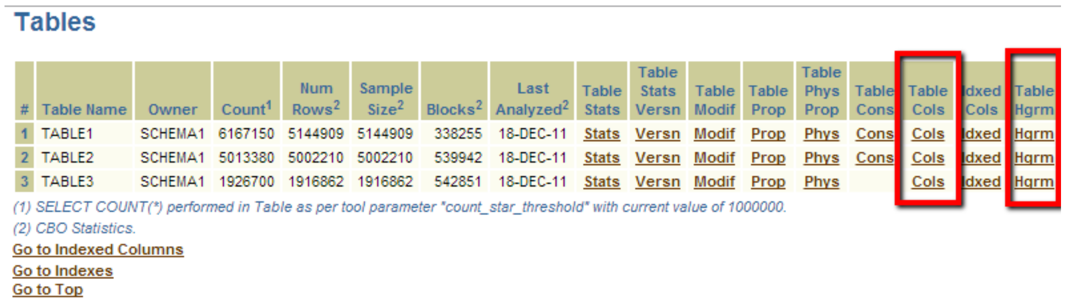
- Tables
- Statistics
- Statistics Versions
- Modifications
- Properties
- Physical Properties
- Constraints
- Columns
- Indexed Columns
- Histograms
- Partitions
- Indexes

Objects

- Objects
- Dependencies
- Fixed Objects
- Fixed Object Columns
- Nested Tables
- Policies
- Audit Policies
- Tablespaces
- Metadata

Figure 4-2. *The top of the SQLXECUTE report*

The “Tables” page shows us the screen in Figure 4-3.



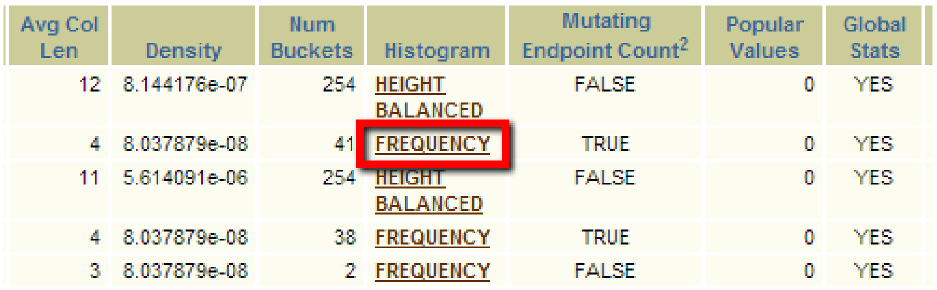
Tables

#	Table Name	Owner	Count[1]	Num Rows[2]	Sample Size[2]	Blocks[2]	Last Analyzed[2]	Table Stats	Table Stats Versn	Table Modif	Table Prop	Table Phys Prop	Table Cons	Table Cols	Idxed Cols	Table Hgrm
1	TABLE1	SCHEMA1	6167150	5144909	5144909	338255	18-DEC-11	Stats	Versn	Modif	Prop	Phys	Cons	Cols	Idxed	Hgrm
2	TABLE2	SCHEMA1	5013380	5002210	5002210	539942	18-DEC-11	Stats	Versn	Modif	Prop	Phys	Cons	Cols	Idxed	Hgrm
3	TABLE3	SCHEMA1	1926700	1916862	1916862	542851	18-DEC-11	Stats	Versn	Modif	Prop	Phys		Cols	Idxed	Hgrm

(1) SELECT COUNT(*) performed in Table as per tool parameter "count_star_threshold" with current value of 1000000.
(2) CBO Statistics.
Go to Indexed Columns
Go to Indexes
Go to Top

Figure 4-3. *The tables section of the report*

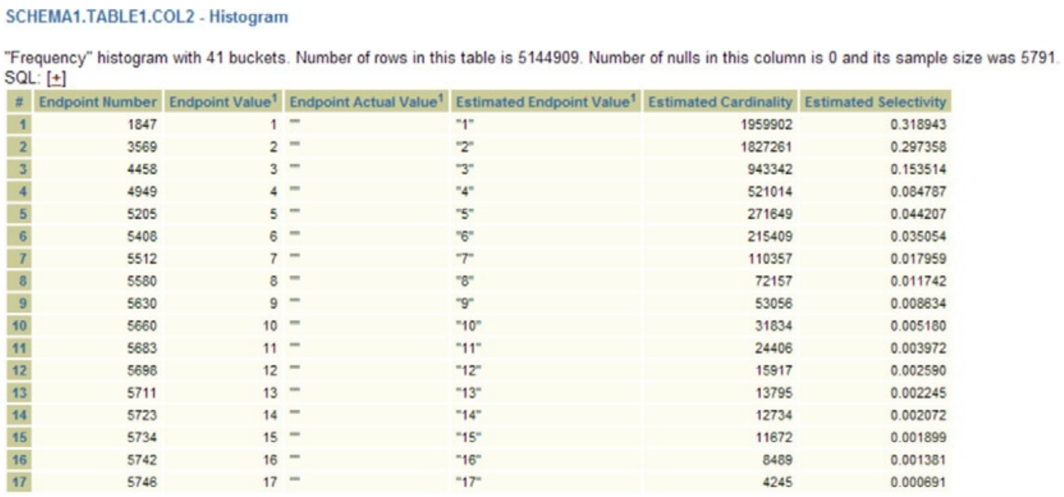
From Figure 4-3, which shows only the left hand side of the screen, we can now click on the “Cols” hyperlink for any individual table to see the column details. In this case we are looking at TABLE 1 so we click on the corresponding link. This gets us to the screen shown in Figure 4-4 (which is the right hand side of the screen), where I have highlighted the “FREQUENCY” hyperlink, which takes us to the histogram for COL2 of TABLE 1.



Avg Col Len	Density	Num Buckets	Histogram	Mutating Endpoint Count^2	Popular Values	Global Stats
12	8.144176e-07	254	HEIGHT BALANCED	FALSE	0	YES
4	8.037879e-08	41	FREQUENCY	TRUE	0	YES
11	5.614091e-06	254	HEIGHT BALANCED	FALSE	0	YES
4	8.037879e-08	38	FREQUENCY	TRUE	0	YES
3	8.037879e-08	2	FREQUENCY	FALSE	0	YES

Figure 4-4. *The hyperlink for the histogram for COL2*

We arrived at the screen shown in Figure 4-5, by clicking on the "FREQUENCY" histogram for column COL2. The two types of histograms, "FREQUENCY" and "HEIGHT BALANCED" will be discussed in a later section, but suffice to say that they are both ways of representing bucket frequencies.



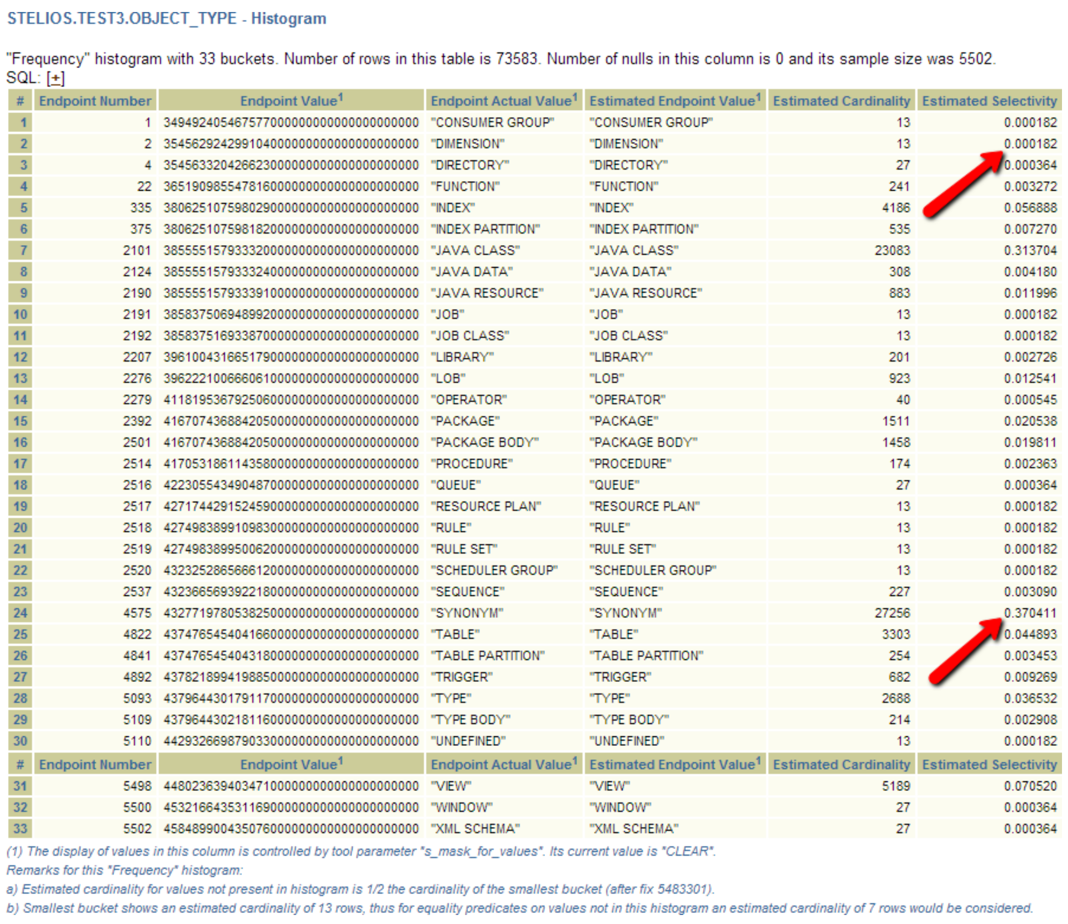
SCHEMA1.TABLE1.COL2 - Histogram

"Frequency" histogram with 41 buckets. Number of rows in this table is 5144909. Number of nulls in this column is 0 and its sample size was 5791.
SQL: [+]

#	Endpoint Number	Endpoint Value[1]	Endpoint Actual Value[1]	Estimated Endpoint Value[1]	Estimated Cardinality	Estimated Selectivity
1	1847	1	""	"1"	1959902	0.318943
2	3569	2	""	"2"	1827261	0.297358
3	4458	3	""	"3"	943342	0.153514
4	4949	4	""	"4"	521014	0.084787
5	5205	5	""	"5"	271649	0.044207
6	5408	6	""	"6"	215409	0.035054
7	5512	7	""	"7"	110357	0.017959
8	5580	8	""	"8"	72157	0.011742
9	5630	9	""	"9"	53056	0.008634
10	5660	10	""	"10"	31834	0.005180
11	5683	11	""	"11"	24406	0.003972
12	5698	12	""	"12"	15917	0.002590
13	5711	13	""	"13"	13795	0.002245
14	5723	14	""	"14"	12734	0.002072
15	5734	15	""	"15"	11672	0.001899
16	5742	16	""	"16"	8489	0.001381
17	5746	17	""	"17"	4245	0.000691

Figure 4-5. *A sample histogram from SQLT*

This figure shows us that for a value of "1" the estimated cardinality is approximately 1.9 million, while the value for "17" is approximately 4000. Is this data skewed? It sure is. In Figure 4-5 the estimated cardinality popularity have been ordered, but this is not always the case. Look at Figure 4-6 from our example `test3` table.



STELIOS.TEST3.OBJECT_TYPE - Histogram

"Frequency" histogram with 33 buckets. Number of rows in this table is 73583. Number of nulls in this column is 0 and its sample size was 5502.
SQL: [±]

#	Endpoint Number	Endpoint Value[1]	Endpoint Actual Value[1]	Estimated Endpoint Value[1]	Estimated Cardinality	Estimated Selectivity
1	1	349492405467577000000000000000000000	"CONSUMER GROUP"	"CONSUMER GROUP"	13	0.000182
2	2	354562924299104000000000000000000000	"DIMENSION"	"DIMENSION"	13	0.000182
3	4	354563320426623000000000000000000000	"DIRECTORY"	"DIRECTORY"	27	0.000364
4	22	365190985547816000000000000000000000	"FUNCTION"	"FUNCTION"	241	0.003272
5	335	380625107598029000000000000000000000	"INDEX"	"INDEX"	4186	0.056888
6	375	380625107598182000000000000000000000	"INDEX PARTITION"	"INDEX PARTITION"	535	0.007270
7	2101	385555157933320000000000000000000000	"JAVA CLASS"	"JAVA CLASS"	23083	0.313704
8	2124	385555157933324000000000000000000000	"JAVA DATA"	"JAVA DATA"	308	0.004180
9	2190	385555157933391000000000000000000000	"JAVA RESOURCE"	"JAVA RESOURCE"	883	0.011996
10	2191	385837506948992000000000000000000000	"JOB"	"JOB"	13	0.000182
11	2192	385837516933870000000000000000000000	"JOB CLASS"	"JOB CLASS"	13	0.000182
12	2207	396100431665179000000000000000000000	"LIBRARY"	"LIBRARY"	201	0.002726
13	2276	396222100666061000000000000000000000	"LOB"	"LOB"	923	0.012541
14	2279	411819536792506000000000000000000000	"OPERATOR"	"OPERATOR"	40	0.000545
15	2392	416707436884205000000000000000000000	"PACKAGE"	"PACKAGE"	1511	0.020538
16	2501	416707436884205000000000000000000000	"PACKAGE BODY"	"PACKAGE BODY"	1458	0.019811
17	2514	417053186114358000000000000000000000	"PROCEDURE"	"PROCEDURE"	174	0.002363
18	2516	422305543490487000000000000000000000	"QUEUE"	"QUEUE"	27	0.000364
19	2517	427174429152459000000000000000000000	"RESOURCE PLAN"	"RESOURCE PLAN"	13	0.000182
20	2518	427498389910983000000000000000000000	"RULE"	"RULE"	13	0.000182
21	2519	427498389950062000000000000000000000	"RULE SET"	"RULE SET"	13	0.000182
22	2520	432325286566612000000000000000000000	"SCHEDULER GROUP"	"SCHEDULER GROUP"	13	0.000182
23	2537	432366569392218000000000000000000000	"SEQUENCE"	"SEQUENCE"	227	0.003090
24	4575	432771978053825000000000000000000000	"SYNONYM"	"SYNONYM"	27256	0.370411
25	4822	437476545404166000000000000000000000	"TABLE"	"TABLE"	3303	0.044893
26	4841	437476545404318000000000000000000000	"TABLE PARTITION"	"TABLE PARTITION"	254	0.003453
27	4892	437821899419885000000000000000000000	"TRIGGER"	"TRIGGER"	682	0.009269
28	5093	437964430179117000000000000000000000	"TYPE"	"TYPE"	2688	0.036532
29	5109	437964430218116000000000000000000000	"TYPE BODY"	"TYPE BODY"	214	0.002908
30	5110	442932669879033000000000000000000000	"UNDEFINED"	"UNDEFINED"	13	0.000182
31	5498	448023639403471000000000000000000000	"VIEW"	"VIEW"	5189	0.070520
32	5500	453216643531169000000000000000000000	"WINDOW"	"WINDOW"	27	0.000364
33	5502	458489900435076000000000000000000000	"XML SCHEMA"	"XML SCHEMA"	27	0.000364

(1) The display of values in this column is controlled by tool parameter "s_mask_for_values". Its current value is "CLEAR".
Remarks for this "Frequency" histogram:
a) Estimated cardinality for values not present in histogram is 1/2 the cardinality of the smallest bucket (after fix 5483301).
b) Smallest bucket shows an estimated cardinality of 13 rows, thus for equality predicates on values not in this histogram an estimated cardinality of 7 rows would be considered.

Figure 4-6. *Another example histogram table for our test3 example. SYNONYMs are popular*

In this example histogram, we see the basic facts shown on the column display; the number of buckets (33), the number of rows in the table, the number of nulls, and the sample size. In this case the sample size is extremely small.

This data looks highly skewed. For the "Endpoint Actual Value" of "DIMENSION" the estimated selectivity is 0.000182. For JAVA_CLASS the selectivity is 0.313704. This means that since there were 0 null values nearly a third of the table will be returned if a predicate against `object_type` uses the value "JAVA CLASS". If the predicate used the value "JAVA RESOURCE" then the selectivity is only 0.011996 or a cardinality of 883, compared to the value of 23,083 values for "JAVA CLASS". Even with this highly skewed data, you could ask yourself: "Is this skewness relevant for my query?" If the column `object_type` is not involved in your query, then the skewness does not matter to you.

How Skewness Affects the Execution Plan

Now that we know what skewness is and how statistics are shown in a SQLT report for a particular SQL statement, we should look at how skewness can cause problems with an execution plan.

Suppose we wanted to know the sum of all the financial transactions we had carried out in the previous month with each financial institution we had done business with. We might well have the name of the institution in the description field, possibly a `company_ID` to represent the company and a field to represent the amount of the financial transaction. We would probably have lots of other fields as well of course, but these are the ones we would be interested with regard to skewness. We could query this table to find the sum of all the records where the `company_ID` matched the company that we were currently looking for. If we chose a company we had not done much business with we might only get a few records. The optimizer might well be expecting only one record for some companies and so might choose to do a Cartesian join. If the company was Bank of America, however, we might expect thousands (perhaps millions) of transactions. In this case a hash join might be a better choice. The "good" plan for these two extreme cases are very different.

If the Cartesian join is used with the Bank of America records, there would be a disastrous drop in performance as Oracle tried to do a Cartesian join on thousands of records. Even Exadata can't cope with errors like this. So depending on the predicate value, we could well need different execution plans. To fix these kinds of problems, we need to consider what the options are: removing the histogram, improving the histogram, using Adaptive Cursor Sharing, or changing the query. There is no hard and fast rule here for what to do in these circumstances. SQLT will show what is happening and how skewness is affecting the execution plan; then you, in conjunction with the developers, can decide which is the best strategy for your situation: which indexes to add or remove and which histograms are helpful to collect.

Histograms

Histograms are a confusing topic, unless you know what they are. The confusing and sometimes contradictory terminology used by different documents does not help. So let me start by defining a bucket. A bucket is a range of values for a particular column. So, for example, if we had a column TRUE_OR_FALSE and it could only have two values (TRUE or FALSE) then we could have up to two buckets: each bucket would have a value describing the number of values that were TRUE and the number that were FALSE. A histogram is a data representation that describes the relative population of different ranges of data value.

Histogram Types

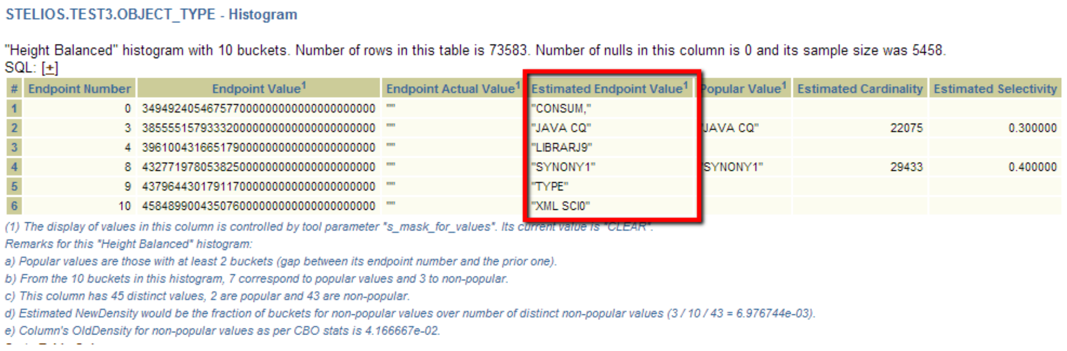
We've talked about histograms in general terms, but Oracle currently has a bucket limit of 254 for any type of histogram. This means it cannot store the frequency of more than 254 types of value for a column. As you can imagine this is a very small number when compared to almost any range of possible values of anything! "Colors of paint", "Names of Banks", "Zip Codes": most histograms will need more than 254 buckets. If the number of distinct values (shown as NDV in SQLT reports) is greater than 254 and the statistics gathering process detects this, then you will have to somehow squeeze the data distribution into fewer buckets than there are distinct values. If this happens, you are exposing the optimizer to the risk of incomplete information that may adversely affect your execution plan. Look back at Figure 4-6. You'll see that there were 33 buckets from the sample of 5,502 rows. When in fact we know there are 45 distinct values. Look at the result of this query, which reads all the data from the table.

```
SQL>select count(distinct object_type) from test3;

COUNT(DISTINCTOBJECT_TYPE)
--------------------------
                        45
```

The true answer to the number of distinct values is 45. The statistically sampled answer was 33. The statistics sampler, even with a sample of 5,502, just wasn't lucky enough to find all the different possible values of `object_type`. Still, it produced a frequency type histogram, because it did not exceed its 254 limit. The point is that despite being a frequency-type histogram not all the values were represented. What happens if we artificially squeeze the number of buckets down to 10?

```
SQL>exec dbms_stats.set_table_prefs(
  ownname=>'STELIOS',
  tabname=>'TEST3',
  pname=>'method_opt',
  pvalue=>'for all columns size 10');

PL/SQL procedure successfully completed.
```

Now we have the other type of histogram (of the two possible types), the height based histogram (see Figure 4-7).



STELIOS.TEST3.OBJECT_TYPE - Histogram

"Height Balanced" histogram with 10 buckets. Number of rows in this table is 73583. Number of nulls in this column is 0 and its sample size was 5458.
SQL: [+]

#	Endpoint Number	Endpoint Value¹	Endpoint Actual Value¹	Estimated Endpoint Value¹	Popular Value¹	Estimated Cardinality	Estimated Selectivity
1	0	349492405467577000000000000000000000	""	"CONSUM,"			
2	3	385555157933320000000000000000000000	""	"JAVA CQ"	JAVA CQ"	22075	0.300000
3	4	396100431665179000000000000000000000	""	"LIBRARJ9"			
4	8	432771978053825000000000000000000000	""	"SYNONY1"	SYNONY1"	29433	0.400000
5	9	437964430179117000000000000000000000	""	"TYPE"			
6	10	458489900435076000000000000000000000	""	"XML SCI0"			

(1) The display of values in this column is controlled by tool parameter "s_mask_for_values". Its current value is "CLEAR".
Remarks for this "Height Balanced" histogram:
a) Popular values are those with at least 2 buckets (gap between its endpoint number and the prior one).
b) From the 10 buckets in this histogram, 7 correspond to popular values and 3 to non-popular.
c) This column has 45 distinct values, 2 are popular and 43 are non-popular.
d) Estimated NewDensity would be the fraction of buckets for non-popular values over number of distinct non-popular values (3 / 10 / 43 = 6.976744e-03).
e) Column's OldDensity for non-popular values as per CBO stats is 4.166667e-02.

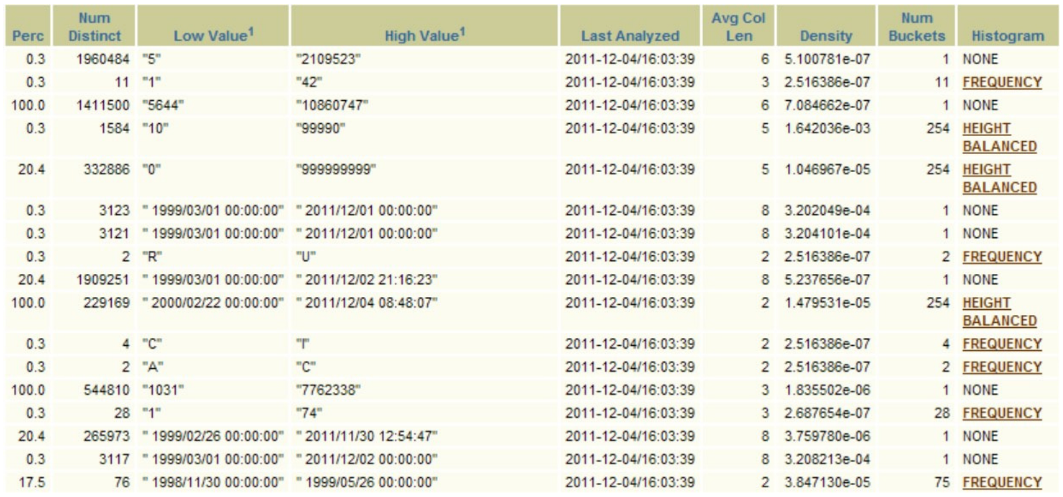
Figure 4-7. *A height-balanced histogram with 10 buckets*

In the height-based histogram, value types are counted against the buckets and are marked as popular or not popular. If they are not popular, then one bucket is allocated. For example, see bucket number 1, as it represents the values for object types named lower in the alphabet than "CONSUM": for example, "CLUSTER" and "CONSUMER GROUP". Because there could be numerous values for one bucket, some of which we might not come across, the end point is estimated. We know there is no object type of "SYNONY1", but it's the label for buckets 5, 6, 7, and 8. As you can see, with only 10 buckets the estimate of the distribution of data for object type (which in fact has 45 distinct types) is very poor. With a histogram of this poor quality we might well end up with some poor plans. It is therefore important that we know when to have histograms and when they are a hindrance.

When to Use Histograms

If you use the default values for optimizer statistics collection then you will most likely have some histograms on columns that you aren't aware of. If you have a query with poor performance, or one you want to tune, you may want to examine what has been collected to make sure it makes sense. There are a few guidelines to consider when deciding if you want a histogram on a column or not. Histograms were designed to take care of skewed data, so if your data is not very skewed then it follows that histograms are not useful. An example might be a date-stamp against a table record. If the database activity happens every day and we have 100 transactions every day each with a timestamp, a histogram on the date column is not going to help. In FREQUENCY histograms this would be equivalent to the height of all buckets being the same. If there is no skewness then histograms do not help us. It just means we spend time overnight gathering statistics that we don't need. As I've mentioned before the auto sample size is very clever about when to quit sampling if the data is hardly skewed, so you probably will not lose much time in this way.

If you have very skewed data, then histograms may be very important for you; but if you do have them, you should check with SQLT that the histograms are appropriate and correct. In Figure 4-8, we see the column statistics for TABLE 1.



Perc	Num Distinct	Low Value[1]	High Value[1]	Last Analyzed	Avg Col Len	Density	Num Buckets	Histogram
0.3	1960484	"5"	"2109523"	2011-12-04/16:03:39	6	5.100781e-07	1	NONE
0.3	11	"1"	"42"	2011-12-04/16:03:39	3	2.516386e-07	11	FREQUENCY
100.0	1411500	"5644"	"10860747"	2011-12-04/16:03:39	6	7.084662e-07	1	NONE
0.3	1584	"10"	"99990"	2011-12-04/16:03:39	5	1.642036e-03	254	HEIGHT BALANCED
20.4	332886	"0"	"999999999"	2011-12-04/16:03:39	5	1.046967e-05	254	HEIGHT BALANCED
0.3	3123	" 1999/03/01 00:00:00"	" 2011/12/01 00:00:00"	2011-12-04/16:03:39	8	3.202049e-04	1	NONE
0.3	3121	" 1999/03/01 00:00:00"	" 2011/12/01 00:00:00"	2011-12-04/16:03:39	8	3.204101e-04	1	NONE
0.3	2	"R"	"U"	2011-12-04/16:03:39	2	2.516386e-07	2	FREQUENCY
20.4	1909251	" 1999/03/01 00:00:00"	" 2011/12/02 21:16:23"	2011-12-04/16:03:39	8	5.237656e-07	1	NONE
100.0	229169	" 2000/02/22 00:00:00"	" 2011/12/04 08:48:07"	2011-12-04/16:03:39	2	1.479531e-05	254	HEIGHT BALANCED
0.3	4	"C"	"I"	2011-12-04/16:03:39	2	2.516386e-07	4	FREQUENCY
0.3	2	"A"	"C"	2011-12-04/16:03:39	2	2.516386e-07	2	FREQUENCY
100.0	544810	"1031"	"7762338"	2011-12-04/16:03:39	3	1.835502e-06	1	NONE
0.3	28	"1"	"74"	2011-12-04/16:03:39	3	2.687654e-07	28	FREQUENCY
20.4	265973	" 1999/02/26 00:00:00"	" 2011/11/30 12:54:47"	2011-12-04/16:03:39	8	3.759780e-06	1	NONE
0.3	3117	" 1999/03/01 00:00:00"	" 2011/12/02 00:00:00"	2011-12-04/16:03:39	8	3.208213e-04	1	NONE
17.5	76	" 1998/11/30 00:00:00"	" 1999/05/26 00:00:00"	2011-12-04/16:03:39	2	3.847130e-05	75	FREQUENCY

Figure 4-8. *A section from some column statistics*

Apart from the very low percentage sample rates (0.3 percent) for many columns we also have some bucket counts that do not match the number of distinct values. The last entry shows 76 distinct values but only 75 buckets. This means there is one value that is not even represented in the sample size of 17.5 percent. In a case such as this, you would have to look at the histogram (a FREQUENCY histogram in this case) and determine if the histogram was useful. If it isn't because you judge the data is not skewed, then you can remove the histogram and change the table preferences to not collect statistics on this column. If you find that the data is indeed skewed then you need to collect better statistics. This will mean collecting a larger sample size for this column. You could do this by setting table preferences or by running specific jobs to collect data after the main statistics job.

How to Add and Remove Histograms

If you want to remove a column's histogram you can simply use the `dbms_stats` procedure as shown in the example below

```
SQL> exec dbms_stats.delete_column_stats(
  ownname=>'STELIOS',
  tabname=>'TEST3',
  colname=>'OBJECT_TYPE',col_stat_type=>'HISTOGRAM');
PL/SQL procedure successfully completed.
```

However, this does not remove the statistics permanently. The next time the statistics job comes across the table, it may decide that the same column needs column statistics again. You would remove statistics from a column on a one-time basis if you were testing the execution of a query. Removing the column statistics might then allow you to re-parse a statement and see if the execution plan is materially affected by the change. If you wanted to remove the column statistics collection forever then you could set the table preferences, as in the example below, by setting the `method_opt` value to "`FOR ALL COLUMNS SIZE 1`", which means no column histograms.

```
SQL> exec dbms_stats.set_table_prefs(
  ownname=>'STELIOS',
  tabname=>'TEST3',
  pname=>'method_opt',
  pvalue=>'FOR ALL COLUMNS SIZE 1');

PL/SQL procedure successfully completed.
```

If the table has other columns you need statistics for, you may decide that you want to set the preferences so that only one column in particular is ignored for column statistics. This is the command example show in the following example.

```
SQL> exec dbms_stats.set_table_prefs(
  ownname=>'STELIOS',
  tabname=>'TEST3',
  pname=>'method_opt',
  pvalue=>'FOR COLUMNS OBJECT_TYPE SIZE 1');

PL/SQL procedure successfully completed.
```

If on the other hand you want to add column statistics you can gather them manually with a `dbms_stats.get_column_stats` command.

Bind Variables

Bind variables are one of those mysteries that a new Oracle DBA might not immediately be aware of. Developers, on the other hand, will use bind variables all the time and be comfortable with the definitions and the meanings. If you are familiar with the concept of a variable and its value, then there is nothing more to understand from a bind variable. It's just Oracle's version of a variable. A variable (and bind variable) is just a symbolic name for a value that can change depending on the needs of the code. Bind peeking and bind capture, however, are ideas that are more specific to Oracle and are techniques allowing the optimizer to make better decisions when skewness is present. Bind variables, skewness, and bind peeking all work with CURSOR_SHARING to improve the CBO plan. We cover all these concepts and how they work for you, in the following sections.

What Are Bind Variables?

Now that we have entered the world of histograms and have looked at skewness, we see that the value used as the value for the predicate could have a material effect on the number of rows returned. In the example below we see that depending on whether we select "EDITION" or "TABLE" we get a different count.

```
SQL> select count(*) from test3 where object_type='EDITION';
  COUNT(*)
----------
         1
SQL> select count(*) from test3 where object_type='TABLE';
  COUNT(*)
----------
      3113
```

If we wanted to avoid literal values in our SQL, which would all be parsed separately (and would have an adverse effect on performance), we might have introduced bind variables. Bind variables are a way to pass parameters to routines by setting the values of the parameters. Below I show a simple procedure to count the number of different object types in TEST3, using a bind variable called `b1`.

```
SQL> set serveroutput on;
SQL> create or replace procedure object_count(b1 in char)
  2  as
  3    object_count number;
  4    begin
  5    select count(*) into object_count from test3 where object_type=b1;
  6    dbms_output.put_line(b1||' = '||object_count);
  7    end;
  8  /

Procedure created.
```

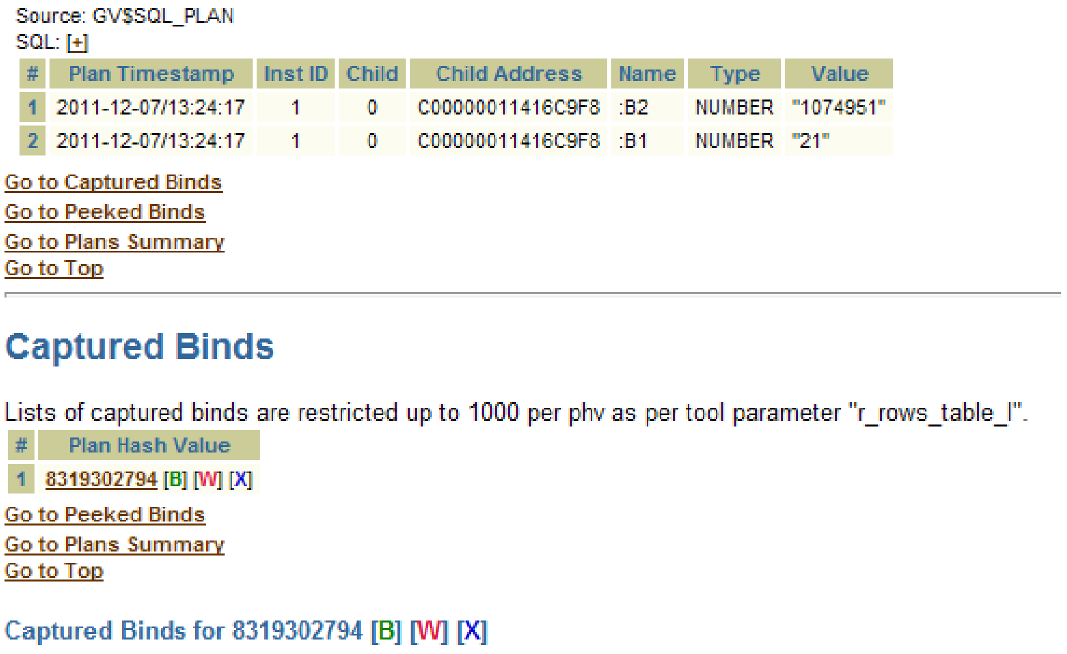
If we ran this procedure a few times we could see that different values are being passed in and being used.

```
SQL> exec object_count('EDITION');
EDITION = 1

PL/SQL procedure successfully completed.

SQL> exec object_count('TABLE');
TABLE = 3113

PL/SQL procedure successfully completed.
```

In the example above, `b1` is taking the value "EDITION" and then the value "TABLE". The text in the procedure that actually does the work is a simple `select` statement. Its predicate is `object_type=b1`. The text of the query did not change as far as the optimizer is concerned.

What Are Bind Peeking and Bind Capture?

By using bind variables we are hiding the actual value of the bind variable from the optimizer. We did this for a good reason. We want to avoid excessive parsing of every SQL with a different predicate value. That's a good strategy when the data is not skewed and your execution plan is not likely to change depending on the value passed in as the bind variable. With skewed data, the optimizer could benefit from knowing the value of the bind variable being passed to the SQL. This process is called "bind peeking" and is done during hard parsing. Bind capture, on the other hand, is a snapshot of the value of actual bind variables being used during execution of the SQL. The values used for the bind, in this case `b1`, are collected and available to SQLT. In the example below (Figure 4-9) we see the section in the SQLT report showing the values of the captured binds.



Source: GV$SQL_PLAN
SQL: [+]

#	Plan Timestamp	Inst ID	Child	Child Address	Name	Type	Value
1	2011-12-07/13:24:17	1	0	C00000011416C9F8	:B2	NUMBER	"1074951"
2	2011-12-07/13:24:17	1	0	C00000011416C9F8	:B1	NUMBER	"21"

Go to Captured Binds
Go to Peeked Binds
Go to Plans Summary
Go to Top

Captured Binds

Lists of captured binds are restricted up to 1000 per phv as per tool parameter "r_rows_table_l".

#	Plan Hash Value
1	8319302794 [B] [W] [X]

Go to Peeked Binds
Go to Plans Summary
Go to Top

Captured Binds for 8319302794 [B] [W] [X]

Figure 4-9. *Example output of captured binds*

In cases where skewness is important and you suspect that the values of the binds are sending the optimizer down the wrong path, you may want to look at the section “Captured Binds.” In conjunction with the captured binds and the time stamps (and any reports of poor performance) you can track down why a particular statement decided to do the wrong thing.

Cursor_Sharing and Its Values

The CURSOR_SHARING parameter was introduced by Oracle to address the issues caused by code, which did not use bind variables as described above. When bind variables are not used, the SQL text has to be parsed by the optimizer. This is a costly operation, and it’s probably a waste of time if the only thing that’s changed is the value of a literal.

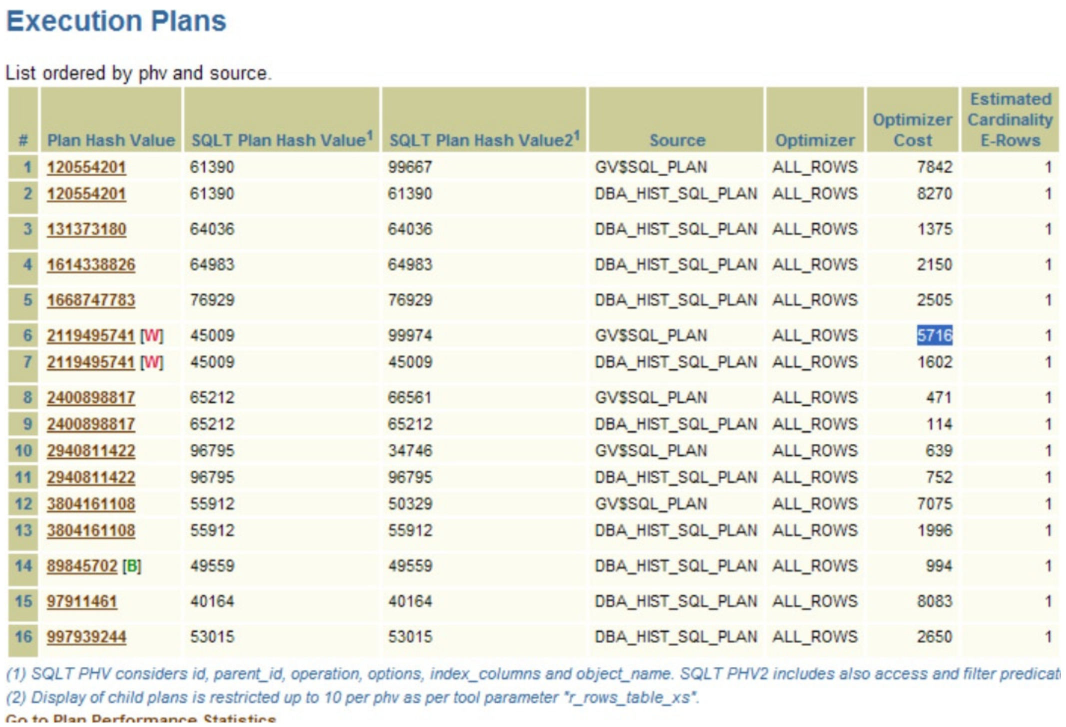
CURSOR_SHARING has three possible values: EXACT, FORCE, and SIMILAR. EXACT is the default and tells the optimizer to consider each SQL text as it comes across it. If your application is well written and uses binds and literals as seems most efficient, then leaving EXACT alone is the best choice.

CURSOR_SHARING set to FORCE tells the optimizer to use binds for all the predicates (and it will make its own names). This is a value you would use if you had a badly written application and you wanted to take advantage of the performance gains from using bind variables, without modifying your application to use bind variables.

CURSOR_SHARING set to SIMILAR is deprecated as of 11g and should not be used, but it’s probably worth mentioning that it was an “intelligent” version of CURSOR_SHARING=FORCE. It created system-defined bind variables in most cases unless the bind variable affected the optimization in some way. It was not a complete solution, and the introduction of Adaptive Cursor Sharing is a big improvement on CURSOR_SHARING=SIMILAR. The value of the parameter CURSOR_SHARING is important to note so that you understand if the bind variables are in play. You will also see in the SQL text section of the SQL report that if literals have been replaced by bind variables they will have system-defined names.

The Case of the Variable Execution Time

Sometimes it is better to have a slightly longer stable execution time than a slightly faster unstable plan. Unstable execution times (where the execution time can vary widely) cause problems with scheduling of batch jobs. In the example case here the execution times have never been stable. You're in a pre-production environment and you want to get some information on what can be done to improve the stability of the execution time. You have a luxury in that you have a test environment with representative queries and data. So after the test you look at the AWR reports and decide that one particular query needs attention, because sometimes it takes a long time. We get the SQL_ID and generate the SQLT. We want to know what's happening with the execution plans, so we look in the "Execution Plans" section as shown in Figure 4-10.


Execution Plans

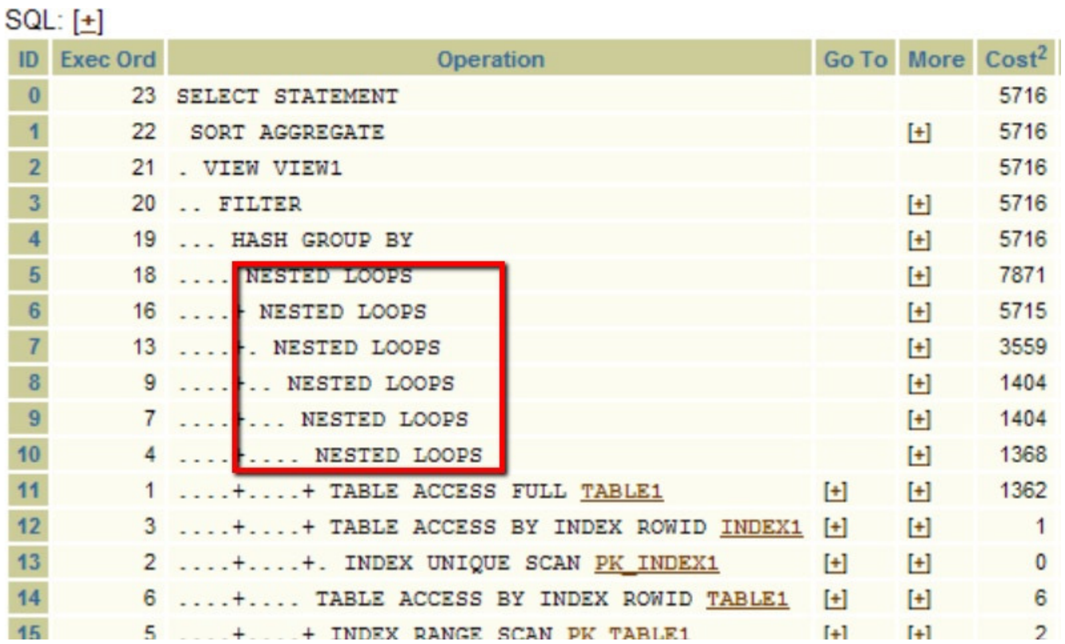
List ordered by phv and source.

#	Plan Hash Value	SQLT Plan Hash Value[1]	SQLT Plan Hash Value2[1]	Source	Optimizer	Optimizer Cost	Estimated Cardinality E-Rows
1	120554201	61390	99667	GV$SQL_PLAN	ALL_ROWS	7842	1
2	120554201	61390	61390	DBA_HIST_SQL_PLAN	ALL_ROWS	8270	1
3	131373180	64036	64036	DBA_HIST_SQL_PLAN	ALL_ROWS	1375	1
4	1614338826	64983	64983	DBA_HIST_SQL_PLAN	ALL_ROWS	2150	1
5	1668747783	76929	76929	DBA_HIST_SQL_PLAN	ALL_ROWS	2505	1
6	2119495741 [W]	45009	99974	GV$SQL_PLAN	ALL_ROWS	5716	1
7	2119495741 [W]	45009	45009	DBA_HIST_SQL_PLAN	ALL_ROWS	1602	1
8	2400898817	65212	66561	GV$SQL_PLAN	ALL_ROWS	471	1
9	2400898817	65212	65212	DBA_HIST_SQL_PLAN	ALL_ROWS	114	1
10	2940811422	96795	34746	GV$SQL_PLAN	ALL_ROWS	639	1
11	2940811422	96795	96795	DBA_HIST_SQL_PLAN	ALL_ROWS	752	1
12	3804161108	55912	50329	GV$SQL_PLAN	ALL_ROWS	7075	1
13	3804161108	55912	55912	DBA_HIST_SQL_PLAN	ALL_ROWS	1996	1
14	89845702 [B]	49559	49559	DBA_HIST_SQL_PLAN	ALL_ROWS	994	1
15	97911461	40164	40164	DBA_HIST_SQL_PLAN	ALL_ROWS	8083	1
16	997939244	53015	53015	DBA_HIST_SQL_PLAN	ALL_ROWS	2650	1

(1) SQLT PHV considers id, parent_id, operation, options, index_columns and object_name. SQLT PHV2 includes also access and filter predicat
(2) Display of child plans is restricted up to 10 per phv as per tool parameter "r_rows_table_xs".
Go to Plan Performance Statistics

Figure 4-10. *Shows many different execution plans*

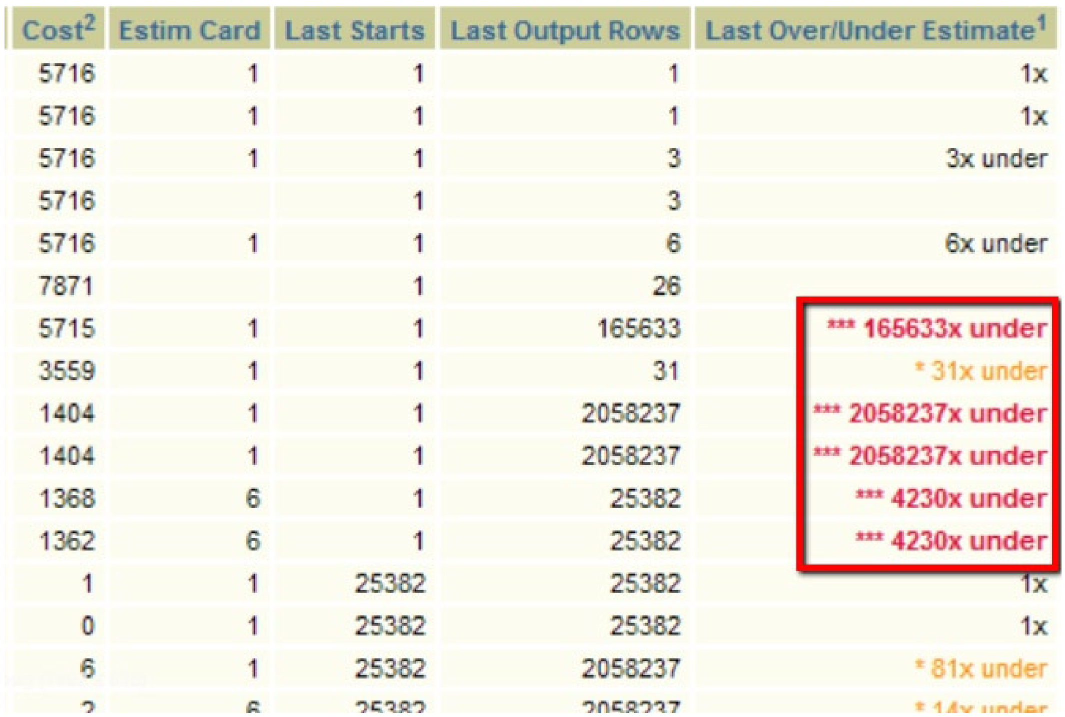
In the "Execution Plans" section we see that there are many different execution plans. Why would we have so many different execution plans? Let's look at the worst execution plan first, as shown in Figure 4-11 (the left side of the screen) and Figure 4-12 (on the right side). We see many Nested Loop steps in Figure 4-11 and many under estimates in Figure 4-12.



SQL: [+]

ID	Exec Ord	Operation	Go To	More	Cost²
0	23	SELECT STATEMENT			5716
1	22	SORT AGGREGATE		[+]	5716
2	21	. VIEW VIEW1			5716
3	20	.. FILTER		[+]	5716
4	19	... HASH GROUP BY		[+]	5716
5	18	....NESTED LOOPS		[+]	7871
6	16	....+ NESTED LOOPS		[+]	5715
7	13	....+. NESTED LOOPS		[+]	3559
8	9	....+.. NESTED LOOPS		[+]	1404
9	7	....+... NESTED LOOPS		[+]	1404
10	4	....+.... NESTED LOOPS		[+]	1368
11	1	....+....+ TABLE ACCESS FULL TABLE1	[+]	[+]	1362
12	3	....+....+ TABLE ACCESS BY INDEX ROWID INDEX1	[+]	[+]	1
13	2	....+....+. INDEX UNIQUE SCAN PK_INDEX1	[+]	[+]	0
14	6	....+.... TABLE ACCESS BY INDEX ROWID TABLE1	[+]	[+]	6
15	5	....+....+ INDEX RANGE SCAN PK_TABLE1	[+]	[+]	2

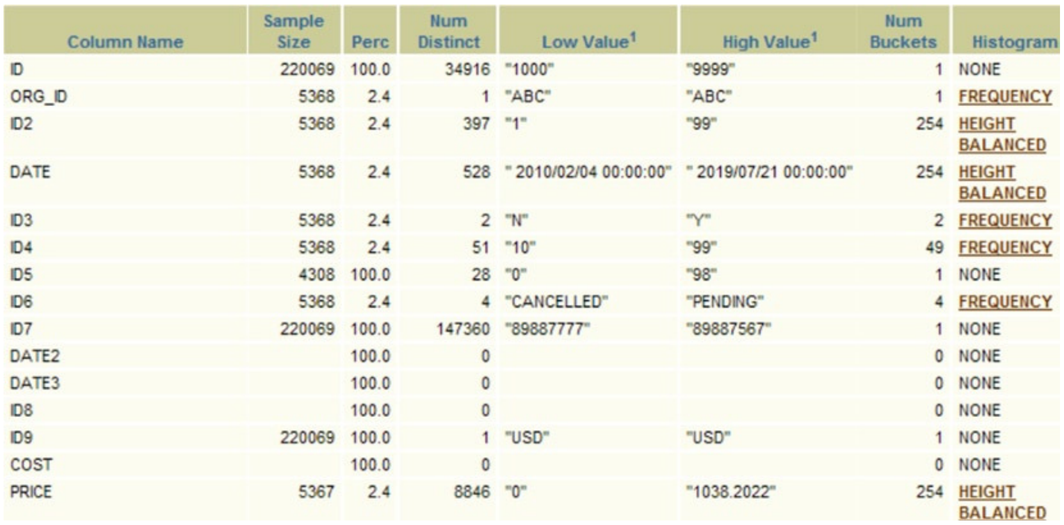
Figure 4-11. *The worst execution plan as selected by clicking on the "W" from the list of execution plans in Figure 4-10*



Cost²	Estim Card	Last Starts	Last Output Rows	Last Over/Under Estimate¹
5716	1	1	1	1x
5716	1	1	1	1x
5716	1	1	3	3x under
5716		1	3	
5716	1	1	6	6x under
7871		1	26	
5715	1	1	165633	*** 165633x under
3559	1	1	31	* 31x under
1404	1	1	2058237	*** 2058237x under
1404	1	1	2058237	*** 2058237x under
1368	6	1	25382	*** 4230x under
1362	6	1	25382	*** 4230x under
1	1	25382	25382	1x
0	1	25382	25382	1x
6	1	25382	2058237	* 81x under
2	6	25382	2058237	* 14x under

Figure 4-12. *The numbers highlighted with *** show very big under estimates by the optimizer*

Since we have actual rows returned, we can look at the over and under estimates. We have quite a few of them, so something is not right. Possibly more than one thing is not right. Since the first step, Line 11, is already "wrong," by a factor of 4,230, we should start there. Look at Figure 4-13, as it shows the column statistics for TABLE 1. Do you see anything of interest? (The actual table has more information than is shown in the figure. I have removed some table columns for readability.)



Column Name	Sample Size	Perc	Num Distinct	Low Value[1]	High Value[1]	Num Buckets	Histogram
ID	220069	100.0	34916	"1000"	"9999"	1	NONE
ORG_ID	5368	2.4	1	"ABC"	"ABC"	1	FREQUENCY
ID2	5368	2.4	397	"1"	"99"	254	HEIGHT BALANCED
DATE	5368	2.4	528	" 2010/02/04 00:00:00"	" 2019/07/21 00:00:00"	254	HEIGHT BALANCED
ID3	5368	2.4	2	"N"	"Y"	2	FREQUENCY
ID4	5368	2.4	51	"10"	"99"	49	FREQUENCY
ID5	4308	100.0	28	"0"	"98"	1	NONE
ID6	5368	2.4	4	"CANCELLED"	"PENDING"	4	FREQUENCY
ID7	220069	100.0	147360	"89887777"	"89887567"	1	NONE
DATE2		100.0	0			0	NONE
DATE3		100.0	0			0	NONE
ID8		100.0	0			0	NONE
ID9	220069	100.0	1	"USD"	"USD"	1	NONE
COST		100.0	0			0	NONE
PRICE	5367	2.4	8846	"0"	"1038.2022"	254	HEIGHT BALANCED

Figure 4-13. *Column statistics for Table 1*

Let's look at the ID column's statistics. The sample size is good at 100 percent, and there are 34,916 distinct values: so that's more values than we can fit into our 254 buckets. So if we had a histogram it would be a HEIGHT BALANCED one, but under the "Histograms" column we see "NONE". So what about ORG_ID? Only a 2.4 percent sample with only one distinct value, and high and low values of "ABC". Also we have a FREQUENCY histogram with only one bucket. Not surprisingly the selectivity will be 1.0, but equally this histogram can't be doing anything for us, as it's pointless. On the other hand, it can't be contributing to a variable execution plan. So let's look at Column ID2, where we have a 2.4 percent sample with 397 distinct values and a height balanced histogram. If we looked at the histogram in detail we'd see there were lots of popular values, but let's keep going. The DATE column has a HEIGHT BALANCED histogram: does that make any sense? It depends on what the DATE column represents. But usually DATE columns with histograms should be viewed with suspicion. ID3, also 2.4 percent, has two distinct values, "Y" and "N". If we look at the histogram we'd see that "N" has a selectivity of 0.03 and "Y" has a selectivity of 0.97. This alone could affect the rows returned. It's looking like there is some skewed data for sure. But let's keep looking. Next we have ID4 with 51 distinct values and 49 buckets, so things don't look quite so good now. The histogram for that shows selectivities between 0.005 and 0.1. That's a factor of 20: which means that's another skewed column, plus the bucket count is wrong. ID5 doesn't have a histogram and ID6 has a wrong bucket count: again it's skewed. You would spend time looking at every column in a report like this because sometimes the real culprit is the last thing you look at; but in this example, we have poor sampling for histograms, inappropriate histograms, and probably missing histograms.

In a case like this you would do your best to set up the right histograms, eliminating them where they are not needed and adding them where they are.

Summary

Skewness is one of the most difficult concepts to understand and one of the most troublesome to deal with in SQL. Many attempts have been made through iterations of the Oracle software to deal with the problem, some attempts were more successful than others, but Adaptive Cursor Sharing is the most successful to date. Throughout the twists and turns that can result from unstable execution plans, SQLTXPLAIN is there with the information to get your plan stable and the right statistics collected. In the next chapter we will look at query transformation that the optimizer does during parsing to get your execution plan to execute quickly.

CHAPTER 5

Troubleshooting Query Transformations

The Oracle query optimizer is an amazing piece of code, developed and improved over the years (with perhaps a few blind alleys along the way) to generate execution plans that are both easy to generate and that run fast. The optimizer uses a number of "tricks" along the way to improve the speed of execution and implements the execution plan in a way that gives the same results as your unmodified SQL. These "tricks" (or heuristics) sometimes include query transformations.

Query transformations, as the name implies, change the SQL query from its original text into something different: but it still gives the same result. This is a little bit like saying you want to go from London to New York, and you have decided to go via Orlando, have a nice rest, visit Disneyworld, soak up the sun and then drive up the East Coast. A query transformation would optimize your journey by using heuristics. One of the heuristics might be: "If there is a direct flight from your starting point to your destination, then avoid using stopovers in other locations". Or perhaps you could rewrite the rule and use "minimize the number of stop over points in your journey". I'm sure you can see this could get pretty complicated, even for my simple example. What if you really wanted to see Mickey Mouse? Then the plan would be more difficult to change. Sometimes even the simplest Mickey Mouse queries can cause problems when you're trying to make the journey quickly and efficiently.

What Are Query Transformations?

To explain what is meant by "query transformation," from the point of view of an SQL query, let's look at a simple example. Ask yourself a question. Is this query something that can be simplified?

```
SQL> select * from (select * from sales); -- Query 1
```

This query "obviously" simplifies to

```
SQL> select * from sales; -- Query 2
```

That's about the simplest example of a query transformation. Query 1 was transformed into query 2. There is no danger with this query transformation that the results will be different between version 1 and version 2 of the query. This particular transformation is called "sub-query unnesting." Each transformation has its name and its set of rules to follow to ensure that the end results are correct as well as optimized. Does the cost-based optimizer recognize this fact? Let's ask the optimizer to calculate a plan for a very simple query; but we'll use a hint that tells it to not use any query transformations. Here is the SQL and the execution plan.

```
SQL> select /*+ no_query_transformation */ * from (select * from sales);
Execution Plan
----------------------------------------------------------
Plan hash value: 2635429107
-----------------------------------------------------------------------------------------------
| Id  | Operation           | Name  | Rows  | Bytes | Cost (%CPU)| Time     | Pstart| Pstop |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |       |   918K|    76M|   503   (5)| 00:00:07 |       |       |
|   1 |  PARTITION RANGE ALL|       |   918K|    76M|   503   (5)| 00:00:07 |     1 |    28 |
|   2 |   VIEW              |       |   918K|    76M|   503   (5)| 00:00:07 |       |       |
|   3 |    TABLE ACCESS FULL| SALES |   918K|    25M|   503   (5)| 00:00:07 |     1 |    28 |
-----------------------------------------------------------------------------------------------
```

We see from this example that the optimizer chose to access SALES with a full table scan. This is required because of the `select * from sales` inside the round brackets. Then the optimizer chose to create a view on that data and select all of the partitions to create the selected data. So the optimizer collected all the data inside the bracket then presented the data to the next `select`. Now, however, let's give the optimizer a chance to show us how clever it is. Now we run the query but without our hint. Now the optimizer can use query optimizations.

```
SQL> select  * from (select * from sales);
Execution Plan
----------------------------------------------------------
Plan hash value: 1550251865
-----------------------------------------------------------------------------------------------
| Id  | Operation           | Name  | Rows  | Bytes | Cost (%CPU)| Time     | Pstart| Pstop |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |       |   918K|    25M|   503   (5)| 00:00:07 |       |       |
|   1 |  PARTITION RANGE ALL|       |   918K|    25M|   503   (5)| 00:00:07 |     1 |    28 |
|   2 |   TABLE ACCESS FULL | SALES |   918K|    25M|   503   (5)| 00:00:07 |     1 |    28 |
-----------------------------------------------------------------------------------------------
```

Now we see that the optimizer has chosen a simpler plan. The `VIEW` step has been eliminated and is in fact now the same execution plan as the select statement with no sub-query:

```
SQL> select * from sales;

Execution Plan
----------------------------------------------------------
Plan hash value: 1550251865

-----------------------------------------------------------------------------------------------
| Id  | Operation           | Name  | Rows  | Bytes | Cost (%CPU)| Time     | Pstart| Pstop |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |       |   918K|    25M|   503   (5)| 00:00:07 |       |       |
|   1 |  PARTITION RANGE ALL|       |   918K|    25M|   503   (5)| 00:00:07 |     1 |    28 |
|   2 |   TABLE ACCESS FULL | SALES |   918K|    25M|   503   (5)| 00:00:07 |     1 |    28 |
-----------------------------------------------------------------------------------------------
```

Just to prove the point that query transformations are doing something, I show an example below that nests the `select` statements, and we see from the execution plan that the optimizer has added a view for every layer. This is because I have told the optimizer to not use any query transformations with the hint `/*+ no_query_transformation */`.

```
SQL> select /*+ no_query_transformation */ * from (select * from (select * from (
select * from sales)));
Execution Plan
----------------------------------------------------------
Plan hash value: 1018107371
---------------------------------------------------------------------------------------------
| Id  |Operation            |Name  |Rows  | Bytes | Cost (%CPU)| Time     | Pstart| Pstop |
---------------------------------------------------------------------------------------------
|   0 |SELECT STATEMENT     |      |  918K|    76M|   503   (5)| 00:00:07 |       |       |
|   1 | PARTITION RANGE ALL |      |  918K|    76M|   503   (5)| 00:00:07 |     1 |    28 |
|   2 |  VIEW               |      |  918K|    76M|   503   (5)| 00:00:07 |       |       |
|   3 |   VIEW              |      |  918K|    76M|   503   (5)| 00:00:07 |       |       |
|   4 |    VIEW             |      |  918K|    76M|   503   (5)| 00:00:07 |       |       |
|   5 |     TABLE ACCESS FULL|SALES |  918K|    25M|   503   (5)| 00:00:07 |     1 |    28 |
---------------------------------------------------------------------------------------------
```

If we go back to the original question at the beginning of this chapter “Can this query be simplified?”, we see that in this simple case the answer is “yes.” This partially answers the question: “Why do we simplify queries?” The answer to this question is that the cost-based optimizer knows some rules it can use to save the execution engine from doing some work. The example given is very simple and for illustrative purposes only, but it does explain why we need to do query transformation.

The next question you might ask is “if the CBO knows all these rules to simplify my query, why do I need to know about it and what has this got to do with SQLT?” Let me answer the second question first.

SQLT, as a matter of routine, produces many different files that are useful. The main one is the HTML file we’ve looked at so far, but there are also trace files. One of the trace files is a “10053” trace file, which I’ll explain in the next section. The answer to the first question is simply that when things go wrong with the CBO we sometimes need to look at the 10053 trace file to get more detailed information on what decisions the optimizer made during parsing. You can then check these steps against what you consider sensible and decide if the optimizer is making a mistake or has insufficient statistics or some other input is influencing the optimizer in some way which is producing poorly performing SQL.

The 10053 Trace File

The 10053 trace file is the result of a request to trace an SQL statement. More properly, it is a request to trace the steps the optimizer makes to produce the execution plan. The contents are verbose and cryptic but extremely useful if you want to understand what is happening with query transformation.

How Do I Get a 10053 Trace File?

SQLTXPLAIN generates the 10053 trace file automatically and places it in the main reports directory. It has 10053 in the name so it’s pretty easy to spot. This is an example file name from the SQLT reports directory:

```
sqlt_s89909_10046_10053_execute.trc
```

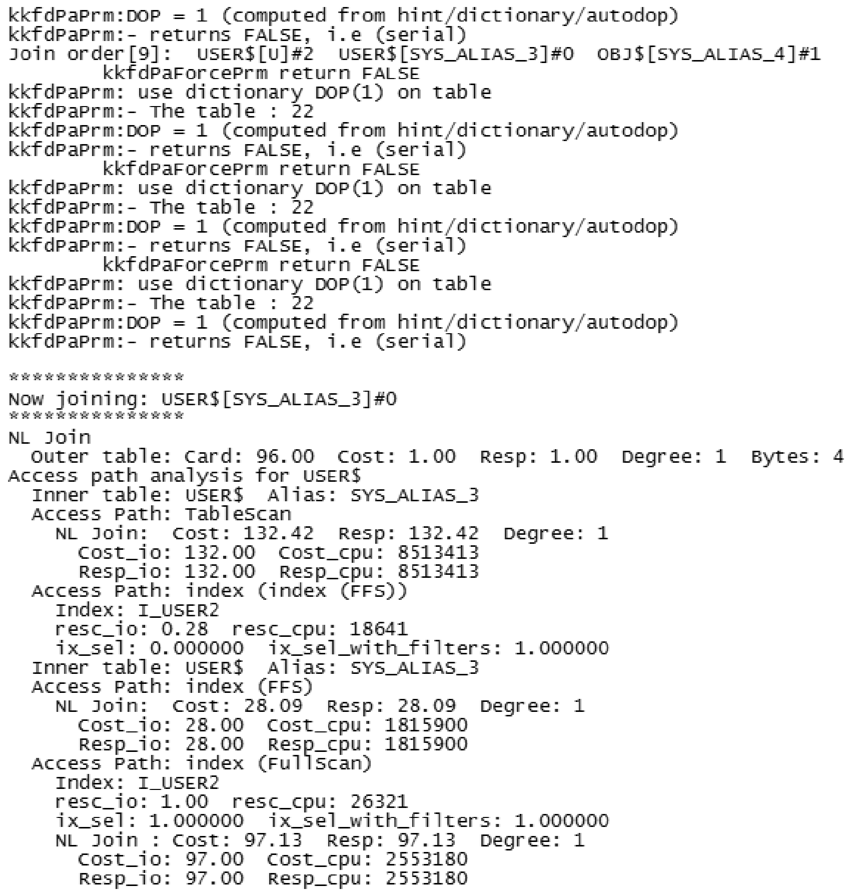
If you want to collect 10053 manually (i.e., without using SQLTXPLAIN) then these are the commands:

```
SQL> ALTER SESSION SET MAX_DUMP_FILE_SIZE = UNLIMITED;
SQL> ALTER SESSION SET TRACEFILE_IDENTIFIER = 'MY_10053_TRACE';
SQL> ALTER SESSION SET EVENTS '10053 TRACE NAME CONTEXT FOREVER, LEVEL 1';
SQL> select sysdate from dual;
SQL> exit
```

In my case this produces a trace file (with a file extenstion of `trc`) in the `F:\app\Stelios\diag\rdbms\snc1\snc1\trace` directory on my PC. The file is called `snc1_ora_2872_MY_10053_TRACE.trc`. The location is dependent on your operating system and system parameters. This is why it's useful to set the `tracefile_identifier` parameter, which allows you an easy way to spot your file among all the others. Notice how in my case the instance name (snc1) and ora are placed at the beginning of the name, then the `tracefile_identifier` value is used, and finally the standard extension makes the full file name. We also set `max_dump_file_size=unlimited` so that the file is not truncated just as we reach the interesting part.

What's in a 10053 Trace File?

I remember when I first looked inside a 10053 (many years ago now) thinking, *What is all this gibberish, and how can I possibly understand it*? None of this is documented, and there are many, many short codes for information that are not explained anywhere. It's not as bad as it sounds though: the 10053 trace file is, simply put, a log of what the cost-based optimizer "thought" as it parsed the query in question. It is literally a log of every step considered by the optimizer. It is not often considered a user facing file, so although it is purely text, it is pretty difficult to understand and pretty verbose. After all, the file is written to allow debugging of the optimization process and so has to include everything the optimizer does in great detail. The file was created so that support can fix problems that are discovered in the optimizer. No one to my knowledge has attempted to make the file user friendly. An example section from a 10053 trace file is shown in Figure 5-1.



```
kkfdPaPrm:DOP = 1 (computed from hint/dictionary/autodop)
kkfdPaPrm:- returns FALSE, i.e (serial)
Join order[9]:  USER$[U]#2  USER$[SYS_ALIAS_3]#0  OBJ$[SYS_ALIAS_4]#1
        kkfdPaForcePrm return FALSE
kkfdPaPrm: use dictionary DOP(1) on table
kkfdPaPrm:- The table : 22
kkfdPaPrm:DOP = 1 (computed from hint/dictionary/autodop)
kkfdPaPrm:- returns FALSE, i.e (serial)
        kkfdPaForcePrm return FALSE
kkfdPaPrm: use dictionary DOP(1) on table
kkfdPaPrm:- The table : 22
kkfdPaPrm:DOP = 1 (computed from hint/dictionary/autodop)
kkfdPaPrm:- returns FALSE, i.e (serial)
        kkfdPaForcePrm return FALSE
kkfdPaPrm: use dictionary DOP(1) on table
kkfdPaPrm:- The table : 22
kkfdPaPrm:DOP = 1 (computed from hint/dictionary/autodop)
kkfdPaPrm:- returns FALSE, i.e (serial)

***************
Now joining: USER$[SYS_ALIAS_3]#0
***************
NL Join
  Outer table: Card: 96.00  Cost: 1.00  Resp: 1.00  Degree: 1  Bytes: 4
Access path analysis for USER$
  Inner table: USER$  Alias: SYS_ALIAS_3
  Access Path: TableScan
    NL Join:  Cost: 132.42  Resp: 132.42  Degree: 1
      Cost_io: 132.00  Cost_cpu: 8513413
      Resp_io: 132.00  Resp_cpu: 8513413
  Access Path: index (index (FFS))
    Index: I_USER2
    resc_io: 0.28  resc_cpu: 18641
    ix_sel: 0.000000  ix_sel_with_filters: 1.000000
  Inner table: USER$  Alias: SYS_ALIAS_3
  Access Path: index (FFS)
    NL Join:  Cost: 28.09  Resp: 28.09  Degree: 1
      Cost_io: 28.00  Cost_cpu: 1815900
      Resp_io: 28.00  Resp_cpu: 1815900
  Access Path: index (FullScan)
    Index: I_USER2
    resc_io: 1.00  resc_cpu: 26321
    ix_sel: 1.000000  ix_sel_with_filters: 1.000000
    NL Join : Cost: 97.13  Resp: 97.13  Degree: 1
      Cost_io: 97.00  Cost_cpu: 2553180
      Resp_io: 97.00  Resp_cpu: 2553180
```

Figure 5-1. *A section of an example 10053 trace file*

Despite the text in a 10053 trace file being difficult to understand, we can see snippets of information that begin to make sense. For example, we've mentioned that NL is short for "nested loop" and that is a kind of join. We see that USER$ is having its Access path analyzed. USER$ was one of the tables involved in this query. We also see references to Cost (we've discussed those in Chapters 1 and 2), and we see references to different types of access. For example, index (FFS) – which is short for Index Fast Full Scan. This is the approach to take with a 10053 trace file. Do not try and understand every line of it. You will not be able to because some lines cannot be decoded unless you are a developer working for Oracle. There are a number of different ways to get 10053 trace file information. The simplest way is to use SQLT XTRACT (as described in Chapter 1). This will generate a 10053 trace file and include it in the ZIP file. Sometimes, however, you do not want all the information that SQLT gives you and maybe you are only after the 10053 trace file, in which case you could use the DBMS_SQLDIAG package as long as you have the SQL ID of the SQL statement you are interested in.

Another advantage DBMS_SQLDIAG has is that you don't need to execute the statement. As long as you have the SQL ID you can get the 10053 trace. This feature is only available from 11g Release 2, however. The steps are as follows:

1. First we find the `sql_id`, by knowing the text in the SQL we can search `v$sql`.
2. Then we can use the `dbms_sqldiag.dump_trace` routine to get the 10053 trace file. This puts the trace file in the `user_dump_dest` location, where we can review it with any text editor.

Let's see those steps in action:

```
SQL> column sql_text format a30
SQL> select sysdate from dual;
SYSDATE
---------
05-OCT-12

SQL> select sql_id from v$sql where sql_text like 'select sysdate from dual%';

SQL_ID
-------------
7h35uxf5uhmm1

SQL> execute dbms_sqldiag.dump_trace(p_sql_id=>'7h35uxf5uhmm1',
  p_child_number=>0,
  p_component=>'Compiler',p_file_id=>'DIAG');

PL/SQL procedure successfully completed.

SQL> show parameter user_dump_Dest

NAME                 TYPE                             VALUE
-------------------- -------------------------------- -------
user_dump_dest       string    f:\app\stelios\diag\rdbms\snc1\snc1\trace

SQL> host dir f:\app\stelios\diag\rdbms\snc1\snc1\trace\*DIAG*.trc
 Volume in drive F is My Passport
 Volume Serial Number is 1E65-69CC

 Directory of f:\app\stelios\diag\rdbms\snc1\snc1\trace
```

```
09/15/2012  12:41 PM              1,096 snc1_diag_2548.trc
10/05/2012  12:54 PM             68,133 snc1_ora_4980_DIAG.trc
              2 File(s)          69,229 bytes
              0 Dir(s)  788,285,857,792 bytes free

SQL> host notepad f:\app\stelios\diag\rdbms\snc1\snc1\trace\snc1_ora_4980_DIAG.trc
```

This method, which is part of the event infrastructure, has the added advantage that it can capture trace for an SQL statement inside a PL/SQL block. An example of this is shown below. A package specification and body are created to calculate the area of a circle. Then we identify the SQL ID inside the PL/SQL package and trace only the matching statement.

```
SQL> host type area.sql
create or replace package getcircarea as
  function getcircarea(radius number)
  return number;
end getcircarea;
/

create or replace package body getcircarea as
  function getcircarea (radius number) return number
  is area number(8,2);
  begin
    select 3.142*radius*radius into area from dual;
    return area;
  end;
  end getcircarea;
/

set serveroutput on size 100000;

declare
  area number(8,2);
  begin
    area:= getcircarea.getcircarea(10);
    dbms_output.put_line('Area is '||area);
  end;
/

SQL> select sql_text, sql_id from v$sqlarea where sql_text like '%3.142%';

SQL_TEXT
--------------------------------------------------------------------------------
SQL_ID
-------------
SELECT 3.142*:B1 *:B1 FROM DUAL
9rjmrhbjuasav

select sql_text, sql_id from v$sqlarea where sql_text like '%3.142%'
ggux6y542z8mr
```

```
alter session set tracefile_identifier='PLSQL';
alter session set events 'trace[rdbms.SQL_Optimizer.*][sql:9rjmrhbjuasav]';

SQL> @area

Package created.


Package body created.

Area is 314.2

PL/SQL procedure successfully completed.


Session altered.

SQL> host dir f:\app\stelios\diag\rdbms\snc1\snc1\trace\*.trc
 Volume in drive F is My Passport
 Volume Serial Number is 1E65-69CC

 Directory of f:\app\stelios\diag\rdbms\snc1\snc1\trace

10/06/2012  09:49 AM             2,593 snc1_dbrm_2404.trc
10/06/2012  09:46 AM             1,175 snc1_j000_4956.trc
10/06/2012  09:46 AM           130,630 snc1_ora_4804_PLSQL.trc
10/06/2012  06:00 AM               973 snc1_vkrm_2948.trc
               4 File(s)        135,371 bytes
               0 Dir(s)  799,840,133,120 bytes free
```

No more excuses about not being able to get the trace file because the SQL is inside a PL/SQL block. You can turn off tracing for that SQL ID afterwards with

```
ALTER SESSION SET EVENTS 'trace[rdbms.SQL_Optimizer.*]off';
```

The old fashioned way to collect 10053 is `alter session`, which was mentioned earlier, and which works well enough. It is also easier to remember and is by far the most popular way to collect trace, as it works with most versions of Oracle.

```
ALTER SESSION SET EVENTS '10053 trace name context forever, level 1';
```

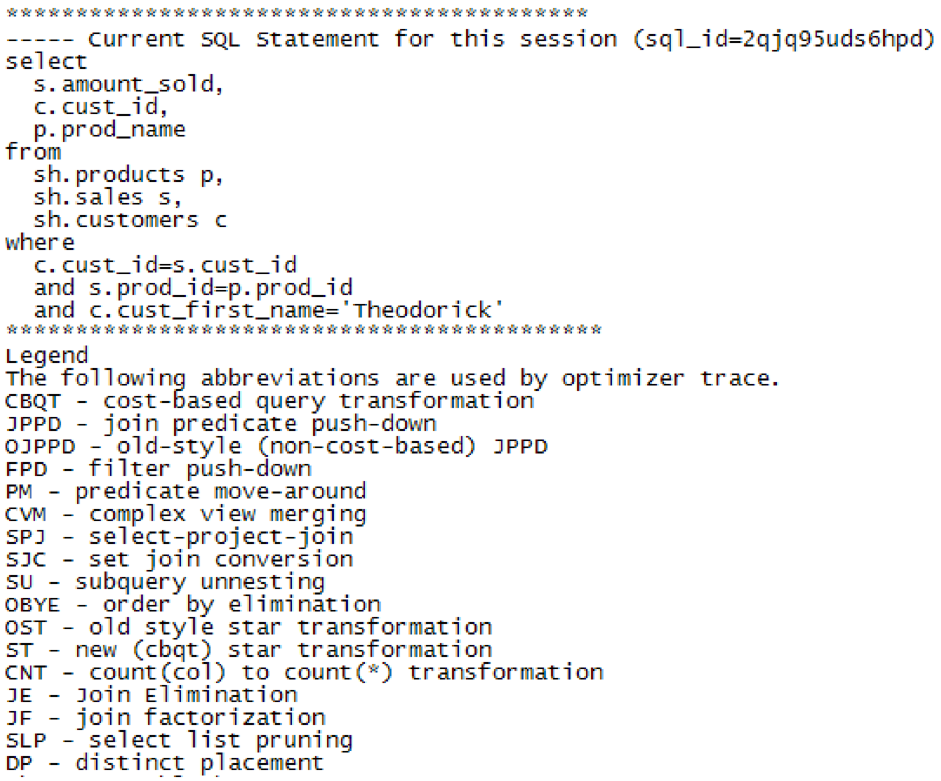
What Is a Query Transformation?

Now that we know how to get a 10053 (and SQLT makes that particularly easy), we need to look at some examples of queries (and some query transformations) that can be carried out on them. As we said earlier in the chapter, query transformation is the process the optimizer carries out to change an SQL statement to another SQL statement that is logically the same (and will give the same result).

Here is a list of common query transformations and their codes:

- Subquery Unnesting: SU
- Complex View Merging: CVM
- Join Predicate Push Down: JPPD

Luckily if you want a full listing of the abbreviations and their meanings you can look in the 10053 trace file in the "Legend" section. Look at Figure 5-2 for an example. A section like this is shown in every 10053 trace file, and it shows all the query transformations that can be used for all SQL statements. This section is not specific to the SQL statement being examined.



```
******************************************
----- Current SQL Statement for this session (sql_id=2qjq95uds6hpd)
select
  s.amount_sold,
  c.cust_id,
  p.prod_name
from
  sh.products p,
  sh.sales s,
  sh.customers c
where
  c.cust_id=s.cust_id
  and s.prod_id=p.prod_id
  and c.cust_first_name='Theodorick'
*******************************************
Legend
The following abbreviations are used by optimizer trace.
CBQT - cost-based query transformation
JPPD - join predicate push-down
OJPPD - old-style (non-cost-based) JPPD
FPD - filter push-down
PM - predicate move-around
CVM - complex view merging
SPJ - select-project-join
SJC - set join conversion
SU - subquery unnesting
OBYE - order by elimination
OST - old style star transformation
ST - new (cbqt) star transformation
CNT - count(col) to count(*) transformation
JE - Join Elimination
JF - join factorization
SLP - select list pruning
DP - distinct placement
```

Figure 5-2. *The Legend section of the 10053 trace file*

Subquery unnesting, the first query transformation on our list, is formally defined as a query transformation that converts a subquery into a join in the outer query, which then allows subquery tables to be considered for join order, access paths and join methods. We'll look at this example of query optimization because it is a commonly used one; examples for this are also easy to explain and to generate!

An example query that can use subquery unnesting is:

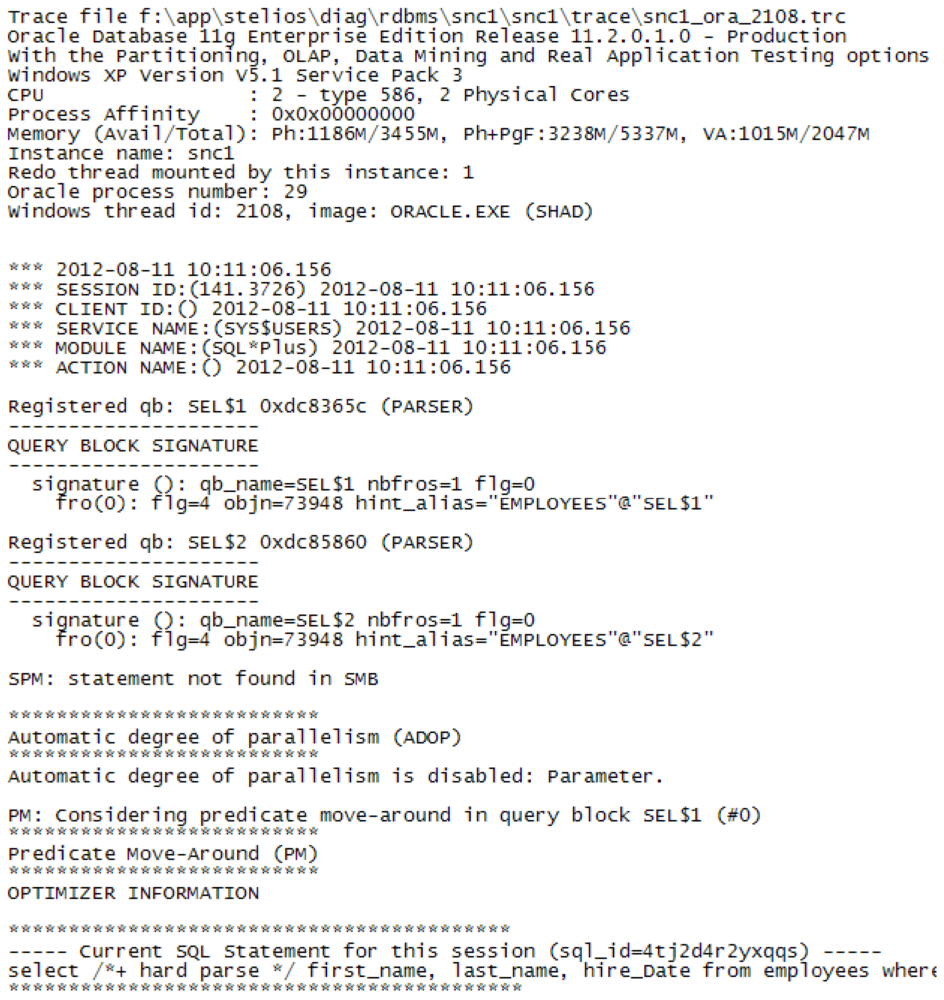
```
Select
first_name,
last_name,
hire_Date
```

```
from employees
where
hire_date IN (
select hire_date from employees where department_id = 30
);
```

The subquery part is inside the brackets, and in this subquery example we are using IN. Let's see what happens when we trace this query with event 10053, as shown below:

```
SQL> connect hr/hr
Connected.
SQL> alter session set max_dump_file_size=unlimited;

Session altered.

SQL> alter session set events '10053 trace name context forever, level 1';

Session altered.

SQL> select /*+ hard parse */ first_name, last_name, hire_Date
  2  from employees where hire_date in
  3  (select hire_date from employees where
  4  department_id=30);

FIRST_NAME           LAST_NAME                 HIRE_DATE
-------------------- ------------------------- ---------
Den                  Raphaely                  07-DEC-02
Alexander            Khoo                      18-MAY-03
Shelli               Baida                     24-DEC-05
Sigal                Tobias                    24-JUL-05
Guy                  Himuro                    15-NOV-06
Karen                Colmenares                10-AUG-07

6 rows selected.

SQL> connect / as sysdba
Connected.
SQL> show parameter user_dump_Dest

NAME                                 TYPE        VALUE
------------------------------------ ----------- ------------------------------
user_dump_dest                       string      f:\app\stelios\diag\rdbms\snc1
                                                 \snc1\trace
```

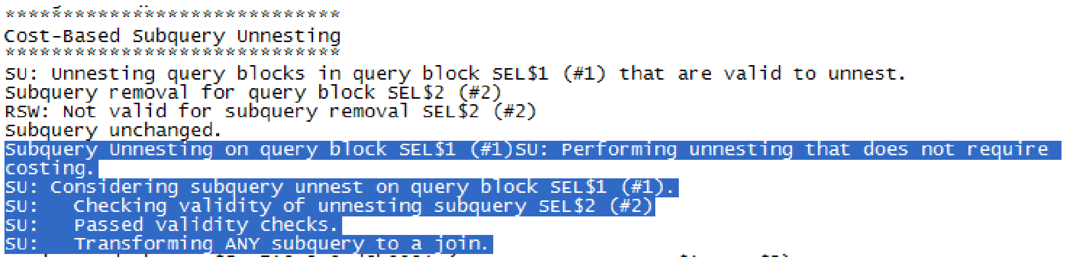
The trace file is located in wherever user_dump_dest points to. If we edit that trace file, we'll see the header shown in Figure 5-3.



```
Trace file f:\app\stelios\diag\rdbms\snc1\snc1\trace\snc1_ora_2108.trc
Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - Production
With the Partitioning, OLAP, Data Mining and Real Application Testing options
Windows XP Version V5.1 Service Pack 3
CPU                 : 2 - type 586, 2 Physical Cores
Process Affinity    : 0x0x00000000
Memory (Avail/Total): Ph:1186M/3455M, Ph+PgF:3238M/5337M, VA:1015M/2047M
Instance name: snc1
Redo thread mounted by this instance: 1
Oracle process number: 29
Windows thread id: 2108, image: ORACLE.EXE (SHAD)


*** 2012-08-11 10:11:06.156
*** SESSION ID:(141.3726) 2012-08-11 10:11:06.156
*** CLIENT ID:() 2012-08-11 10:11:06.156
*** SERVICE NAME:(SYS$USERS) 2012-08-11 10:11:06.156
*** MODULE NAME:(SQL*Plus) 2012-08-11 10:11:06.156
*** ACTION NAME:() 2012-08-11 10:11:06.156

Registered qb: SEL$1 0xdc8365c (PARSER)
---------------------
QUERY BLOCK SIGNATURE
---------------------
  signature (): qb_name=SEL$1 nbfros=1 flg=0
    fro(0): flg=4 objn=73948 hint_alias="EMPLOYEES"@"SEL$1"

Registered qb: SEL$2 0xdc85860 (PARSER)
---------------------
QUERY BLOCK SIGNATURE
---------------------
  signature (): qb_name=SEL$2 nbfros=1 flg=0
    fro(0): flg=4 objn=73948 hint_alias="EMPLOYEES"@"SEL$2"

SPM: statement not found in SMB

**************************
Automatic degree of parallelism (ADOP)
**************************
Automatic degree of parallelism is disabled: Parameter.

PM: Considering predicate move-around in query block SEL$1 (#0)
**************************
Predicate Move-Around (PM)
**************************
OPTIMIZER INFORMATION

******************************************
----- Current SQL Statement for this session (sql_id=4tj2d4r2yxqqs) -----
select /*+ hard parse */ first_name, last_name, hire_Date from employees wher
*******************************************
```

Figure 5-3. *The first page of the example 10053 trace file*

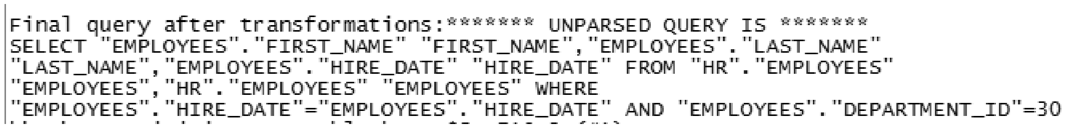
We see the usual signature text, telling us about the operating system, the version of Oracle and so on. The important part for this discussion is to check that the SQL statement we thought we were parsing is found under the “Current SQL statement for this session” section (at the bottom of Figure 5-3). In our case we put the hint `/*+ hard parse */` in the original SQL, and this has appeared in the 10053 section under the “Current SQL section”. So we’re pretty sure it’s the right SQL. So how do we know if subquery unnesting is taking place? We search the 10053 trace file for text similar to that in Figure 5-4. We would search for “subquery unnesting”, but I’ve highlighted the section relating to subquery unnesting. Notice the “SU” at the beginning of the lines. This tells you the optimizer is looking at the query with regard to subquery unnesting.



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

Figure 5-4. *The text in the 10053 trace file shows subquery unnesting taking place*

We can also see what the new SQL is after it has been transformed. See Figure 5-5 to see what our test query has been changed into.



```
Final query after transformations:******* UNPARSED QUERY IS *******
SELECT "EMPLOYEES"."FIRST_NAME" "FIRST_NAME","EMPLOYEES"."LAST_NAME"
"LAST_NAME","EMPLOYEES"."HIRE_DATE" "HIRE_DATE" FROM "HR"."EMPLOYEES"
"EMPLOYEES","HR"."EMPLOYEES" "EMPLOYEES" WHERE
"EMPLOYEES"."HIRE_DATE"="EMPLOYEES"."HIRE_DATE" AND "EMPLOYEES"."DEPARTMENT_ID"=30
```

Figure 5-5. *The final query after transformation*

Does this query look semantically the same as the original query to you? This was the original query:

```
Select
first_name,
last_name,
hire_Date
from employees
where
hire_date IN (
select hire_date from employees where department_id = 30
);
```

and this is the new query:

```
Select
first_name,
last_name,
hire_date
from employees A, employees B
where
A.hire_date = B.hire_date
and
A.department_id=30;
```

We see from the above example what subquery unnesting is, from a fairly simple example. The objective of subquery unnesting is to allow the optimizer to possibly use other joins or table orders to get satisfactory results more efficiently. In other words, by removing the subquery we give the optimizer more freedom to use other joins because we've moved the table name up a level into the main query. There are many variations on subquery unnesting.

For example, the subquery could utilize `NOT IN` or `NOT EXISTS`. We will not cover all the variations and combinations of this technique or cover the other query transformations. (You could easily write a whole book on just query transformations.) Suffice it to say the 10053 trace file will list what it has considered and show what it has done. The question you should be asking at this point is "Why do I care, what the optimizer is doing 'under the hood' if it works properly?"

Why Would We Want to Disable Query Transformations?

There are situations in which you may want to turn off particular transformations: for example, where an optimizer bug is causing a problem. This may be because Oracle support has suggested changes or indicated that there is a problem with a query transformation. You may also see some of the hidden parameters shown below on a system that you are managing. Here are some cost-based optimizer parameters that can influence the CBO with regard to query transformation:

- `_complex_view_merging`
- `_convert_set_to_join`
- `_unnest_subquery`
- `_optimizer_cost_based_transformation`
- `_optimizer_extend_jppd_view_types`
- `_optimizer_filter_pred_pullup`
- `_optimizer_group_by_placement`
- `_optimizer_improve_selectivity`
- `_optimizer_join_elimination_enabled`
- `_optimizer_join_factorization`
- `_optimizer_multi_level_push_pred`
- `_optimizer_native_full_outer_join`
- `_optimizer_order_by_elimination_enabled`
- `_optimizer_push_pred_cost_based`
- `_optimizer_unnest_all_subqueries`
- `_optimizer_unnest_corr_set_subq`
- `_optimizer_squ_bottomup`
- `_optimizer_null_aware_antijoin`
- `_pred_move_around`
- `_push_join_predicate`

These are some hints that influence this behavior:

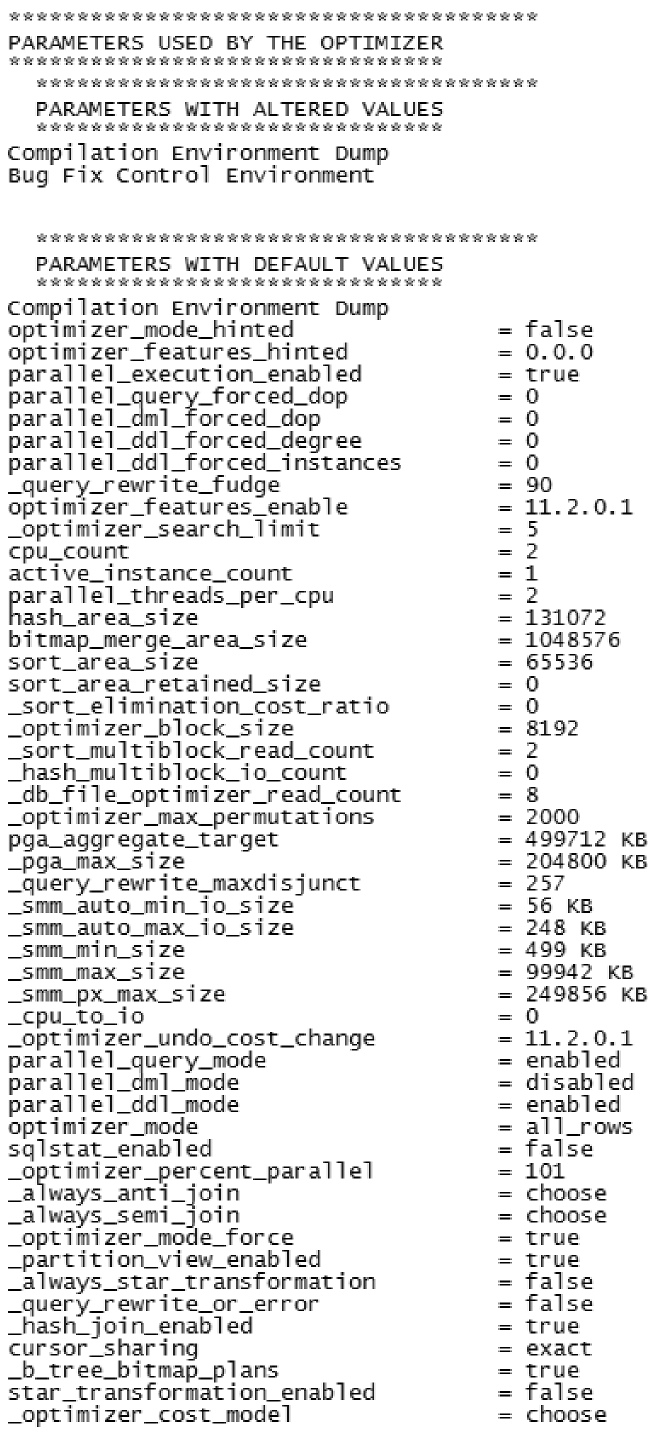
- `first_rows(n)`
- `no_query_transformation`
- `unnest`

- `no_unnest`
- `push_pred`
- `no_push_pred`
- `push_subq`
- `native_full_outer_join`
- `no_push_subq`
- `no_set_to_join`
- `qb_name`

This is by no means a complete list of all the hidden parameters influencing the optimizer. These parameters can be used to turn on and off certain features. For example, "`unnest_subquery`" has a default value of TRUE (for versions of Oracle after 9.0) or later. In most situations you would only notice the problems if you were tuning SQL and found that some hint or some system parameter was not changing the execution plan the way you would expect. In some situations it is merely a lack of performance (or a sudden change of performance) that is the clue that something may be wrong. These parameters should only be set if Oracle support asks you to set them. They are generally only set for a short time to debug certain issues associated with these transformations and bugs associated with them, but there's no reason not to open a Service Request with Oracle and ask if you can try something. Support people at Oracle are all very helpful and accommodating (honest).

Optimizer Parameters

Luckily the 10053 trace file lists optimizer parameters and their default values so that you can tell if there have been any changes. In the section titled "PARAMETERS USED BY THE OPTIMIZER" (as shown in Figure 5-6, we see the first few entries of these parameters. Note that "Compilation Environment Dump" and "Bug Fix Control Environment" are subheadings present in both the altered values and default values section. In our case there are no altered values for either of these subsections in the altered values section. In the default values section the first actual parameter shown with a default value is `optimizer_mode_hinted` (which is set to false). All the parameters are listed (including the hidden ones). Play with these at your peril, however. If support asks you to change one of these parameters, it is because they are helping you with some problem related to the optimizer. Also note that any non-default parameters are shown in a separate section entitled "PARAMETERS WITH ALTERED VALUES".



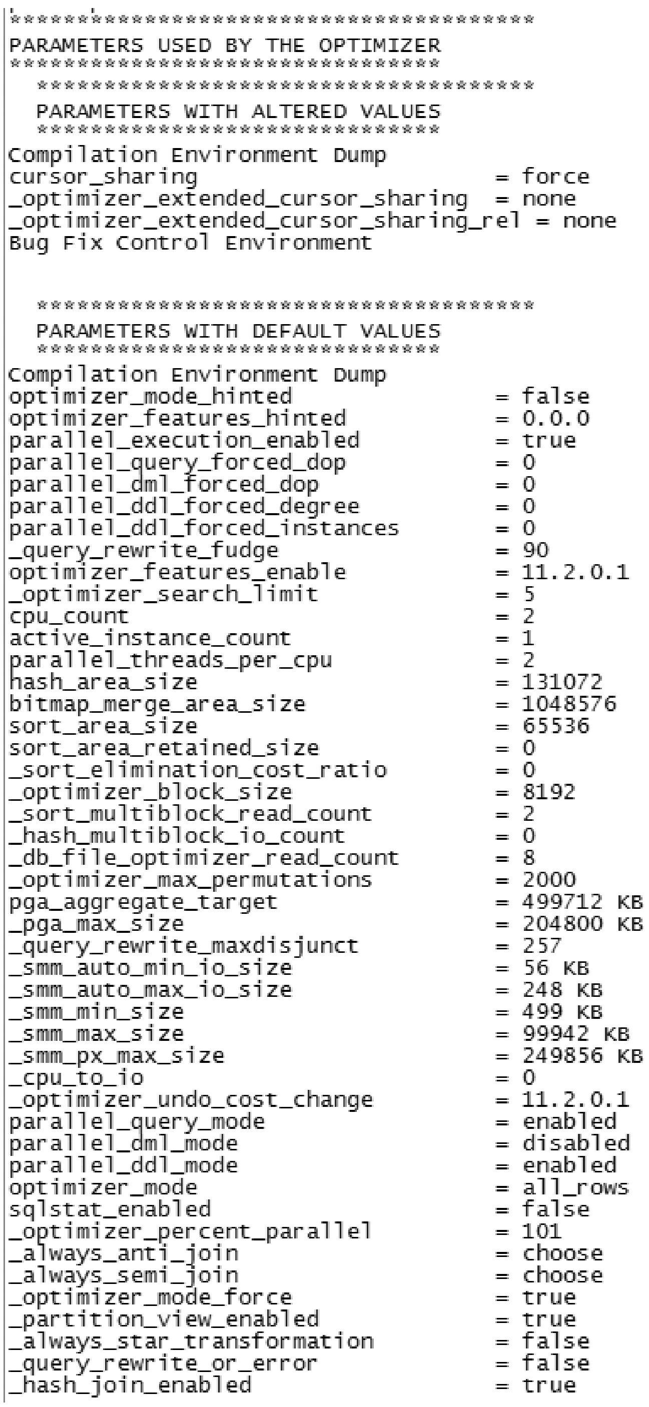
```
***************************************
PARAMETERS USED BY THE OPTIMIZER
********************************
  ***************************************
  PARAMETERS WITH ALTERED VALUES
  ********************************
Compilation Environment Dump
Bug Fix Control Environment


  ***************************************
  PARAMETERS WITH DEFAULT VALUES
  ********************************
Compilation Environment Dump
optimizer_mode_hinted                = false
optimizer_features_hinted            = 0.0.0
parallel_execution_enabled           = true
parallel_query_forced_dop            = 0
parallel_dml_forced_dop              = 0
parallel_ddl_forced_degree           = 0
parallel_ddl_forced_instances        = 0
_query_rewrite_fudge                 = 90
optimizer_features_enable            = 11.2.0.1
_optimizer_search_limit              = 5
cpu_count                            = 2
active_instance_count                = 1
parallel_threads_per_cpu             = 2
hash_area_size                       = 131072
bitmap_merge_area_size               = 1048576
sort_area_size                       = 65536
sort_area_retained_size              = 0
_sort_elimination_cost_ratio         = 0
_optimizer_block_size                = 8192
_sort_multiblock_read_count          = 2
_hash_multiblock_io_count            = 0
_db_file_optimizer_read_count        = 8
_optimizer_max_permutations          = 2000
pga_aggregate_target                 = 499712 KB
_pga_max_size                        = 204800 KB
_query_rewrite_maxdisjunct           = 257
_smm_auto_min_io_size                = 56 KB
_smm_auto_max_io_size                = 248 KB
_smm_min_size                        = 499 KB
_smm_max_size                        = 99942 KB
_smm_px_max_size                     = 249856 KB
_cpu_to_io                           = 0
_optimizer_undo_cost_change          = 11.2.0.1
parallel_query_mode                  = enabled
parallel_dml_mode                    = disabled
parallel_ddl_mode                    = enabled
optimizer_mode                       = all_rows
sqlstat_enabled                      = false
_optimizer_percent_parallel          = 101
_always_anti_join                    = choose
_always_semi_join                    = choose
_optimizer_mode_force                = true
_partition_view_enabled              = true
_always_star_transformation          = false
_query_rewrite_or_error              = false
_hash_join_enabled                   = true
cursor_sharing                       = exact
_b_tree_bitmap_plans                 = true
star_transformation_enabled          = false
_optimizer_cost_model                = choose
```

Figure 5-6. *The optimizer parameter section of the 10053 trace file*

As an example, let's see what happens if we change the parameter cursor sharing from its default value of EXACT to a value of FORCE. Here we set cursor_sharing to FORCE (it was EXACT before). Then we ask for a 10053 trace and issue a query; then we go to the user_dump_dest and look at the 10053 trace file and find the section with the parameters. Here are the commands issued:

```
SQL> show parameter cursor_sharing
NAME                                 TYPE        VALUE
------------------------------------ ----------- ---------------------------
cursor_sharing                       string      EXACT
_optimizer_extended_cursor_sharing  = none
_optimizer_extended_cursor_sharing_rel = none

SQL> alter system set cursor_sharing=FORCE scope=memory;

System altered.

SQL> alter session set events '10053 trace name context forever, level 1';

Session altered.

SQL> explain plan for select count(*) from dba_objects;

Explained.
```

And in the figure below (Figure 5-7) we see that cursor_sharing has been changed to force.



```
***************************************
PARAMETERS USED BY THE OPTIMIZER
********************************
  *************************************
  PARAMETERS WITH ALTERED VALUES
  ******************************
Compilation Environment Dump
cursor_sharing                       = force
_optimizer_extended_cursor_sharing   = none
_optimizer_extended_cursor_sharing_rel = none
Bug Fix Control Environment


  *************************************
  PARAMETERS WITH DEFAULT VALUES
  ******************************
Compilation Environment Dump
optimizer_mode_hinted                = false
optimizer_features_hinted            = 0.0.0
parallel_execution_enabled           = true
parallel_query_forced_dop            = 0
parallel_dml_forced_dop              = 0
parallel_ddl_forced_degree           = 0
parallel_ddl_forced_instances        = 0
_query_rewrite_fudge                 = 90
optimizer_features_enable            = 11.2.0.1
_optimizer_search_limit              = 5
cpu_count                            = 2
active_instance_count                = 1
parallel_threads_per_cpu             = 2
hash_area_size                       = 131072
bitmap_merge_area_size               = 1048576
sort_area_size                       = 65536
sort_area_retained_size              = 0
_sort_elimination_cost_ratio         = 0
_optimizer_block_size                = 8192
_sort_multiblock_read_count          = 2
_hash_multiblock_io_count            = 0
_db_file_optimizer_read_count        = 8
_optimizer_max_permutations          = 2000
pga_aggregate_target                 = 499712 KB
_pga_max_size                        = 204800 KB
_query_rewrite_maxdisjunct           = 257
_smm_auto_min_io_size                = 56 KB
_smm_auto_max_io_size                = 248 KB
_smm_min_size                        = 499 KB
_smm_max_size                        = 99942 KB
_smm_px_max_size                     = 249856 KB
_cpu_to_io                           = 0
_optimizer_undo_cost_change          = 11.2.0.1
parallel_query_mode                  = enabled
parallel_dml_mode                    = disabled
parallel_ddl_mode                    = enabled
optimizer_mode                       = all_rows
sqlstat_enabled                      = false
_optimizer_percent_parallel          = 101
_always_anti_join                    = choose
_always_semi_join                    = choose
_optimizer_mode_force                = true
_partition_view_enabled              = true
_always_star_transformation          = false
_query_rewrite_or_error              = false
_hash_join_enabled                   = true
```

Figure 5-7. *Here we see the parameter cursor_sharing has been changed to* `FORCE`

We see that the 10053 trace file is useful for tracking optimizer parameters both hidden and unhidden. Now we'll look at optimizer hints.

Optimizer Hints

If you use any optimizer hints in your SQL, it is a good idea to check that these hints are being used. The optimizer will do its best to use hints given to it but will not use the hints if there is a problem with such use. Reasons why the optimizer will ignore hints are incorrect syntax or a conflict in the hint with another hint. If the hints are correct the optimizer will use them. Let's look at an example involving the query we were previously working on. Now, armed with our knowledge of subquery unnesting, we'll write the query a different way and try a few hints. First we'll run the query with no hint and look at the execution plan. It will show a hash join. Then we'll hint a nested loop join. We'll confirm that but look at the execution plan again. Then finally we'll use conflicting hints and see that the execution plan reverts to the hash join.

```
SQL> set autotrace traceonly explain;
set lines 100
select
  a.first_name, a.last_name, a.hire_date
  from employees a, employees b
  where a.hire_date = b.hire_date
  and a.department_id=30
/
Execution Plan
----------------------------------------------------------
Plan hash value: 2254211361

-------------------------------------------------------------------------------------------------
| Id  | Operation                    | Name               | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
| 0   | SELECT STATEMENT             |                    |     7 |   238 |     6  (17)| 00:00:01 |
|*1   |  HASH JOIN                   |                    |     7 |   238 |     6  (17)| 00:00:01 |
| 2   |   TABLE ACCESS BY INDEX ROWID| EMPLOYEES          |     6 |   156 |     2   (0)| 00:00:01 |
|*3   |    INDEX RANGE SCAN          | EMP_DEPARTMENT_IX  |     6 |       |     1   (0)| 00:00:01 |
| 4   |   TABLE ACCESS FULL          | EMPLOYEES          |   107 |   856 |     3   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------

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

   1 - access("A"."HIRE_DATE"="B"."HIRE_DATE")
   3 - access("A"."DEPARTMENT_ID"=30)
SQL> select /*+ use_nl(a b) */
  a.first_name, a.last_name, a.hire_date
  from employees a, employees b
  where a.hire_date = b.hire_date
  and a.department_id=30
/
Execution Plan
----------------------------------------------------------
Plan hash value: 3321434377
```

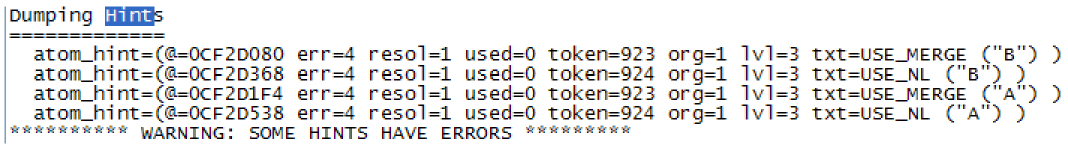
```
---------------------------------------------------------------------------------------------
| Id  | Operation                  | Name              | Rows  | Bytes | Cost (%CPU)| Time     |
---------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT           |                   |     7 |   238 |    12   (0)| 00:00:01 |
|   1 |  NESTED LOOPS              |                   |     7 |   238 |    12   (0)| 00:00:01 |
|   2 | TABLE ACCESS BY INDEX ROWID| EMPLOYEES         |     6 |   156 |     2   (0)| 00:00:01 |
|*  3 |  INDEX RANGE SCAN          | EMP_DEPARTMENT_IX |     6 |       |     1   (0)| 00:00:01 |
|*  4 | TABLE ACCESS FULL          | EMPLOYEES         |     1 |     8 |     2   (0)| 00:00:01 |
---------------------------------------------------------------------------------------------

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

   3 - access("A"."DEPARTMENT_ID"=30)
   4 - filter("A"."HIRE_DATE"="B"."HIRE_DATE")

SQL> select /*+ use_nl(a b) use_merge(a b) */
  a.first_name, a.last_name, a.hire_date
  from employees a, employees b
  where a.hire_date = b.hire_date
  and a.department_id=30
/

Execution Plan
----------------------------------------------------------
Plan hash value: 2254211361

---------------------------------------------------------------------------------------------
| Id  | Operation                   | Name              | Rows  | Bytes | Cost (%CPU)| Time     |
---------------------------------------------------------------------------------------------
|   0 |SELECT STATEMENT             |                   |     7 |   238 |     6  (17)| 00:00:01 |
|*  1 | HASH JOIN                   |                   |     7 |   238 |     6  (17)| 00:00:01 |
|   2 |  TABLE ACCESS BY INDEX ROWID|EMPLOYEES          |     6 |   156 |     2   (0)| 00:00:01 |
|*  3 |   INDEX RANGE SCAN          |EMP_DEPARTMENT_IX  |     6 |       |     1   (0)| 00:00:01 |
|   4 |  TABLE ACCESS FULL          |EMPLOYEES          |   107 |   856 |     3   (0)| 00:00:01 |
---------------------------------------------------------------------------------------------

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

   1 - access("A"."HIRE_DATE"="B"."HIRE_DATE")
   3 - access("A"."DEPARTMENT_ID"=30)
```

Let's look at each step and see if we can understand what happened. In the first execution we allowed the optimizer to do its own thing. We could also confirm this by looking at the 10053 trace if we wanted to make sure there was no influence on the optimizer from hidden parameters. So in the first step the optimizer went with a hash join. In the second execution, we decide (for whatever reason) we'd like to try a nested loop instead, so we use a hint in the SQL `use_nl (a b)`. Remember that `(a b)` represents the aliases of the two tables that are being joined in this hint. Gratifyingly the new execution plan uses a nested loop. So far so good.

Now, in the third execution, we want to refine things a little further and use an additional hint, `use_merge(a b)`. This seems somewhat contradictory, but we want to know what the optimizer will do. Will it use a hash join, will it use a sort merge join, or will it pick the best option from both of these? If we look at the resulting execution plan we see that it does neither of these things. Instead it uses a hash join. So rather than guessing we now generate a 10053 trace file of the "errant" behavior. In these steps we set the dump file size to unlimited ask for a 10053 trace file run the SQL with the two hints, that we want to investigate and then run the SQL.

```
SQL> alter session set max_dump_file_size=unlimited;

Session altered.

SQL> alter session set events '10053 trace name context forever, level 1';

Session altered.

SQL> set autotrace off;
SQL> select /*+ use_nl(a b) user_merge (a b) */
  2  a.first_name, a.last_name, a.hire_date
  3  from employees a, employees b
  4  where a.hire_date=b.hire_date
  5  and a.department_id=30
  6  /

FIRST_NAME           LAST_NAME                 HIRE_DATE
-------------------- ------------------------- ---------
Den                  Raphaely                  07-DEC-02
Alexander            Khoo                      18-MAY-03
Shelli               Baida                     24-DEC-05
Sigal                Tobias                    24-JUL-05
Guy                  Himuro                    15-NOV-06
Karen                Colmenares                10-AUG-07

6 rows selected.
```

We confirm we are looking at the right 10053 trace file by checking the text, and true enough, the hints match. If we now search for the word "hint" we'll find a section at the end of the trace file called "Dumping Hints". See Figure 5-8.



```
Dumping Hints
=============
  atom_hint=(@=0CF2D080 err=4 resol=1 used=0 token=923 org=1 lvl=3 txt=USE_MERGE ("B") )
  atom_hint=(@=0CF2D368 err=4 resol=1 used=0 token=924 org=1 lvl=3 txt=USE_NL ("B") )
  atom_hint=(@=0CF2D1F4 err=4 resol=1 used=0 token=923 org=1 lvl=3 txt=USE_MERGE ("A") )
  atom_hint=(@=0CF2D538 err=4 resol=1 used=0 token=924 org=1 lvl=3 txt=USE_NL ("A") )
********** WARNING: SOME HINTS HAVE ERRORS *********
```

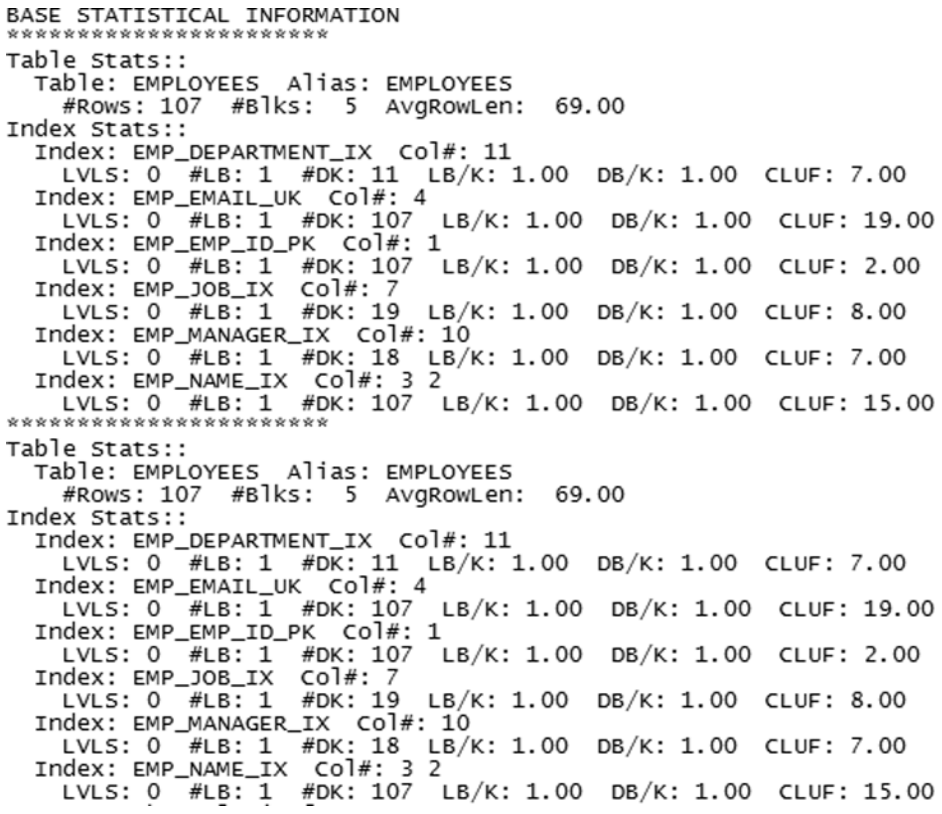
Figure 5-8. *The hints section of the 10053 trace file*

The important thing about this small section of the file is to note that used=0. This means the hint was not used. It was ignored completely. In fact both hints were ignored, which then allowed the optimizer to make its own decisions, which resulted in the hash join being used. The execution of the SQL resulted in no error. We did not get an error message from the optimizer saying there was something wrong with the hints. This information was hidden away in a 10053 trace file (if we asked for it). So what was wrong? The err code is 4, which means that there was a conflict with another hint. In this case both hints are taking each other out. The important thing to note here is that the third execution of the SQL above did not generate an error. The optimizer recognized the hints, found they were conflicting with each other, ignored them and developed a plan without them.

Cost Calculations

The most important job that the optimizer does is to compare costs between different execution plans. That's the overall plan of course. The cost of the execution plan is made up of the cost of each line in the execution plan, and each line is based on calculations that depend on statistics. The different plans calculated are based on different access paths, join methods and order of access of different parts of the query. Sounds like fun doesn't it? If you had seven different tables you could choose an order of processing in seven ways (that's seven factorial, which translates to 7x6x5x4x3x2x1)! That's 5,040 ways total, and that's before you consider the different ways of accessing the tables in the first place. The optimizer chooses a reasonable plan, picks the outer table and works down, tries a few combinations, and quickly eliminates those options that already exceed the cost of previous plans it has tried (this is highly simplified). So if, for example, the first plan the optimizer guesses is going to cost 1,000, the second plan will be abandoned if halfway through the cost calculations the plan reaches 1,500. In this case there is no need to keep calculating. It does this to save time in the overall hard parsing elapsed time. It knows you are hungry for results. Once you know that the 10053 is a story about how the optimizer derived its plan based on this approach, the trace file begins to make some sense.

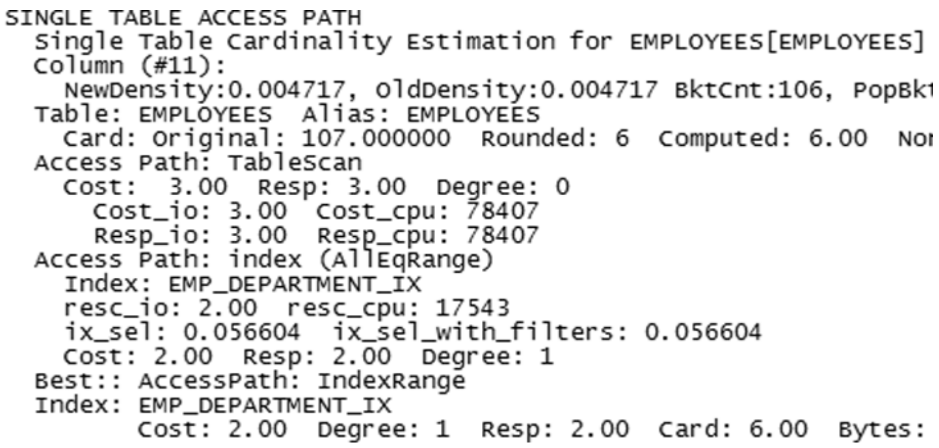
After it gives some basic information about parameters and the SQL being considered and any possible query transformations that might apply, the cost-based optimizer launches into some basic information about the cost of accessing the objects involved. So for example in Figure 5-9 there is shown some basic information about the table EMPLOYEES (twice in this case).



```
BASE STATISTICAL INFORMATION
***********************
Table Stats::
  Table: EMPLOYEES  Alias: EMPLOYEES
    #Rows: 107  #Blks:  5  AvgRowLen:  69.00
Index Stats::
  Index: EMP_DEPARTMENT_IX  Col#: 11
    LVLS: 0  #LB: 1  #DK: 11  LB/K: 1.00  DB/K: 1.00  CLUF: 7.00
  Index: EMP_EMAIL_UK  Col#: 4
    LVLS: 0  #LB: 1  #DK: 107  LB/K: 1.00  DB/K: 1.00  CLUF: 19.00
  Index: EMP_EMP_ID_PK  Col#: 1
    LVLS: 0  #LB: 1  #DK: 107  LB/K: 1.00  DB/K: 1.00  CLUF: 2.00
  Index: EMP_JOB_IX  Col#: 7
    LVLS: 0  #LB: 1  #DK: 19  LB/K: 1.00  DB/K: 1.00  CLUF: 8.00
  Index: EMP_MANAGER_IX  Col#: 10
    LVLS: 0  #LB: 1  #DK: 18  LB/K: 1.00  DB/K: 1.00  CLUF: 7.00
  Index: EMP_NAME_IX  Col#: 3 2
    LVLS: 0  #LB: 1  #DK: 107  LB/K: 1.00  DB/K: 1.00  CLUF: 15.00
***********************
Table Stats::
  Table: EMPLOYEES  Alias: EMPLOYEES
    #Rows: 107  #Blks:  5  AvgRowLen:  69.00
Index Stats::
  Index: EMP_DEPARTMENT_IX  Col#: 11
    LVLS: 0  #LB: 1  #DK: 11  LB/K: 1.00  DB/K: 1.00  CLUF: 7.00
  Index: EMP_EMAIL_UK  Col#: 4
    LVLS: 0  #LB: 1  #DK: 107  LB/K: 1.00  DB/K: 1.00  CLUF: 19.00
  Index: EMP_EMP_ID_PK  Col#: 1
    LVLS: 0  #LB: 1  #DK: 107  LB/K: 1.00  DB/K: 1.00  CLUF: 2.00
  Index: EMP_JOB_IX  Col#: 7
    LVLS: 0  #LB: 1  #DK: 19  LB/K: 1.00  DB/K: 1.00  CLUF: 8.00
  Index: EMP_MANAGER_IX  Col#: 10
    LVLS: 0  #LB: 1  #DK: 18  LB/K: 1.00  DB/K: 1.00  CLUF: 7.00
  Index: EMP_NAME_IX  Col#: 3 2
    LVLS: 0  #LB: 1  #DK: 107  LB/K: 1.00  DB/K: 1.00  CLUF: 15.00
```

Figure 5-9. *A section in the 10053 showing BASE STATISTICAL INFORMATION*

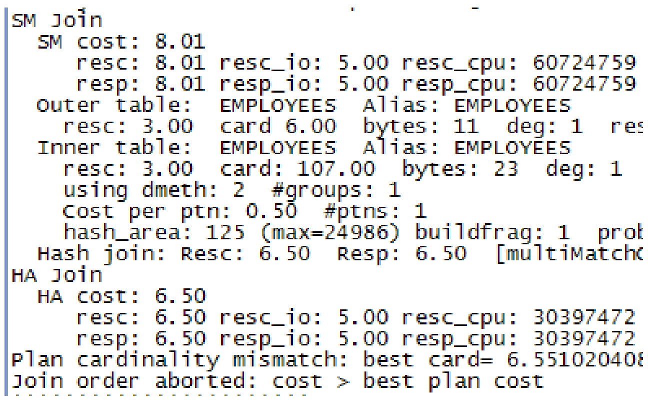
Once we have all this base information about the table and its indexes (note we also have the clustering factor `CLUF` in this basic information), we can start to determine how to get the information the cheapest way. For example, the CBO will consider a single table access path and from that will consider a `Tablescan` (cost 3) or `index (AllEqRange)` (cost 2) and then tell us which was cheapest: `Best:: AccessPath: IndexRange`. Figure 5-10 illustrates the section in the 10053 that shows this.



```
SINGLE TABLE ACCESS PATH
  Single Table Cardinality Estimation for EMPLOYEES[EMPLOYEES]
  Column (#11):
    NewDensity:0.004717, OldDensity:0.004717 BktCnt:106, PopBk
  Table: EMPLOYEES  Alias: EMPLOYEES
    Card: Original: 107.000000  Rounded: 6  Computed: 6.00  No
  Access Path: TableScan
    Cost:  3.00  Resp: 3.00  Degree: 0
      Cost_io: 3.00  Cost_cpu: 78407
      Resp_io: 3.00  Resp_cpu: 78407
  Access Path: index (AllEqRange)
    Index: EMP_DEPARTMENT_IX
    resc_io: 2.00  resc_cpu: 17543
    ix_sel: 0.056604  ix_sel_with_filters: 0.056604
    Cost: 2.00  Resp: 2.00  Degree: 1
  Best:: AccessPath: IndexRange
  Index: EMP_DEPARTMENT_IX
         Cost: 2.00  Degree: 1  Resp: 2.00  Card: 6.00  Bytes:
```

Figure 5-10. *what the optimizer thought of table EMPLOYEES*

Once this is done the optimizer considers join orders, in this case only to another version of the table EMPLOYEES. An example of the optimizer abandoning its calculations is shown in Figure 5-11.



```
SM Join
  SM cost: 8.01
     resc: 8.01 resc_io: 5.00 resc_cpu: 60724759
     resp: 8.01 resp_io: 5.00 resp_cpu: 60724759
  Outer table:  EMPLOYEES  Alias: EMPLOYEES
    resc: 3.00  card 6.00  bytes: 11  deg: 1  re
  Inner table:  EMPLOYEES  Alias: EMPLOYEES
    resc: 3.00  card: 107.00  bytes: 23  deg: 1
    using dmeth: 2  #groups: 1
    Cost per ptn: 0.50  #ptns: 1
    hash_area: 125 (max=24986) buildfrag: 1  pro
  Hash join: Resc: 6.50  Resp: 6.50  [multiMatch
HA Join
  HA cost: 6.50
     resc: 6.50 resc_io: 5.00 resc_cpu: 30397472
     resp: 6.50 resp_io: 5.00 resp_cpu: 30397472
Plan cardinality mismatch: best card= 6.5510204
Join order aborted: cost > best plan cost
```

Figure 5-11. *The optimizer abandons a join because it's greater than the best so far cost*

Now that we’ve covered some of the information that you can get from a 10053 trace file, you know how useful it is. From a careful reading of the trace file you should now be able to see why the optimizer chose to do what it did: which access paths it considered and why it rejected some of them. And if you feel that some of the access paths should not have been rejected, you should be able to work back to the root cause and determine the answer to that age old question of all DBAs: “Why did it not use my index?”

Summary

SQLTXPLAIN is a wonderful tool, and it generates the 10053 trace file by default in most cases. While it is sometimes difficult to understand, there is a mine of information that helps you tune your SQL when things are going wrong. In the next chapter we’ll cover SQL*Profiles and what to do when an emergency strikes and we need the right performance, fast!

CHAPTER 6

Forcing Execution Plans Through Profiles

Sometimes it's best to fix a problem and figure out afterward what went wrong and determine the root cause. By this I mean that when there are production time pressures to get something done, any way that works is good enough. The idea is to get the business back up and running, then sit at your leisure and work out what went wrong, figure out how to stop it from happening again, and develop a plan to implement the permanent fix in a controlled manner during a planned outage if required. Management is not usually very tolerant of perfectionists who want to understand every detail of a problem and its solution before acting: and if they are tolerant, it's because they have another system in their pocket that's keeping the lights on and the business going.

The scenario I am about to describe is rare, but it does happen. An overnight batch job, one that is time critical (has to be finished by a certain time), suddenly starts taking much longer to run. The online day is affected, management reports are not produced on time (if at all), and the DBAs and developers are at a loss. Nothing was changed (which is never true), no releases to production, no parameter changes, no changes in statistics gathering. This is when you often hear the plaintive cry: "What's it doing?"

This is a good question to ask when you are trying to fix a problem properly; but when you need the system to behave the way it did before, there's a short cut. This technique of fixing the problem without understanding the problem is often referred to as the "Severity 1 killer". This is because Severity 1 service requests at Oracle are the highest priority: if there is a way to fix the problem, then go for it. Understand it later.

So how do you do this? SQLT has the tools to get the job done (most of the time). As part of SQLT modus operandi, SQLT produces a script that uses an SQL profile. This SQL profile (if it is the "good" execution plan), can be applied to the troublesome SQL, which then forces the SQL to run "properly" despite whatever else is happening on the system (wrong statistics, changed parameters, etc). SQL profiles, despite the introduction of SQL plan management, are still useful. In the version of Oracle to come, however, expect SQL plan baselines to become a better product. For now, however, let's look at what we can do with SQL profiles.

What is an SQL Profile?

Although the title of this chapter is "Forcing Execution Plans Through Profiles," SQL profiles do not exactly force an execution plan. They are more like "super hints." With hints there is always the possibility that the optimizer will find yet another way of doing the query, with your hint that isn't what you want. Super hints, on the other hand, have specified so many constraints on the execution plan that the optimizer (usually) has no choices left. The word "profile" in this context is quite descriptive. The optimizer has seen this SQL statement go through many times and has determined that it behaves in a certain way: it accesses rows in a certain order and consistently uses a particular index. All this profiling information is gathered and kept for the query and can be used to "force" the statement to behave in the same way in the future if its new behavior is suboptimal.

Can SQL profiles be a bad thing? Let's examine an analogy. Everyone has their routines they stick to. Get up, brush your teeth, eat breakfast, go to work, drive home, read the newspaper, etc. The "profile" in this case is fairly simple. Make sure to brush your teeth before going to work, otherwise you might have to come back home to do it. If somebody wiped your memory (perhaps a tuning book fell off a shelf and bumped you on the head), and you forgot your routine, you might want a reminder of your previous routine until your memory returned. This routine forcing, or "profile," might be a good thing if nothing else had changed. Suppose, however, that you had also moved into a new house and now worked from home. Your profile might now be completely out of date. Driving to work would no longer make sense. Here we see that a profile might be good for a short while to get things back to normal, but every so often you need to check that the profile still makes sense.

So what does a super hint look like? Here's one:

```
h := SYS.SQLPROF_ATTR(
  '[BEGIN_OUTLINE_DATA]',
  '[IGNORE_OPTIM_EMBEDDED_HINTS]',
  '[OPTIMIZER_FEATURES_ENABLE('11.2.0.1')]',
  '[DB_VERSION('11.2.0.1')]',
  '[ALL_ROWS]',
  '[OUTLINE_LEAF(@"SEL$E9AF9BDE")]',
  '[MERGE(@"SEL$CE1D94FA")]',
  '[OUTLINE(@"SEL$98588D1C")]',
  '[UNNEST(@"SEL$6")]',
  '[OUTLINE(@"SEL$CE1D94FA")]',
  '[OUTLINE(@"SEL$C50F9DEF")]',
  '[OUTLINE(@"SEL$6")]',
  '[OUTLINE(@"SEL$81719215")]',
  '[MERGE(@"SEL$EE94F965")]',
  '[OUTLINE(@"SEL$5")]',
  '[OUTLINE(@"SEL$EE94F965")]',
  '[MERGE(@"SEL$9E43CB6E")]',
  '[OUTLINE(@"SEL$4")]',
  '[OUTLINE(@"SEL$9E43CB6E")]',
  '[MERGE(@"SEL$58A6D7F6")]',
  '[OUTLINE(@"SEL$3")]',
  '[OUTLINE(@"SEL$58A6D7F6")]',
  '[MERGE(@"SEL$1")]',
  '[OUTLINE(@"SEL$2")]',
  '[OUTLINE(@"SEL$1")]',
  '[INDEX_RS_ASC(@"SEL$E9AF9BDE" "TABLE1"@"SEL$4" "TABLE1"."TABLE_NAME" "TABLE1"."STATUS"
"TABLE1"."CATEGORY"))]',
  '[INDEX_RS_ASC(@"SEL$E9AF9BDE" "TABLE2"@"SEL$3" ("TABLE2"."COL2" TABLE2"."COL1" "TABLE2"."COL3"))]',
  '[INDEX(@"SEL$E9AF9BDE" "TABLE3"@"SEL$2" ("TABLE3"."COL2"  TABLE3"."COL1"))]',
  '[FULL(@"SEL$E9AF9BDE" "TABLE4"@"SEL$1")]',
  '[INDEX_RS_ASC(@"SEL$E9AF9BDE" "TABLE1"@"SEL$6" ("TABLE1"."TABLE_NAME" "TABLE1"."STATUS"
"TABLE1"."CATEGORY"))]',
  '[INDEX_RS_ASC(@"SEL$E9AF9BDE" "TABLE2"@"SEL$6" ("TABLE2"."COL2" "TABLE2"."COL1" "TABLE2"."COL3"))]',
  '[LEADING(@"SEL$E9AF9BDE" "TABLE1"@"SEL$4" "TABLE2"@"SEL$3" "TABLE3"@"SEL$2" "TABLE4"@"SEL$1"
"TABLE1"@"SEL$6" "TABLE2"@"SEL$6")]',
  '[USE_NL(@"SEL$E9AF9BDE" "TABLE2"@"SEL$3")]',
  '[USE_NL(@"SEL$E9AF9BDE" "TABLE3"@"SEL$2")]',
  '[NLJ_BATCHING(@"SEL$E9AF9BDE" "TABLE3"@"SEL$2")]',
'[USE_HASH(@"SEL$E9AF9BDE" "TABLE4"@"SEL$1")]',
'[USE_MERGE_CARTESIAN(@"SEL$E9AF9BDE" "TABLE1"@"SEL$6")]',
'[USE_NL(@"SEL$E9AF9BDE" "TABLE2"@"SEL$6")]',
'[PX_JOIN_FILTER(@"SEL$E9AF9BDE" "TABLE4"@"SEL$1")]',
'[USE_HASH_AGGREGATION(@"SEL$E9AF9BDE")]',
'[END_OUTLINE_DATA]');
```

Notice how the hint is more verbose than the normal hint you would use to tune an SQL statement. Still, there are recognizable parts after the standard preamble (which includes `BEGIN_OUTLINE_DATA`, `IGNORE_OPTIM_EMBEDDED_HINTS`, `OPTIMIZER_FEATURES_ENABLE('11.2.0.1')` and `DB_VERSION('11.2.0.1')`). Let's list the more familiar parts:

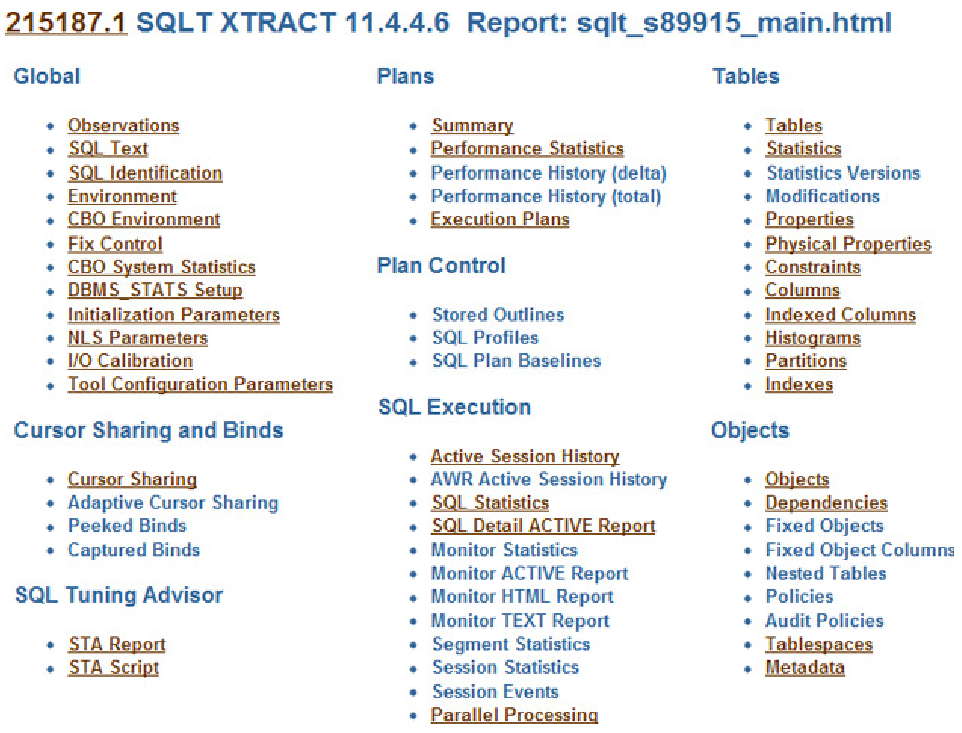
- `ALL_ROWS`
- `MERGE`
- `INDEX_RS_ASC`
- `FULL`
- `USE_NL`
- `NLJ_BATCHING`
- `PX_JOIN_FILTERING`
- `USE_MERGE_CARTESIAN`

You'll also notice that the references made are to obscure objects such as `SEL$3` and `SEL$E9AF9BDE` (which are properly called "query block names"). Remember that the optimizer has transformed the SQL (through the use of query transformations as described in Chapter 5) before it gets to do more detailed cost-based tuning. During the transformation, the various sections of the query get these exotic names (which are unrecognizable). The profile hints then refer to these names.

So now we know what an SQL profile is and how it looks to the optimizer. We also know that keeping an unchanging profile on an SQL statement might be a bad plan. Having said all that we still know getting the production database back to an operational state might be the difference between success and failure. So with all these provisos and limitations, let's find out where SQLT gets its SQL profile.

Where Does SQLT Get Its SQL Profile?

To get the script that creates the SQL profile for you is simple. SQLT generates the required script from both the SQLT XTRACT method and from the XECUTE method. We saw in Chapter 1 how to generate an XTRACT report, and we covered XECUTE in Chapter 3. Make a note of the SQLT ID (not the SQL ID) of the report and the plan hash value that you want. See Figure 6-1 below. This shows the SQLT ID, it's the number in the title "Report: sqlt_s89915_main.html". In this case the SQLT ID is 89915.



215187.1 SQLT XTRACT 11.4.4.6 Report: sqlt_s89915_main.html

Global

- Observations
- SQL Text
- SQL Identification
- Environment
- CBO Environment
- Fix Control
- CBO System Statistics
- DBMS_STATS Setup
- Initialization Parameters
- NLS Parameters
- I/O Calibration
- Tool Configuration Parameters

Cursor Sharing and Binds

- Cursor Sharing
- Adaptive Cursor Sharing
- Peeked Binds
- Captured Binds

SQL Tuning Advisor

- STA Report
- STA Script

Plans

- Summary
- Performance Statistics
- Performance History (delta)
- Performance History (total)
- Execution Plans

Plan Control

- Stored Outlines
- SQL Profiles
- SQL Plan Baselines

SQL Execution

- Active Session History
- AWR Active Session History
- SQL Statistics
- SQL Detail ACTIVE Report
- Monitor Statistics
- Monitor ACTIVE Report
- Monitor HTML Report
- Monitor TEXT Report
- Segment Statistics
- Session Statistics
- Session Events
- Parallel Processing

Tables

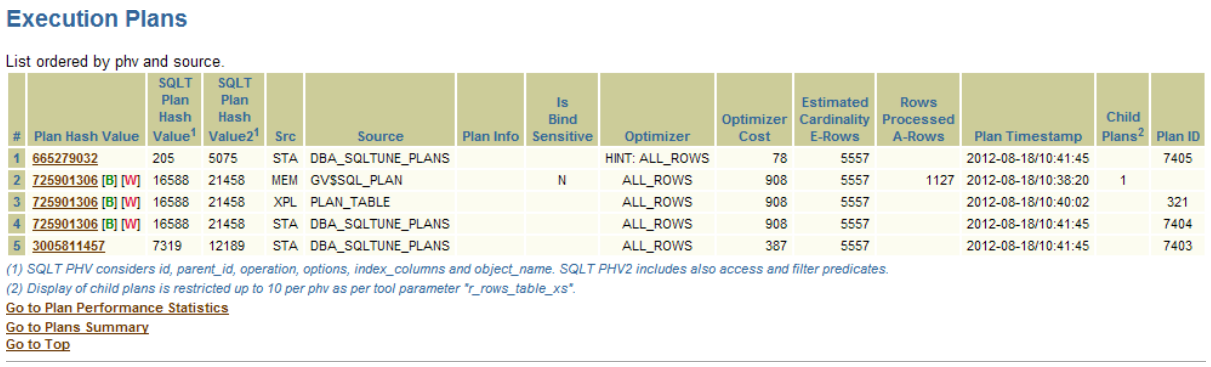
- Tables
- Statistics
- Statistics Versions
- Modifications
- Properties
- Physical Properties
- Constraints
- Columns
- Indexed Columns
- Histograms
- Partitions
- Indexes

Objects

- Objects
- Dependencies
- Fixed Objects
- Fixed Object Columns
- Nested Tables
- Policies
- Audit Policies
- Tablespaces
- Metadata

***Figure 6-1.** The header page shows the SQLT ID, needed for generating the profile*

The plan hash value (PHV) can be obtained from the "Execution Plans" section of the report. See Figure 6-2 for the section showing which plan hash values are available.



Execution Plans

List ordered by phv and source.

#	Plan Hash Value	SQLT Plan Hash Value[1]	SQLT Plan Hash Value2[1]	Src	Source	Plan Info	Is Bind Sensitive	Optimizer	Optimizer Cost	Estimated Cardinality E-Rows	Rows Processed A-Rows	Plan Timestamp	Child Plans[2]	Plan ID
1	665279032	205	5075	STA	DBA_SQLTUNE_PLANS			HINT: ALL_ROWS	78	5557		2012-08-18/10:41:45		7405
2	725901306 [B] [W]	16588	21458	MEM	GV$SQL_PLAN		N	ALL_ROWS	908	5557	1127	2012-08-18/10:38:20	1	
3	725901306 [B] [W]	16588	21458	XPL	PLAN_TABLE			ALL_ROWS	908	5557		2012-08-18/10:40:02		321
4	725901306 [B] [W]	16588	21458	STA	DBA_SQLTUNE_PLANS			ALL_ROWS	908	5557		2012-08-18/10:41:45		7404
5	3005811457	7319	12189	STA	DBA_SQLTUNE_PLANS			ALL_ROWS	387	5557		2012-08-18/10:41:45		7403

(1) SQLT PHV considers id, parent_id, operation, options, index_columns and object_name. SQLT PHV2 includes also access and filter predicates.
(2) Display of child plans is restricted up to 10 per phv as per tool parameter "r_rows_table_xs".

Go to Plan Performance Statistics
Go to Plans Summary
Go to Top

***Figure 6-2.** Execution plans available and the Plan Hash Values (PHV)*

So now we have both the SQLT ID and the PHV. In the `utl` directory under the SQLT installation area we can now run the script that will generate the profile, as shown in the following example. However, before you do that on a production system you should consult Oracle Support to make sure your steps are validated and supported actions. Just open a ticket and the friendly people at tuning support will be more than happy to help you. Here is the directory listing showing the files in the `utl` directory. From that directory we then enable the profile creation and then run the routine to generate the profile, making sure to pass in the SQLT ID and the PHV. I've shown the output from the script, and then at the end I show the new file created. It's that easy.

```
C:\Documents and Settings\Stelios\Desktop\SQLT\sqlt\utl>dir
 Volume in drive C has no label.
 Volume Serial Number is 77E9-80B4

 Directory of C:\Documents and Settings\Stelios\Desktop\SQLT\sqlt\utl

09/01/2012  09:23 AM    <DIR>          .
09/01/2012  09:23 AM    <DIR>          ..
07/02/2011  12:49 AM               130 10053.sql
04/02/2012  12:43 PM             4,828 coe_gen_sql_profile.sql
08/18/2012  11:25 AM             1,185 coe_gen_sql_profile_.zip
06/02/2012  05:28 AM            10,305 coe_load_sql_baseline.sql
04/02/2012  12:43 PM            12,007 coe_load_sql_profile.sql
05/02/2012  11:27 AM            18,248 coe_xfr_sql_profile.sql
07/02/2011  12:49 AM               101 flush.sql
08/18/2012  11:25 AM                33 missing_file.txt
07/02/2011  12:49 AM               184 plan.sql
06/02/2012  05:28 AM            22,527 profiler.sql
06/02/2012  05:28 AM            73,472 pxhcdr.sql
06/02/2012  05:28 AM            71,213 roxecute.sql
06/02/2012  05:28 AM            70,126 roxtract.sql
08/11/2011  03:46 AM               475 sel.sql
02/02/2012  01:19 PM               435 sel_aux.sql
06/02/2012  05:28 AM           160,393 sqlhc.sql
01/03/2012  12:04 AM             2,891 sqltcdirs.sql
08/11/2011  03:46 AM             4,014 sqlthistfile.sql
08/11/2011  03:46 AM             3,116 sqlthistpurge.sql
04/02/2012  12:43 PM             3,694 sqltimp.sql
04/02/2012  12:43 PM             2,927 sqltimpfo.sql
01/03/2012  12:04 AM             3,545 sqltlite.sql
03/02/2012  04:06 PM             3,900 sqltmain.sql
09/01/2012  09:23 AM             6,582 sqltprofile.log
01/03/2012  12:04 AM             5,469 sqltprofile.sql <<<Script we are going to use
08/18/2012  09:33 AM                74 x.sql
02/18/2012  10:04 AM    <DIR>          xgram
02/02/2012  12:15 PM    <DIR>          xhume
06/01/2012  09:39 AM    <DIR>          xplore
              27 File(s)        486,021 bytes
               5 Dir(s)  11,102,261,248 bytes free

C:\Documents and Settings\Stelios\Desktop\SQLT\sqlt\utl>sqlplus stelios/oracle

SQL*Plus: Release 11.2.0.1.0 Production on Sat Sep 1 09:24:34 2012

Copyright (c) 1982, 2010, Oracle.  All rights reserved.

Connected to:
Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - Production
With the Partitioning, OLAP, Data Mining and Real Application Testing options
```

```
SQL> EXEC sqltxplain.sqlt$a.set_param('custom_sql_profile', 'Y');

PL/SQL procedure successfully completed.

SQL> @sqltprofile 89915 3005811457
```

Please note that the `sqltxplain.sqlt$a.set_param` procedure is required to set this functionality. When we run `sqltprofile.sql` (with a valid SQLT ID and a valid plan hash value [the first and second numbers respectively]) we will see a result similar to the one shown below:

```
... please wait ...

STAID MET INSTANCE SQL_TEXT
----- --- -------- ------------------------------------------------------------
89906 XTR snc1     select count(*) from dba_objects
89909 XEC snc1     select   s.amount_sold,   c.cust_id,   p.prod_name from   sh
89910 XEC snc1     select  s.amount_sold,c.cust_id,p.prod_name from sh.products
89911 XEC snc1     select  s.amount_sold,c.cust_id,p.prod_name from sh.products
89912 XTR snc1     select  s.amount_sold,c.cust_id,p.prod_name from sh.products
89913 XTR snc1     select sql_id from v$sql where sql_text like '%select count(
89914 XTR snc1     select count(*) from test3 where object_type like '%TAB%' an
89915 XTR snc1     select  s.amount_sold,c.cust_id,p.prod_name from sh.products
89916 XTR snc1     select  s.amount_sold,c.cust_id,p.prod_name from sh.products
Parameter 1:
STATEMENT_ID (required)

PLAN_HASH_VALUE ATTRIBUTE
--------------- ---------
      665279032
      725901306 [B][W]
     3005811457

Parameter 2:
PLAN_HASH_VALUE (required)

Values passed to sqltprofile:
~~~~~~~~~~~~~~~~~~~~~~~~~~~~
STATEMENT_ID   : "89915"
PLAN_HASH_VALUE: "3005811457"
... getting sqlt_s89915_p3005811457_sqlprof.sql out of sqlt repository ...
sqlt_s89915_p3005811457_sqlprof.sql has been generated
SQLTPROFILE completed.
SQL>
```

Now that the script has run, where do you find the script to create a SQL Profile? The SQL script can be found in the same directory where the profile script was. The directory command below shows the file in the directory, which I've indicated with a New File pointer.

```
C:\Documents and Settings\Stelios\Desktop\SQLT\sqlt\utl>dir *.sql
 Volume in drive C has no label.
 Volume Serial Number is 77E9-80B4
```

```
 Directory of C:\Documents and Settings\Stelios\Desktop\SQLT\sqlt\utl

07/02/2011  12:49 AM               130 10053.sql
04/02/2012  12:43 PM             4,828 coe_gen_sql_profile.sql
06/02/2012  05:28 AM            10,305 coe_load_sql_baseline.sql
04/02/2012  12:43 PM            12,007 coe_load_sql_profile.sql
05/02/2012  11:27 AM            18,248 coe_xfr_sql_profile.sql
07/02/2011  12:49 AM               101 flush.sql
07/02/2011  12:49 AM               184 plan.sql
06/02/2012  05:28 AM            22,527 profiler.sql
06/02/2012  05:28 AM            73,472 pxhcdr.sql
06/02/2012  05:28 AM            71,213 roxecute.sql
06/02/2012  05:28 AM            70,126 roxtract.sql
08/11/2011  03:46 AM               475 sel.sql
02/02/2012  01:19 PM               435 sel_aux.sql
06/02/2012  05:28 AM           160,393 sqlhc.sql
01/03/2012  12:04 AM             2,891 sqltcdirs.sql
08/11/2011  03:46 AM             4,014 sqlthistfile.sql
08/11/2011  03:46 AM             3,116 sqlthistpurge.sql
04/02/2012  12:43 PM             3,694 sqltimp.sql
04/02/2012  12:43 PM             2,927 sqltimpfo.sql
01/03/2012  12:04 AM             3,545 sqltlite.sql
03/02/2012  04:06 PM             3,900 sqltmain.sql
01/03/2012  12:04 AM             5,469 sqltprofile.sql
09/01/2012  09:29 AM             4,147 sqlt_s89915_p3005811457_sqlprof.sql <<<New file
08/18/2012  09:33 AM                74 x.sql
              24 File(s)        478,221 bytes
               0 Dir(s)  11,099,074,560 bytes free
```

What Can You Do with a SQL Profile?

Now that we have the sqlt_s89915_p3005811457_sqlprof.sql script, what can we do with it? In broad terms, we can run this script on the database where the SQL is executing and this will freeze the execution plan for that SQL ID. It can also be used to freeze an execution plan on another system. So for example if your development system has the right execution plan and your production system does not, you can transfer the execution plan. In a situation where the production database is in an unusable state because some vital piece of SQL is using the wrong execution plan and consequently running too slowly to fit into the execution window, this script alone is worth the price of this book. Let's look at the script.

```
SPO sqlt_s89915_p3005811457_sqlprof.log;
SET ECHO ON TERM ON LIN 2000 TRIMS ON NUMF 99999999999999999999;
REM
REM $Header: 215187.1 sqlt_s89915_p3005811457_sqlprof.sql 11.4.4.6 2012/09/01 carlos.sierra $
REM
REM Copyright (c) 2000-2012, Oracle Corporation. All rights reserved.
REM
REM AUTHOR
REM   carlos.sierra@oracle.com
REM
REM SCRIPT
REM   sqlt_s89915_p3005811457_sqlprof.sql
REM
REM SOURCE
REM   Host    : LOCUTUS
REM   DB Name : SNC1
REM   Platform: 32-bit Windows
REM   Product : Oracle Database 11g Enterprise Edition (Production)
REM   Version : 11.2.0.1.0
REM   Language: US:AMERICAN_AMERICA.WE8MSWIN1252
REM   EBS     : NO
REM   Siebel  : NO
REM   PSFT    : NO
REM
REM DESCRIPTION
REM   This script is generated automatically by the SQLT tool.
REM   It contains the SQL*Plus commands to create a custom
REM   SQL Profile based on plan hash value 3005811457.
REM   The custom SQL Profile to be created by this script
REM   will affect plans for SQL commands with signature
REM   matching the one for SQL Text below.
REM   Review SQL Text and adjust accordingly.
REM
REM PARAMETERS
REM   None.
REM
REM EXAMPLE
REM   SQL> START sqlt_s89915_p3005811457_sqlprof.sql; <<< Note 1.
REM
REM NOTES
REM   1. Should be run as SYSTEM or SYSDBA.
REM   2. User must have CREATE ANY SQL PROFILE privilege.
REM   3. SOURCE and TARGET systems can be the same or similar.
REM   4. To drop this custom SQL Profile after it has been created:
REM      EXEC DBMS_SQLTUNE.DROP_SQL_PROFILE('sqlt_s89915_p3005811457'); <<<Note 2.
REM   5. Be aware that using DBMS_SQLTUNE requires a license
REM      for the Oracle Tuning Pack. <<< Note 3.
REM   6. If you modified a SQL putting Hints in order to produce a desired
REM      Plan, you can remove the artifical Hints from SQL Text pieces below.
REM      By doing so you can create a custom SQL Profile for the original
REM      SQL but with the Plan captured from the modified SQL (with Hints).
REM
WHENEVER SQLERROR EXIT SQL.SQLCODE;

VAR signature NUMBER;

DECLARE
  sql_txt CLOB;
  h       SYS.SQLPROF_ATTR;
  PROCEDURE wa (p_line IN VARCHAR2) IS
  BEGIN
    DBMS_LOB.WRITEAPPEND(sql_txt, LENGTH(p_line), p_line);
  END wa;
```

```
BEGIN
  DBMS_LOB.CREATETEMPORARY(sql_txt, TRUE); <<< Note 4
  DBMS_LOB.OPEN(sql_txt, DBMS_LOB.LOB_READWRITE);
  -- SQL Text pieces below do not have to be of same length.
  -- So if you edit SQL Text (i.e. removing temporary Hints),
  -- there is no need to edit or re-align unmodified pieces.
  wa(q'[select  s.amount_sold,c.cust_id,p.prod_name from sh.products p,s]'); <<< Note 5
  wa(q'[h.sales s,sh.customers c where
  c.cust_id=s.cust_id and s.prod_]');
  wa(q'[id=p.prod_id and c.cust_first_name='Theodorick' ]');
  DBMS_LOB.CLOSE(sql_txt);
  h := SYS.SQLPROF_ATTR(
  q'[BEGIN_OUTLINE_DATA]',
  q'[SWAP_JOIN_INPUTS(@"SEL$1" "P"@"SEL$1")]',
  q'[USE_HASH(@"SEL$1" "P"@"SEL$1")]',
  q'[NLJ_BATCHING(@"SEL$1" "S"@"SEL$1")]',
  q'[USE_NL(@"SEL$1" "S"@"SEL$1")]',
  q'[LEADING(@"SEL$1" "C"@"SEL$1" "S"@"SEL$1" "P"@"SEL$1")]',
  q'[FULL(@"SEL$1" "P"@"SEL$1")]',
  q'[INDEX(@"SEL$1" "S"@"SEL$1" ("SALES"."CUST_ID"))]',
  q'[INDEX(@"SEL$1" "C"@"SEL$1" ("CUSTOMERS"."CUST_FIRST_NAME" "CUSTOMERS"."CUST_ID"))]',
  q'[OUTLINE_LEAF(@"SEL$1")]',
  q'[ALL_ROWS]',
  q'[DB_VERSION('11.2.0.1')]',
  q'[OPTIMIZER_FEATURES_ENABLE('11.2.0.1')]',
  q'[IGNORE_OPTIM_EMBEDDED_HINTS]',
  q'[END_OUTLINE_DATA]');

  :signature := DBMS_SQLTUNE.SQLTEXT_TO_SIGNATURE(sql_txt); <<< Note 6

  DBMS_SQLTUNE.IMPORT_SQL_PROFILE ( <<< Note 7
    sql_text    => sql_txt,
    profile     => h,
    name        => 'sqlt_s89915_p3005811457',
    description => 's89915_snc1_locutus 6ga32aw0dn2sd 3005811457 '||:signature,
    category    => 'DEFAULT',
    validate    => TRUE,
    replace     => TRUE,
    force_match => FALSE /* TRUE:FORCE (match even when different literals in SQL).
FALSE:EXACT (similar to CURSOR_SHARING) */ );
    DBMS_LOB.FREETEMPORARY(sql_txt);
END;
/

WHENEVER SQLERROR CONTINUE;
SET ECHO OFF;
PRINT signature
PRO
PRO ... manual custom SQL Profile has been created
PRO
SET TERM ON ECHO OFF LIN 80 TRIMS OFF NUMF "";
```

```
SPO OFF;
PRO
PRO SQLPROFILE completed.
```

You do not need to know any of the details of how this code works; in an emergency situation you would just follow the instructions and then make your system behave as required.

1. `SQL> START sqlt_s89915_p3005811457_sqlprof.sql;` - This is how you run the SQL script. As it says in the notes you should use a suitably privileged account, such as SYS.
2. `SQL> EXEC DBMS_SQLTUNE.DROP_SQL_PROFILE('sqlt_s89915_p3005811457');` If for some reason you do not want to keep the profile (for example, you've found the long term solution to your SQL performance problem) you can execute this command line, and the SQL profile will be dropped.
3. You must have the Oracle license for the tuning pack for the system on which you apply this code.
4. This is the code where a `LOB` is created to store your SQL text.
5. A series of calls to the procedure `wa` (Write Append) are used to store more and more of your SQL text until it is all stored in `sql_text`.
6. Create a signature for the SQL text.
7. Import the SQL profile. The procedure `IMPORT_SQL_PROFILE` takes the following parameters:
    a. The `sql_text` of the query
    b. The hint (h) for the query
    c. A name for the SQL profile (sqlt_s89915_p3005811457 in this example)
    d. Some text to describe the profile, based on the SQLT ID, the SQL ID the plan hash value and the signature.
    e. A category, which is set to `DEFAULT`
    f. A setting to validate the SQL profile
    g. A setting to replace any existing profiles on the same SQL text
    h. A `force_match` flag, which is set to `FALSE` by default but should be set to `TRUE` if your SQL uses literal values and you want all occurrences of the SQL to use the same profile.

Let's see an example run of the script. I haven't set the `force_match` flag in this example because I don't need it, but on a system with SQL using literals I will set the `force_match` flag to TRUE. In the example below I have run the profile script that was created above. This script is now a stand-alone script that generates the profile for a particular SQL ID and uses the plan for a particular PHV. It has no other purpose. Hence it takes no parameters. We know that the script has completed because we see the message "SQLPROFILE completed". Once this script has finished there is nothing more to do except find the long term solution!

```
C:\Documents and Settings\Stelios\Desktop\SQLT\sqlt\utl>sqlplus stelios/oracle

SQL*Plus: Release 11.2.0.1.0 Production on Sat Sep 1 10:53:17 2012

Copyright (c) 1982, 2010, Oracle.  All rights reserved.
```

```
Connected to:
Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - Production
With the Partitioning, OLAP, Data Mining and Real Application Testing options

SQL> @sqlt_s89915_p3005811457_sqlprof.sql
SQL> REM
SQL> REM $Header: 215187.1 sqlt_s89915_p3005811457_sqlprof.sql 11.4.4.6 2012/09/01 carlos.sierra $
SQL> REM
SQL> REM Copyright (c) 2000-2012, Oracle Corporation. All rights reserved.
SQL> REM
SQL> REM AUTHOR
SQL> REM   carlos.sierra@oracle.com
SQL> REM
SQL> REM SCRIPT
SQL> REM   sqlt_s89915_p3005811457_sqlprof.sql
SQL> REM
SQL> REM SOURCE
SQL> REM   Host     : LOCUTUS
SQL> REM   DB Name  : SNC1
SQL> REM   Platform: 32-bit Windows
SQL> REM   Product  : Oracle Database 11g Enterprise Edition (Production)
SQL> REM   Version  : 11.2.0.1.0
SQL> REM   Language: US:AMERICAN_AMERICA.WE8MSWIN1252
SQL> REM   EBS      : NO
SQL> REM   Siebel   : NO
SQL> REM   PSFT     : NO
SQL> REM
SQL> REM DESCRIPTION
SQL> REM   This script is generated automatically by the SQLT tool.
SQL> REM   It contains the SQL*Plus commands to create a custom
SQL> REM   SQL Profile based on plan hash value 3005811457.
SQL> REM   The custom SQL Profile to be created by this script
SQL> REM   will affect plans for SQL commands with signature
SQL> REM   matching the one for SQL Text below.
SQL> REM   Review SQL Text and adjust accordingly.
SQL> REM
SQL> REM PARAMETERS
SQL> REM   None.
SQL> REM
SQL> REM EXAMPLE
SQL> REM   SQL> START sqlt_s89915_p3005811457_sqlprof.sql;
SQL> REM
SQL> REM NOTES
SQL> REM   1. Should be run as SYSTEM or SYSDBA.
SQL> REM   2. User must have CREATE ANY SQL PROFILE privilege.
SQL> REM   3. SOURCE and TARGET systems can be the same or similar.
SQL> REM   4. To drop this custom SQL Profile after it has been created:
SQL> REM         EXEC DBMS_SQLTUNE.DROP_SQL_PROFILE('sqlt_s89915_p3005811457');
SQL> REM   5. Be aware that using DBMS_SQLTUNE requires a license
SQL> REM         for the Oracle Tuning Pack.
SQL> REM   6. If you modified a SQL putting Hints in order to produce a desired
SQL> REM         Plan, you can remove the artifical Hints from SQL Text pieces below.
SQL> REM         By doing so you can create a custom SQL Profile for the original
SQL> REM         SQL but with the Plan captured from the modified SQL (with Hints).
SQL> REM
SQL> WHENEVER SQLERROR EXIT SQL.SQLCODE;
SQL>
SQL> VAR signature NUMBER;
SQL>
SQL> DECLARE
  2    sql_txt CLOB;
  3    h          SYS.SQLPROF_ATTR;
  4    PROCEDURE wa (p_line IN VARCHAR2) IS
  5    BEGIN
  6      DBMS_LOB.WRITEAPPEND(sql_txt, LENGTH(p_line), p_line);
  7    END wa;
  8  BEGIN
  9    DBMS_LOB.CREATETEMPORARY(sql_txt, TRUE);
 10    DBMS_LOB.OPEN(sql_txt, DBMS_LOB.LOB_READWRITE);
 11    -- SQL Text pieces below do not have to be of same length.
 12    -- So if you edit SQL Text (i.e. removing temporary Hints),
 13    -- there is no need to edit or re-align unmodified pieces.
 14    wa(q'[select     s.amount_sold,c.cust_id,p.prod_name from sh.products p,s]');
 15    wa(q'[h.sales s,sh.customers c where
 16    c.cust_id=s.cust_id and s.prod_]');
 17    wa(q'[id=p.prod_id and c.cust_first_name='Theodorick' ]');
 18    DBMS_LOB.CLOSE(sql_txt);
 19    h := SYS.SQLPROF_ATTR(
 20    q'[BEGIN_OUTLINE_DATA]',
 21    q'[SWAP_JOIN_INPUTS(@"SEL$1" "P"@"SEL$1")]',
 22    q'[USE_HASH(@"SEL$1" "P"@"SEL$1")]',
 23    q'[NLJ_BATCHING(@"SEL$1" "S"@"SEL$1")]',
 24    q'[USE_NL(@"SEL$1" "S"@"SEL$1")]',
 25    q'[LEADING(@"SEL$1" "C"@"SEL$1" "S"@"SEL$1" "P"@"SEL$1")]',
 26    q'[FULL(@"SEL$1" "P"@"SEL$1")]',
 27    q'[INDEX(@"SEL$1" "S"@"SEL$1" ("SALES"."CUST_ID"))]',
 28    q'[INDEX(@"SEL$1" "C"@"SEL$1" ("CUSTOMERS"."CUST_FIRST_NAME" "CUSTOMERS"."CUST_ID"))]',
 29    q'[OUTLINE_LEAF(@"SEL$1")]',
 30    q'[ALL_ROWS]',
 31    q'[DB_VERSION('11.2.0.1')]',
 32    q'[OPTIMIZER_FEATURES_ENABLE('11.2.0.1')]',
 33    q'[IGNORE_OPTIM_EMBEDDED_HINTS]',
 34    q'[END_OUTLINE_DATA]');
 35
 36    :signature := DBMS_SQLTUNE.SQLTEXT_TO_SIGNATURE(sql_txt);
 37
 38    DBMS_SQLTUNE.IMPORT_SQL_PROFILE (
 39      sql_text    => sql_txt,
 40      profile     => h,
 41      name        => 'sqlt_s89915_p3005811457',
 42      description => 's89915_snc1_locutus 6ga32aw0dn2sd 3005811457 '||:signature,
 43      category    => 'DEFAULT',
 44      validate    => TRUE,
 45      replace     => TRUE,
 46      force_match => FALSE /* TRUE:FORCE (match even when different literals in SQL).
FALSE:EXACT (similar to CURSOR_SHARING) */ );
 47      DBMS_LOB.FREETEMPORARY(sql_txt);
 48  END;
 49  /

PL/SQL procedure successfully completed.

SQL>
SQL> WHENEVER SQLERROR CONTINUE;
SQL> SET ECHO OFF;

            SIGNATURE
---------------------
     4850905917262832

... manual custom SQL Profile has been created

SQLPROFILE completed.
SQL>
```

At the end of this script the SQL profile is attached to the SQL ID and should use the same plan as shown in the outline section of the script. The three-step plan in case of emergency (emergency drop in SQL performance)

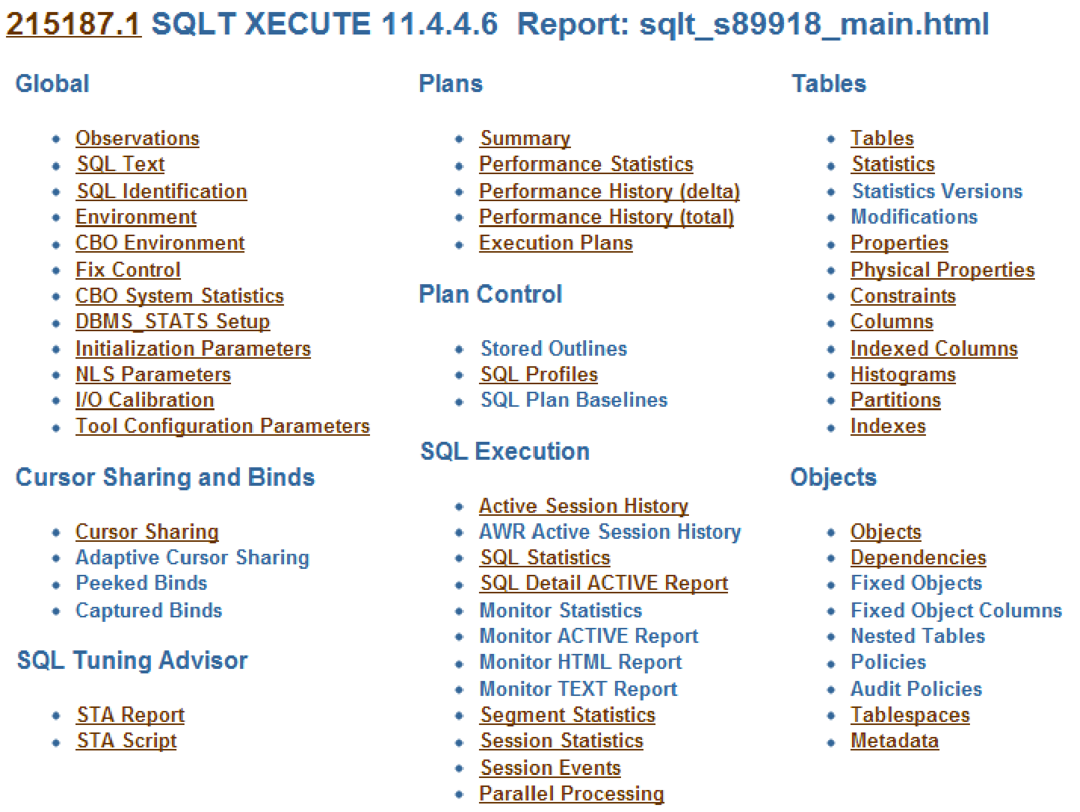
1. Break open SQLT
2. Run `sqltprofile.sql` (license permitting)
3. Set the `force_match` flag to TRUE (if using literals) and run the script.

How Do You Confirm You Are Using an SQL Profile?

To confirm that the SQL Profile is working we need to run the SQL from the SQL prompt or re-run SQLT and look at the display showing the SQL profiles being used. Below is an example where I have run the SQL manually and got the execution plan.

```
SQL> @q1
1127 rows selected.

Execution Plan
----------------------------------------------------------
Plan hash value: 1574422790
---------------------------------------------------------------------------------------------
| Id  |Operation                            |Name                |Rows |Cost (%CPU)|Time    |
---------------------------------------------------------------------------------------------
|   0 |SELECT STATEMENT                     |                    | 5557| 7035   (1)|00:01:25|
|*  1 | HASH JOIN                           |                    | 5557| 7035   (1)|00:01:25|
|   2 |  TABLE ACCESS FULL                  |PRODUCTS            |   72|    3   (0)|00:00:01|
|   3 |  NESTED LOOPS                       |                    |     |           |        |
|   4 |   NESTED LOOPS                      |                    | 5557| 7032   (1)|00:01:25|
|*  5 |    TABLE ACCESS BY INDEX ROWID      |CUSTOMERS           |   43| 2366   (1)|00:00:29|
|   6 |     BITMAP CONVERSION TO ROWIDS     |                    |     |           |        |
|   7 |      BITMAP INDEX FULL SCAN         |CUSTOMERS_GENDER_BIX|     |           |        |
|   8 |    PARTITION RANGE ALL              |                    |     |           |        |
|   9 |     BITMAP CONVERSION TO ROWIDS     |                    |     |           |        |
|* 10 |      BITMAP INDEX SINGLE VALUE      |SALES_CUST_BIX      |     |           |        |
|  11 |   TABLE ACCESS BY LOCAL INDEX ROWID|SALES               |  130| 7032   (1)|00:01:25|
---------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - access("S"."PROD_ID"="P"."PROD_ID")
   5 - filter("C"."CUST_FIRST_NAME"='Theodorick')
  10 - access("C"."CUST_ID"="S"."CUST_ID")
Note
-----
   - SQL profile "sqlt_s89915_p3005811457" used for this statement
Statistics
----------------------------------------------------------
          0  recursive calls
          0  db block gets
       5782  consistent gets
          0  physical reads
          0  redo size
      49680  bytes sent via SQL*Net to client
       1241  bytes received via SQL*Net from client
         77  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
       1127 rows processed
```

If you look at the execution plan, you'll see that a note is appended stating `SQL profile "sqlt_s89915_p3005811457" used for this statement`. If a SQLT XECUTE report was run against this system and this SQL ID you would see this header page (see Figure 6-3) from which you could look at the execution plans created.


215187.1 SQLT XECUTE 11.4.4.6 Report: sqlt_s89918_main.html

Global

- Observations
- SQL Text
- SQL Identification
- Environment
- CBO Environment
- Fix Control
- CBO System Statistics
- DBMS_STATS Setup
- Initialization Parameters
- NLS Parameters
- I/O Calibration
- Tool Configuration Parameters

Cursor Sharing and Binds

- Cursor Sharing
- Adaptive Cursor Sharing
- Peeked Binds
- Captured Binds

SQL Tuning Advisor

- STA Report
- STA Script

Plans

- Summary
- Performance Statistics
- Performance History (delta)
- Performance History (total)
- Execution Plans

Plan Control

- Stored Outlines
- SQL Profiles
- SQL Plan Baselines

SQL Execution

- Active Session History
- AWR Active Session History
- SQL Statistics
- SQL Detail ACTIVE Report
- Monitor Statistics
- Monitor ACTIVE Report
- Monitor HTML Report
- Monitor TEXT Report
- Segment Statistics
- Session Statistics
- Session Events
- Parallel Processing

Tables

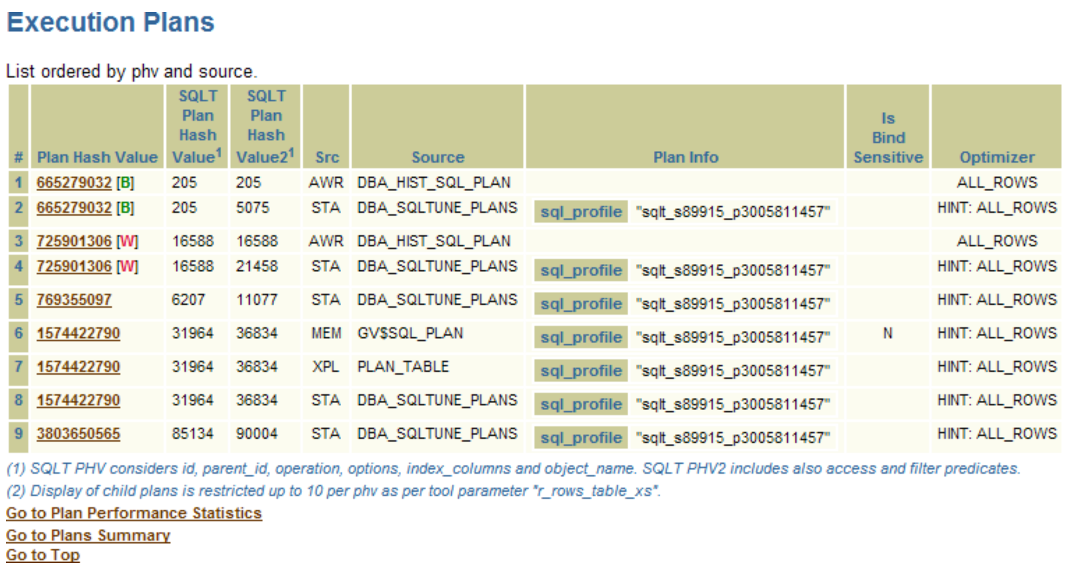
- Tables
- Statistics
- Statistics Versions
- Modifications
- Properties
- Physical Properties
- Constraints
- Columns
- Indexed Columns
- Histograms
- Partitions
- Indexes

Objects

- Objects
- Dependencies
- Fixed Objects
- Fixed Object Columns
- Nested Tables
- Policies
- Audit Policies
- Tablespaces
- Metadata

Figure 6-3. *Shows the top of the SQLT HTML report sqlt_s89918_main.html file*

From here, click on Execution Plans. We see now that the `sql_profile` text is present and highighted in the Plan Info column. See Figure 6-4, which shows this section of the report.



Execution Plans

List ordered by phv and source.

#	Plan Hash Value	SQLT Plan Hash Value[1]	SQLT Plan Hash Value2[1]	Src	Source	Plan Info	Is Bind Sensitive	Optimizer
1	665279032 [B]	205	205	AWR	DBA_HIST_SQL_PLAN			ALL_ROWS
2	665279032 [B]	205	5075	STA	DBA_SQLTUNE_PLANS	sql_profile "sqlt_s89915_p3005811457"		HINT: ALL_ROWS
3	725901306 [W]	16588	16588	AWR	DBA_HIST_SQL_PLAN			ALL_ROWS
4	725901306 [W]	16588	21458	STA	DBA_SQLTUNE_PLANS	sql_profile "sqlt_s89915_p3005811457"		HINT: ALL_ROWS
5	769355097	6207	11077	STA	DBA_SQLTUNE_PLANS	sql_profile "sqlt_s89915_p3005811457"		HINT: ALL_ROWS
6	1574422790	31964	36834	MEM	GV$SQL_PLAN	sql_profile "sqlt_s89915_p3005811457"	N	HINT: ALL_ROWS
7	1574422790	31964	36834	XPL	PLAN_TABLE	sql_profile "sqlt_s89915_p3005811457"		HINT: ALL_ROWS
8	1574422790	31964	36834	STA	DBA_SQLTUNE_PLANS	sql_profile "sqlt_s89915_p3005811457"		HINT: ALL_ROWS
9	3803650565	85134	90004	STA	DBA_SQLTUNE_PLANS	sql_profile "sqlt_s89915_p3005811457"		HINT: ALL_ROWS

(1) SQLT PHV considers id, parent_id, operation, options, index_columns and object_name. SQLT PHV2 includes also access and filter predicates.
(2) Display of child plans is restricted up to 10 per phv as per tool parameter "r_rows_table_xs".
Go to Plan Performance Statistics
Go to Plans Summary
Go to Top

Figure 6-4. *shows the Execution Plans section of the SQLT HTML report. This is the left hand side of the page. There are more details available on the right hand side of the page. Note the use of sql_profile in the Plan Info column*

How Do You Transfer an SQL Profile from One Database to Another?

In many cases SQLT is considered too intrusive to database operations (it isn't) or there are strict rules on a particular site that disallow the use of SQLT directly on a production database. In such cases there will often be a development or staging environment where test SQL can be run against a representative workload (i.e., one with enough data, which is up to date). In cases like these you can often get the right execution plan in the development or staging environment, but can't in the production environment. Either because you don't have time to correct statistics or some other factor prevents you from correcting production in a timely manner. In such cases, after confirming with Oracle support, you can create an SQL profile on one system and transfer it to production. You can follow the steps described in the Oracle Support Note "How To Move SQL Profiles From One Database To Another Database [ID 457531.1]" or follow the steps described here, which are a simplified version of the note steps using SQLT.

1. Create an SQL profile script on your staging or development system (as described above). The SQL profile should be based on SQL text that matches the SQL text on the production system where you are going to transfer the SQL profile. The end result of this step is an SQL Profile.

2. On the Development or Staging system create a staging table with the commands

```
SQL> exec dbms_sqltune.create_stgtab_sqlprof(table_name=>'STAGE', schema_name='STELIOS');
```

3. Pack the SQL profile into the staging area just created with the commands

```
SQL> exec dbms_sqltune.pack_stgtab_sqlprof(staging_table_name=>'STAGE',  profile_name=>'
sqlt_s89915_p3005811457');
```

4. Export from the development or staging system the SQL Profile Staging table either with `exp` or `expdp`.
5. Import the dump file using `imp` or `impdp` into the production environment
6. Unpack the staging table using the command

```
SQL> exec dbms_sqltune.unpack_stgtab_sqlprof(replace=>TRUE, staging_table_name=>'STAGE');
```

Once you have carried out these steps your target SQL should run with the execution plan from your development or staging environment. You can check it by getting the execution plan as we did earlier on.

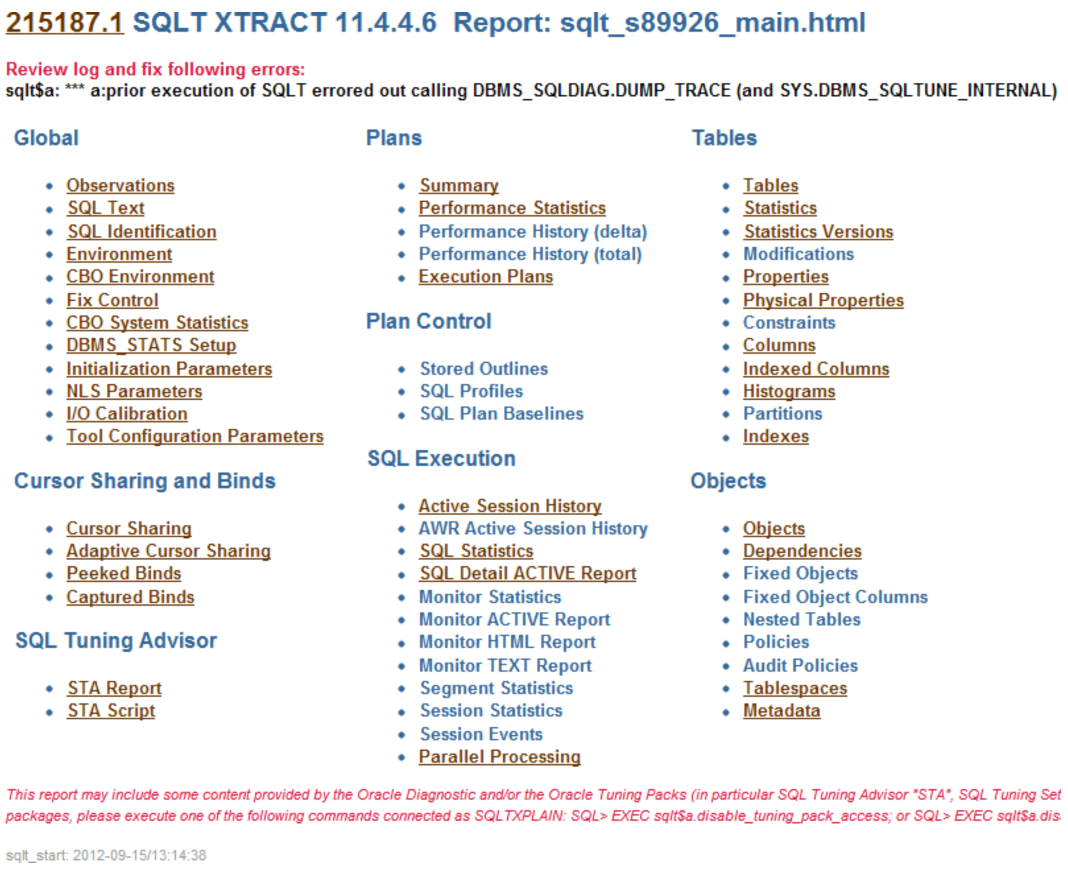
Summary

As you can see SQLT is a powerful tool when used properly. Traditional tuning can seem slow and cumbersome in comparison after you have used SQLT for a while. This is the main objective of this book: to make you familiar with SQLT and then for you to use SQLT often enough to become fast and efficient at tuning SQL. In the next chapter we'll look at a feature introduced in 11g called Adaptive Cursor Sharing. There is much confusion about how this feature works and when it is being used and under what circumstances. SQLT has a section in the main report, which is just for Adaptive Cursor Sharing.

CHAPTER 7

Adaptive Cursor Sharing

There's no doubt that Adaptive Cursor Sharing is one of the most misunderstood and confusing optimizer areas. It doesn't help that it is sometimes referred to as Intelligent Cursor Sharing or Bind Aware Peeking. Adaptive Cursor Sharing was introduced in 11g to take care of those pesky bind variables that keep changing. SQLTXTRACT has a section called "Cursor Sharing and Binds," which can be found at the top of the main report. See Figure 7-1 to remind yourself. The Adaptive Cursor Sharing section is at the bottom left of the screen.


215187.1 SQLT XTRACT 11.4.4.6 Report: sqlt_s89926_main.html

Review log and fix following errors:
sqlt$a: *** a:prior execution of SQLT errored out calling DBMS_SQLDIAG.DUMP_TRACE (and SYS.DBMS_SQLTUNE_INTERNAL)

Global
- Observations
- SQL Text
- SQL Identification
- Environment
- CBO Environment
- Fix Control
- CBO System Statistics
- DBMS_STATS Setup
- Initialization Parameters
- NLS Parameters
- I/O Calibration
- Tool Configuration Parameters

Cursor Sharing and Binds
- Cursor Sharing
- Adaptive Cursor Sharing
- Peeked Binds
- Captured Binds

SQL Tuning Advisor
- STA Report
- STA Script

Plans
- Summary
- Performance Statistics
- Performance History (delta)
- Performance History (total)
- Execution Plans

Plan Control
- Stored Outlines
- SQL Profiles
- SQL Plan Baselines

SQL Execution
- Active Session History
- AWR Active Session History
- SQL Statistics
- SQL Detail ACTIVE Report
- Monitor Statistics
- Monitor ACTIVE Report
- Monitor HTML Report
- Monitor TEXT Report
- Segment Statistics
- Session Statistics
- Session Events
- Parallel Processing

Tables
- Tables
- Statistics
- Statistics Versions
- Modifications
- Properties
- Physical Properties
- Constraints
- Columns
- Indexed Columns
- Histograms
- Partitions
- Indexes

Objects
- Objects
- Dependencies
- Fixed Objects
- Fixed Object Columns
- Nested Tables
- Policies
- Audit Policies
- Tablespaces
- Metadata

This report may include some content provided by the Oracle Diagnostic and/or the Oracle Tuning Packs (in particular SQL Tuning Advisor "STA", SQL Tuning Set packages, please execute one of the following commands connected as SQLTXPLAIN: SQL> EXEC sqlt$a.disable_tuning_pack_access; or SQL> EXEC sqlt$a.dis

sqlt_start: 2012-09-15/13:14:38

Figure 7-1. *The Adaptive Cursor Sharing section can be found in the bottom left hand corner of the screen*

Understanding this area of the optimizer is not helped by the documentation either. It is not exactly crystal clear and lacks some details that are required to understand what happens when you use this feature. I will explain this feature in simple terms and show examples where it has kicked into action. If you have a system where this feature is important, you will better be able to understand what is happening and why. You will also have the option of disabling the feature completely.

Bind Variables and Why We Need Them

Before we go into Adaptive Cursor Sharing (ACS), however, we need to cover some prerequisite knowledge, which will be required for you to follow the discussion and understand the history of ACS. Bind variables are the key component of SQL, which ACS relies on. They are used by programmers in Oracle systems, because over the years Oracle has told them that systems with many literal variables are a bad thing. This is true in the majority of cases, but even today you will see systems with hundreds of thousands of cursors being generated because of literal values. Often because SQL is being created by an application that generates SQL automatically. This is an example of a bind variable in an SQL statement and a literal value of an equivalent statement:

```
variable b varchar2(5);
exec :b := 'TABLE'
select count(object_type) from acs where object_type=:b;
select count(object_type) from acs where object_type='TABLE';
```

Here the bind variable is `b`, and it is being used in the SQL statement (a `select` in this case) to stand in for the value `TABLE`. It would be a reasonable question to ask, "Why do we need bind variables," when we could just as well do

```
select count(object_type) from acs where object_type='TABLE';
```

The answer is that the overhead of hard parsing is prohibitive on busy systems and each SQL cursor's details have to be kept in the shared pool, which can be a big memory overhead. Each of the statements would be exactly the same; we don't want the cost-based optimizer to look in detail at each statement each time the bind variable changed. Back in the dim and distant past (8i), bind variables were available, but were not used very often. Oracle decided that more bind variables should be used to improve performance but without necessarily having bind variables. The `cursor_sharing` parameter was introduced.

The CURSOR_SHARING Parameter

The solution introduced was the `cursor_sharing` parameter, which had possible values of EXACT or FORCE. The default value was EXACT, which resulted in no change in behavior: i.e., if there was a literal value in an SQL statement then it was parsed and executed. If the overhead of the parsing was too high, then you could consider setting the value to FORCE. In this case, the literal value (in the example from above this would be 'TABLE'), would be replaced by a system-defined bind variable (an example would be SYS_B_0). This resulted in an improvement in performance in some cases, but caused problems in some situations where the value of the binds made a difference to the execution plan (in other words, skewed data). A predicate with a rare value and a common value would have the same execution plan as they were using the same system-generated bind variable. This sometimes caused problems, so bind peeking was created to help. Here is an example to show the behavior:

```
SQL> alter session set cursor_sharing='FORCE';
Session altered.
SQL> @q4
  COUNT(*)
----------
         0
```

```
SQL> host type q4.sql
select  count(*) from sh.products p,sh.sales s,sh.customers c where
  c.cust_id=s.cust_id and s.prod_id=p.prod_id and
  c.cust_first_name='Stelios' ;
SQL> select sql_id from v$sqlarea where sql_text like 'select  count(*) from sh.products
p,sh.sales s,sh.customers c where%';
SQL_ID
-------------
7d16bm8ub2w1g
SQL> select sql_text from v$sqlarea where sql_id='7d16bm8ub2w1g';
SQL_TEXT
-----------------------------------------------------------
select  count(*) from sh.products p,sh.sales s,sh.customers c where   c.cust_id=s.cust_id and
s.prod_id=p.prod_id and c.cust_first_name=:"SYS_B_0"

SQL> alter session set cursor_sharing='EXACT';
Session altered.
SQL> @q4
  COUNT(*)
----------
         0
SQL> select sql_id from v$sqlarea where sql_text like 'select  count(*) from sh.products
p,sh.sales s,sh.customers c where%';
SQL_ID
-------------
7d16bm8ub2w1g
fqsukcqvrby36
SQL> select sql_text from v$sqlarea where sql_id='fqsukcqvrby36';
SQL_TEXT
-----------------------------------------------------------
select  count(*) from sh.products p,sh.sales s,sh.customers c where   c.cust_id=s.cust_id and
s.prod_id=p.prod_id and c.cust_first_name='Stelios'
```

In the example above I changed the default value of `cursor_sharing` to `FORCE`. My SQL then changed so that it included a system defined bind variable called `SYS_B_0`. Then I set the value back to `EXACT`, and the bind value disappeared and was replaced by a literal. This is the expected behavior.

Bind Peeking

In 9i Oracle introduced bind peeking. Bind peeking is used to look at the value being used for a bind variable, during the hard parsing process, to determine an appropriate execution plan. A new possible value of SIMILAR for `cursor_sharing` was also introduced. If it was appropriate, during parsing, a new plan was generated; otherwise (based on peeking) the plan was unchanged. The problem with this was that if the first execution plan resulted in a plan that was not good for the subsequent plans then you were stuck with a poorly executing plan.

Bind Sensitive and Bind Aware Cursors

A new hybrid system was introduced by 11g called Adaptive Cursor Sharing (ACS), which is a little more subtle about when to introduce a new plan. The concepts of "bind sensitive" and "bind aware" were introduced. Bind sensitive is true when you have bind variables whose value *may* affect the execution plan. Bind sensitive means that the optimizer suspects that a new plan may be appropriate for some values, but it's not sure. Think of bind sensitive as the first step in becoming aware that more execution plans are needed. You have to be sensitive to the bind variables before you become aware. If you run the SQL a number of times so that the number of buffer gets significantly changed for different bind variables, eventually the cursor will be marked "bind aware." In other words, we went from bind sensitive (we have bind variables) to bind aware (these bind variables make a significant difference to the number of buffer gets). You can track the progress of the bind sensitivity and bind awareness values by looking at the SQLTXPLAIN report.

Setting Up a Bind Sensitive Cursor

To show the behavior of ACS will take a little setting up. In broad terms, here are the steps we'll carry out in the example code. We'll show the code for doing the following below:

1. Creating a test table
2. Inserting skewed data into the test table. There should be enough rows to allow the possibility of multiple execution plans.
3. Showing how skewed the data is
4. Creating indexes to allow them to be used in the execution plan
5. Gathering statistics including histograms on all columns
6. Selecting common and rare values in a number of executions until ACS is activated
7. Checking the results in SQLTXECUTE

```
-- Set up
drop table acs;
create table acs(object_id number, object_type varchar2(19));
insert into acs select object_id, object_type from dba_objects;
insert into acs select object_id, object_type from dba_objects;
select object_type, count(*) from acs group by object_type order by count(object_type);
create index i1 on acs(object_id);
create index i2 on acs(object_type);
exec dbms_stats.gather_table_stats('STELIOS','ACS', estimate_percent=>100,
method_opt=>'FOR ALL COLUMNS SIZE 254');
prompt Now we have a table, ACS with skewed data.
-- Now let's do some selecting...
variable b varchar2(19);
@@acs_query 'SYNONYM'
@@acs_query 'TABLE'
@@acs_query 'SYNONYM'
@@acs_query 'DIMENSION'
@@acs_query 'DIMENSION'
@@acs_query 'DIMENSION'
@@acs_query 'DIMENSION'
@@acs_query 'DIMENSION'
```

```
          @@acs_query 'DIMENSION'
          @@acs_query 'DIMENSION'
          @@acs_query 'DIMENSION'
          @@acs_query 'DIMENSION'
```

We've selected `dba_objects` yet again as a good source for psuedo-random data. In this case, we are interested in the fact that there are many more SYNONYM types in this table than there are DIMENSIONs. The script `acs_query.sql` contains the following:

```
exec :b := '&1'
select count(object_type) from acs where object_type=:b;
```

The output from this script looks like this:

```
SQL>@acs

Table dropped.

Table created.

73378 rows created.

73378 rows created.

OBJECT_TYPE           COUNT(*)
------------------- ----------
RULE                         2
LOB PARTITION                2
EDITION                      2
DESTINATION                  4
JAVA SOURCE                  4
SCHEDULE                     6
MATERIALIZED VIEW            6
SCHEDULER GROUP              8
DIMENSION                   10 <<<Count for DIMENSION
CONTEXT                     14
INDEXTYPE                   18
UNDEFINED                   18
WINDOW                      18
CLUSTER                     20
RESOURCE PLAN               20
JOB CLASS                   26
DIRECTORY                   28
JOB                         28
EVALUATION CONTEXT          30
PROGRAM                     38
RULE SET                    46
CONSUMER GROUP              50
QUEUE                       80
XML SCHEMA                 104
OPERATOR                   110
```

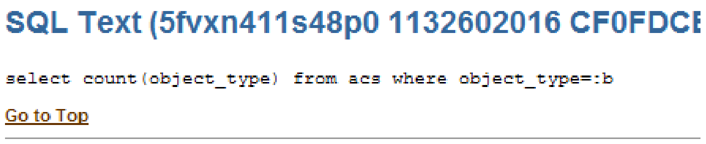
```
PROCEDURE                    320
LIBRARY                      366
TABLE PARTITION              478
TYPE BODY                    480
SEQUENCE                     484
FUNCTION                     604
JAVA DATA                    656
INDEX PARTITION              800
TRIGGER                     1234
JAVA RESOURCE               1668
LOB                         2032
PACKAGE BODY                2536
PACKAGE                     2658
TYPE                        5656
TABLE                       6192 <<<Count for Tables
INDEX                       8122
VIEW                       10340
JAVA CLASS                 45834
SYNONYM                    55604 <<<Count for Synonyms

44 rows selected.

Index created.

Index created.

PL/SQL procedure successfully completed.

Now we have a table, ACS with skewed data.

PL/SQL procedure successfully completed.

COUNT(OBJECT_TYPE)
------------------
             55604 <<<This is the count for SYNONYMS

PL/SQL procedure successfully completed.

COUNT(OBJECT_TYPE)
------------------
              6192 <<<This is the count for TABLES

PL/SQL procedure successfully completed.

COUNT(OBJECT_TYPE)
------------------
             55604
```

```
PL/SQL procedure successfully completed.

COUNT(OBJECT_TYPE)
------------------
                10 <<<Count for DIMENSION

PL/SQL procedure successfully completed.

COUNT(OBJECT_TYPE)
------------------
                10

PL/SQL procedure successfully completed.

COUNT(OBJECT_TYPE)
------------------
                10

PL/SQL procedure successfully completed.

COUNT(OBJECT_TYPE)
------------------
                10

PL/SQL procedure successfully completed.

COUNT(OBJECT_TYPE)
------------------
                10

PL/SQL procedure successfully completed.

COUNT(OBJECT_TYPE)
------------------
                10

PL/SQL procedure successfully completed.

COUNT(OBJECT_TYPE)
------------------
                10

PL/SQL procedure successfully completed.

COUNT(OBJECT_TYPE)
------------------
                10

PL/SQL procedure successfully completed.

COUNT(OBJECT_TYPE)
------------------
                10
```

Let's summarize what happened. We created a test table, populated it with skewed data, selected a popular value, a moderately popular value, a popular value, and finally a rare value a number of times. We would expect the rare value would use an index range scan and the popular value would use a full table scan or fast full index scan. If we generate a SQLT XTRACT report (a SQLTXECUTE report would be just as good) we can now examine what happened.

Examining ACS with a SQLTXTRACT Report

ACS information is gathered for both SQLTXTRACT and for SQLTXECUTE reports. Both of these types of SQLTXPLAIN reports collect the bind variable information (if available) and show you the information in a clear report. Our example report happens to be a SQLTXTRACT report, but the navigation and examples work for both kinds of SQLTXPLAIN reports. First we can check to see if the right SQL was picked up: see Figure 7-2, which shows the SQL with a bind variable as expected.



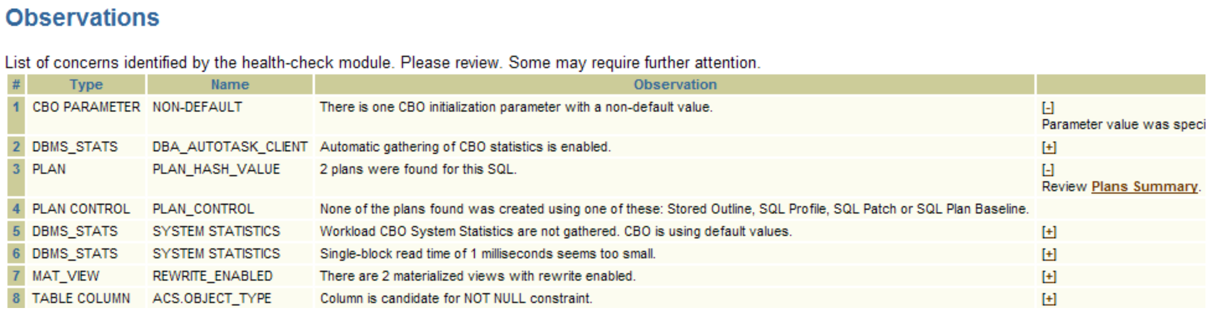
SQL Text (5fvxn411s48p0 1132602016 CF0FDCE

```
select count(object_type) from acs where object_type=:b
```

Go to Top

Figure 7-2. *Shows the SQL with the bind variable*

If we follow the hyperlink from the top of the main SQLTXTRACT report to the Observations section, we can see that, there is an observation that there are multiple plans for the SQL statement. See Figure 7-3, which shows this:



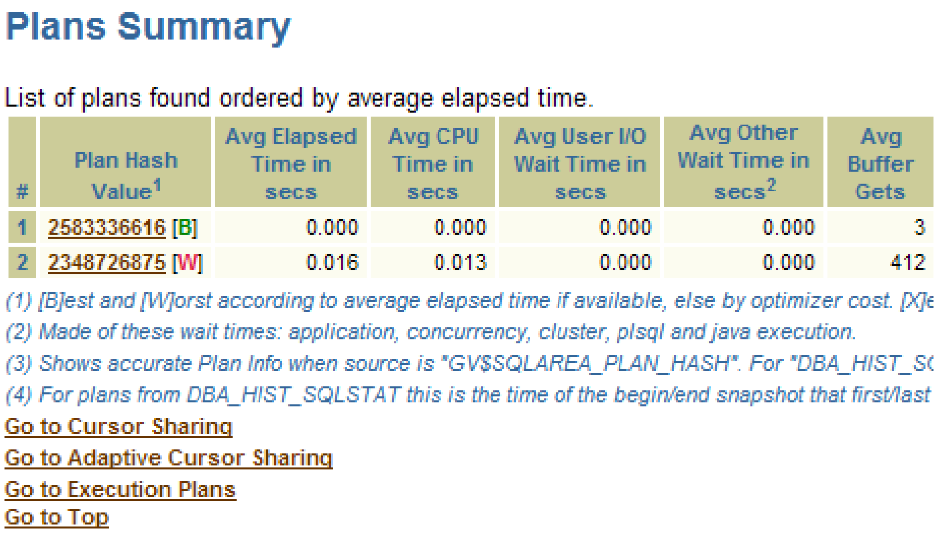
Observations

List of concerns identified by the health-check module. Please review. Some may require further attention.

#	Type	Name	Observation	
1	CBO PARAMETER	NON-DEFAULT	There is one CBO initialization parameter with a non-default value.	[-] Parameter value was speci
2	DBMS_STATS	DBA_AUTOTASK_CLIENT	Automatic gathering of CBO statistics is enabled.	[+]
3	PLAN	PLAN_HASH_VALUE	2 plans were found for this SQL.	[-] Review Plans Summary.
4	PLAN CONTROL	PLAN_CONTROL	None of the plans found was created using one of these: Stored Outline, SQL Profile, SQL Patch or SQL Plan Baseline.	
5	DBMS_STATS	SYSTEM STATISTICS	Workload CBO System Statistics are not gathered. CBO is using default values.	[+]
6	DBMS_STATS	SYSTEM STATISTICS	Single-block read time of 1 milliseconds seems too small.	[+]
7	MAT_VIEW	REWRITE_ENABLED	There are 2 materialized views with rewrite enabled.	[+]
8	TABLE COLUMN	ACS.OBJECT_TYPE	Column is candidate for NOT NULL constraint.	[+]

Figure 7-3. *Shows the Observations section (notice that two plans were found for an SQL)*

If we now click on "Plans Summary" we end up at the Plans Summary section, which shows two exectution plans (as expected). One plan is almost instantaneous and the other takes 0.016 seconds in elapsed time. You can guess that the faster execution time is associated with the value of the predicate when it is set to DIMENSION (the rarer value). See Figure 7-4 for the details of the execution plans.



Plans Summary

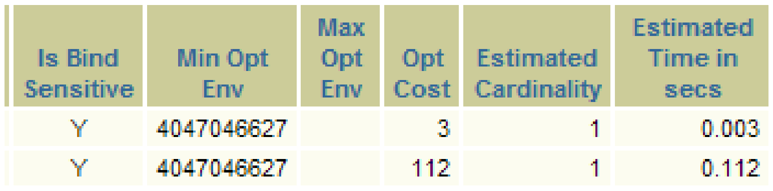
List of plans found ordered by average elapsed time.

#	Plan Hash Value[1]	Avg Elapsed Time in secs	Avg CPU Time in secs	Avg User I/O Wait Time in secs	Avg Other Wait Time in secs[2]	Avg Buffer Gets
1	2583336616 [B]	0.000	0.000	0.000	0.000	3
2	2348726875 [W]	0.016	0.013	0.000	0.000	412

(1) [B]est and [W]orst according to average elapsed time if available, else by optimizer cost. [X]e
(2) Made of these wait times: application, concurrency, cluster, plsql and java execution.
(3) Shows accurate Plan Info when source is "GV$SQLAREA_PLAN_HASH". For "DBA_HIST_S(
(4) For plans from DBA_HIST_SQLSTAT this is the time of the begin/end snapshot that first/last
Go to Cursor Sharing
Go to Adaptive Cursor Sharing
Go to Execution Plans
Go to Top

Figure 7-4. *Two execution plans, a fast and a slow one. One relates to DIMENSION and the other relates to SYNONYM*

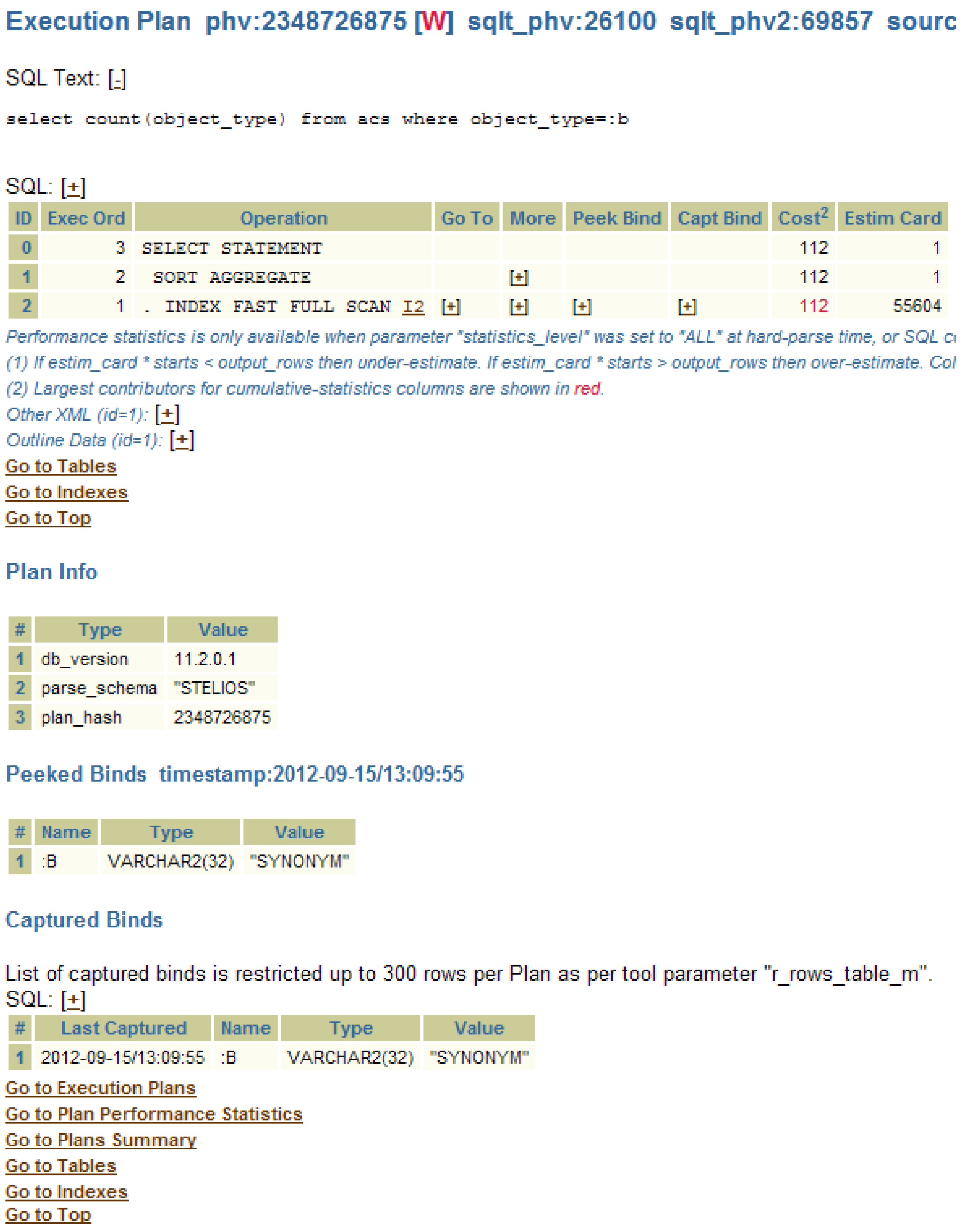
The figure above is the left hand side of the section for the Plans Summary. As expected, we see slower execution, more CPU usage and more buffer gets for the value `Plan Hash Value 2348726875`, because this is the plan associated with the predicate value SYNONYM. If we look at the right hand side of the same section we see more corroborating evidence (see Figure 7-5).



Is Bind Sensitive	Min Opt Env	Max Opt Env	Opt Cost	Estimated Cardinality	Estimated Time in secs
Y	4047046627		3	1	0.003
Y	4047046627		112	1	0.112

Figure 7-5. *The right hand side of the Plan Summary section shows a higher cost and higher estimated time in seconds for the second plan hash value*

To confirm this click on the hyperlink of the plan hash value for the slower plan. This gets us to the details for this plan. See Figure 7-6.



Execution Plan phv:2348726875 [W] sqlt_phv:26100 sqlt_phv2:69857 sourc

SQL Text: [-]

```
select count(object_type) from acs where object_type=:b
```

SQL: [+]

ID	Exec Ord	Operation	Go To	More	Peek Bind	Capt Bind	Cost[2]	Estim Card
0	3	SELECT STATEMENT					112	1
1	2	SORT AGGREGATE		[+]			112	1
2	1	. INDEX FAST FULL SCAN I2	[+]	[+]	[+]	[+]	112	55604

Performance statistics is only available when parameter "statistics_level" was set to "ALL" at hard-parse time, or SQL c
*(1) If estim_card * starts < output_rows then under-estimate. If estim_card * starts > output_rows then over-estimate. Col*
(2) Largest contributors for cumulative-statistics columns are shown in red.
Other XML (id=1): [+]
Outline Data (id=1): [+]
Go to Tables
Go to Indexes
Go to Top

Plan Info

#	Type	Value
1	db_version	11.2.0.1
2	parse_schema	"STELIOS"
3	plan_hash	2348726875

Peeked Binds timestamp:2012-09-15/13:09:55

#	Name	Type	Value
1	:B	VARCHAR2(32)	"SYNONYM"

Captured Binds

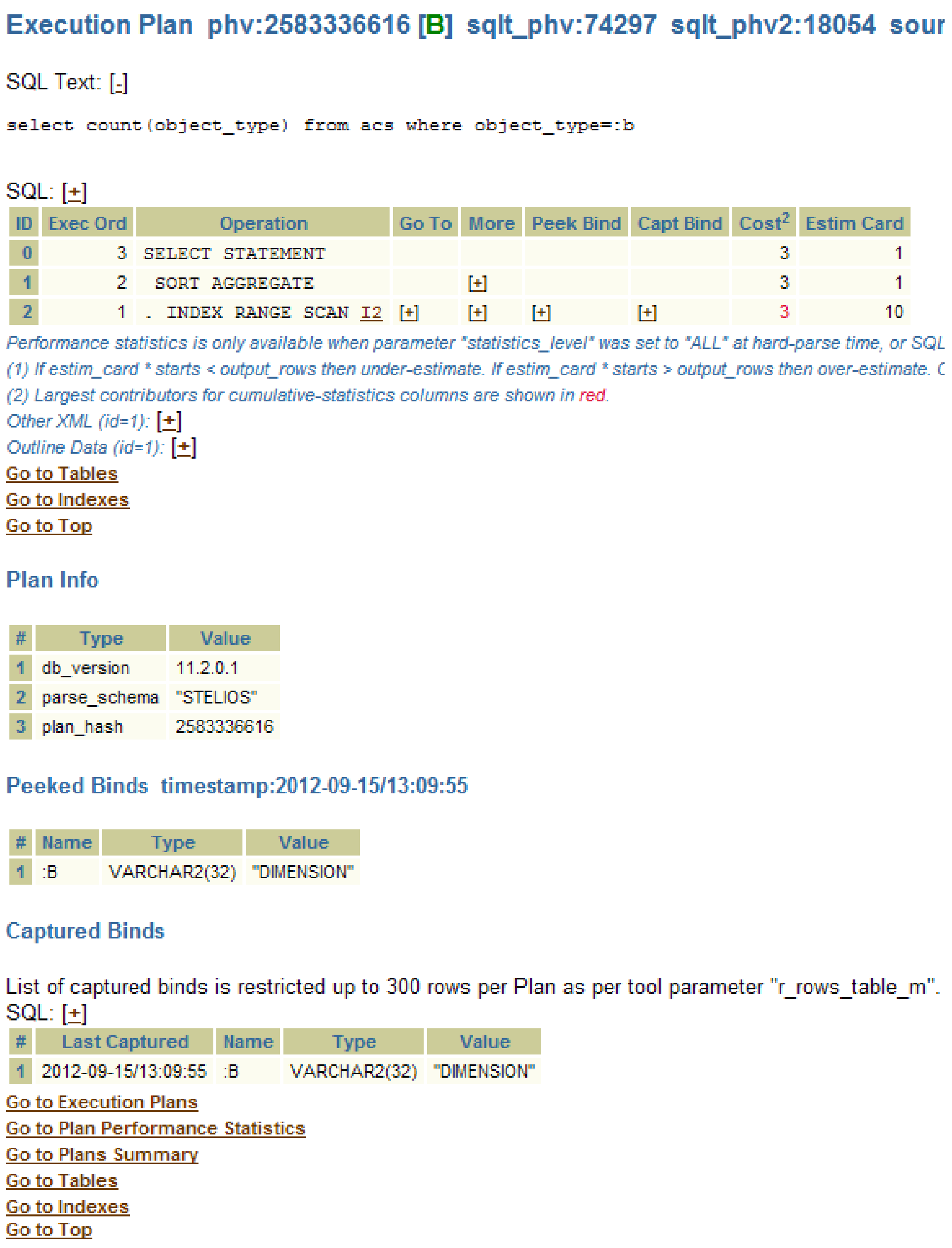
List of captured binds is restricted up to 300 rows per Plan as per tool parameter "r_rows_table_m".
SQL: [+]

#	Last Captured	Name	Type	Value
1	2012-09-15/13:09:55	:B	VARCHAR2(32)	"SYNONYM"

Go to Execution Plans
Go to Plan Performance Statistics
Go to Plans Summary
Go to Tables
Go to Indexes
Go to Top

Figure 7-6. *Here we see the execution plan, the peeked binds and the captured binds*

As expected a slower execution resulted when the value SYNONYM was used. This was because there were 55,604 values in the table that matched "SYNONYM". A Fast Full Scan therefore seems appropriate. If we look at the execution plan for DIMENSION (which only matches 10 rows) we'd see a cost of only three and a plan based around and INDEX RANGE SCAN. See Figure 7-7, which shows this.



Execution Plan phv:2583336616 [B] sqlt_phv:74297 sqlt_phv2:18054 sour

SQL Text: [-]

```
select count(object_type) from acs where object_type=:b
```

SQL: [+]

ID	Exec Ord	Operation	Go To	More	Peek Bind	Capt Bind	Cost^2	Estim Card
0	3	SELECT STATEMENT					3	1
1	2	SORT AGGREGATE		[+]			3	1
2	1	. INDEX RANGE SCAN I2	[+]	[+]	[+]	[+]	3	10

Performance statistics is only available when parameter "statistics_level" was set to "ALL" at hard-parse time, or SQL
*(1) If estim_card * starts < output_rows then under-estimate. If estim_card * starts > output_rows then over-estimate. C*
(2) Largest contributors for cumulative-statistics columns are shown in red.
Other XML (id=1): [+]
Outline Data (id=1): [+]
Go to Tables
Go to Indexes
Go to Top

Plan Info

#	Type	Value
1	db_version	11.2.0.1
2	parse_schema	"STELIOS"
3	plan_hash	2583336616

Peeked Binds timestamp:2012-09-15/13:09:55

#	Name	Type	Value
1	:B	VARCHAR2(32)	"DIMENSION"

Captured Binds

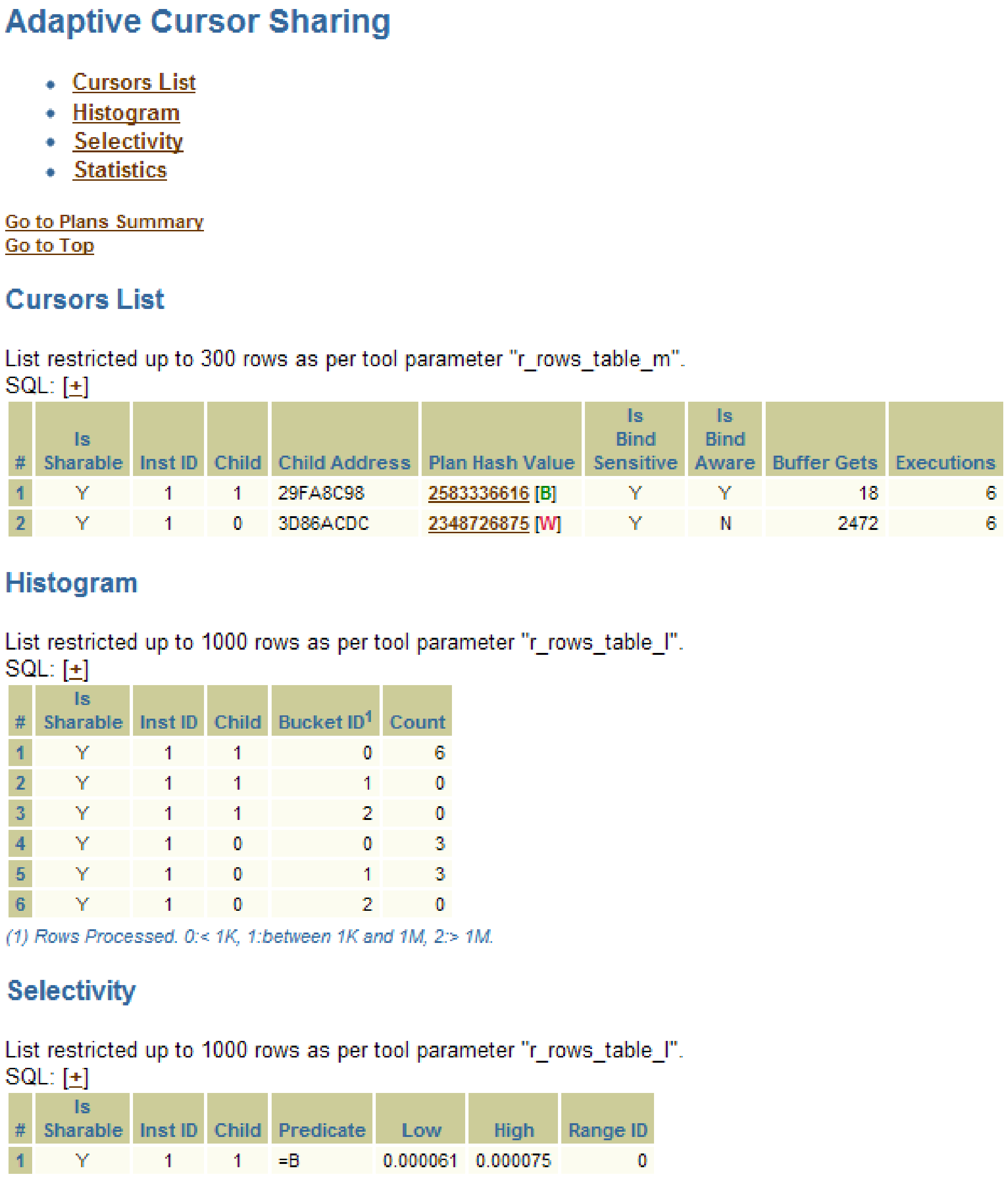
List of captured binds is restricted up to 300 rows per Plan as per tool parameter "r_rows_table_m".
SQL: [+]

#	Last Captured	Name	Type	Value
1	2012-09-15/13:09:55	:B	VARCHAR2(32)	"DIMENSION"

Go to Execution Plans
Go to Plan Performance Statistics
Go to Plans Summary
Go to Tables
Go to Indexes
Go to Top

Figure 7-7. *Shows the execution plan details for the plan associated with DIMENSION, a rare value*

Now let's go back to the top of the report and click on "Adaptive Cursor Sharing". This brings us to the section shown in Figure 7-8.



Adaptive Cursor Sharing

- Cursors List
- Histogram
- Selectivity
- Statistics

Go to Plans Summary
Go to Top

Cursors List

List restricted up to 300 rows as per tool parameter "r_rows_table_m".
SQL: [+]

#	Is Sharable	Inst ID	Child	Child Address	Plan Hash Value	Is Bind Sensitive	Is Bind Aware	Buffer Gets	Executions
1	Y	1	1	29FA8C98	2583336616 [B]	Y	Y	18	6
2	Y	1	0	3D86ACDC	2348726875 [W]	Y	N	2472	6

Histogram

List restricted up to 1000 rows as per tool parameter "r_rows_table_l".
SQL: [+]

#	Is Sharable	Inst ID	Child	Bucket ID(1)	Count
1	Y	1	1	0	6
2	Y	1	1	1	0
3	Y	1	1	2	0
4	Y	1	0	0	3
5	Y	1	0	1	3
6	Y	1	0	2	0

(1) Rows Processed. 0:< 1K, 1:between 1K and 1M, 2:> 1M.

Selectivity

List restricted up to 1000 rows as per tool parameter "r_rows_table_l".
SQL: [+]

#	Is Sharable	Inst ID	Child	Predicate	Low	High	Range ID
1	Y	1	1	=B	0.000061	0.000075	0

Figure 7-8. *Shows the section on adaptive cursor sharing*

The section on Adaptive Cursor Sharing needs some explanation. The cursor list part of this section, shows all child cursors related to the SQL text. There is a child cursor 0 and 1. The child cursor (1) relates to the DIMENSION value (as this came after the SYNONYM value). We also see under the "Is Bind Sensitive" column that both child cursors are bind sensitive, and in this case (because we ran the statement relating to DIMENSION a few times), we also have an "Is Bind Aware" value of "Y" for the Child Cursor 1. A cursor is always bind sensitive when there is a bind variable with an associated histogram. If we look at the values of IS_BIND_SENSITIVE and IS_BIND_AWARE throughout this process we'll see the following:

```
SQL> select IS_BIND_SENSITIVE S, IS_BIND_AWARE A from v$sqlarea where sql_id=' 5fvxn411s48p0';

S A
- -
Y N <<<AFTER FIRST EXECUTION (SYNONYM)

Y N <<<AFTER SECOND EXECUTION (TABLE)

Y N <<<AFTER THIRD EXECUTION (SYNONYM)

Y N <<<AFTER FOURTH EXECUTION (DIMENSION)

Y N <<<AFTER FIFTH EXECUTION (DIMENSION)

Y N <<<AFTER SIXTH EXECUTION (DIMENSION)

Y N <<<AFTER SEVENTH EXECUTION (DIMENSION)
Y Y

Y N <<<AFTER EIGHT EXECUTION (DIMENSION)
Y Y

Y N <<<AFTER NINTH EXECUTION (DIMENSION)
Y Y

Y N <<<AFTER TENTH EXECUTION (DIMENSION)
Y Y

Y N <<<AFTER ELEVENTH EXECUTION (DIMENSION)
Y Y

Y N <<<AFTER TWELFTH EXECUTION (DIMENSION)
Y Y
```

The Bind Sensitive column is always set to "yes" in this case, for all the children. But how does the second child get created and when? The algorithm is not published or documented, but the clue is under the histogram heading in the Adaptive Cursor Sharing section. Look at the Bucket ID column. There appear to be only three buckets, 0, 1, and 2. The annotation at the bottom mentions that a 0 bucket represents less than 1k. The 1 bucket represents greater than 1k but less than 1M, and bucket 2 represents everything that is more than 1M. So for cursor 0 (remember this was the SYNONYM and TABLE values) there were a total of six executions, all of them with fewer than 1M buffer gets. For child number 1 the situation is different. Every one of its six executions had fewer than 1K in buffer gets. This is why after the seventh execution a new child is created that is bind aware. Some DBAs call this "warming the bucket." The idea is there is no point in creating a new bucket (and the overhead that goes with that) if the predicate value is rarely going to be seen. We also judge if a predicate is the same as another child by this broad criteria. Once the cursor is bind aware it can be used for matching bind values that are peeked. In this simple example this would mean that two execution plans are possible and available based on the bind values.

This chapter covered what Adaptive Cursor Sharing is and how it works. The example shown is a simple one-dimensional example; there was only one column with skewed data. If you had a table with multiple columns of skewed data then you could potentially get different combinations of the columns, which would each generate their own child cursor. ACS allows for this by creating a range of selectivities for each column.

Does ACS Go Wrong?

As with all features that attempt to improve performance, there are rare cases where performance actually deteriorates. If you feel your execution plans are unstable, or the wrong plans have been chosen, you can always disable ACS by setting the following paramters (after first checking with Oracle Support):

```
SQL> alter system set "_optimizer_extended_cursor_sharing_rel"=NONE scope=both;
SQL> alter system set "_optimizer_extended_cursor_sharing"=none scope=both;
```

Now that we have disabled ACS what is the result or running acs.sql again?

```
SQL> @acs
Table dropped.
Table created.
73377 rows created.
73377 rows created.
OBJECT_TYPE           COUNT(*)
------------------- ----------
EDITION                      2
RULE                         2
LOB PARTITION                2
DESTINATION                  4
JAVA SOURCE                  4
SCHEDULE                     6
MATERIALIZED VIEW            6
SCHEDULER GROUP              8
DIMENSION                   10
CONTEXT                     14
INDEXTYPE                   18
UNDEFINED                   18
WINDOW                      18
RESOURCE PLAN               20
CLUSTER                     20
JOB CLASS                   26
JOB                         28
DIRECTORY                   28
EVALUATION CONTEXT          30
PROGRAM                     38
RULE SET                    46
CONSUMER GROUP              50
QUEUE                       80
XML SCHEMA                 104
OPERATOR                   110
PROCEDURE                  320
```

```
LIBRARY                     366
TABLE PARTITION             478
TYPE BODY                   480
SEQUENCE                    484
FUNCTION                    604
JAVA DATA                   656
INDEX PARTITION             800
TRIGGER                    1234
JAVA RESOURCE              1668
LOB                        2032
PACKAGE BODY               2536
PACKAGE                    2658
TYPE                       5648
TABLE                      6194
INDEX                      8126
VIEW                      10340
JAVA CLASS                45834
SYNONYM                   55604

44 rows selected.

Index created.

Index created.

PL/SQL procedure successfully completed.

Now we have a table, ACS with skewed data.
PL/SQL procedure successfully completed.

COUNT(OBJECT_TYPE)
------------------
             55604
S A
- -
N N

PL/SQL procedure successfully completed.

COUNT(OBJECT_TYPE)
------------------
              6194

S A
- -
N N
```

```
PL/SQL procedure successfully completed.

COUNT(OBJECT_TYPE)
------------------
             55604

S A
- -
N N

PL/SQL procedure successfully completed.

COUNT(OBJECT_TYPE)
------------------
                10

S A
- -
N N

PL/SQL procedure successfully completed.

COUNT(OBJECT_TYPE)
------------------
                10

S A
- -
N N

PL/SQL procedure successfully completed.

COUNT(OBJECT_TYPE)
------------------
                10

S A
- -
N N

PL/SQL procedure successfully completed.

COUNT(OBJECT_TYPE)
------------------
                10

S A
- -
N N
```

```
PL/SQL procedure successfully completed.

COUNT(OBJECT_TYPE)
------------------
                10

S A
- -
N N

PL/SQL procedure successfully completed.

COUNT(OBJECT_TYPE)
------------------
                10

S A
- -
N N

PL/SQL procedure successfully completed.

COUNT(OBJECT_TYPE)
------------------
                10

S A
- -
N N

PL/SQL procedure successfully completed.

COUNT(OBJECT_TYPE)
------------------
                10

S A
- -
N N

PL/SQL procedure successfully completed.

COUNT(OBJECT_TYPE)
------------------
                10

S A
- -
N N
```

Here we see that IS-BIND_SENSITIVE and IS_BIND_AWARE do not change throughout the exercise. No cursors were marked as bind sensitive or bind aware.

Summary

In this chapter, we covered the basic behavior of Adaptive Curosr Sharing. We also saw how it was an evolution of various features introduced over the years by Oracle. No doubt there will be further development in this area. With SQLTXPLAIN we can collect information on ACS and get enough information to decide if it is going to be helpful. In the next chapter we'll cover another couple of features, Dynamic Sampling and Cardinality Feedback, which try to fix problems with execution plans as they occur.

CHAPTER 8

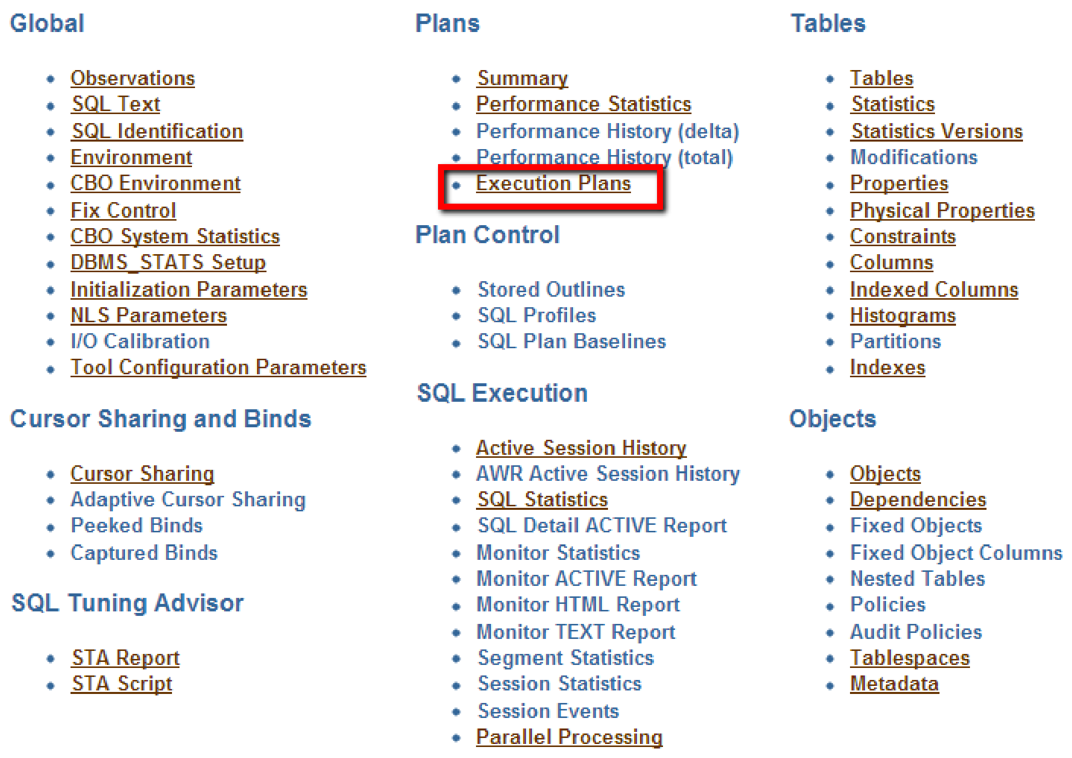
Dynamic Sampling and Cardinality Feedback

Imagine you've been given the job of sorting through some books in a house to find all the references to the word "Vienna" in the book titles. You've been told there are 2 or 3 books in the study, and that there is an index in the desk if you need it. Naturally with 2 or 3 books or even a dozen books, there's no point in using the index in the desk, you'll just look at the book titles one by one and quickly find all of the books with "Vienna" in the title. Unfortunately, what you didn't know is that since the last person looked at the books the owner has bought the entire contents of the local public library, but you only discover this after you start. It's going to be a long day if you stick to your plan.

In this chapter we'll see how Oracle's dynamic sampling (DS) and cardinality feedback (CFB) features could help our poor librarian. We'll see that these features both work together at the beginning and at the end of the parsing process to get the right answer fast. We'll learn that DS and CFB are not always used, and we'll learn when those occasions arise. With this knowledge fresh in our minds we'll also look at an example mystery case that we'll follow through with SQLTXPLAIN. Who will be the villain and who will be the hero this time?

Dynamic Sampling?

The poor optimizer has to cope with occasionally coming across poor objects statistics (we covered that in Chapter 3), or completely missing statistics. Rather than just letting this situation occur and accepting that missing or poor statistics sometimes cause poor execution plans, Oracle developed dynamic sampling and cardinality feedback. Both of these great features work together to correct missing or poor statistics. SQLT reports when it detects that dynamic sampling is used as well as cardinality feedback. Let's look at an example report. Figure 8-1 shows the top of the now familiar SQLTXPLAIN main report.



Global

- Observations
- SQL Text
- SQL Identification
- Environment
- CBO Environment
- Fix Control
- CBO System Statistics
- DBMS_STATS Setup
- Initialization Parameters
- NLS Parameters
- I/O Calibration
- Tool Configuration Parameters

Cursor Sharing and Binds

- Cursor Sharing
- Adaptive Cursor Sharing
- Peeked Binds
- Captured Binds

SQL Tuning Advisor

- STA Report
- STA Script

Plans

- Summary
- Performance Statistics
- Performance History (delta)
- Performance History (total)
- Execution Plans

Plan Control

- Stored Outlines
- SQL Profiles
- SQL Plan Baselines

SQL Execution

- Active Session History
- AWR Active Session History
- SQL Statistics
- SQL Detail ACTIVE Report
- Monitor Statistics
- Monitor ACTIVE Report
- Monitor HTML Report
- Monitor TEXT Report
- Segment Statistics
- Session Statistics
- Session Events
- Parallel Processing

Tables

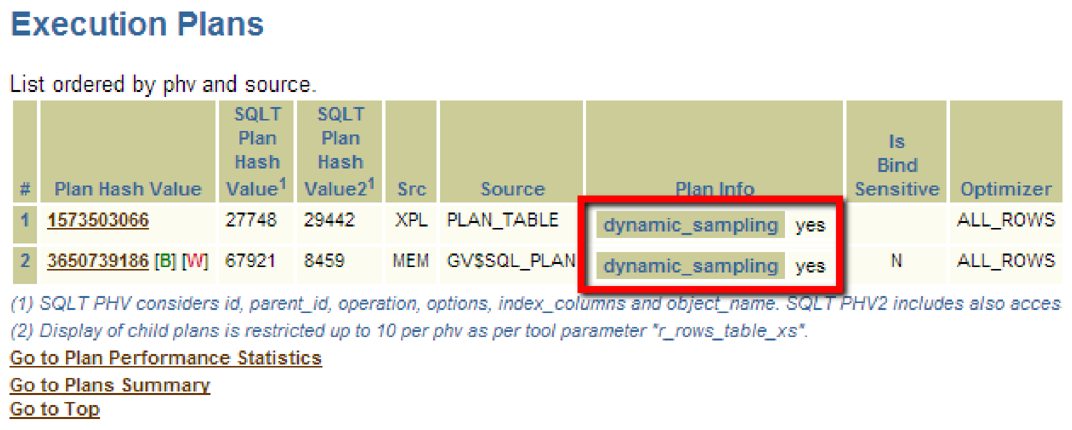
- Tables
- Statistics
- Statistics Versions
- Modifications
- Properties
- Physical Properties
- Constraints
- Columns
- Indexed Columns
- Histograms
- Partitions
- Indexes

Objects

- Objects
- Dependencies
- Fixed Objects
- Fixed Object Columns
- Nested Tables
- Policies
- Audit Policies
- Tablespaces
- Metadata

Figure 8-1. *The top of the SQLTXPLAIN report*

From here if we click on "Execution Plans" we naturally end up at the section of the report showing all the known execution plans for our current SQL. If dynamic sampling was used we see a message in the report under "Plan Info". See Figure 8-2 for an example report showing dynamic sampling being used. This figure only shows the left hand side of the page.



Execution Plans

List ordered by phv and source.

#	Plan Hash Value	SQLT Plan Hash Value[1]	SQLT Plan Hash Value2[1]	Src	Source	Plan Info		Is Bind Sensitive	Optimizer
1	1573503066	27748	29442	XPL	PLAN_TABLE	dynamic_sampling	yes		ALL_ROWS
2	3650739186 [B] [W]	67921	8459	MEM	GV$SQL_PLAN	dynamic_sampling	yes	N	ALL_ROWS

(1) SQLT PHV considers id, parent_id, operation, options, index_columns and object_name. SQLT PHV2 includes also acces
(2) Display of child plans is restricted up to 10 per phv as per tool parameter "r_rows_table_xs".

Go to Plan Performance Statistics
Go to Plans Summary
Go to Top

Figure 8-2. *Both execution plans used dynamic sampling*

What Is Dynamic Sampling?

So dynamic sampling was used, but what is it exactly? It is a way to improve the statistics of a query by collecting those statistics *during* the compilation process. Don't assume from this that you can ignore the collection of statistics, however. You can't. Dynamic sampling is not a substitute for good statistics. It's a last attempt to avoid a bad execution plan. In the simplest terms the steps the optimizer goes through are shown below:

1. The optimizer starts parsing the query.
2. During the parsing process the optimizer assesses the state of the object statistics.
3. If the optimizer finds that some statistics are missing, it may do some dynamic sampling. The amount, and even whether dynamic sampling is done, will depend on the value of `optimizer_dynamic_sampling`.
4. If no dynamic sampling is to be done the rest of the optimization process continues, and statistics are used where available.
5. If dynamic sampling is to be done, the amount of sampling is determined from `optimizer_dynamic_sampling`, and a dynamic query is generated to gather the information.
6. If dynamic sampling was done then the statistics gathered are used to generate a better execution plan.

In Figure 8-2 we saw that dynamic sampling was used, but nowhere is there an indication of the value of the parameter. This can be seen in the CBO Environment part of the report. (See the section below "How to find the value of `optimizer_dynamic_sampling`.") Remember, this all happens *before* the query executes and during the parsing process.

How to Control Dynamic Sampling

Dynamic sampling is controlled by the value of `optimizer_dynamic_sampling` as already mentioned. It can be set at the system level or at the session level or by using hints in the SQL. These different options allow the behavior of dynamic sampling to be very carefully controlled and be set to behave in different ways for different SQL (with hints) and different sessions (log on triggers for example). Here are examples of all of these options. (I've set the value to 4. You can set any value from 0 to 10.) First set it at the session level.

```
SQL> alter session set optimizer_dynamic_sampling=4 ;
Session altered.
```

You may want to set the parameter at the session level if you are testing some SQL and want to see the effect on the execution plan for a number of different values and you want to see it quickly without affecting anybody else. You can also set the value in the current instance without making the value permanent.

```
SQL> alter system set optimizer_dynamic_sampling=4 scope=memory;
System altered.
```

You might want to do this if you are testing at the system level and want to be sure it is the right choice across the system before making the change permanent. Once you've decided that setting this value at the system level is appropriate, you can set it as shown below.

```
SQL> alter system set optimizer_dynamic_sampling=4 scope=spfile;
System altered.
```

Setting the value at the system level makes changes to the spfile so that it is applied to the database the next time it starts. If you want to make changes on a more granular level, perhaps individual SQLs you may want to use a hint. The hint version of this parameter can take two forms: a cursor level and a table level version. So, for example, to set for the cursor:

```
SQL> select /*+ dynamic_sampling (4) */ count(*) from dba_objects;
  COUNT(*)
----------
     73454
```

There is a different form of this hint that allows yout to set the sampling level on a table. This is getting very specific: and you have to wonder, if you know the object statistics on this particular object are missing, why haven't you collected real statistics?

```
SQL> select /*+ dynamic_sampling (dba_objects 4) */ count(*) from dba_objects;

  COUNT(*)
----------
     73454
```

As I mentioned earlier, the amount of sampling and whether sampling is done depends on the value of the dynamic sampling parameter value. Most systems will have the default value of 2.

```
C:\Documents and Settings\Stelios>sqlplus / as sysdba
SQL*Plus: Release 11.2.0.1.0 Production on Sat Oct 13 11:47:40 2012
Copyright (c) 1982, 2010, Oracle.  All rights reserved.
Connected to:
Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - Production
With the Partitioning, OLAP, Data Mining and Real Application Testing options
SQL> show parameter optimizer_dynamic_sampling
NAME                                 TYPE        VALUE
------------------------------------ ----------- ----------------------------
optimizer_dynamic_sampling           integer     2
```

If the default value is set and dynamic sampling is used, then the optimizer will attempt to sample 64 blocks of data, unless the query is parallelized (see section below, "Dynamic Sampling and Parallel Statements"). This is not a percentage of the table or index size, it is a fixed number of blocks. The number of rows sampled is dependent on how many rows fit into a block. Remember the objective of dynamic sampling is to get some very basic statistics at the last moment just before the query is executed. To minimize this overhead the sampling size is set in blocks to, a clearly defined value that cannot expand or contract with the size of the table rows. Dynamic sampling was designed this way to stop the parsing process from consuming too many resources on large tables. If the dynamic sampling process takes place (and we can check in the SQLTXPLAIN report) the samples collected may help to make the execution plan better than it would otherwise be. The values for the parameter (as mentioned earlier) vary from 0 to 10, and control the operation of dynamic sampling. If optimizer_dynamic_sampling is set to

- 0: No dynamic sampling is used under any circumstances.
- 1: If there is at least 1 unanalyzed, unindexed, nonpartitioned table and this table is bigger than 32 blocks, then 32 blocks are sampled. This means that if the table is indexed or is partitioned or is smaller than 32 blocks no dynamic sampling will take place.
- 2: If at least one table has no statistics, whether it has been indexed or not, then 64 blocks are sampled. Partitioned and indexed tables are included in this. This will apply to all tables with no statistics unlike level 1, where some tables will be excluded.

- 3: If at least one table has no statistics and if there is a where clause with an expression, 64 blocks are sampled. This is trying to fix the problem of expressions on where clauses where it can be tricky to develop the right execution plan. This is more restrictive than level 2. This still applies to all tables just like level 2.
- 4: If at least one table has no statistics and if an OR or AND operator is used on predicates on the same table, then 64 blocks are sampled. This is attempting to deal with the problem of complex predicates. This is more restrictive than level 2. This level also applies to indexed and partitioned tables just like level 2.

For values between 5 and 8 the rules are unchanged from the value for optimizer_dynamic_sampling set to 4 but the sample sizes increase, doubling each time so that for 5 the sample size is 128 blocks and for 8 the sample size is 1024. For level 9, the sample size is 4086. For a value of 10 all blocks are sampled. As you can imagine, setting this value can be a very big overhead. If we generate the 10053 trace for a query against the sales table, we would type the following commands:

```
SQL> ALTER SESSION SET MAX_DUMP_FILE_SIZE = UNLIMITED;
SQL> ALTER SESSION SET TRACEFILE_IDENTIFIER = '10053_TRACE';
SQL> ALTER SESSION SET EVENTS '10053 TRACE NAME CONTEXT FOREVER, LEVEL 1';
SQL> select /*+ PARSE 5 / count(*) from sales;
SQL> exit
```

Here I set the dump file size (the trace file) to an unlimited size and appended a string to the automatically generated file name so I can easily find the file by setting the TRACEFILE_IDENTIFIER value to 10053_TRACE. The generated name for the trace file will be made up from the sid (in this case snc1), the string "ora", a session number (5556 in this case), and then my appended string. Your file name will be different, but if you use a sensible TRACEFILE_IDENTIFIER value you should be able to find your trace file easily. If you wanted to see the overhead, you could look in the 10053 trace file for a query that was using dynamic sampling. If you search the 10053 trace file, you'll see a section similar to the one below. (I've removed some of the text for clarity.)

Now if we look in the user_dump_dest location, we will find a file called snc1_ora_5556_10053_TRACE.trc. If we then search this file for the string "dynamic sampling", we'll see the section below.

```
*** 2012-12-01 12:59:17.671
** Performing dynamic sampling initial checks. **
** Dynamic sampling initial checks returning TRUE (level = 4).<<<Dynamic sampling value
*** 2012-12-01 12:59:18.078
** Generated dynamic sampling query: <<<A dynamic query is generated
    query text :
SELECT /* OPT_DYN_SAMP */ /*+ ALL_ROWS IGNORE_WHERE_CLAUSE NO_PARALLEL(SAMPLESUB) opt_
param('parallel_execution_enabled', 'false') NO_PARALLEL_INDEX(SAMPLESUB) NO_SQL_TUNE */
NVL(SUM(C1),0), NVL(SUM(C2),0) FROM (SELECT /*+ NO_PARALLEL("SALES") FULL("SALES") NO_PARALLEL_
INDEX("SALES") */ 1 AS C1, 1 AS C2 FROM "SALES" SAMPLE BLOCK (2.035048 , 1) SEED (1) "SALES")
SAMPLESUB

*** 2012-12-01 12:59:18.093
** Executed dynamic sampling query:
    level : 4
    sample pct. : 2.035048 <<<2 percent of the table was sampled
    total partitions : 28 <<<There were 28 partitions in the table.
      partitions for sampling : 28
    actual sample size : 18860 <<<Sample size used
    filtered sample card. : 18860
    orig. card. : 145484 <<<Original estimate for cardinality
    block cnt. table stat. : 1769
    block cnt. for sampling: 1769
    max. sample block cnt. : 64
    sample block cnt. : 36
    min. sel. est. : -1.00000000
** Using dynamic sampling card. : 926759 <<<New estimated cardinality
** Dynamic sampling updated table card.
```

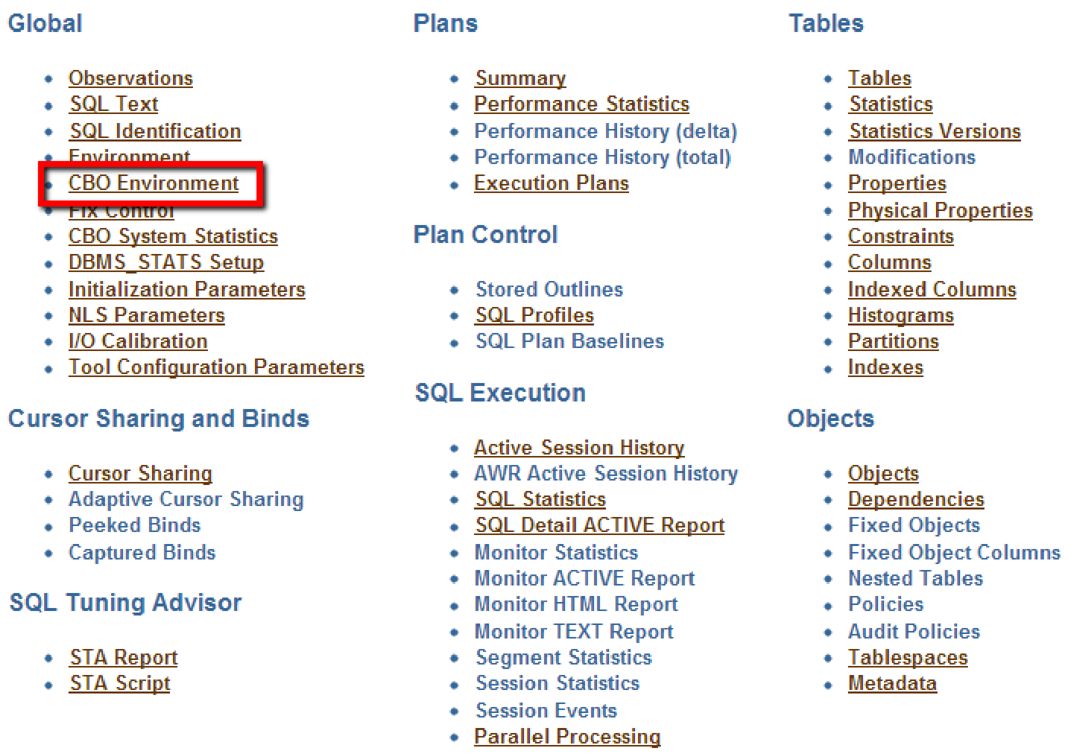
Let me step through what happens in this 10053 trace file. First we see that `optimizer_dynamic_sampling` is detected at level 4. Then a dynamic sampling query is generated. The query text is shown. There are a number of interesting options used for the hints in this query

- `/* OPT_DYN_SAMP */` - This is not a hint it is just a comment.
- `/*+ ALL_ROWS` – The `ALL_ROWS` hint, a standard hint.
- `IGNORE_WHERE_CLAUSE` – This ignores any `WHERE` clauses.
- `NO_PARALLEL(SAMPLESUB)` – No parallel execution, the overhead from this dynamic query must not be allowed to take too many resources.
- `opt_param('parallel_execution_enabled','false')` – No parallel execution.
- `NO_PARALLEL_INDEX` – No parallel Index plans.
- `NO_SQL_TUNE */` - Undocumented hint.

Then the dynamic sampling query is executed and from the value 4 of the parameter we were able to sample approximately 2 percent of the rows. You can see in the query that the blocks sampled are randomized (`SEED (1)` and that we are using the `SAMPLE` clause, which samples blocks from a table. So did the dynamic sampling query do any good? The original estimate for the cardinality was 145,484. After the dynamic sampling query is executed the new estimate is 926,759. This is much closer to the actual value of 918,843. The value of the controlling parameter is pretty important; so next we'll see how to find out its value.

How to Find the Value of optimizer_dynamic_sampling

We can see the actual value used for `optimizer_dynamic_sampling` by looking at the "CBO Environment" section of the SQLT report. The hyperlink is shown in Figure 8-3 below.



Global

- Observations
- SQL Text
- SQL Identification
- Environment
- CBO Environment
- Fix Control
- CBO System Statistics
- DBMS_STATS Setup
- Initialization Parameters
- NLS Parameters
- I/O Calibration
- Tool Configuration Parameters

Cursor Sharing and Binds

- Cursor Sharing
- Adaptive Cursor Sharing
- Peeked Binds
- Captured Binds

SQL Tuning Advisor

- STA Report
- STA Script

Plans

- Summary
- Performance Statistics
- Performance History (delta)
- Performance History (total)
- Execution Plans

Plan Control

- Stored Outlines
- SQL Profiles
- SQL Plan Baselines

SQL Execution

- Active Session History
- AWR Active Session History
- SQL Statistics
- SQL Detail ACTIVE Report
- Monitor Statistics
- Monitor ACTIVE Report
- Monitor HTML Report
- Monitor TEXT Report
- Segment Statistics
- Session Statistics
- Session Events
- Parallel Processing

Tables

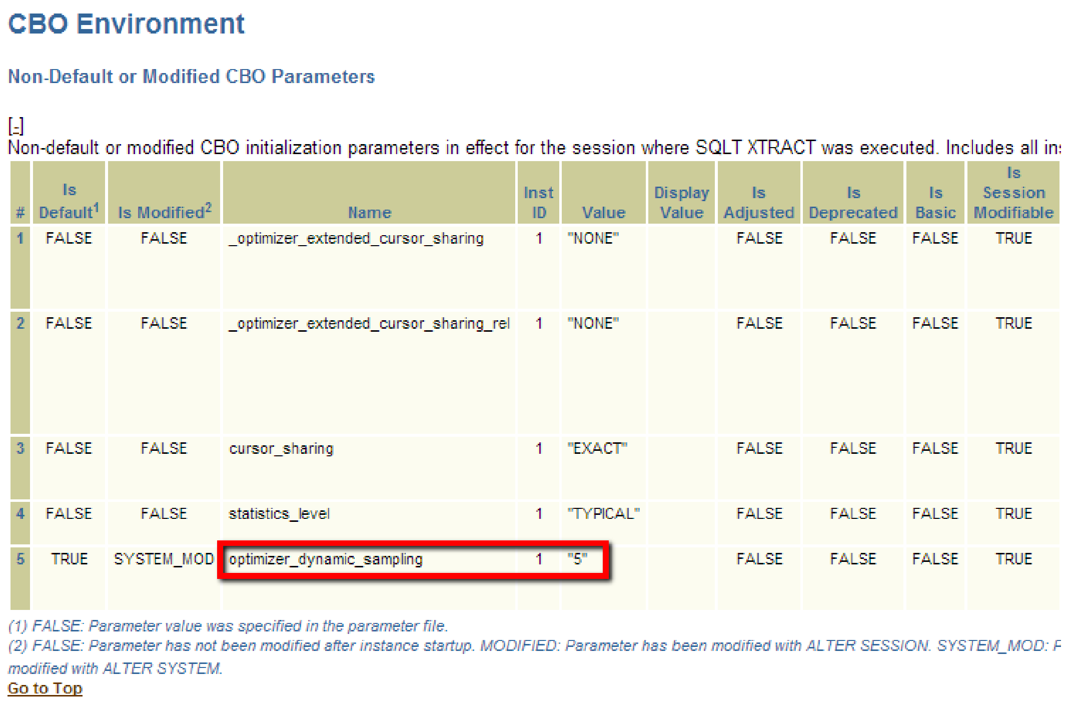
- Tables
- Statistics
- Statistics Versions
- Modifications
- Properties
- Physical Properties
- Constraints
- Columns
- Indexed Columns
- Histograms
- Partitions
- Indexes

Objects

- Objects
- Dependencies
- Fixed Objects
- Fixed Object Columns
- Nested Tables
- Policies
- Audit Policies
- Tablespaces
- Metadata

Figure 8-3. *The hyperlink that takes you to the non-default CBO parameters*

Once we've clicked on this we see the "CBO Environment" part of the report, which shows, amongst other things, the value of `optimizer_dynamic_sampling`. See Figure 8-4.



CBO Environment

Non-Default or Modified CBO Parameters

[-]
Non-default or modified CBO initialization parameters in effect for the session where SQLT XTRACT was executed. Includes all in:

#	Is Default[1]	Is Modified[2]	Name	Inst ID	Value	Display Value	Is Adjusted	Is Deprecated	Is Basic	Is Session Modifiable
1	FALSE	FALSE	_optimizer_extended_cursor_sharing	1	"NONE"		FALSE	FALSE	FALSE	TRUE
2	FALSE	FALSE	_optimizer_extended_cursor_sharing_rel	1	"NONE"		FALSE	FALSE	FALSE	TRUE
3	FALSE	FALSE	cursor_sharing	1	"EXACT"		FALSE	FALSE	FALSE	TRUE
4	FALSE	FALSE	statistics_level	1	"TYPICAL"		FALSE	FALSE	FALSE	TRUE
5	TRUE	SYSTEM_MOD	optimizer_dynamic_sampling	1	"5"		FALSE	FALSE	FALSE	TRUE

(1) FALSE: Parameter value was specified in the parameter file.
(2) FALSE: Parameter has not been modified after instance startup. MODIFIED: Parameter has been modified with ALTER SESSION. SYSTEM_MOD: F
modified with ALTER SYSTEM.
Go to Top

Figure 8-4. *The value of `optimizer_dynamic_sampling` set to the non-default value of 5*

Figure 8-4 also shows us that the value `optimizer_dynamic_sampling` was changed by an `alter system` statement. We can see this because under the “Is Modified” column we see SYSTEM_MOD. We can also find out what the value is by looking at the 10053 trace of a SQL statement (as we did above) or we can show the value

```
SQL> show parameter optimizer_dynamic_sampling

NAME                                 TYPE        VALUE
------------------------------------ ----------- --------------
optimizer_dynamic_sampling           integer     4
```

Dynamic Sampling and Parallel Statements

The rules for deciding if dynamic sampling should be used for parallel execution plans are slightly different than for serially executed statements. Parallel statements are already expected to be resource intensive so the small overhead in dynamic sampling is worth it to ensure a good execution plan. The logic is that if the value is set to the default (`optimizer_dynamic_sampling=2`) then the 64 block sample size is ignored and the actual sample size is determined by looking at the table sizes and the predicate complexity. If there is a non-default value then the rules are applied as for serially executing statements.

What Dynamic Sampling Value Should I Set?

The general rule of thumb for dynamic sampling is that first of all you should not rely on this feature. Remember dynamic sampling is there to catch potential problems in the optimization process caused by missing statistics.

If your query is not performing as expected, dynamic sampling should not be your first port of call. You should get a SQLTXPLAIN report and look at that first. For parallel statements, especially if you have complex predicates and at least one table with missing statistics, you may well end up using dynamic sampling.

If you see dynamic sampling being used (as shown earlier in Figure 8-2) then you should check to see why it was used and what you can do to avoid using it. If you have statistics on a table and still see dynamic sampling, then one possibility is that you have complex predicates and have not used extended statistics (mentioned in Chapter 3).

If for some reason you want to use dynamic sampling but find its sampling level too low or the expected plan is not produced, you can increase the sampling level by changing the value of `optimizer_dynamic_sampling`, but take care to test these changes on a test system and make small changes to see that the overhead is not too great. Pick a representative SQL and see how it performs with different values. If you have done no testing on this parameter, then keep the value at its default.

If you want to disable this feature completely then set the value to 0.

There are cases where dynamic sampling is the only option left. For example, tables that are populated during the query will not have good statistics (as they are most likely empty during the maintenance window when statistics are gathered). Global temporary tables are a good example where dynamic sampling is a good idea.

If dynamic sampling is your last chance to get your execution plan right, then cardinality feedback is your chance to get the execution plan right the second time around.

Cardinality Feedback

Cardinality feedback is a simple yet elegant way of correcting cardinality. Rather than going to endless complications to determine the right cardinality, we just wait for the result of each step in the execution plan, store it in the shared pool and reference it on subsequent executions, in the hope that the information will give us a good idea of how well we did the last time. This simple technique naturally has its own pitfalls, how do we stop results bouncing from one estimate to another for example? Let's look at some of the details.

How Does Cardinalty Feedback Work?

Cardinality feedback could not work if information about every SQL was not stored in memory, to be accessed by later executions of the same SQL. Let's see what information we can access. We'll run a simple query, then get the Actual and Estimated cardinalities for the execution and then run for two queries and compare estimates and actual rows returned, with both dynamic sampling and cardinality feedback disabled. Then we'll enable cardinality feedback and repeat the experiment. In our first step we check the value of `optimizer_dynamic_sampling` and see that it is set to 0, which means this feature is disabled.

```
SQL> show parameter optimizer_dynamic_sampling
NAME                                 TYPE        VALUE
------------------------------------ ----------- ------------
optimizer_dynamic_sampling           integer     0 <<<DS disabled
SQL> alter system set "_optimizer_use_feedback"=FALSE; <<CFB disabled
System altered.
```

■ **Note** We could also have disabled cardinality feedback with a hint `/*+ opt_param('_optimizer_use_feedback' 'false') */`.

Next we create a test table that is populated from dba_objects. We use the hint /*+ gather_plan_statistics */ to ensure we have good statistics for the execution plan we want to look at.

```
SQL> create table test1 as select (object_id) from dba_objects;
Table created.
SQL> select /*+ gather_plan_statistics */ count(*) from test1;
  COUNT(*)
----------
     73532
```

Now we use the dbms_xplan.display_cursor to get the execution plan and the statistics associated with the execution. This is a pretty nice feature introduced in 11g of Oracle.

```
SQL> select * from table(dbms_xplan.display_cursor(null,null,'ALLSTATS LAST'));
PLAN_TABLE_OUTPUT
SQL_ID  gtukt6kw8yjm6, child number 0
-------------------------------------
select /*+ gather_plan_statistics */ count(*) from test1
Plan hash value: 3896847026
-----------------------------------------------------------------------------------------------
| Id  | Operation          | Name  | Starts | E-Rows | A-Rows |   A-Time   | Buffers | Reads  |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |       |      1 |        |      1 |00:00:00.09 |     116 |    112 |
|   1 |  SORT AGGREGATE    |       |      1 |      1 |      1 |00:00:00.09 |     116 |    112 |
|   2 |   TABLE ACCESS FULL| TEST1 |      1 |   9965 |  73532 |00:00:00.18 |     116 |    112 |
-----------------------------------------------------------------------------------------------
14 rows selected.
```

We see on this line that the estimate (E-Rows) is pretty poor compared to the actual number of rows (A-Rows). This estimate counts as bad enough to merit cardinality feedback use, but in this case we have the feature turned off. So if we run the query a second time we would expect no improvement in E-Rows; and indeed this is what happens.

```
SQL> select /*+ gather_plan_statistics */  count(*) from test1;
  COUNT(*)
----------
     73532
SQL> select * from table(dbms_xplan.display_cursor(null,null,'ALLSTATS LAST'));
PLAN_TABLE_OUTPUT
SQL_ID  gtukt6kw8yjm6, child number 0
-------------------------------------
select /*+ gather_plan_statistics */ count(*) from test1
Plan hash value: 3896847026
-----------------------------------------------------------------------------------------------
| Id  | Operation          | Name  | Starts | E-Rows | A-Rows |   A-Time   | Buffers | Reads  |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |       |      1 |        |      1 |00:00:00.09 |     116 |    112 |
|   1 |  SORT AGGREGATE    |       |      1 |      1 |      1 |00:00:00.09 |     116 |    112 |
|   2 |   TABLE ACCESS FULL| TEST1 |      1 |   9965 |  73532 |00:00:00.18 |     116 |    112 |
-----------------------------------------------------------------------------------------------
14 rows selected.
```

No improvement has occurred in the estimated cardinality (E-Rows) for this plan. The E-Rows value has not changed; the optimizer did not make any changes even though its estimate was so far out. The next step in our experiment is to enable cardinality feedback.

```
SQL> alter system set "_optimizer_use_feedback"=TRUE;
System altered.
```

Now we repeat the entire sequence of steps and we find that for the last step we get

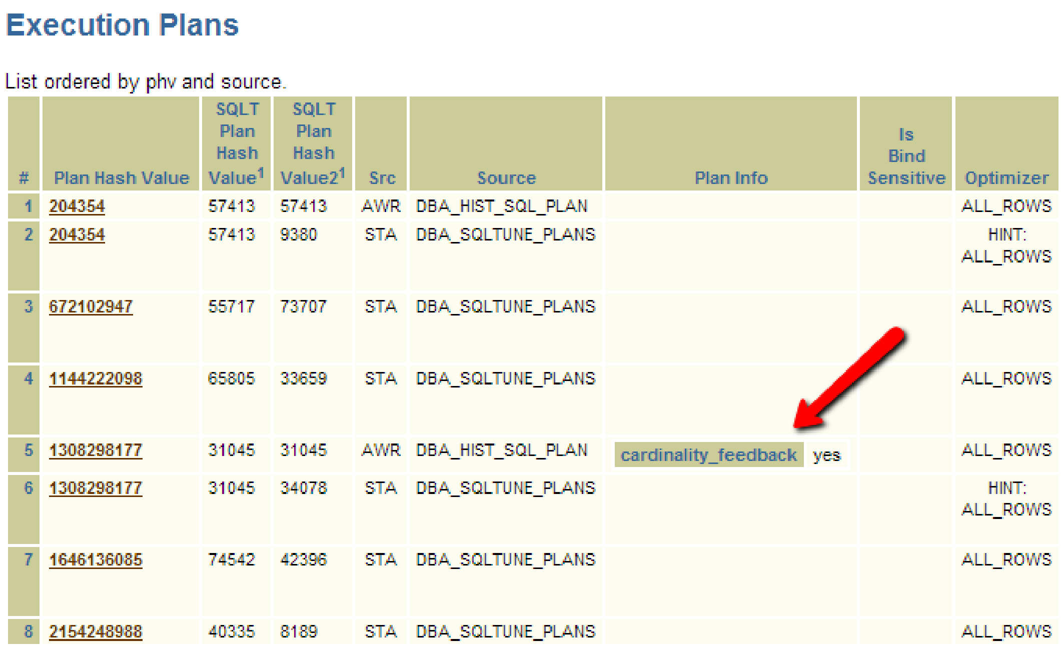
```
SQL> select * from table(dbms_xplan.display_cursor(null,null,'ALLSTATS LAST'));
PLAN_TABLE_OUTPUT
SQL_ID  gtukt6kw8yjm6, child number 3
-------------------------------------
select /*+ gather_plan_statistics */ count(*) from test1
Plan hash value: 3896847026
-------------------------------------------------------------------------------------
| Id  | Operation          | Name  | Starts | E-Rows | A-Rows |   A-Time    | Buffers |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |       |      1 |        |      1 |00:00:00.02 |     116 |
|   1 |  SORT AGGREGATE    |       |      1 |      1 |      1 |00:00:00.02 |     116 |
|   2 |   TABLE ACCESS FULL| TEST1 |      1 |  73532 |  73532 |00:00:00.18 |     116 |
-------------------------------------------------------------------------------------
Note
-----
cardinality feedback used for this statement
```

We see that in the `Note` section the optimizer has left us a message indicating that cardinality feedback was used. In our example above we had an estimated 9,965 rows to begin with and an actual number of 73,532 rows. Without cardinality feedback and dynamic sampling or statistics on the new object our estimate is pretty poor to begin with. Because the number of estimated rows was “significantly” different, cardinality feedback was used. It is not documented what a “significant” difference is, but approximately 8-fold difference is enough.

There are safety features in place to stop the cardinalities bouncing back and forth between estimates, so after a small number of iterations the plan is stabilized. The fact that the actual cardinalities are stored in the SGA also explains why cardinality feedback information is not persistent, that is if the instance is restarted then the cardinality feedback information will be lost, as the information held in memory is not persistent.

How Can You Tell If Cardinality Feedback Is Used?

The simplest way to tell if cardinality feedback has been used is to use the SQLTXPLAIN report. Click on “Execution Plans” (as shown in Figure 8-1) and if cardinality feedback has been used for some of your execution plans you will see “cardinality_feedback yes” under the “Plan Info” column. See figure Figure 8-5 for an example.



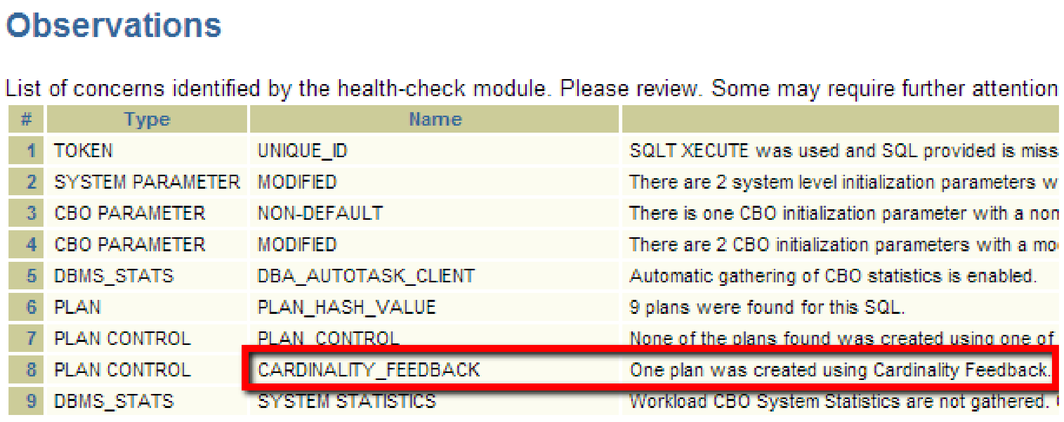
Execution Plans

List ordered by phv and source.

#	Plan Hash Value	SQLT Plan Hash Value[1]	SQLT Plan Hash Value2[1]	Src	Source	Plan Info	Is Bind Sensitive	Optimizer
1	204354	57413	57413	AWR	DBA_HIST_SQL_PLAN			ALL_ROWS
2	204354	57413	9380	STA	DBA_SQLTUNE_PLANS			HINT: ALL_ROWS
3	672102947	55717	73707	STA	DBA_SQLTUNE_PLANS			ALL_ROWS
4	1144222098	65805	33659	STA	DBA_SQLTUNE_PLANS			ALL_ROWS
5	1308298177	31045	31045	AWR	DBA_HIST_SQL_PLAN	cardinality_feedback yes		ALL_ROWS
6	1308298177	31045	34078	STA	DBA_SQLTUNE_PLANS			HINT: ALL_ROWS
7	1646136085	74542	42396	STA	DBA_SQLTUNE_PLANS			ALL_ROWS
8	2154248988	40335	8189	STA	DBA_SQLTUNE_PLANS			ALL_ROWS

Figure 8-5. *"Execution Plans" section shows that cardinality feedback was used for an execution plan*

To emphasize the point that cardinality feedback is a backup mechanism, its use is also highlighted in the "Observations" section of the SQLTXPLAIN report. See section in Figure 8-6, which you can reach by clicking on the "Observations" hyperlink from the top of the main SQLTXPLAIN report.



Observations

List of concerns identified by the health-check module. Please review. Some may require further attention

#	Type	Name	
1	TOKEN	UNIQUE_ID	SQLT XECUTE was used and SQL provided is miss
2	SYSTEM PARAMETER	MODIFIED	There are 2 system level initialization parameters w
3	CBO PARAMETER	NON-DEFAULT	There is one CBO initialization parameter with a non
4	CBO PARAMETER	MODIFIED	There are 2 CBO initialization parameters with a mo
5	DBMS_STATS	DBA_AUTOTASK_CLIENT	Automatic gathering of CBO statistics is enabled.
6	PLAN	PLAN_HASH_VALUE	9 plans were found for this SQL.
7	PLAN CONTROL	PLAN_CONTROL	None of the plans found was created using one of
8	PLAN CONTROL	CARDINALITY_FEEDBACK	One plan was created using Cardinality Feedback.
9	DBMS_STATS	SYSTEM STATISTICS	Workload CBO System Statistics are not gathered.

Figure 8-6. *Cardinality feedback usage is shown as an observation of type PLAN_CONTROL*

When is Cardinality Feedback used?

Lack of statistics or “complex” predicates that create queries with hard-to-determine cardinalities will give cardinality feedback a chance to improve E-Rows. Here is a SQL with a “complex” predicate:

```
SQL> select product_name from order_items ord, product_information pro
  where ord.unit_price= 15 and quantity > 1
  and pro.product_id = ord.product_id;
```

Here we have a filter on `unit_price` (must be equal to 15 and quantity must be > 1). This situation is not very rare so cardinality feedback could be used often. However, remember that we mentioned that the statement needs to execute at least once for the optimizer to store the actual rows so that it can compare them to the estimated number of rows. If dynamic sampling has already been used (because it was needed and it was not disabled), then cardinality feedback will not be used. Also because of the problems that can be introduced by bind variables (especially if you have skewed data), cardinality feedback will not be used for parts of the statement that involve bind variables.

If you find cardinality feedback is not useful for your site or SQL statement you can, with the assistance of support, disable it with

```
SQL> alter system set "_optimizer_use_feedback" = FALSE;
```

If you want to disable an individual statement, then you can put this hint in the SQL

```
/*+ opt_param('_optimizer_use_feedback' 'false') */
```

A select sysdate from dual becomes

```
SQL> select /*+ opt_param('_optimizer_use_feedback' 'false') */ sysdate from dual;
```

Cardinality feedback is not persistent through instance restarts, so it is better to get your statistics from other sources, preferably from `dbms_stats`, but remember that cardinality feedback is enabled by default.

How Do Cardinality Feedback and Dynamic Sampling Work Together?

I’m sure you can imagine that with dynamic sampling and cardinality feedback both working on the same statement, there could be conflicts, and the amount of overhead could be doubled. There are no controls on cardinality feedback (except to disable it), unlike dynamic sampling, which can be set for different criteria and different levels of sample collection. Oracle has thought about these potential conflicts and overheads by having built-in safeties. For example, if dynamic sampling is used, cardinality feedback will not be used. Dynamic sampling can be used more than once for an individual statement, whereas collection of information for an individual SQL statement by cardinality feedback is only allowed to run a limited number of times. The safeties are there to catch problems, but the care has been taken to ensure the safeties are not too much of an overhead.

Now that we’ve got some understanding about these two features let’s see an example mystery case involving both of these features.

The Case of the Identical Twins

This is the kind of situation that occurs frequently in the DBA world: two apparently identical systems, one cloned from the other but with widely different performance in some SQL. Naturally you can suspect different DDL, statistics, or operational procedures, different resource allocation, different workloads, etc. When the hardware is identical and the databases are cloned from each other, the number of choices becomes more limited. In this case we discover by experimentation that one particular SQL is behaving well on system A (say, New York) and the cloned system B (say, London) is behaving badly. The SQL is the same, the systems are the same, the parameter settings are the same. Let me show you the steps you could follow to solve a problem like this using SQLT.

There are many ways you could solve this problem. No doubt you could do it just by collecting 10046 trace files, but we're looking at how you would do this with SQLTXPLAIN. Step by step. First, if we collect SQLT for both SQL statements, one SQLT XECUTE report for New York (as it executes normally) and one SQLT XTRACT report for the evil twin in London (as it takes too long to execute). From the top of SQLTXPLAIN report (see Figure 8-7) we look at the list of execution plans by clicking on "Execution Plans".

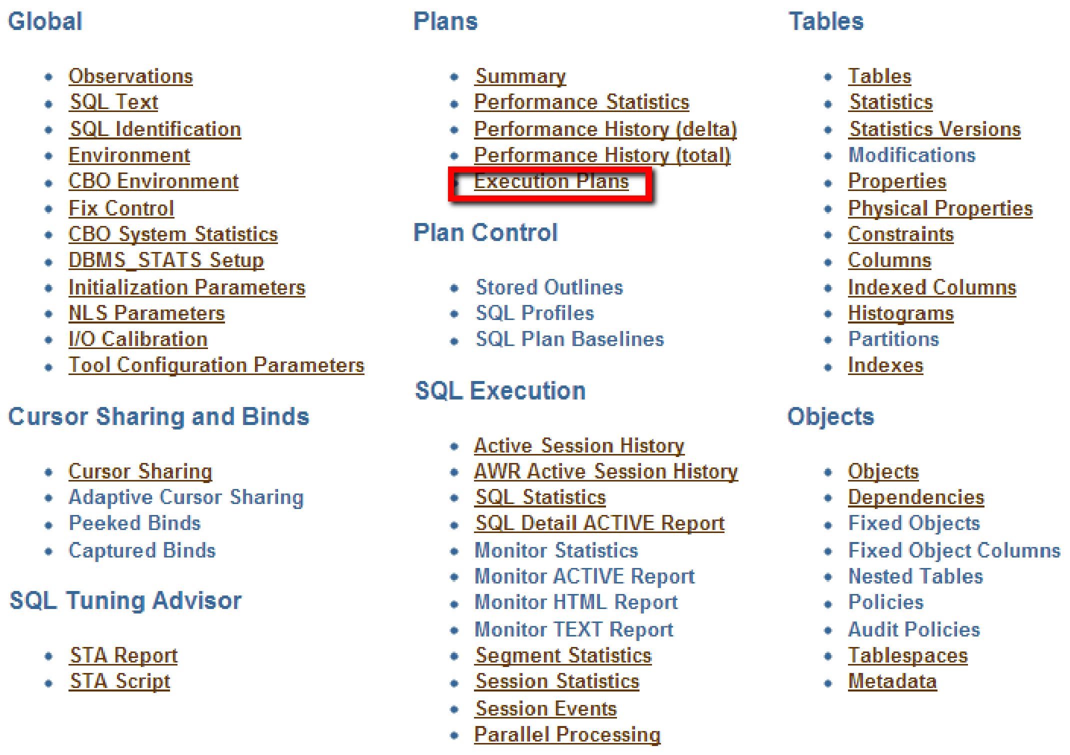
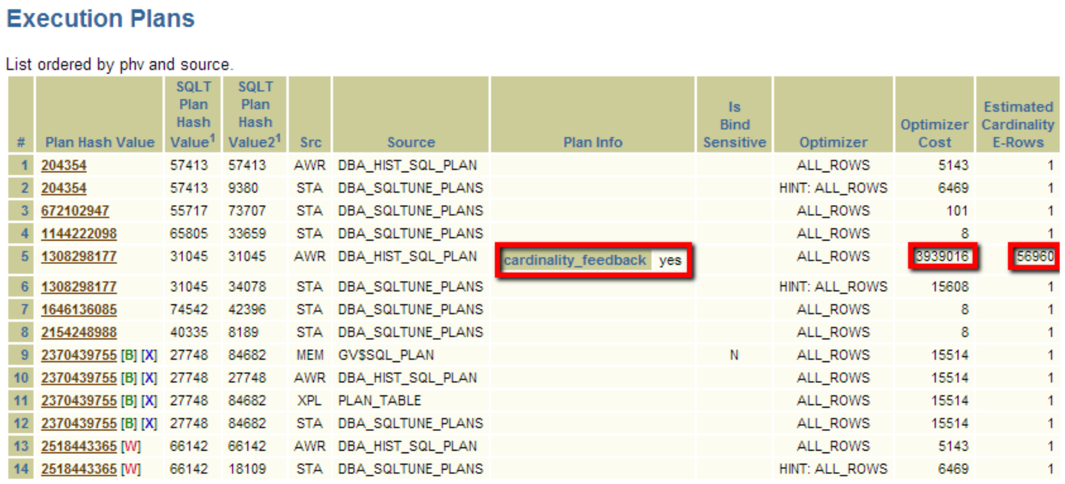


Figure 8-7. *The top of the SQLT report. We click on "Execution Plans"*

This is for the evil system. We see that cardinality feedback is in play, so something must have happened to make this feature kick in. We also see (Figure 8-8) that the estimated cardinality is very different for some of the executions (where cardinality feedback was used). We also see that the optimizer_cost is very high where cardinality feedback was used.



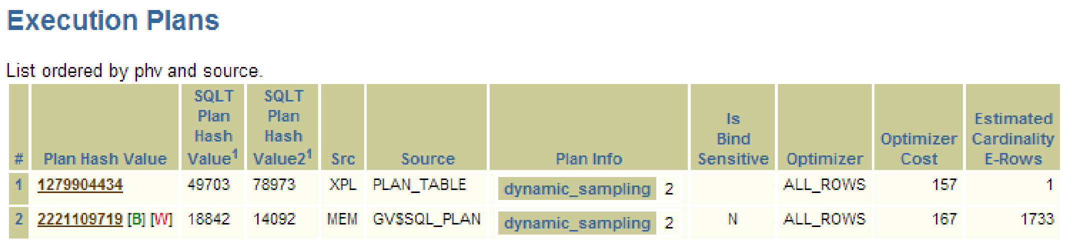
Execution Plans

List ordered by phv and source.

#	Plan Hash Value	SQLT Plan Hash Value[1]	SQLT Plan Hash Value2[1]	Src	Source	Plan Info	Is Bind Sensitive	Optimizer	Optimizer Cost	Estimated Cardinality E-Rows
1	204354	57413	57413	AWR	DBA_HIST_SQL_PLAN			ALL_ROWS	5143	1
2	204354	57413	9380	STA	DBA_SQLTUNE_PLANS			HINT: ALL_ROWS	6469	1
3	672102947	55717	73707	STA	DBA_SQLTUNE_PLANS			ALL_ROWS	101	1
4	1144222098	65805	33659	STA	DBA_SQLTUNE_PLANS			ALL_ROWS	8	1
5	1308298177	31045	31045	AWR	DBA_HIST_SQL_PLAN	cardinality_feedback yes		ALL_ROWS	3939016	56960
6	1308298177	31045	34078	STA	DBA_SQLTUNE_PLANS			HINT: ALL_ROWS	15608	1
7	1646136085	74542	42396	STA	DBA_SQLTUNE_PLANS			ALL_ROWS	8	1
8	2154248988	40335	8189	STA	DBA_SQLTUNE_PLANS			ALL_ROWS	8	1
9	2370439755 [B] [X]	27748	84682	MEM	GV$SQL_PLAN		N	ALL_ROWS	15514	1
10	2370439755 [B] [X]	27748	27748	AWR	DBA_HIST_SQL_PLAN			ALL_ROWS	15514	1
11	2370439755 [B] [X]	27748	84682	XPL	PLAN_TABLE			ALL_ROWS	15514	1
12	2370439755 [B] [X]	27748	84682	STA	DBA_SQLTUNE_PLANS			ALL_ROWS	15514	1
13	2518443365 [W]	66142	66142	AWR	DBA_HIST_SQL_PLAN			ALL_ROWS	5143	1
14	2518443365 [W]	66142	18109	STA	DBA_SQLTUNE_PLANS			HINT: ALL_ROWS	6469	1

Figure 8-8. *The London evil twin shows cardinality feedback being used and high values for estimated cardinality*

Since we are comparing the good and bad systems, we should now look at the execution plans for the good system in New York. Remember these two systems are identical (same hardware, same database versions, similar volumes of data, same tables, and indexes). Here is the same part of the report for the good system (see Figure 8-9)



Execution Plans

List ordered by phv and source.

#	Plan Hash Value	SQLT Plan Hash Value[1]	SQLT Plan Hash Value2[1]	Src	Source	Plan Info	Is Bind Sensitive	Optimizer	Optimizer Cost	Estimated Cardinality E-Rows
1	1279904434	49703	78973	XPL	PLAN_TABLE	dynamic_sampling 2		ALL_ROWS	157	1
2	2221109719 [B] [W]	18842	14092	MEM	GV$SQL_PLAN	dynamic_sampling 2	N	ALL_ROWS	167	1733

Figure 8-9. *The New York good twin shows dynamic sampling and smaller values for estimated cardinality*

We also see that there are many more different plans in London than there are in New York. We know that dynamic sampling and cardinality feedback are sometimes used when poor statistics are involved, so it is a reasonable route of inquiry to look at the statistics of the main objects in the query. We should be thinking throughout our investigation of the information presented by SQLT: "Why was dynamic sampling used and why was cardinality feedback used and why are they different?" From the top of the report we click on "Statistics". We see for the good system (Figure 8-10) that we have a "Y" under the "Temp" column. Also we have no values under "Num Rows" or "Sample Size" or "Perc". TABLE4 is of interest because as we'll see the same part of the report for the bad system is very different in this respect.

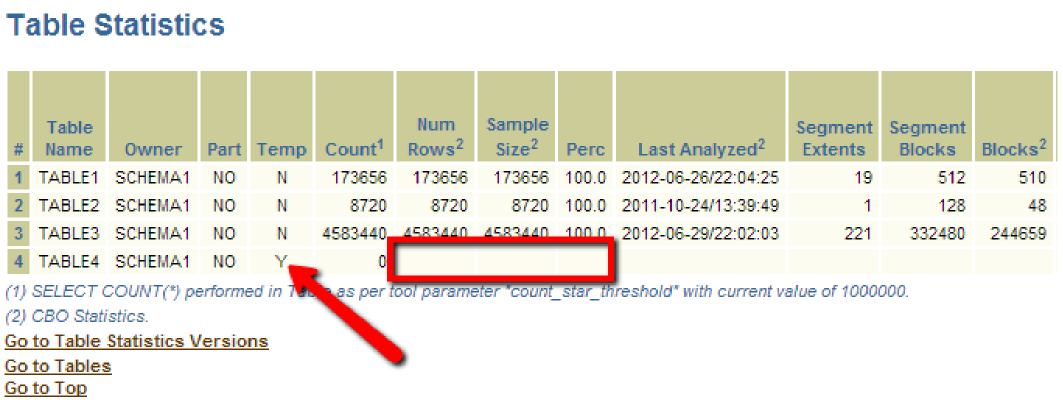


Table Statistics

#	Table Name	Owner	Part	Temp	Count[1]	Num Rows[2]	Sample Size[2]	Perc	Last Analyzed[2]	Segment Extents	Segment Blocks	Blocks[2]
1	TABLE1	SCHEMA1	NO	N	173656	173656	173656	100.0	2012-06-26/22:04:25	19	512	510
2	TABLE2	SCHEMA1	NO	N	8720	8720	8720	100.0	2011-10-24/13:39:49	1	128	48
3	TABLE3	SCHEMA1	NO	N	4583440	4583440	4583440	100.0	2012-06-29/22:02:03	221	332480	244659
4	TABLE4	SCHEMA1	NO	Y	0							

(1) SELECT COUNT(*) performed in Table as per tool parameter "count_star_threshold" with current value of 1000000.
(2) CBO Statistics.
Go to Table Statistics Versions
Go to Tables
Go to Top

Figure 8-10. *The table statistics for the good system*

Before we leap to conclusions (and this is always to be avoided in investigating performance issues), we need to look at the bad system and do a comparison (see Figure 8-11). Our plan is to figure out what could be different between these two systems to cause one to use dynamic sampling and the other to use cardinality feedback.

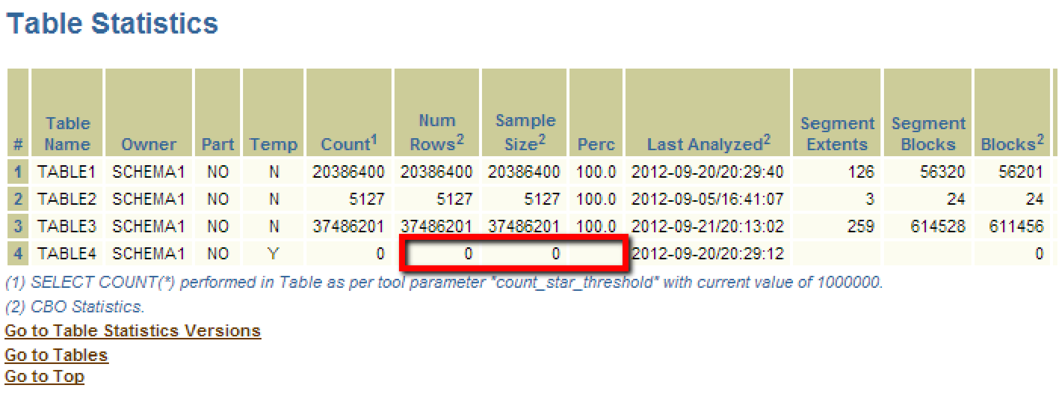


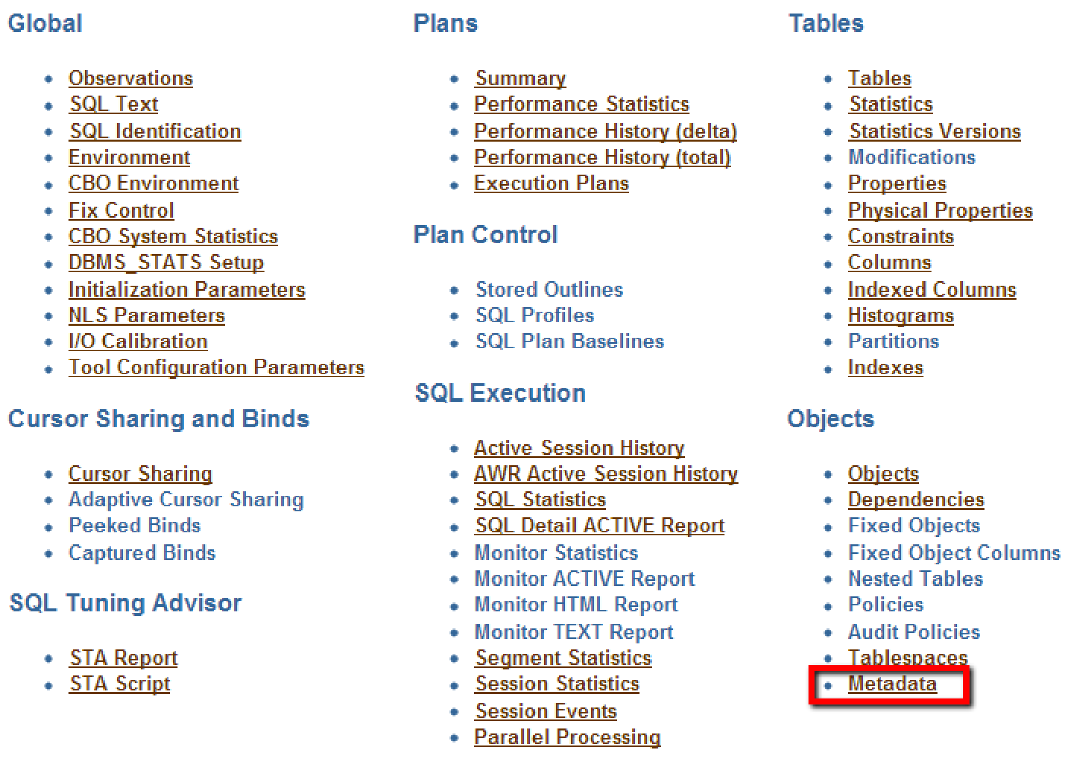
Table Statistics

#	Table Name	Owner	Part	Temp	Count[1]	Num Rows[2]	Sample Size[2]	Perc	Last Analyzed[2]	Segment Extents	Segment Blocks	Blocks[2]
1	TABLE1	SCHEMA1	NO	N	20386400	20386400	20386400	100.0	2012-09-20/20:29:40	126	56320	56201
2	TABLE2	SCHEMA1	NO	N	5127	5127	5127	100.0	2012-09-05/16:41:07	3	24	24
3	TABLE3	SCHEMA1	NO	N	37486201	37486201	37486201	100.0	2012-09-21/20:13:02	259	614528	611456
4	TABLE4	SCHEMA1	NO	Y	0	0	0		2012-09-20/20:29:12			0

(1) SELECT COUNT(*) performed in Table as per tool parameter "count_star_threshold" with current value of 1000000.
(2) CBO Statistics.
Go to Table Statistics Versions
Go to Tables
Go to Top

Figure 8-11. *The table statistics for the bad system*

Now that we see both together we see something very interesting. Both databases show that TABLE4 is a temporary table and that the statistics gathering is different for these two systems for this one table. We also see that the data volume for TABLE1 is different, but TABLE4 seems much more interesting in terms of difference, at least for now because it has a count of 0 and row count of 0 also. We'll keep the TABLE1 idea as a backup. So let's check on the metadata for TABLE4 and see how it was defined. (We can get this easily enough by clicking on the "Metadata" from the top of the report. See Figure 8-12.



Global

- Observations
- SQL Text
- SQL Identification
- Environment
- CBO Environment
- Fix Control
- CBO System Statistics
- DBMS_STATS Setup
- Initialization Parameters
- NLS Parameters
- I/O Calibration
- Tool Configuration Parameters

Cursor Sharing and Binds

- Cursor Sharing
- Adaptive Cursor Sharing
- Peeked Binds
- Captured Binds

SQL Tuning Advisor

- STA Report
- STA Script

Plans

- Summary
- Performance Statistics
- Performance History (delta)
- Performance History (total)
- Execution Plans

Plan Control

- Stored Outlines
- SQL Profiles
- SQL Plan Baselines

SQL Execution

- Active Session History
- AWR Active Session History
- SQL Statistics
- SQL Detail ACTIVE Report
- Monitor Statistics
- Monitor ACTIVE Report
- Monitor HTML Report
- Monitor TEXT Report
- Segment Statistics
- Session Statistics
- Session Events
- Parallel Processing

Tables

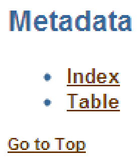
- Tables
- Statistics
- Statistics Versions
- Modifications
- Properties
- Physical Properties
- Constraints
- Columns
- Indexed Columns
- Histograms
- Partitions
- Indexes

Objects

- Objects
- Dependencies
- Fixed Objects
- Fixed Object Columns
- Nested Tables
- Policies
- Audit Policies
- Tablespaces
- Metadata

Figure 8-12. *From the top of the SQLTXECUTE report we can navigate to the metadata for all objects for this query*

Then from the part of the report labeled "Metadata" we can select the table metadata by clicking on "Table" hyperlink, which is shown in the Figure 8-13. Notice how we also have a link to index metadata if we wanted to investigate that.



Metadata

- Index
- Table

Go to Top

Figure 8-13. *The list of object types for which we have metadata, in this case tables and indexes*

The table metadata part of the SQLTXPLAIN report show links to all the tables, which are in the query for which the report was created. See in Figure 8-14. In this case we have four tables. We are interested in the fourth table in this case, so we click on the "TABLE4" hyperlink.

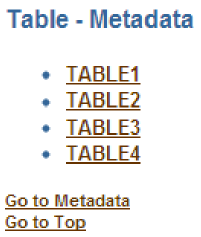


Table - Metadata

- TABLE1
- TABLE2
- TABLE3
- TABLE4

Go to Metadata
Go to Top

Figure 8-14. *The list of table objects for which we have metadata*

This gets us to the part of the report which shows the metadata for TABLE4. We see that is is global temporary table with the clause `ON COMMIT PRESERVE ROWS`. This is the DDL we see:

```
CREATE GLOBAL TEMPORARY TABLE "SCHEMA1"."TABLE4"
 (   "COLUMN1" NUMBER(10,0),
     "COLUMN2" NUMBER(10,0),
     "COLUMN3" NUMBER(10,0)
 ) ON COMMIT PRESERVE ROWS
```

There's nothing remarkable about this table. This is a default creation of a global temporary table that can be used by more than one session during SQL processing. This is often included in application designs if the developer wants to keep some temporary data in a table for processing in later steps in the application. The `on commit preserve` part of the DDL (metadata) ensures that data committed to the table is preserved. In this kind of table you would normally expect the table to be cleaned out at the end of processing, or sometimes at the beginning of processing. The key thing to note here is that on the bad system, statistics were collected for this table (see Figure 8-11). There is a last analyzed date for TABLE4 but not on the good system. If we look at the statistics for TABLE4 where they were collected we see that the table was empty. These statistics would then prevent dynamic sampling from being activated (as there are "good" statistics for TABLE4), but would not prevent cardinality feedback because data loaded during processing would make the cardinality estimates wrong. This sounds like a working theory. Somehow TABLE4 has had statistics collected on it on the bad system but not on the good system. On the good system the table was not analyzed. This would have allowed dynamic sampling to take an estimate of the statistics at run time and determine a good execution plan. With a working theory we can now build a test case (described in Chapter 13) and attempt to get the good execution plan from the bad test case by deleting the statistics for the global temporary table.

Summary

Dynamic sampling and cardinality feedback are useful features, for those rare occasions when statistics are missing. There is no substitute for good statistics, however. With the interplay of complex features situations can be created that show strange behavior. Even seemingly identical systems can behave very differently if key components are changed, sometimes unwittingly. SQLTXPLAIN, because it gathers everything, is the quickest and easiest way to solve most SQL tuning mysteries. In the next chapter we'll take a closer look at the special Data Guard physical standby environment and how SQLT can help.

CHAPTER 9

Using SQLTXPLAIN with Data Guard Physical Standby Databases

SQLTXPLAIN is a great tool designed to help with SQL tuning problems, but its effectiveness is limited when the database is a read-only Data Guard physical standby database. Read only databases like a Data Guard physical standby databases cannot be written to by utilities such as SQLTXPLAIN. In this chapter we explore the special tools created just to deal with Data Guard.

Data Guard Physical Standby Database

Data Guard is a piece of technology, developed by Oracle in response to a “what if” scenario. What if my data center is completely wiped out by flooding? What if a fire destroys the building my database is housed in? Traditionally this has been answered by timely backups, shipped off site and stored in a secure location that can then be accessed within an agreed time scale and restored to backup hardware made available at another site (presumably a site not in the disaster zone). This strategy had many difficulties: taking the backup, shipping the backup to a safe location, getting the right backup from the secure location, recreating your systems with the use of the backup and your restore procedures, and finally getting access to those systems for the staff required to use them. Finally and most importantly, you need to test these procedures on a regular basis, otherwise come the day of the disaster you may have all the data and equipment, but you’ll have no idea how to put it all together to make a working system. Of course, this restore procedure can take a considerable amount of time. You have tested your recovery procedure haven’t you? One site I worked with took this so seriously that on the day of the test, they would go around and put red stickers on people and equipment and tell the people they were not available for this test (presumably they had been abducted by the aliens in the “Aliens abduct your data center” scenario). It was hard to test these scenarios; and not surprisingly, many sites did not do adequate testing and crossed their IT fingers and hoped for the best.

Data Guard makes preserving your data site much simpler. All of the aforementioned complications can potentially be eliminated. The technology that makes Data Guard physical standby work is in concept very simple. The archive logs (or redo logs), that track every change in your database are transferred by special processes on your source system to special processes on the standby system, where the changes are applied, block by block to a copy of your source database. This process called propagation can apply data on the standby database. The data can be applied in lock step with the primary database or it can lag behind the primary database by a preset amount. You can even have multiple standby databases all collecting changes from the primary database. With Data Guard active standby, you can even allow the failover (the process whereby the primary fails and the standby has to take over) to happen with the minimum of fuss.

Data Guard is a huge leap forward in disaster-recovery technology. Complete coverage goes well beyond the scope of this book; but a basic, high-level understanding of the tool is useful.

■ **Note** I could not hope to cover any more than the briefest details of Data Guard here. It is a book- length topic by itself. The latest book is entitled *Oracle Data Guard 11g Handbook* by Larry Carpenter, et. al. (Oracle Press 2009).

Suffice it to say that Oracle Data Guard Physical Standby allows the data center to be online all of the time. There is no down time. Naturally you still need backups for those pesky systematic errors (you deleted the wrong data). Data Guard makes your testing scenario much simpler. Now you need only carry out a switchover (a controlled switch of the computer roles as opposed to the failover), and once the switchover is complete you can start your testing immediately.

In 11g release 1, Oracle changed Data Guard so that not only was the physical standby ready for a failover or switchover operation, it was also available for read-only operations. Typically this would be to allow reports to run on the standby database. Many sites found that this was a massive boon to their operations. Now they could move their expensive reporting operations (in terms of resource usage) to the standby database and free up the primary for On-Line Transaction Processing (OLTP).

SQLTXTRSBY

Once reporting operations moved to the physical standby (and logical standby) we could once more have poorly performing SQL. Performance problems that are on a read/write database are the sort of thing that SQLT can help us with. The only problem being that SQLTXTRACT and SQLTXECUTE need read/write access to the database (to store data in the SQLT repository and to install packages and procedures). How is it possible for SQLT to help us if we can't even store data about the performance on the database with the performance problem? This is where SQLTXTRSBY comes into play.

It provides some special procedures that deal with these special circumstances. As Data Guard (and other read-only databases) became more popular more of the performance problems mentioned above appeared on Data Guard instances. To deal with these problems SQLTXTRACT was adapted to work without any local storage. This was done by making special routines that ran from a read/write database and reached out across a database link to the read-only database, collected the required information and collated it and presented it on the read-write database. Let's take a more detailed look at some of SQLTXTRACT's limitations and then discuss using SQLTXTRSBY.

SQLTXTRACT Limitations

I always think of SQLTXTRACT (alias XTRACT) and SQLTXTRSBY (alias XTRSBY) as a superhero and his (or her) sidekick. XTRACT the superhero can seemingly do it all, but every superhero has their Achilles heel, in this case it is the inability to work on a read-only database. XTRSBY has those special skills that the main hero does not have. In this case XTRSBY can go into a read-only Data Guard system and get out with the information, where XTRACT cannot. Physical standbys cannot tolerate write operations. There cannot be two systems potentially updating the same data (unless you have multi-master replication of course, but that's a whole other book), not unless there was some way for the primary to know what the failover system was doing, and Data Guard does not allow for that. This is why the physical standby database is open in read-only mode. If you do happen to wander onto a read-only database and try and insert some data you'll get an error message like this:

```
SQL> insert into test1 values (1);
insert into test1 values (1)
            *
ERROR at line 1:
ORA-00604: error occurred at recursive SQL level 1
ORA-16000: database open for read-only access
```

What we see in the above example is that inserting data into a read-only database is not possible. The error message is even somewhat descriptive of what happened. XTRACT needs to keep some data in the database to do its analysis, so this step (saving the data in the local database) will not work for XTRACT. XTRSBY solves this problem by pulling the required data across a database link to a read/write database and stores it locally rather than on the read-only database.

XTRACT also relies on creating a user so that it can store its objects safely. XTRACT cannot do that on a read-only database.

```
SQL> connect / as sysdba
Connected.
SQL> create user sqltxplain_read_only identified by orcle;
create user sqltxplain_read_only identified by orcle
            *
ERROR at line 1:
ORA-00604: error occurred at recursive SQL level 1
ORA-16000: database open for read-only access
```

Even a simple procedure to collect trace cannot be done:

```
SQL> CREATE OR REPLACE PROCEDURE set_trace
       /* grant alter session to <user> */
       /* create this procedure on both sites */
       as
         c1  integer;
         r1 integer;
         c2  integer;
         r2 integer;
         BEGIN
           c1:=dbms_sql.open_cursor;
          dbms_sql.parse(c1,
            'alter session set events
            ''10046 trace name context forever, level 8''',dbms_sql.v7);
            r1:=dbms_sql.execute(c1);
            dbms_sql.close_cursor(c1);
            c2:=dbms_sql.open_cursor;
            dbms_sql.parse(c2,
              'alter session set events
              ''10053 trace name context forever, level 1''', dbms_sql.v7);
            r2:=dbms_sql.execute(c2);
            dbms_sql.close_cursor(c2);
          END;
    /
CREATE OR REPLACE PROCEDURE set_trace
*
ERROR at line 1:
ORA-00604: error occurred at recursive SQL level 1
ORA-16000: database open for read-only access
```

Clearly this is not going to be allowed. So if we can't create the SQLTXPLAIN schema, or any procedures, how do we use XTRACT against this database? XTRSBY solves this problem by using local users (on a read/write database) and creating procedures that use database links to the read-only database.

How Do We Use XTRSBY?

We've just seen some of the things that cannot happen on a Data Guard Physical standby database. This is when XTRSBY can step forward and do most of what XTRACT can do. I say "most" because XTRSBY is not exactly the same as XTRACT in terms of the results (we'll see the differences in the "What does an XTRSBY report look like?") There's enough in the report, however, to help tune queries on the Data Guard instance. So how do we get XTRSBY to work on a Data Guard database? The brief steps in getting XTRSBY to work on a Data Guard standby database are as follows:

1. Install SQLTXPLAIN on the primary (we already covered this in Chapter 1 of this book).
2. Allow the DDL to be propagated to the standby database.
3. Create a database link accessible to the SQLTXPLAIN schema linking to the standby database. I'll give an example below.
4. Run XTRSBY from the primary specifying the SQL ID of the SQL on the standby and the database link.
5. The report is produced on the primary.

Step 1, above is covered in Chapter 1. Step 2 is a matter of waiting for the schema to be propagated (or copied as part of the normal Data Guard operations) to the standby. Step 3 is merely creating a public database link (or link available from SQLTXPLAIN): These are the commands I typed on my read-write database to create a publicly available database link that connects to the read-only database.

```
SQL> show user
USER is "SYS"
SQL> create public database link "TO_STANDBY" connect to sqltxplain identified by oracle using
'(DESCRIPTION=(ADDRESS=(PROTOCOL=TCP)(HOST=localhost)(Port=1521))(connect_data=(SID=SNC2)))';

Database link created.

SQL> select sysdate from dual@to_standby;

SYSDATE
---------
28-OCT-12
```

Here I have simulated the standby database with a read-only database, hence in my connect string for the public database link I have used `localhost`. In a real world example you would use the hostname of the standby database. Also in my example I have used a `PUBLIC` database link, but you may have standards on your site for creating database links, which require you to create a private link or a two-part link where the link information and the password are specified separately. As long as SQLTXPLAIN has access to the link to find the standby database the routine `xtrsby.sql` should work. Once these pre-requisites are in place we can use XTRSBY to collect information about a standby SQL, by running `sqlxtrsby` on the primary.

First let's create some SQL on the standby. This would be equivalent to the reports that are run on the standby database.

```
SQL> select count(1) from dba_objects where object_type ='DIMENSION';

  COUNT(1)
----------
         5
```

```
SQL> select sql_id from v$sql where sql_text like 'select count(1) from dba_objects where
object_type =''DIMENSION''';

SQL_ID
-------------
gaz0dgkpffqrr

SQL>
```

In the steps above we ran some arbitrary SQL and got the SQL ID for that SQL. Remember we ran the SQL on the standby database (where our reports might have run). We can't store any data on the Data Guard Physical Standby database so now we have to capture information about the SQL from across the database link from the primary database:

```
SQL> @sqltxtrsby gaz0dgkpffqrr to_standby

PL/SQL procedure successfully completed.

 Parameter 1:
SQL_ID or HASH_VALUE of the SQL to be extracted (required)

Parameter 2:
DBLINK to SQLTXPLAIN in the stand-by database (required)

Paremeter 3:
SQLTXPLAIN password (required)

Enter value for 3:

At this point enter the password for SQLTXPLAIN

... please wait ...
```

This is the output, shortened for brevity:

```
Archive:  sqlt_s96365_xtrsby_gaz0dgkpffqrr.zip
  Length      Date    Time    Name
---------  ---------- -----   ----
     3060  10/30/2012 13:00   sqlt_s96365_driver.zip
     1159  10/30/2012 13:00   sqlt_s96365_export_driver.sql
    46903  10/30/2012 13:00   sqlt_s96365_lite.html
    12448  10/30/2012 13:00   sqlt_s96365_log.zip
  1788199  10/30/2012 13:00   sqlt_s96365_main.html
    15498  10/30/2012 13:00   sqlt_s96365_readme.html
      222  10/30/2012 13:00   sqlt_s96365_remote_driver.sql
   217088  10/30/2012 13:00   sqlt_s96365_tc.zip
      193  10/30/2012 13:00   sqlt_s96365_tcb_driver.sql
      172  10/30/2012 13:00   sqlt_s96365_tc_script.sql
       65  10/30/2012 13:00   sqlt_s96365_tc_sql.sql
  3183654  10/30/2012 13:00   sqlt_s96365_trc.zip
---------                     -------
  5268661                     12 files
```

```
File sqlt_s96365_xtrsby_gaz0dgkpffqrr.zip for gaz0dgkpffqrr has been created.

SQLTXTRSBY completed.
```

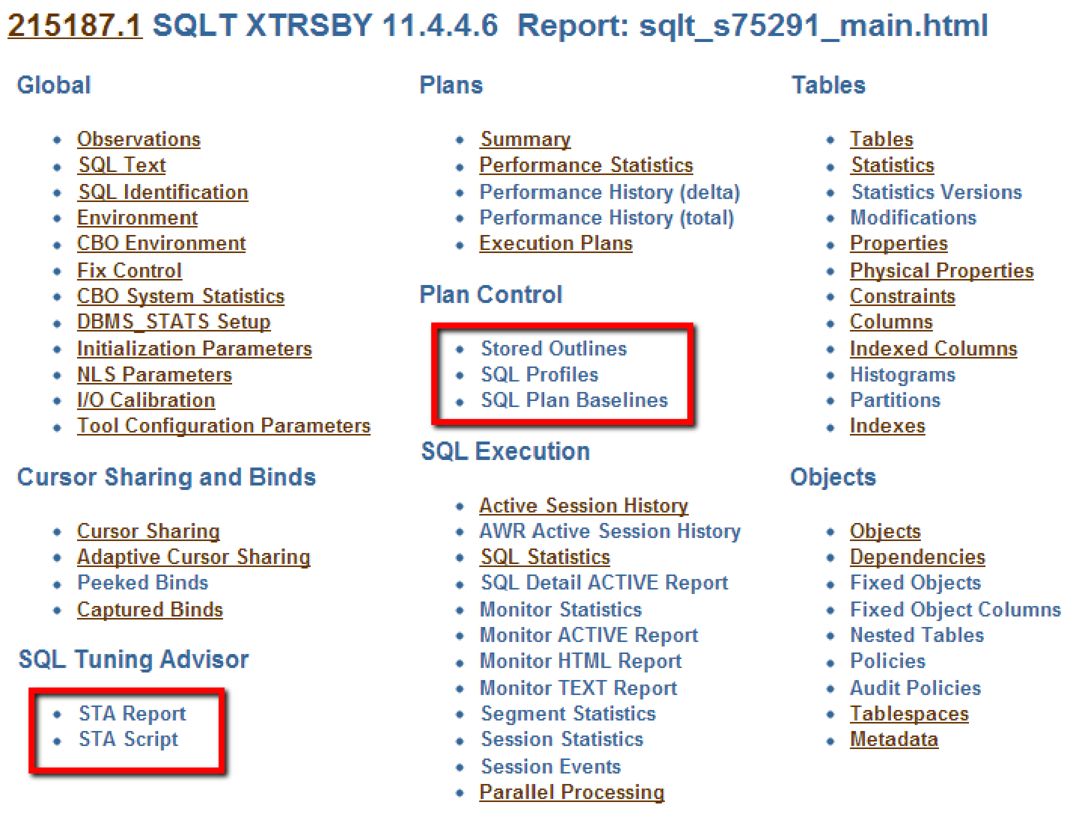
This produces the zip file in the directory where the SQLTXPLAIN routine was run. If we expand the zip file, we'll see the following:

```
10/31/2012  12:03 PM    <DIR>          .
10/31/2012  12:03 PM    <DIR>          ..
10/30/2012  01:00 PM             3,060 sqlt_s96365_driver.zip
10/30/2012  01:00 PM             1,159 sqlt_s96365_export_driver.sql
10/30/2012  01:00 PM            46,903 sqlt_s96365_lite.html
10/30/2012  01:00 PM            12,448 sqlt_s96365_log.zip
10/30/2012  01:00 PM         1,788,199 sqlt_s96365_main.html
10/30/2012  01:00 PM            15,498 sqlt_s96365_readme.html
10/30/2012  01:00 PM               222 sqlt_s96365_remote_driver.sql
10/30/2012  01:00 PM           217,088 sqlt_s96365_tc.zip
10/30/2012  01:00 PM               193 sqlt_s96365_tcb_driver.sql
10/30/2012  01:00 PM               172 sqlt_s96365_tc_script.sql
10/30/2012  01:00 PM                65 sqlt_s96365_tc_sql.sql
10/30/2012  01:00 PM         3,183,654 sqlt_s96365_trc.zip
              12 File(s)      5,268,661 bytes
               2 Dir(s)   7,211,745,280 bytes free
```

We'll see the main `sqlt_s96365_main.html` file, but fewer files than for a "normal" `sqltxtract` run: no 10053 trace file, no sql profile script and no SQL Tuning Advisor reports. This is because the read-only status of the standby restricts what can be done. Production of the XTRSBY report, as you can see, is only a little more complicated to produce. The additional steps already described are that one additional link is required, and the link name needs to be passed in the call to the routine. These are all that is required to get XTRSBY working.

What Does a XTRSBY Report Look Like?

Now that you have a report, what can we do with it? Let's look at the top of another example report. See Figure 9-1.



215187.1 SQLT XTRSBY 11.4.4.6 Report: sqlt_s75291_main.html

Global

- Observations
- SQL Text
- SQL Identification
- Environment
- CBO Environment
- Fix Control
- CBO System Statistics
- DBMS_STATS Setup
- Initialization Parameters
- NLS Parameters
- I/O Calibration
- Tool Configuration Parameters

Cursor Sharing and Binds

- Cursor Sharing
- Adaptive Cursor Sharing
- Peeked Binds
- Captured Binds

SQL Tuning Advisor

- STA Report
- STA Script

Plans

- Summary
- Performance Statistics
- Performance History (delta)
- Performance History (total)
- Execution Plans

Plan Control

- Stored Outlines
- SQL Profiles
- SQL Plan Baselines

SQL Execution

- Active Session History
- AWR Active Session History
- SQL Statistics
- SQL Detail ACTIVE Report
- Monitor Statistics
- Monitor ACTIVE Report
- Monitor HTML Report
- Monitor TEXT Report
- Segment Statistics
- Session Statistics
- Session Events
- Parallel Processing

Tables

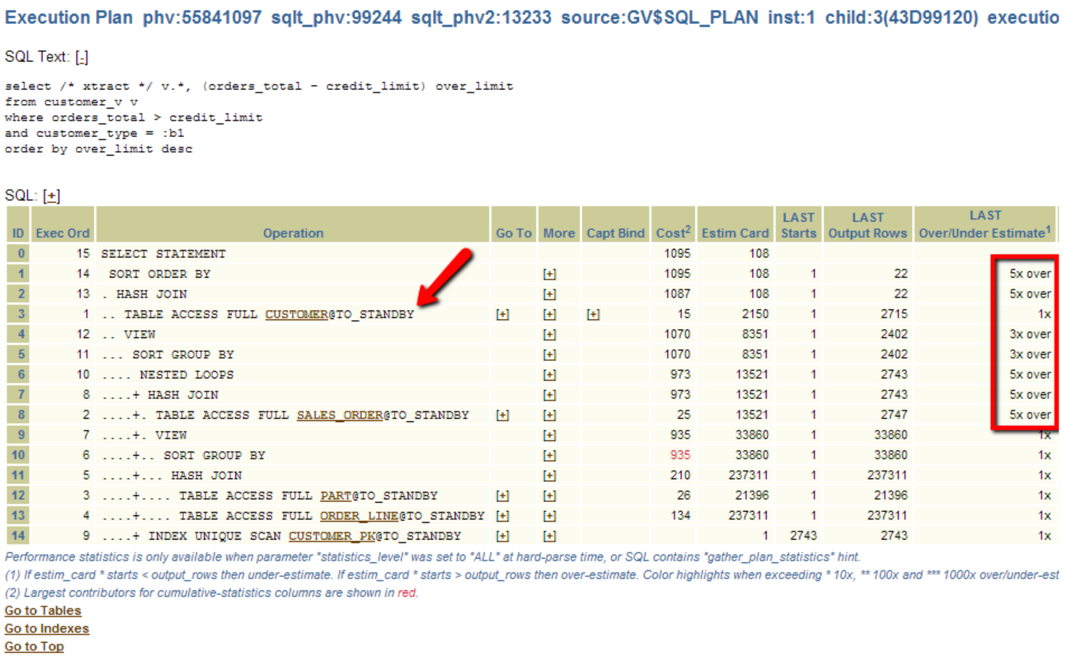
- Tables
- Statistics
- Statistics Versions
- Modifications
- Properties
- Physical Properties
- Constraints
- Columns
- Indexed Columns
- Histograms
- Partitions
- Indexes

Objects

- Objects
- Dependencies
- Fixed Objects
- Fixed Object Columns
- Nested Tables
- Policies
- Audit Policies
- Tablespaces
- Metadata

Figure 9-1. *The top of a SQLT XTRSBY report. Some features are not available*

Some features are not available. For example, in Figure 9-1 I've highlighted two sections that do not have links, as they are not available in XTRSBY. This is because routines on the standby cannot be run (we saw earlier in the chapter what happens). What about the execution plans? They are visible as usual. You'll see in Figure 9-2 that there is no real difference in the information collected.



Execution Plan phv:55841097 sqlt_phv:99244 sqlt_phv2:13233 source:GV$SQL_PLAN inst:1 child:3(43D99120) executio

SQL Text: [-]

```
select /* xtract */ v.*, (orders_total - credit_limit) over_limit
from customer_v v
where orders_total > credit_limit
and customer_type = :b1
order by over_limit desc
```

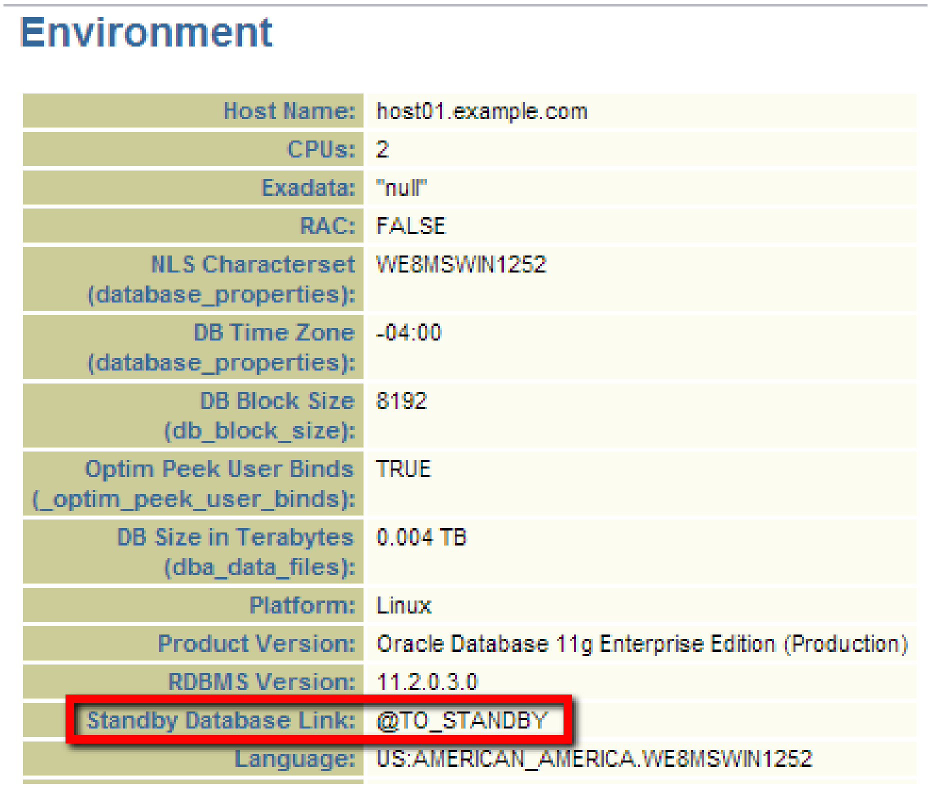
SQL: [+]

ID	Exec Ord	Operation	Go To	More	Capt Bind	Cost[2]	Estim Card	LAST Starts	LAST Output Rows	LAST Over/Under Estimate[1]
0	15	SELECT STATEMENT				1095	108			
1	14	SORT ORDER BY		[+]		1095	108	1	22	5x over
2	13	. HASH JOIN		[+]		1087	108	1	22	5x over
3	1	.. TABLE ACCESS FULL CUSTOMER@TO_STANDBY	[+]	[+]	[+]	15	2150	1	2715	1x
4	12	.. VIEW		[+]		1070	8351	1	2402	3x over
5	11	... SORT GROUP BY		[+]		1070	8351	1	2402	3x over
6	10	.... NESTED LOOPS		[+]		973	13521	1	2743	5x over
7	8	....+ HASH JOIN		[+]		973	13521	1	2743	5x over
8	2	....+. TABLE ACCESS FULL SALES_ORDER@TO_STANDBY	[+]	[+]		25	13521	1	2747	5x over
9	7	....+. VIEW		[+]		935	33860	1	33860	1x
10	6	....+.. SORT GROUP BY		[+]		935	33860	1	33860	1x
11	5	....+... HASH JOIN		[+]		210	237311	1	237311	1x
12	3	....+.... TABLE ACCESS FULL PART@TO_STANDBY	[+]	[+]		26	21396	1	21396	1x
13	4	....+.... TABLE ACCESS FULL ORDER_LINE@TO_STANDBY	[+]	[+]		134	237311	1	237311	1x
14	9	....+ INDEX UNIQUE SCAN CUSTOMER_PK@TO_STANDBY	[+]	[+]			1	2743	2743	1x

Performance statistics is only available when parameter "statistics_level" was set to "ALL" at hard-parse time, or SQL contains "gather_plan_statistics" hint.
*(1) If estim_card * starts < output_rows then under-estimate. If estim_card * starts > output_rows then over-estimate. Color highlights when exceeding * 10x, ** 100x and *** 1000x over/under-est*
(2) Largest contributors for cumulative-statistics columns are shown in red.
Go to Tables
Go to Indexes
Go to Top

Figure 9-2. *An execution plan collected by XTRSBY shows the links used*

In Figure 9-2 we see that all the usual features described in Chapter 2 and 3 are present and can be interpreted in the same way. In this new example (created on a different machine) the link name, still called TO_STANDBY, is shown in the execution plan. We also see that under the “Environment” heading (which we can reach by clicking on “Environment” from the top of the report), we see that the standby database link is shown. See Figure 9-3.



Environment

Host Name:	host01.example.com
CPUs:	2
Exadata:	"null"
RAC:	FALSE
NLS Characterset (database_properties):	WE8MSWIN1252
DB Time Zone (database_properties):	-04:00
DB Block Size (db_block_size):	8192
Optim Peek User Binds (_optim_peek_user_binds):	TRUE
DB Size in Terabytes (dba_data_files):	0.004 TB
Platform:	Linux
Product Version:	Oracle Database 11g Enterprise Edition (Production)
RDBMS Version:	11.2.0.3.0
Standby Database Link:	@TO_STANDBY
Language:	US:AMERICAN_AMERICA.WE8MSWIN1252

Figure 9-3. *The standby database link is shown in the Environment section*

Apart from those few minor differences the SQLTXPLAIN report is the same. Even though XTRSBY is very useful in its limited way, we still need to collect some information on the standby system itself. In earlier versions of SQLT (for example, 11.4.4.6), the read-only tool `roxtract.sql` would be used. This tool is now replaced with SQLHC chapter 14). The read-only (ro) tool is still useful, and if you have an older version of SQLT you may want to use this tool anyway. The tool is named with an RO prefix to indicate it is a read-only tool and does most of what the main XTRACT tool does. It is however not available from version 11.4.5.1 of SQLT onward.

The roxtract Tool

The `utl` area of the SQLT directory has many useful utilities in it. One of these is `roxtract.sql`. This read-only routine is specially designed for running in a read-only database and so is ideal for a Data Guard Physical Standby Database open in read-only mode. I'm sure you're already asking how this routine is different from the `xtrsby.sql` routine we've just been looking at. The simple answer is that running remotely (like `xtrsby.sql`) across a database link doesn't get us everything. Some things are better done locally, and we'll see shortly what `roxtract.sql` produces. This routine runs directly on the read-only Data Guard standby database (with a suitably privileged account) and produces the reports locally unlike `xtrsby.sql`, which produces its reports on the read-write database. To get at this extra information we have to use the routines provided in the `utl` directory. Here's the directory listing on a windows system.

```
C:\Documents and Settings\Stelios\Desktop\SQLT\sqlt\utl>dir
 Volume in drive C has no label.
 Volume Serial Number is 77E9-80B4

    Directory of C:\Documents and Settings\Stelios\Desktop\SQLT\sqlt\utl

11/03/2012  10:08 AM    <DIR>          .
11/03/2012  10:08 AM    <DIR>          ..
07/02/2011  12:49 AM               130 10053.sql
04/02/2012  12:43 PM             4,828 coe_gen_sql_profile.sql
08/18/2012  11:25 AM             1,185 coe_gen_sql_profile_.zip
06/02/2012  05:28 AM            10,305 coe_load_sql_baseline.sql
04/02/2012  12:43 PM            12,007 coe_load_sql_profile.sql
05/02/2012  11:27 AM            18,248 coe_xfr_sql_profile.sql
07/02/2011  12:49 AM               101 flush.sql
08/18/2012  11:25 AM                33 missing_file.txt
07/02/2011  12:49 AM               184 plan.sql
06/02/2012  05:28 AM            22,527 profiler.sql
06/02/2012  05:28 AM            73,472 pxhcdr.sql
06/02/2012  05:28 AM            71,213 roxecute.sql
11/03/2012  10:08 AM             3,441 roxtract.log
06/02/2012  05:28 AM            70,126 roxtract.sql <<< Read Only Procedure
08/11/2011  03:46 AM               475 sel.sql
02/02/2012  01:19 PM               435 sel_aux.sql
06/02/2012  05:28 AM           160,393 sqlhc.sql
01/03/2012  12:04 AM             2,891 sqltcdirs.sql
08/11/2011  03:46 AM             4,014 sqlthistfile.sql
08/11/2011  03:46 AM             3,116 sqlthistpurge.sql
04/02/2012  12:43 PM             3,694 sqltimp.sql
04/02/2012  12:43 PM             2,927 sqltimpfo.sql
01/03/2012  12:04 AM             3,545 sqltlite.sql
03/02/2012  04:06 PM             3,900 sqltmain.sql
09/01/2012  12:16 PM             6,802 sqltprofile.log
01/03/2012  12:04 AM             5,469 sqltprofile.sql
09/01/2012  12:16 PM             4,021 sqlt_s89910_p725901306_sqlprof.sql
09/01/2012  10:53 AM             4,548 sqlt_s89915_p3005811457_sqlprof.log
09/01/2012  09:29 AM             4,147 sqlt_s89915_p3005811457_sqlprof.sql
08/18/2012  09:33 AM                74 x.sql
02/18/2012  10:04 AM    <DIR>          xgram
02/02/2012  12:15 PM    <DIR>          xhume
06/01/2012  09:39 AM    <DIR>          xplore
              30 File(s)        498,251 bytes
               5 Dir(s)   7,103,299,584 bytes free
```

How Do We Use roxtract?

As we saw above `roxtract.sql` can be found in the `utl` directory of the `SQLT` unzipped area. You still need this procedure to be present on the target system, but you do not need the SQLTXPLAIN schema, in fact in the case of `roxtract.sql` you run the script as SYS. This is the target SQL in this case:

```
SQL>

select /*+ GATHER_PLAN_STATISTICS MONITOR*/
  cust_first_name,
  amount_sold
from
  customers C,
  sales S
where
  c.cust_id=s.cust_id
  and amount_sold > 1750
```

Now we get the SQL ID:

```
SQL> select sql_id from v$sqlarea where sql_text like 'select /*+ GATHER_PLAN_STATISTICS%';
SQL_ID
-------------
dck1xz983xa8c
```

Now that we have the SQL ID we can run the `roxtract` script from the SYS account. Notice that we have to enter the SQL ID the licensing level and if we want to select 10053 information. Notice there is no prompt for the SQLTXPLAIN schema password, because we do not use it.

```
SQL>@roxtract
Parameter 1:
Oracle Pack License (Tuning, Diagnostics or None) [T|D|N] (required)
Enter value for 1: T <<<You must enter a value here
PL/SQL procedure successfully completed.

Parameter 2:
SQL_ID of the SQL to be analyzed (required)

Enter value for 2: dck1xz983xa8c <<<This is the SQL ID we are investigating

Parameter 3:
EVENT 10053 Trace (on 11.2 and higher) [Y|N]

Enter value for 3: Y <<<We want a 10053 trace file.

PL/SQL procedure successfully completed.

Values passed:
~~~~~~~~~~~~~
License: "T"
SQL_ID : " dck1xz983xa8c"
Trace  : "Y"
```

```
PL/SQL procedure successfully completed.

SQL>
SQL>DEF script = 'roxtract';
SQL>DEF method = 'ROXTRACT';
SQL>
SQL>--
SQL>-- begin common
SQL>--
SQL>
SQL>DEF mos_doc = '215187.1';
SQL>DEF doc_ver = '11.4.4.6';
SQL>DEF doc_date = '2012/06/02';
SQL>DEF doc_link = 'https://support.oracle.com/CSP/main/article?cmd=show&type=NOT&id=';
SQL>DEF bug_link = 'https://support.oracle.com/CSP/main/article?cmd=show&type=BUG&id=';
SQL>
SQL>-- tracing script in case it takes long to execute so we can diagnose it
SQL>ALTER SESSION SET TRACEFILE_IDENTIFIER = "^^script._^^unique_id.";

Session altered.

Elapsed: 00:00:00.00
SQL>ALTER SESSION SET STATISTICS_LEVEL = 'ALL';
```

I've excluded most of this output, because it is very long. The output finishes with this:

```
Ignore CP or COPY error below
'cp' is not recognized as an internal or external command,
operable program or batch file.
f:\app\stelios\diag\rdbms\snc2\snc2\trace\snc2_ora_5768_DBMS_SQLDIAG_10053_20121103_111817.trc
        1 file(s) copied.
  adding: roxtract_SNC2_locutus_11.2.0.1.0_dck1xz983xa8c_20121103_111855_10053_trace_from_cursor.trc
(164 bytes security) (deflated 80%)
test of roxtract_SNC2_locutus_11.2.0.1.0_dck1xz983xa8c_20121103_111855.zip OK

ROXTRACT files have been created:
roxtract_SNC2_locutus_11.2.0.1.0_dck1xz983xa8c_20121103_111855_main.html.
roxtract_SNC2_locutus_11.2.0.1.0_dck1xz983xa8c_20121103_111855_monitor.html.
```

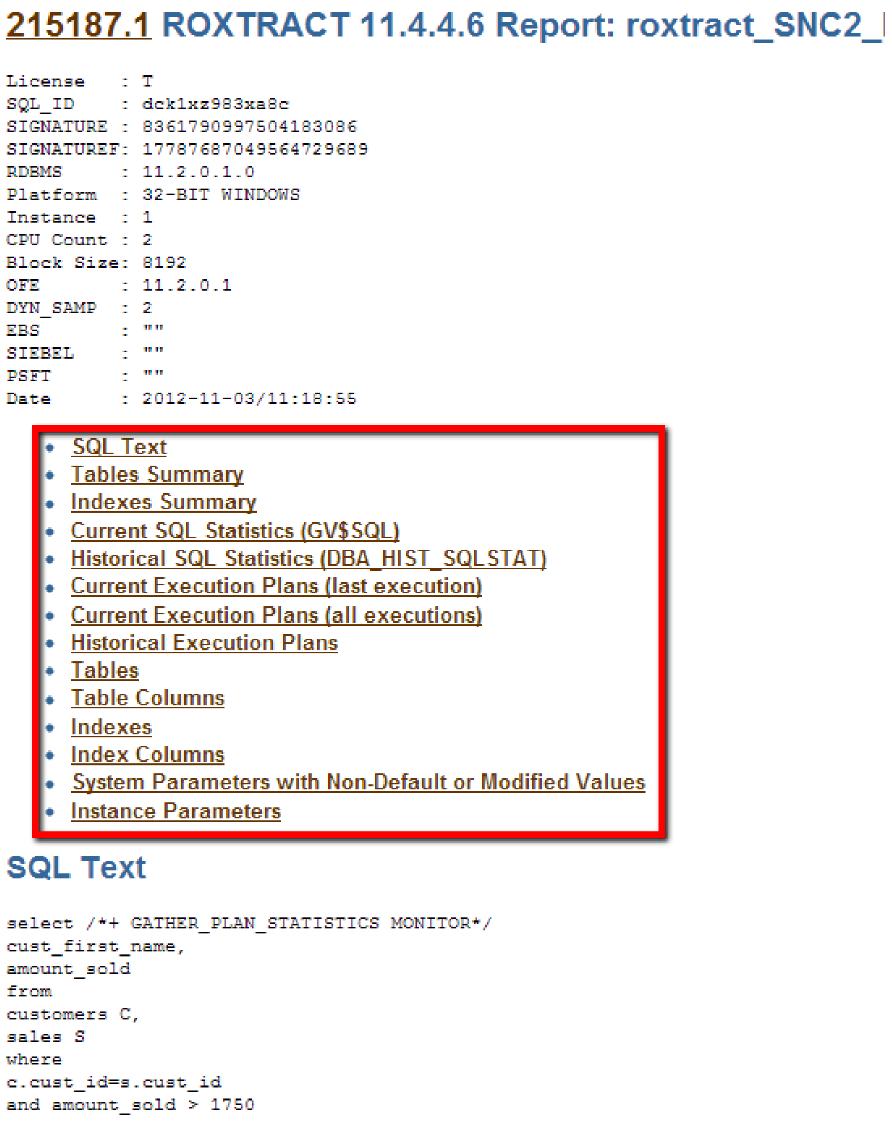
We can see from the above that a zip file has been created. If we extract the contents of this zip file into a directory we will see a number of files.

```
11/03/2012  11:22 AM    <DIR>          .
11/03/2012  11:22 AM    <DIR>          ..
11/03/2012  11:18 AM            17,035 roxtract.log
11/03/2012  11:19 AM           133,666 roxtract_SNC2_locutus_11.2.0.1.0_
dck1xz983xa8c_20121103_111855_10053_trace_from_cursor.trc
11/03/2012  11:19 AM           122,477 roxtract_SNC2_locutus_11.2.0.1.0_
dck1xz983xa8c_20121103_111855_main.html
11/03/2012  11:19 AM            10,113 roxtract_SNC2_locutus_11.2.0.1.0_
dck1xz983xa8c_20121103_111855_monitor.html
               4 File(s)        283,291 bytes
               2 Dir(s)   7,103,459,328 bytes free
```

We see four files, one of which is the log of the roxtract.sql procedure. The remaining three files represent some of the missing output from XTRSBY. The first is the 10053 trace file, which we specifically requested. We discussed this extensively in Chapter 5, so there's no need to go through its contents. The other two are HTML files, the "main" HTML file and the "monitor" HTML file. Let's look at what information roxtract.sql provides.

What Do roxtract Reports Look Like?

The first report roxtract_SNC2_locutus_11.2.0.1.0_dck1xz983xa8c_20121103_111855_main.html (in this example) is the roxtract main report. It is a trimmed down version of the SQLTXTRACT report, but nonetheless an amazingly useful HTML file. Let's look at the top of this HTML file, as shown in Figure 9-4.



215187.1 ROXTRACT 11.4.4.6 Report: roxtract_SNC2_

```
License   : T
SQL_ID    : dck1xz983xa8c
SIGNATURE : 8361790997504183086
SIGNATUREF: 1778768704956472968
RDBMS     : 11.2.0.1.0
Platform  : 32-BIT WINDOWS
Instance  : 1
CPU Count : 2
Block Size: 8192
OFE       : 11.2.0.1
DYN_SAMP  : 2
EBS       : ""
SIEBEL    : ""
PSFT      : ""
Date      : 2012-11-03/11:18:55
```

- SQL Text
- Tables Summary
- Indexes Summary
- Current SQL Statistics (GV$SQL)
- Historical SQL Statistics (DBA_HIST_SQLSTAT)
- Current Execution Plans (last execution)
- Current Execution Plans (all executions)
- Historical Execution Plans
- Tables
- Table Columns
- Indexes
- Index Columns
- System Parameters with Non-Default or Modified Values
- Instance Parameters

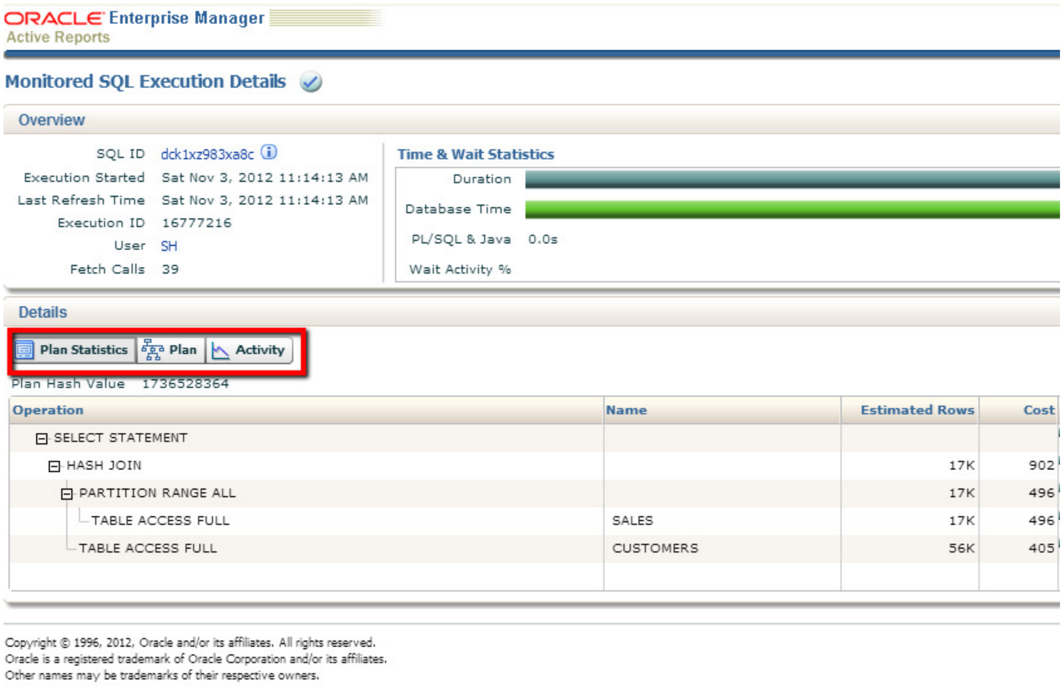
SQL Text

```
select /*+ GATHER_PLAN_STATISTICS MONITOR*/
cust_first_name,
amount_sold
from
customers C,
sales S
where
c.cust_id=s.cust_id
and amount_sold > 1750
```

Figure 9-4. *The top of the HTML ROXTRACT report*

As you can see, roxtract is weaker than the powerful XTRACT report. The number of sections it covers is reduced, but you still have the execution plans, table information and many other sections.

The other file supplied by roxtract is `roxtract_SNC2_locutus_11.2.0.1.0_dck1xz983xa8c_20121103_111855_monitor.html`. The HTML defaults to the Plan Statistics tab as shown in Figure 9-5.



ORACLE Enterprise Manager
Active Reports

Monitored SQL Execution Details

Overview

SQL ID	dck1xz983xa8c
Execution Started	Sat Nov 3, 2012 11:14:13 AM
Last Refresh Time	Sat Nov 3, 2012 11:14:13 AM
Execution ID	16777216
User	SH
Fetch Calls	39

Time & Wait Statistics

Duration	
Database Time	
PL/SQL & Java	0.0s
Wait Activity %	

Details

Plan Statistics | Plan | Activity

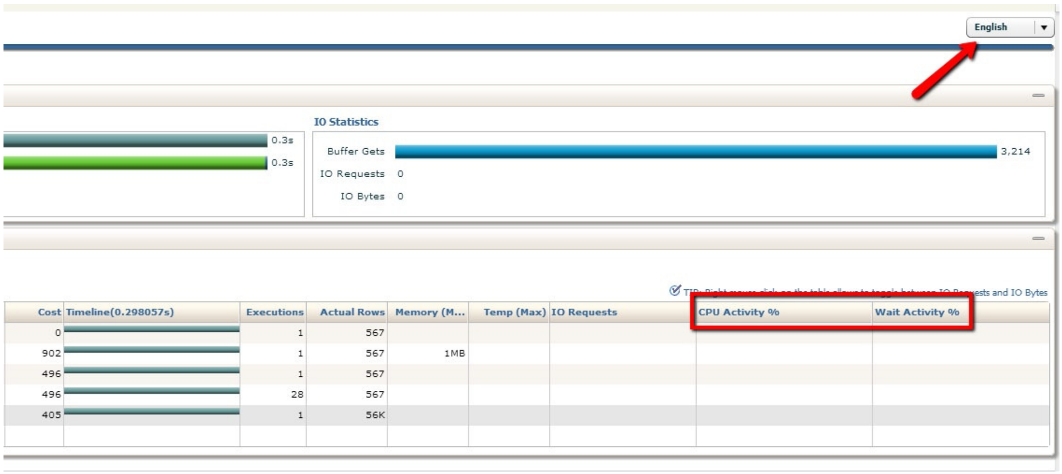
Plan Hash Value 1736528364

Operation	Name	Estimated Rows	Cost
SELECT STATEMENT			
HASH JOIN		17K	902
PARTITION RANGE ALL		17K	496
TABLE ACCESS FULL	SALES	17K	496
TABLE ACCESS FULL	CUSTOMERS	56K	405

Copyright © 1996, 2012, Oracle and/or its affiliates. All rights reserved.
Oracle is a registered trademark of Oracle Corporation and/or its affiliates.
Other names may be trademarks of their respective owners.

Figure 9-5. *The left-hand side of the page of the Monitored Execution Details page*

The figure above shows only the left hand side of the report. We see the expected execution steps and the estimated number of rows and the associated cost. We also have access to the menu, which allows us to choose between different displays. The right hand of the report shows us more information about the actual rows returned I/O requests and any wait activity. For this SQL there is nothing much to say, but you need to know those entries are there. See Figure 9-6.



IO Statistics

Buffer Gets		3,214
IO Requests	0	
IO Bytes	0	

TIP: Right mouse click on the table allows to toggle between IO Requests and IO Bytes

Cost	Timeline(0.298057s)	Executions	Actual Rows	Memory (M...	Temp (Max)	IO Requests	CPU Activity %	Wait Activity %
0		1	567					
902		1	567	1MB				
496		1	567					
496		28	567					
405		1	56K					

Figure 9-6. *The right hand side of the Monitored Execution Details page*

There's even a button to change the language of the report. The drop-down list includes English, German, Spanish, French, Italian, Korean, and other languages. If you click on the Plan tab, highlighted in Figure 9-5, you can also get a flow diagram version of the execution plan (as seen in Figure 9-7).

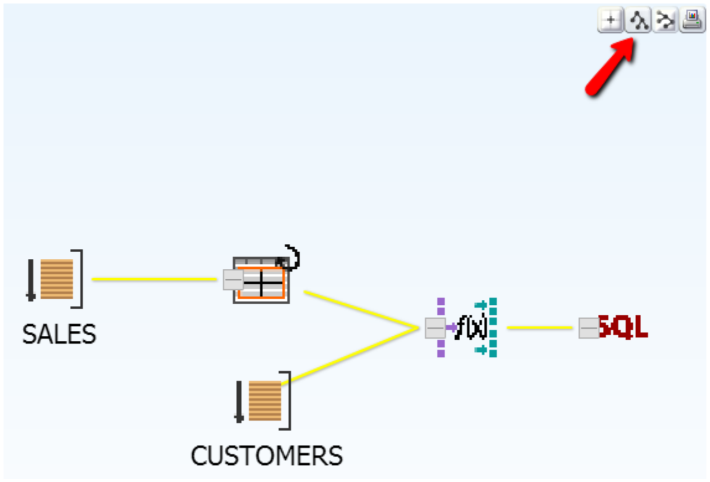


Figure 9-7. *Shows the flow chart representation of the execution plan*

Some developers prefer this representation of the execution plan, and they even get a choice of the flow going left to right or bottom to top.

Summary

In this chapter we covered the special case of Data Guard tuning and the utilities SQLTXPLAIN gives us to deal with this situation. You should not view Data Guard as a black box where no tuning can be done if it is open in read-only mode. Many companies do this, and you should be able to investigate performance if you need to. Now you can with XTRSBY and its trusty sidekick `roxtract.sql`. In the next chapter we will deal with comparison of execution plans. This is a very powerful technique for determining what has changed in an execution plan: and believe me, in an execution plan, nothing ever stays the same.

CHAPTER 10

Comparing Execution Plans

Comparing execution plans is one of the most useful diagnostic tests you can do. If you're lucky enough to have a good execution plan, you can use that plan as a guide to bring the bad execution plan into compliance. In Chapter 6 we talked about using SQL Profiles to apply a plan onto another database's SQL, in order to have an emergency option. Suppose, however, you have more time to consider the difference and try to fix it properly, rather than using the SQL Profile sticking plaster. SQLTCOMPARE is the tool for the job. It will compare two target systems for the same SQL and produce a report that describes the differences. You can even import the SQLTXPLAIN repositories from the two target systems and do the comparison there. This chapter is a pure SQLTXPLAIN chapter, we will be using SQLTXPLAIN's method SQLTCOMPARE and showing how our tuning knowledge from previous chapters can be used to diagnose a problem.

How Do You Use SQLTCOMPARE?

We will go through the steps for using SQLTCOMPARE in more detail with a practical example, but in broad terms the steps to use SQLTCOMPARE are as follows:

1. Get the SQL ID of the SQL you are investigating on both target databases. It should be the same.
2. Then run a main method on this SQL on BOTH target systems.
3. Make a note of the statement ID from both runs of the main method.
4. Create a directory to keep the files from your main method zip files.
5. Unzip both main methods into their respective directories.
6. Create another directory inside the main method directory to keep just the test case file (which is from a zip file inside the main method files: yes a zip file within a zip file). We will look at test cases in detail in Chapter 11.
7. Unzip the test case files into the test case dedicated area.
8. Then import one of the repositories of the statement ID to the other database or import both repositories to a third database (this is the method we will use).
9. Then finally run SQLTCOMPARE.

SQLTCOMPARE relies on the information in the SQLT repository to do its comparison and must have the information that's collected from one of the main methods. The main methods are:

- XTRACT (We covered this in Chapter 1)
- XECUTE (Also covered in Chapter 1)

- XTRXEC
- XPLAIN
- XTRSBY (We covered this in Chapter 9)

Because SQLTCOMPARE can be used with any of the main methods, it can also be used to compare SQL statements from two different systems, or from a primary and a standby database (if you use XTRSBY). It can even be used to compare execution on two different platforms: for example, a Linux and Windows and from a different version of the database as well. For example, 10g on Linux could be compared with 11g from Solaris.

A Practical Example

In this example we will compare the executions of the following statement against two target databases, but the steps apply to any SQL:

```
SQL> host type chapter10_01.sql
select
  s.amount_sold,
  c.cust_id,
  p.prod_name
from
  sh.products p,
  sh.sales s,
  sh.customers c
where
  c.cust_id=s.cust_id
  and s.prod_id=p.prod_id
  and c.cust_first_name='Theodorick';
```

Usually you would notice a difference in performance of a particular SQL if your batch times were different or if your OLTP performance was markedly different and you were investigating the databases looking for differences. You could also be doing an evaluation of the relative performance of two different platforms before migration. For the purposes of this example we have noticed that our target SQL has a different plan on system 1 than on system 2. We created the tables on system 2 from system 1, and the databases are on the same platform and hardware, but we see some differences in the execution plan. On the first database we see this execution plan:

```
SQL> set autotrace traceonly explain
SQL> set lines 200
SQL> /
Execution Plan
----------------------------------------------------------
Plan hash value: 725901306
```

```
-----------------------------------------------------------------------------------------------
| Id |Operation              |Name      | Rows  | Bytes | Cost (%CPU)| Time     | Pstart| Pstop
-----------------------------------------------------------------------------------------------
|  0 |SELECT STATEMENT       |          | 5557 |   303K|   908   (3)| 00:00:11 |       |      |
|* 1 | HASH JOIN             |          | 5557 |   303K|   908   (3)| 00:00:11 |       |      |
|  2 |  TABLE ACCESS FULL    |PRODUCTS  |   72 |  2160 |     3   (0)| 00:00:01 |       |      |
|* 3 |  HASH JOIN            |          | 5557 |   141K|   904   (3)| 00:00:11 |       |      |
|* 4 |   TABLE ACCESS FULL   |CUSTOMERS |   43 |   516 |   405   (1)| 00:00:05 |       |      |
|  5 |   PARTITION RANGE ALL |          |  918K|    12M|   494   (3)| 00:00:06 |     1 |   28 |
|  6 |    TABLE ACCESS FULL  |SALES     |  918K|    12M|   494   (3)| 00:00:06 |     1 |   28 |
-----------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------

   1 - access("S"."PROD_ID"="P"."PROD_ID")
   3 - access("C"."CUST_ID"="S"."CUST_ID")
   4 - filter("C"."CUST_FIRST_NAME"='Theodorick')
```

On the second system we see the execution plan is slightly different:

```
SQL> set autotrace traceonly explain
SQL> set lines 200
SQL> /

Execution Plan
----------------------------------------------------------
Plan hash value: 3857478275

----------------------------------------------------------------------------------
| Id  | Operation           | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |           |  2789 |   473K|  1652   (2)| 00:00:20 |
|*  1 |  HASH JOIN          |           |  2789 |   473K|  1652   (2)| 00:00:20 |
|   2 |   TABLE ACCESS FULL | PRODUCTS  |    72 |  6480 |     3   (0)| 00:00:01 |
|*  3 |   HASH JOIN         |           |  2789 |   228K|  1648   (1)| 00:00:20 |
|*  4 |    TABLE ACCESS FULL| CUSTOMERS |    17 |   765 |   409   (1)| 00:00:05 |
|   5 |    TABLE ACCESS FULL| SALES     |  860K|    31M|  1235   (1)| 00:00:15 |
----------------------------------------------------------------------------------

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

   1 - access("S"."PROD_ID"="P"."PROD_ID")
   3 - access("C"."CUST_ID"="S"."CUST_ID")
   4 - filter("C"."CUST_FIRST_NAME"='Theodorick')

Note
-----
dynamic sampling used for this statement (level=2)
```

The cost is different, the details of the plan are different, there could also be other differences, but that's not the point here. What can SQLTCOMPARE tell us about the difference between these two identical pieces of SQL that behave differently? In the non-SQLTXPLAIN world, we would now be searching for differences between the systems, looking at all the likely suspects one by one, with all their special tools, which would be a long-winded and error-prone process. With SQLTCOMPARE, however, we collect all the information and do a comparison of everything in one go. Then we decide what's important.

Collecting the Main Method Repository Data

The steps in collecting the SQLTXPLAIN data are just the standard methods we have already used. In this example we'll use SQLTXECUTE. To recap, the steps are to identify the SQL ID, create a file with the SQL statement in it (including any bind variables), and then run SQLTXECUTE on both systems.

We already mentioned the SQL statement above, and we can check on both systems that the SQL ID is the same. On both systems 1 and 2 we get:

```
SQL> select sql_id from v$sqlarea where sql_text like 'select  %Theodorick%';

SQL_ID
-------------
2qjq95uds6hpd
```

This is a simple step, but an important one. Without verifying that the SQL_ID is the same we could be comparing slightly different SQLs. In which case the comparison will not work. For each main method we use we are collecting information about the SQL statement and we need to take note of the statement ID (not the SQL IDs). That's the statement number that SQLTXPLAIN assigns to the run of the SQLT method. So on the first system we run SQLTXECUTE like this:

```
SQL> @sqltxecute chapter10_01.sql
```

The file `chapter10_01.sql` contains the SQL mentioned above. When the SQLTXECUTE method completes we see this output.

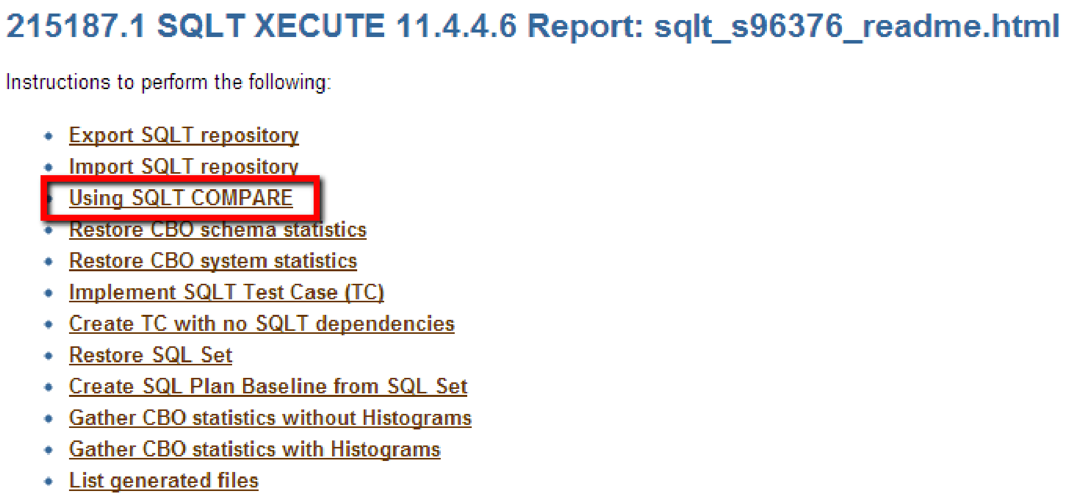
```
File sqlt_s96376_xecute.zip for chapter10_01.sql has been created.

SQLTXECUTE completed.
```

Here we take a note of the statement ID, in this case it is 96376. Similarly on the second system we run the same SQL (remember we checked the SQL ID) and use the same file to run SQLTXECUTE again. The output from the second system is this:

```
File sqlt_s73560_xecute.zip for chapter10_01.sql has been created.

SQLTXECUTE completed.
```

Preparing the Main Method Repository Data

The execution of `sqltxecute` creates zip files that contain important information. The information in this zip file contains instructions on how to import the data into another SQLT repository, but first you must unzip the main zip file. From the execution of the two main methods on the target systems we see that the statement IDs are 96376 (low CBO cost) and 73560 (higher CBO cost). In this case I created two directories, sqlt_s96376 and sqlt_s73560, each with the corresponding zip file in it, which I then unzipped. Each directory contains many files as we've learned, but in this case we are interested in the readme file, called `sql_snnnnn_readme.html`. This is your go to file in case of doubt (and this book of course). In this file the exact steps are described to carry out many functions including SQLTCOMPARE actions. This is the top of the readme file for the SQLTXECUTE report we ran on the first target system (see Figure 10-1).



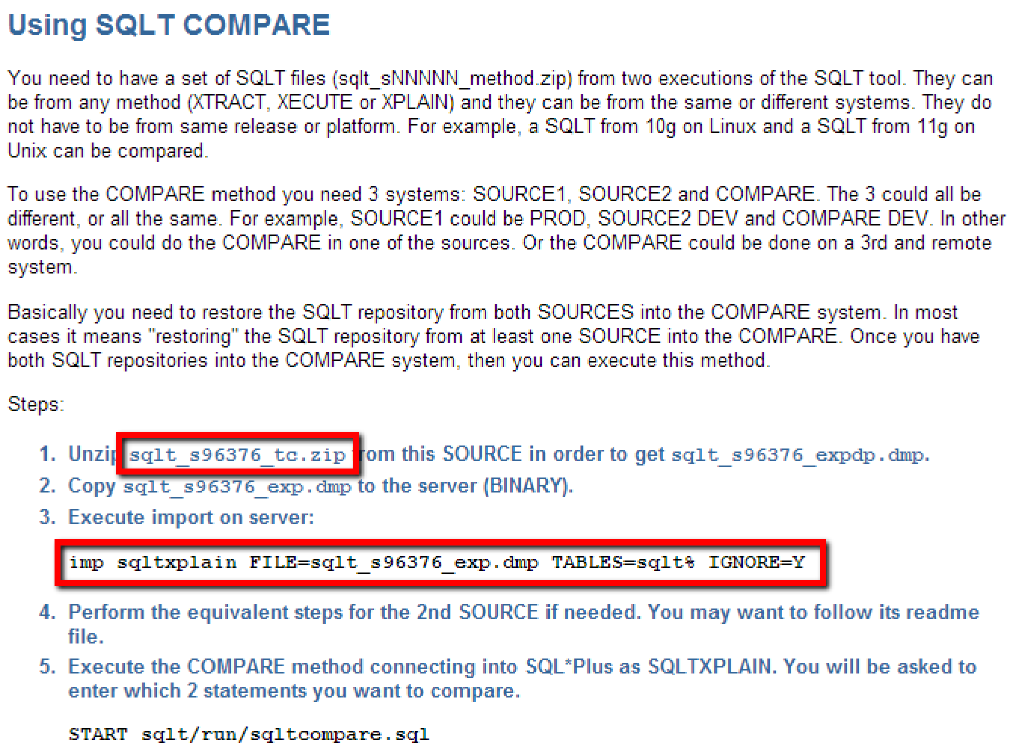
215187.1 SQLT XECUTE 11.4.4.6 Report: sqlt_s96376_readme.html

Instructions to perform the following:

- Export SQLT repository
- Import SQLT repository
- Using SQLT COMPARE
- Restore CBO schema statistics
- Restore CBO system statistics
- Implement SQLT Test Case (TC)
- Create TC with no SQLT dependencies
- Restore SQL Set
- Create SQL Plan Baseline from SQL Set
- Gather CBO statistics without Histograms
- Gather CBO statistics with Histograms
- List generated files

Figure 10-1. *The top of the readme file*

If we click on the “Using SQLT COMPARE” hyperlink in this part of the report we are taken to the steps describing the actions we need to take to import the data into a new database (or in fact the same database) to do the comparison. See Figure 10-2 for the readme section for s96376.



Using SQLT COMPARE

You need to have a set of SQLT files (sqlt_sNNNNN_method.zip) from two executions of the SQLT tool. They can be from any method (XTRACT, XECUTE or XPLAIN) and they can be from the same or different systems. They do not have to be from same release or platform. For example, a SQLT from 10g on Linux and a SQLT from 11g on Unix can be compared.

To use the COMPARE method you need 3 systems: SOURCE1, SOURCE2 and COMPARE. The 3 could all be different, or all the same. For example, SOURCE1 could be PROD, SOURCE2 DEV and COMPARE DEV. In other words, you could do the COMPARE in one of the sources. Or the COMPARE could be done on a 3rd and remote system.

Basically you need to restore the SQLT repository from both SOURCES into the COMPARE system. In most cases it means "restoring" the SQLT repository from at least one SOURCE into the COMPARE. Once you have both SQLT repositories into the COMPARE system, then you can execute this method.

Steps:

1. Unzip `sqlt_s96376_tc.zip` from this SOURCE in order to get `sqlt_s96376_expdp.dmp`.
2. Copy `sqlt_s96376_exp.dmp` to the server (BINARY).
3. Execute import on server:

```
imp sqltxplain FILE=sqlt_s96376_exp.dmp TABLES=sqlt% IGNORE=Y
```

4. Perform the equivalent steps for the 2nd SOURCE if needed. You may want to follow its readme file.
5. Execute the COMPARE method connecting into SQL*Plus as SQLTXPLAIN. You will be asked to enter which 2 statements you want to compare.

```
START sqlt/run/sqltcompare.sql
```

Figure 10-2. *The instructions for SQLTCOMPARE show the import*

Notice that we have to unzip the `sqlt_s96376_tc.zip` file, which is found inside the main SQLT zip file. We'll do more with the test case file in Chapter 11, which is on building good test cases. For the moment, however, we need only concern ourselves with the dump file found inside the test case zip file. I've unzipped this file into a directory called "TC" inside the unzipped area. Now making sure that your SID is pointing to the right database (in my case I am importing into a third database), we execute the instructions to import the data into our third database.

Importing the Respository Data

Importing the repository data is the last step before we can actually run SQLTCOMPARE. As long as we've kept track of the various zip files along the way, it should only be a matter of importing the files.

```
imp sqltxplain file=sqlt_s96376_exp.dmp tables=sqlt% ignore=y

Import: Release 11.2.0.1.0 - Production on Sat Nov 17 14:38:09 2012

Copyright (c) 1982, 2009, Oracle and/or its affiliates.  All rights reserved.

Password:

Connected to: Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - Production
With the Partitioning, OLAP, Data Mining and Real Application Testing options

Export file created by EXPORT:V11.02.00 via conventional path
import done in WE8MSWIN1252 character set and AL16UTF16 NCHAR character set
. importing SQLTXPLAIN's objects into SQLTXPLAIN
. importing SQLTXPLAIN's objects into SQLTXPLAIN
. . importing table          "SQLT$_SQL_STATEMENT"          1 rows imported
. . importing table             "SQLT$_AUX_STATS$"         13 rows imported
. . importing table    "SQLT$_DBA_AUTOTASK_CLIENT"          1 rows imported
. . importing table "SQLT$_DBA_COL_STATS_VERSIONS"        248 rows imported
. . importing table         "SQLT$_DBA_COL_USAGE$"         13 rows imported

---Output removed for clarity

. . importing table          "SQLT$_STGTAB_SQLSET"          7 rows imported
. . importing table  "SQLT$_V$SESSION_FIX_CONTROL"        406 rows imported
. . importing table     "SQLT$_WRI$_ADV_RATIONALE"          2 rows imported
. . importing table         "SQLT$_WRI$_ADV_TASKS"          2 rows imported
Import terminated successfully without warnings.
```

This has imported the required information from the first database (the good execution plan) into the SQLTXPLAIN schema on the third database. Now we repeat the entire process for the second database.

```
imp sqltxplain file=sqlt_s73560_exp.dmp tables=sqlt% ignore=y
Import: Release 11.2.0.1.0 - Production on Sat Nov 17 14:48:31 2012

Copyright (c) 1982, 2009, Oracle and/or its affiliates.  All rights reserved.

Password:

Connected to: Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - Production
With the Partitioning, OLAP, Data Mining and Real Application Testing options
```

```
Export file created by EXPORT:V11.02.00 via conventional path
import done in WE8MSWIN1252 character set and AL16UTF16 NCHAR character set
. importing SQLTXPLAIN's objects into SQLTXPLAIN
. importing SQLTXPLAIN's objects into SQLTXPLAIN
. . importing table          "SQLT$_SQL_STATEMENT"          1 rows imported
. . importing table             "SQLT$_AUX_STATS$"         13 rows imported
. . importing table    "SQLT$_DBA_AUTOTASK_CLIENT"          1 rows imported
. . importing table         "SQLT$_DBA_COL_USAGE$"          5 rows imported
---Output removed for clarity
. . importing table          "SQLT$_STGTAB_SQLSET"          6 rows imported
. . importing table  "SQLT$_V$SESSION_FIX_CONTROL"        406 rows imported
. . importing table     "SQLT$_WRI$_ADV_RATIONALE"          8 rows imported
. . importing table         "SQLT$_WRI$_ADV_TASKS"          2 rows imported
. . importing table "SQLT$_WRI$_OPTSTAT_AUX_HISTORY"         18 rows imported
Import terminated successfully without warnings.
```

Now all that remains to be done is to run the SQLTCOMPARE method. The script for this is in the "run" area. Logged in as SQLTXPLAIN we run SQLTCOMPARE.

Running SQLTCOMPARE

Once we have imported the data we can run SQLTCOMPARE. We can run it with or without the parameters on the command line. In this case I'll omit the parameters so that we can see what the prompts are:

```
SQL> @sqltcompare
... please wait ...

STAID MET INSTANCE SQL_TEXT
----- --- -------- ------------------------------------------------------------
73560 XEC snc2     select   s.amount_sold,   c.cust_id,   p.prod_name from   sh
96376 XEC snc1     select   s.amount_sold,   c.cust_id,   p.prod_name from   sh

Parameter 1:
STATEMENT_ID1 (required)

Enter value for 1: 96376

Parameter 2:
STATEMENT_ID2 (required)

Enter value for 2: 73560
```

We see that the SQLTCOMPARE method immediately identifies the statements that we could compare and shows them to us (because we did not specify the statement Ids on the command line). Our good execution plan was run on snc1 (statement ID 96376), and our not so good execution plan was run on snc2 (a newly created database). I always like to compare good to bad, so for the first statement ID I'll enter 96376 and for the second I'll enter the only remaining statement ID 73560. (When you do your own comparisons you can compare the SQLs anyway you like, but I like the good vs. bad convention to keep me focused on finding the differences the right way around).

Now we are presented with the plan hash value history for each of the statements on each instance so we can choose which plan hash values to compare. In this case I'll compare the best plan on both systems.

```
PLAN_HASH_VALUE SQLT_PLAN_HASH_VALUE STATEMENT_ID ATTRIBUTE
--------------- -------------------- ------------ ---------
      725901306                16588        96376 [B][W]
     2025531852                21473        96376
     3852404249                68517        96376

Parameter 3:
PLAN_HASH_VALUE1 (required if more than one)

Enter value for 3: 725901306


PLAN_HASH_VALUE SQLT_PLAN_HASH_VALUE STATEMENT_ID ATTRIBUTE
--------------- -------------------- ------------ ---------
     3850511567                 8739        73560
     3857478275                 9628        73560 [B][W]
     3858714362                69559        73560

Parameter 4:
PLAN_HASH_VALUE2 (required if more than one)

Enter value for 4: 3857478275
Values passed to sqltcompare:
~~~~~~~~~~~~~~~~~~~~~~~~~~~~
STATEMENT_ID1   : "96376"
STATEMENT_ID2   : "73560"
PLAN_HASH_VALUE1: "725901306"
PLAN_HASH_VALUE2: "3857478275"


... please wait ...

15:03:03 sqlt$c: => compare_report
15:03:03 sqlt$a: -> common_initialization
15:03:03 sqlt$a: ALTER SESSION SET NLS_NUMERIC_CHARACTERS = ".,"
15:03:03 sqlt$a: <- common_initialization
15:03:03 sqlt$c: -> header
15:03:03 sqlt$c: -> sql_text
15:03:03 sqlt$c: -> sql_identification
15:03:03 sqlt$c: -> environment
15:03:03 sqlt$c: -> nls_parameters
15:03:03 sqlt$c: -> io_calibration
15:03:03 sqlt$c: -> cbo_environment
15:03:03 sqlt$c: -> fix_control
15:03:03 sqlt$c: -> cbo_system_statistics
15:03:03 sqlt$c: -> execution_plan
15:03:03 sqlt$c: plan 96376 "GV$SQL_PLAN" "725901306" "-1" "1" "1" "43F746F4"
15:03:03 sqlt$c: plan 73560 "GV$SQL_PLAN" "3857478275" "-1" "1" "0" "2A13E128"
15:03:03 sqlt$c: -> tables
15:03:04 sqlt$c: -> table_partitions
15:03:04 sqlt$c: -> indexes
```

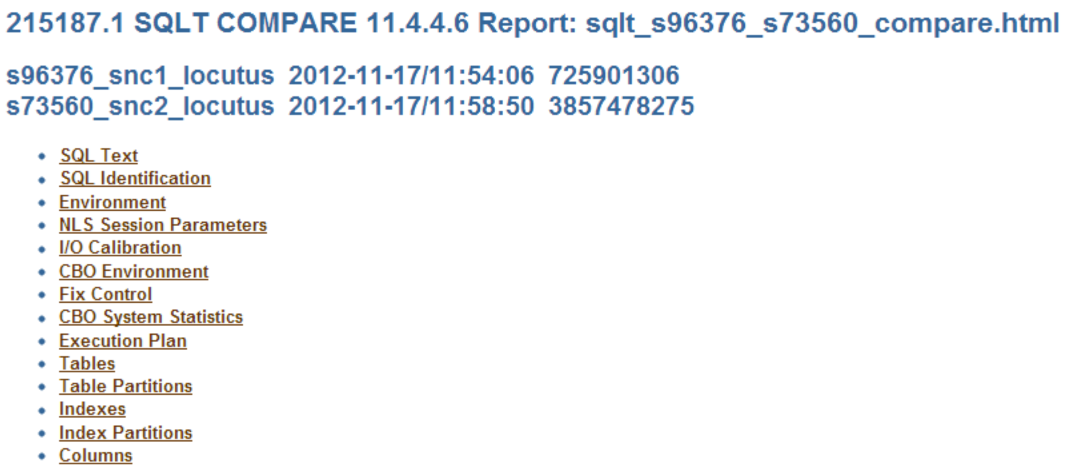
```
15:03:04 sqlt$c: -> index_partitions
15:03:04 sqlt$c: -> columns
15:03:05 sqlt$c: -> footer_and_closure
15:03:05 sqlt$c: <= compare_report

sqlt_s96376_s73560_compare.html has been generated

SQLTCOMPARE completed.
```

Now we have a file called `sqlt_s96376_s73560_compare.html`:

```
C:\Documents and Settings\Stelios\Desktop\SQLT\sqlt\run>dir *compare*.html
 Volume in drive C has no label.
 Volume Serial Number is 77E9-80B4

 Directory of C:\Documents and Settings\Stelios\Desktop\SQLT\sqlt\run

11/17/2012  03:03 PM           355,957 sqlt_s96376_s73560_compare.html
               1 File(s)        355,957 bytes
               0 Dir(s)   7,088,697,344 bytes free
```

Reading the SQLTCOMPARE Report

The HTML file produced by running SQLTCOMPARE contains information that compares the plan hash value from each of the two target databases. In this case the first system was slightly faster than the second system. There could be many reasons for this, but the key element here is to see which sections are there for investigation. We already know the execution plan is slightly different. Here is the top of the SQLTCOMPARE report for our example (see Figure 10-3).



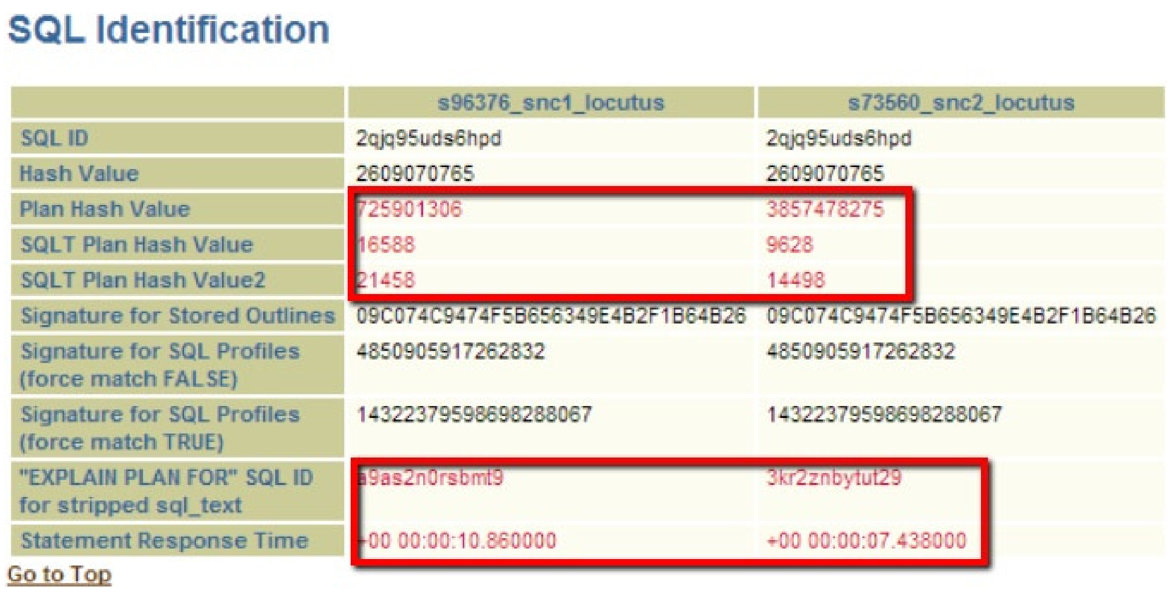
215187.1 SQLT COMPARE 11.4.4.6 Report: sqlt_s96376_s73560_compare.html

s96376_snc1_locutus 2012-11-17/11:54:06 725901306
s73560_snc2_locutus 2012-11-17/11:58:50 3857478275

- SQL Text
- SQL Identification
- Environment
- NLS Session Parameters
- I/O Calibration
- CBO Environment
- Fix Control
- CBO System Statistics
- Execution Plan
- Tables
- Table Partitions
- Indexes
- Index Partitions
- Columns

Figure 10-3. *The top of the SQLTCOMPARE report*

If we click on the SQL Identification hyperlink from the top of the report, we'll see differences (if there are any) as shown in Figure 10-4.



SQL Identification

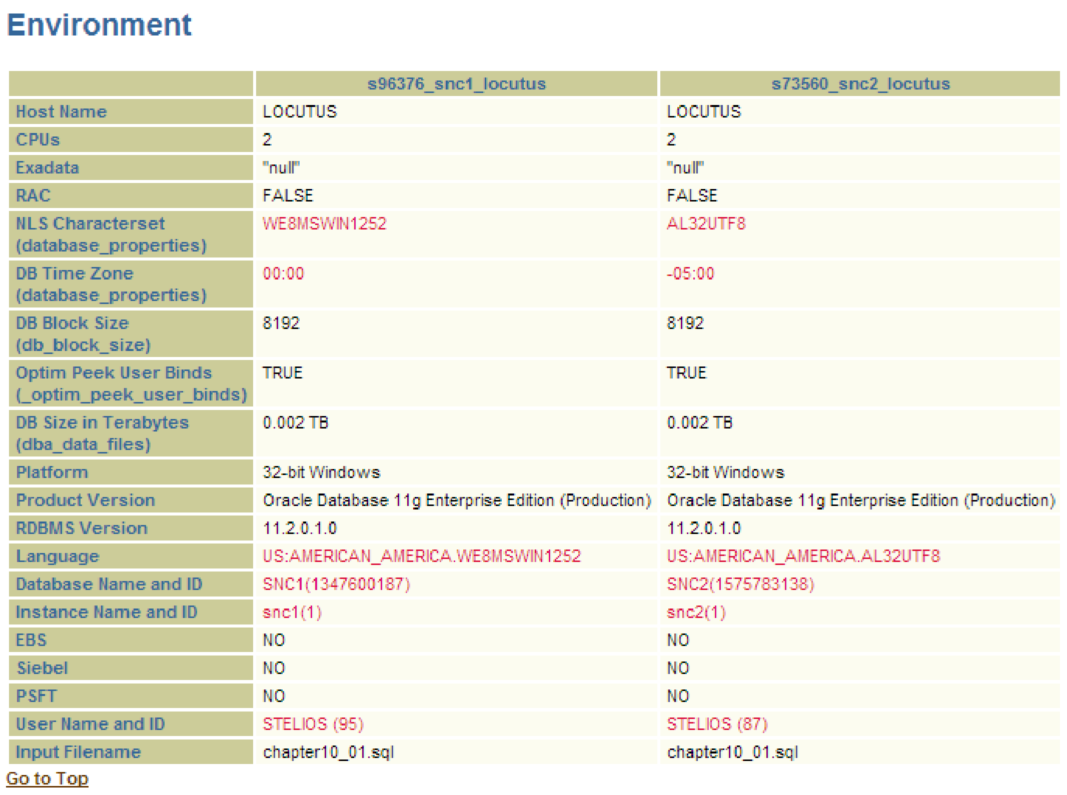
	s96376_snc1_locutus	s73560_snc2_locutus
SQL ID	2qjq95uds6hpd	2qjq95uds6hpd
Hash Value	2609070765	2609070765
Plan Hash Value	725901306	3857478275
SQLT Plan Hash Value	16588	9628
SQLT Plan Hash Value2	21458	14498
Signature for Stored Outlines	09C074C9474F5B656349E4B2F1B64B26	09C074C9474F5B656349E4B2F1B64B26
Signature for SQL Profiles (force match FALSE)	4850905917262832	4850905917262832
Signature for SQL Profiles (force match TRUE)	14322379598698288067	14322379598698288067
"EXPLAIN PLAN FOR" SQL ID for stripped sql_text	a9as2n0rsbmt9	3kr2znbytut29
Statement Response Time	+00 00:00:10.860000	+00 00:00:07.438000

Go to Top

Figure 10-4. *The SQL Identification shows some basic differences*

Notice we did indeed choose the same SQL ID, but that some entries are marked in red highlighted by the boxes in Figure 10-4), to indicate that there are differences. This method of showing differences is not consistently used throughout the report. Sometimes sections are marked in amber for differences that are not so important, but generally differences marked in red are important and should be noted and understood before moving on.

The next section is the Environment section, for which an example is shown in Figure 10-5.



Environment

	s96376_snc1_locutus	s73560_snc2_locutus
Host Name	LOCUTUS	LOCUTUS
CPUs	2	2
Exadata	"null"	"null"
RAC	FALSE	FALSE
NLS Characterset (database_properties)	WE8MSWIN1252	AL32UTF8
DB Time Zone (database_properties)	00:00	-05:00
DB Block Size (db_block_size)	8192	8192
Optim Peek User Binds (_optim_peek_user_binds)	TRUE	TRUE
DB Size in Terabytes (dba_data_files)	0.002 TB	0.002 TB
Platform	32-bit Windows	32-bit Windows
Product Version	Oracle Database 11g Enterprise Edition (Production)	Oracle Database 11g Enterprise Edition (Production)
RDBMS Version	11.2.0.1.0	11.2.0.1.0
Language	US:AMERICAN_AMERICA.WE8MSWIN1252	US:AMERICAN_AMERICA.AL32UTF8
Database Name and ID	SNC1(1347600187)	SNC2(1575783138)
Instance Name and ID	snc1(1)	snc2(1)
EBS	NO	NO
Siebel	NO	NO
PSFT	NO	NO
User Name and ID	STELIOS (95)	STELIOS (87)
Input Filename	chapter10_01.sql	chapter10_01.sql

Go to Top

Figure 10-5. *The Environment section of the SQLTCOMPARE report*

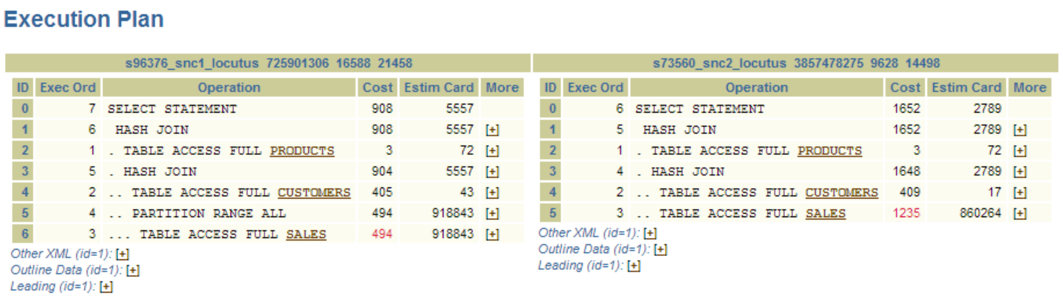
Here the things that are the same are just as important as the things that are different. We can see immediately that the CPU count is the same, the RAC environment is the same, and the block size is the same. Some differences exist: the character set, time zone, etc.

In the main HTML Compare report (as shown in Figure 10-3) we can scan from top to bottom. This is not a long complicated report like the main SQLTXTRACT or SQLTXECUTE report. There are also sections on:

- SQL Text
- SQL Identification
- Environment
- NLS session parameters
- I/O calibration
- CBO parameters
- Fix control
- CBO System statistics
- Execution plans
- Table information

- Table partition information
- Index information
- Index partition information
- Column information
- Peeked binds
- Captured binds

And of course there are the execution plans (as shown in Figure 10-6). Much of this information is exactly the same as you would see in a SQLTXRACT report, so we won't dwell on it too much here. The important thing to note is that it gives you an easy side-by-side comparison for many features including the execution plan (see Figure 10-6). This is a great way to quickly spot differences, especially if you have hundreds of lines in your execution plan and only one line is different (yes, it does happen and yes, it can make a vast difference in the execution time).



Execution Plan

s96376_snc1_locutus 725901306 16588 21458

ID	Exec Ord	Operation	Cost	Estim Card	More
0	7	SELECT STATEMENT	908	5557	
1	6	HASH JOIN	908	5557	[+]
2	1	. TABLE ACCESS FULL PRODUCTS	3	72	[+]
3	5	. HASH JOIN	904	5557	[+]
4	2	.. TABLE ACCESS FULL CUSTOMERS	405	43	[+]
5	4	.. PARTITION RANGE ALL	494	918843	[+]
6	3	... TABLE ACCESS FULL SALES	494	918843	[+]

Other XML (id=1): [+]
Outline Data (id=1): [+]
Leading (id=1): [+]

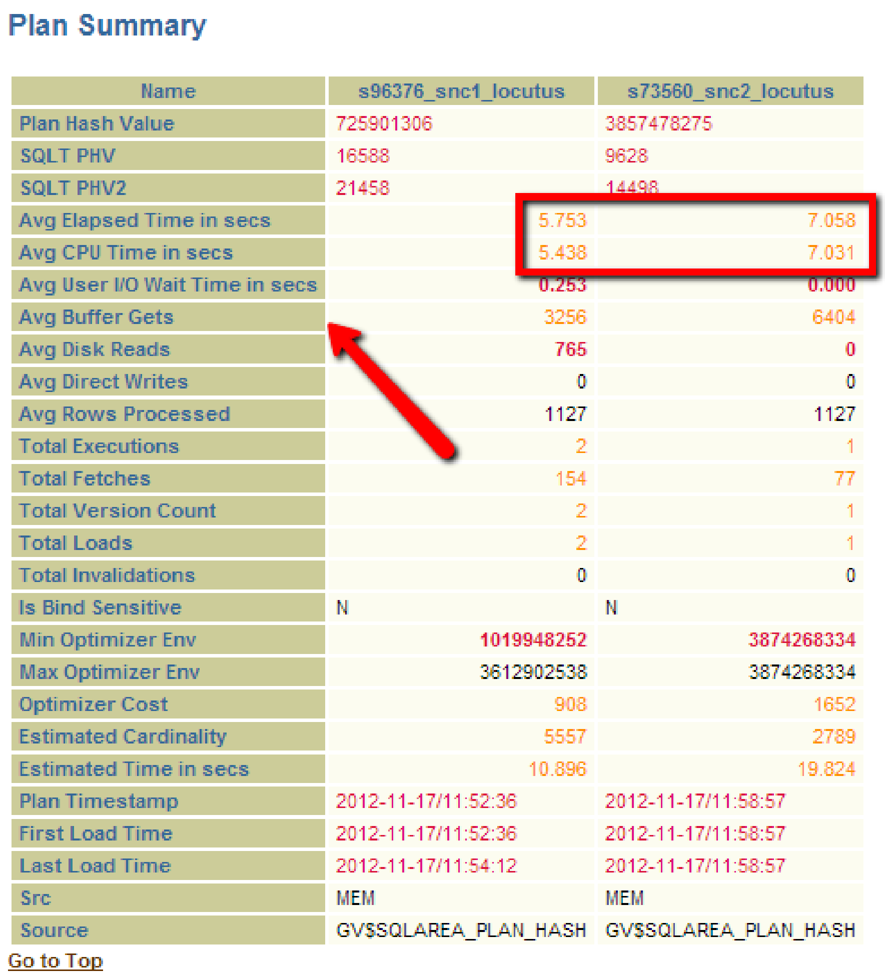
s73560_snc2_locutus 3857478275 9628 14498

ID	Exec Ord	Operation	Cost	Estim Card	More
0	6	SELECT STATEMENT	1652	2789	
1	5	HASH JOIN	1652	2789	[+]
2	1	. TABLE ACCESS FULL PRODUCTS	3	72	[+]
3	4	. HASH JOIN	1648	2789	[+]
4	2	.. TABLE ACCESS FULL CUSTOMERS	409	17	[+]
5	3	.. TABLE ACCESS FULL SALES	1235	860264	[+]

Other XML (id=1): [+]
Outline Data (id=1): [+]
Leading (id=1): [+]

***Figure 10-6.** The Execution plans side by side from the two systems*

The most important section is the Plan Summary. This shows the average elapsed time and CPU time for both compared SQL statements. We see an example of this in Figure 10-7.



Plan Summary

Name	s96376_snc1_locutus	s73560_snc2_locutus
Plan Hash Value	725901306	3857478275
SQLT PHV	16588	9628
SQLT PHV2	21458	14498
Avg Elapsed Time in secs	5.753	7.058
Avg CPU Time in secs	5.438	7.031
Avg User I/O Wait Time in secs	0.253	0.000
Avg Buffer Gets	3256	6404
Avg Disk Reads	765	0
Avg Direct Writes	0	0
Avg Rows Processed	1127	1127
Total Executions	2	1
Total Fetches	154	77
Total Version Count	2	1
Total Loads	2	1
Total Invalidations	0	0
Is Bind Sensitive	N	N
Min Optimizer Env	1019948252	3874268334
Max Optimizer Env	3612902538	3874268334
Optimizer Cost	908	1652
Estimated Cardinality	5557	2789
Estimated Time in secs	10.896	19.824
Plan Timestamp	2012-11-17/11:52:36	2012-11-17/11:58:57
First Load Time	2012-11-17/11:52:36	2012-11-17/11:58:57
Last Load Time	2012-11-17/11:54:12	2012-11-17/11:58:57
Src	MEM	MEM
Source	GV$SQLAREA_PLAN_HASH	GV$SQLAREA_PLAN_HASH

Go to Top

Figure 10-7. *The Plan Summary section of the report*

Not only can we see how the two statements compare in terms of elapsed time (which might be misleading if the load on the two systems is different), but we also see the CPU time in seconds and the average number of buffer gets, which is usually a good measure of how good an execution plan is compared to a different run of the same SQL. I've highlighted the buffer gets with an arrow. Naturally every line of the plan summary is telling us something. I've listed below the items I consider important:

- Avg. Elapsed Time in secs: clearly important if the load on the systems is comparable as it is an indication of the performance of the SQL.
- Avg. CPU Time in secs: the CPU time in comparison to the total elapsed time can tell you if the SQL is CPU bound or not.
- Avg. User I/O Wait Time in secs: a measure of how much time is spent on I/O could tell you if you are I/O bound.
- Avg. Buffer Gets: can be a good comparison between individual runs of the same SQL on busy systems with much activity where relative performance from one run to another is hard to gauge.

- Avg. Disk Reads: shows how many of your reads are going to disk.
- Avg. Direct Writes: indicates how much of your I/O is a direct write.
- Avg. Rows Processed: in comparisons you would expect these to be the same if the data was the same.
- Total Executions: this gives you an idea of how many times the SQL was run on each system.
- Total Fetches: a measure of the data returned in all executions.
- Total Version Count: a count of the number of versions of the cursor found on the system.
- Total Loads: the number of times the SQL was loaded.
- Total Invalidations: shows how many times the SQL was made invalid.
- Is Bind Sensitive: discussed in Chapter 7.
- Min Optimizer Environment: shows the minimum optimizer environment hash value.
- Max Optimizer Environment: shows the maximum optimizer environment hash value (this and the previous value are useful for comparing environment values).
- Optimizer Cost: useful to compare overall cost.
- Estimated Cardinality: useful to see what the optimizer estimated.
- Estimated Time in Seconds: another measure of the optimizer calculations.
- Plan Time Stamp: shows when the execution plan was calculated.
- First Load Time: when the plan was first loaded into memory.
- Last Load Time: when the plan was loaded into memory last.
- Src: shows where the information was derived from.
- Source: shows which source was used in this case GV$SQLAREA_PLAN_HASH.

We see a lot of sections that are different from one another, but Compare is not intelligent enough to tell us what is important in all cases. In this case we can confirm that the elapsed time was slightly higher on the second target system than on the first system. In this case the amber color used in the report indicates that the comparisons are not so different as to cause a problem but should probably be investigated if there aren't any other more pressing problems. In percentage terms "Avg User I/O Wait Time in secs" is very different (as one of the values is 0.000), so this is highlighted in red. The same applies to disk reads. Other differences are not so important (such as plan time stamp), but these are shown in red anyway. Remember these are only guidelines, so you must decide what's important in terms of a difference because you know your environment and circumstances. SQLTCOMPARE is there to show you the differences so as to allow you to make a good decision quickly.

Summary

Comparison of two SQL statement IDs and plan hash values is an invaluable method for highlighting differences, which sometimes do not stand out on individual inspection. SQLTCOMPARE is also invaluable in confirming the differences detected by users or developers and thereby eliminating other sources as the cause of differences. Different environments can often be challenging to compare because there are so many differences that can affect the execution plan. SQLTCOMPARE grabs all the information available and shows it side by side in a simple-to-read report. Yet another case of SQLTXPLAIN making a complex job fast and easy. In the next chapter we look at building good test cases and how these can be used to modify your execution plan.

CHAPTER 11

Building Good Test Cases

Occasionally there are tuning problems where the solution is not obvious from the examination of the SQLTXTRACT or SQLTXECUTE report. All the usual suspects have been eliminated and a more thorough investigation may be required. In cases like these it is useful to build a test case to allow experimentation away from the main environment. If you are investigating a problematic SQL in a test environment, normally you would try and make the environment the same as the environment you are attempting to replicate. This is often very difficult to do because there are many factors affecting an environment that need to be copied. Here are some of them.

- Hardware
- Software
- CBO parameters
- Object Statistics
- Dictionary statistics
- Data
- Table clustering factors.
- Indexes
- Sequences
- Tables

There are many other factors; some factors may be crucial to your problem, but others may not be. You can try and build a test case the old fashioned way, by exporting the data and associated statistics to a test environment, but this is an error prone manual process, unless you've built a script to do this. Luckily SQLT does everything needed to build a test case in a few very simple steps. It's platform independent, so you can run a Solaris test case on Windows for example: it's easy to use and automated. The only thing that is not usually replicated is the data (because that would usually make the test case too big). However, data can be included if needed (we include an example at the end of this chapter).

Warning Only run test case setup scripts in environments that you can easily replace. Some of the scripts involved in setting up the test environment will change the operational environment. There is a good reason you need access to SYS to carry out these operations. They will change system setup. Typically, you would create a stand-alone database for a test case and work with only that test case in your environment. When you are finished with the test case you would drop the entire database. Re-using such a database would be a dangerous course of action. Any number of CBO parameters may have been changed. Only set up test case in scrap databases. You have been warned.

What Can You Do with Test Cases?

In some situations tuning can be a tricky job. Before you can start to investigate the problem, you have to understand what all the parts of the execution plan are doing. Sometimes this understanding comes from tinkering with a copy of the SQL and its environment. You can add indexes, change hints, change optimizer parameters, drop histogram statistics—the list is endless. These kinds of things are done not at random but based on some understanding (or at least as much as you have at the time) of the problem at hand.

All this kind of tinkering cannot be done on a production environment, and it is often difficult to do on a development environment because the risk is too high that some aspect of your case will affect someone else: for example, public synonyms or sequences. On the other hand if you can set up a little private database with a good test case you can change all sorts of parameters without affecting anyone directly. As long as the versions of the database are the same you will get mostly the same behavior. I say "mostly" because in some cases (such as Exadata) there are specific issues that need an environment closer to the one you are replicating. Hybrid columnar compression, for example, cannot be done on a non-Exadata environment. DBAs and developers have been creating scripts to create test cases for their environment ever since Oracle was released, but as the complexity of environments increased setting up good robust cases has taken longer and longer. These hand-crafted test cases are useful but too time consuming to create. SQLT does all of the hard work for you by automatically collecting all the required metadata and statistical information.

Building a Test Case

To get a test case requires no additional effort above running a SQLT report. Running SQLTXTRACT alone creates enough information for a test case. Here is an example directory showing the files from the main SQLT zip file.

```
12/11/2012  08:47 PM           6,705 sqlt_s11992_driver.zip
12/11/2012  08:46 PM           3,452 sqlt_s11992_lite.html
12/11/2012  08:47 PM          12,646 sqlt_s11992_log.zip
12/11/2012  08:46 PM         507,040 sqlt_s11992_main.html
12/11/2012  08:46 PM          11,979 sqlt_s11992_readme.html
12/11/2012  08:46 PM           2,722 sqlt_s11992_sql_detail_active.html
12/11/2012  08:46 PM         909,869 sqlt_s11992_tc.zip <<<Test Case File
12/11/2012  08:46 PM          30,764 sqlt_s11992_tcx.zip <<<Test Case Express File
12/11/2012  08:46 PM             404 sqlt_s11992_tc_script.sql
12/11/2012  08:46 PM             297 sqlt_s11992_tc_sql.sql
12/11/2012  08:47 PM       1,012,953 sqlt_s11992_trc.zip
```

I've highlighted two of the files from this directory listing. The first one `sqlt_s11992_tc.zip` is the standard test case file. We'll deal with what you can do with this file in the next section. The second file `sqlt_s11992_tcx.zip` is a recent innovation (as of December 2012) and we'll discuss that in the section on test case express.

A Test Case Step by Step

The first step in creating your test case is to create a sub-directory of the main SQLT area. Then copy the `sqlt_s11992_tc.zip` file into this area and unzip the file. A number of files are created and we'll describe each of them. Here is a sample listing.

```
12/11/2012  08:46 PM             132 10053.sql
12/11/2012  08:46 PM             103 flush.sql
12/11/2012  08:46 PM             279 plan.sql
12/11/2012  08:46 PM             404 q.sql
12/11/2012  08:46 PM              38 readme.txt
```

```
12/11/2012  08:46 PM               424 sel.sql
12/11/2012  08:46 PM               378 sel_aux.sql
12/11/2012  08:46 PM               448 setup.sql
12/11/2012  08:46 PM               267 sqlt_s11992_del_hgrm.sql
12/11/2012  08:46 PM         7,708,672 sqlt_s11992_exp.dmp
12/11/2012  08:46 PM               683 sqlt_s11992_import.sh
12/11/2012  08:46 PM             5,077 sqlt_s11992_metadata.sql
12/11/2012  08:46 PM               244 sqlt_s11992_purge.sql
12/11/2012  08:46 PM             9,720 sqlt_s11992_readme.txt
12/11/2012  08:46 PM               362 sqlt_s11992_restore.sql
12/11/2012  08:46 PM            84,648 sqlt_s11992_set_cbo_env.sql
12/11/2012  08:46 PM             1,285 sqlt_s11992_system_stats.sql
12/11/2012  08:46 PM           909,869 sqlt_s11992_tc.zip
12/11/2012  08:46 PM               141 tc.sql
12/11/2012  08:46 PM               965 tc_pkg.sql
12/11/2012  08:46 PM               117 xpress.sh
12/11/2012  08:46 PM             1,087 xpress.sql
```

The files in this directory are all that is needed (along with SQLT to create a test case on your database). In the next section we'll look at these files in detail.

The Test Case Files

The scripts and other files in this directory are designed to work together to produce a test case simply and quickly. I list below each of the files and what they are for. Later we'll see how these files work together to produce a realistic test environment.

- `10053.sql` – Sets 10053 tracing at level 1
- `flush.sql` – Flushes the shared pool
- `plan.sql` – Displays the execution plan for the most recently executed SQL and spools the output to `plan.log`. It uses the `dbms_xplan.display_cursor` procedure.
- `q.sql` – The SQL being investigated.
- `readme.txt` – The instructions. They are very brief: for example, "connect as sys and execute setup.sql."
- `sel.sql` – Computes predicate selectivity. This little script relies on `sel_aux.sql` (see below) and prompts for a table name and predicate and then gives you the predicted cardinality and selectivity. There is an example of the use of `sel.sql` below.
- `sel_aux.sql` – Produces expected cardinality and selectivity with `sel.sql`, based on different predicates.
- `setup.sql` – Sets up system statistics, creates the test user and metadata, imports the object statistics and the optimizer environment, executes the test query and displays the execution plan.
- `sqlt_snnnnn_del_hgrm.sql` – Deletes histograms for the TC schema.
- `sqlt_snnnnn_exp.dmp` – Dump file containing the statistics.
- `sqlt_snnnnn_import.sh` – Unix version of the import script for the SQLT objects.
- `sqlt_snnnnn_metadat.sql` – Script called from `setup.sql`; creates the test case user and user objects.

- `sqlt_snnnnn_purge.sql` – Removes the reference to the test case SQL from the SQLT repository.
- `sqlt_snnnnn_readme.txt` – Documentation for the specific test case, containing brief instructions on exporting and importing the SQLT repository, using SQLT COMPARE, (covered in chapter 10), restoring CBO statistics and implementing a test case in both express mode and custom mode.
- `sqlt_snnnnn_restore.sql` – Imports the CBO statistics into the test case.
- `sqlt_snnnnn_set_cbo_env.sql` – Sets the CBO environment for the test case.
- `sqlt_snnnnn_system_stats.sql` – Sets the system statistics on the test system.
- `tc.sql` – Runs the test sql and displays the execution plan.
- `tc_pkg.sql` – Creates a `tc.zip` file for a small test case file.
- `xpress.sh` – The unix version of the script that runs `xpress.sql`. Builds the entire test case.
- `xpress.sql` – The SQL script that builds the entire test case in express mode.

As you can see there are quite a few files in the `TC` directory once you've unzipped it. Most of them are called from `xpress.sql` and `setup.sql` so we will not go into great detail on each of them, but some have some interesting stand-alone functions that we'll look at after we've built the test case.

The SQL Statement

Remember that SQLT is about tuning one SQL statement at a time. In our example case we will be using this SQL:

```
select
  country_name, sum(AMOUNT_SOLD)
from sales s, customers c, countries co
where
  s.cust_id=c.cust_id
  and co.country_id=c.country_id
  and country_name in (
    'Ireland','Denmark','Poland','United Kingdom',
     'Germany','France','Spain','The Netherlands','Italy')
  group by country_name order by sum(AMOUNT_SOLD);
```

This example SQL totals the spending from each country in the European Union. This is the result of running this query:

```
COUNTRY_NAME                             SUM(AMOUNT_SOLD)
---------------------------------------- ----------------
Poland                                            8447.14
Denmark                                        1977764.79
Spain                                          2090863.44
France                                         3776270.13
Italy                                          4854505.28
United Kingdom                                 6393762.94
Germany                                        9210129.22

7 rows selected.
```

What it does is not important for this example. We only need the SQL ID (`fp2wkm07dr0nd`) and a SQLT XTRACT report. The intention here is to build a test case in another schema and then adjust the environment so that we can test different theories in isolation. In the normal way (as discussed in Chapter 1 and Chapter 3) we collect a SQLT XTRACT report and then place the zip file in a convenient place. In this case in the `/run` directory.

How to Build a Test Case Fast (XPRESS.sql)

First we create a directory to put the SQLTXTRACT report files in. In this directory I created a `TC` directory and put the test case zip file in the `TC` directory. Now that we have a test case directory and we've unzipped the test case files the quickest and simplest thing to do is to set up the test case user. Remember you need to be logged into SYS to do this operation. This is because some of the steps in `xpress.sql` will change the database environment. This environment cannot be shared with anyone else. So with all the warnings out of the way let's proceed to create a test case.

First we'll run `xpress.sql`. This will pause in various sections to give you a chance to review the steps and to check for any errors. If there are none, then normally you would press return to go to the next section. The steps in the script are each highlighted by a heading such as this:

```
1/7 Press ENTER to create TC user and schema objects for statement_id 64661.
```

The following list provides a high-level view of the script's steps. Next, we'll look at each of the several steps in more detail:

1. Create the test case user, the user objects (including tables and indexes), the name of the user is of the form TCnnnn. In step 1 you are prompted for a suffix for the name of the test case user. You can just hit Return at this point, or enter a suffix such as DEV. At the end of this section you should check to see if there are any invalid objects. Valid objects are also listed.
2. The SQL repository is purged of any previous incarnation of the SQL statement.
3. The SQL statement is imported into the SQL repository and system environment is restored using the import utility. You will be prompted to enter the password for the SQLTXPLAIN user. This step is one of the reasons why you should not run xpress on a system you cannot afford to lose.
4. The test user schema object statistics are restored.
5. The system statistics are restored.
6. You are connected to the test schema and the CBO environment is set up.
7. The test schema environment is set up the SQL is executed and the execution plan displayed.

Once you reach this stage, assuming there are no errors along the way, you are free to make changes to the test case environment (remember that this is a system that can be junked after this testing). You can change whatever you need to change to improve the performance of your test case, or sometimes you may want to change something about the test case to more fully understand what is happening and why the execution plan is the way it is. Next, we'll look at each of the seven steps above in much more detail. We'll see example output and explain what is happening. We'll go through all the steps to set up a test case that you can work.

1. In step 1 you are asked to confirm that you want to create a test case user in my case TC64661 and the schema objects.

   ```
   1/7 Press ENTER to create TC user and schema objects for statement_id 64661.
   ```

If you press Enter you complete step 1. In this step the script `sqlt_snnnnn_metadata.sql` is run. You are prompted for a test case user suffix. A typical value might be "_1", but you can press RETURN to accept the default, which is no suffix. This then creates the metadata objects, such as tables and indexes as well as any constraints, functions, packages, views or any other metadata. At the end of this step you are shown the status of these objects. They should all be valid. Here is the screen from my example.

```
SQL>
SQL>
SQL> /*************************************************************************/
SQL>
SQL> REM PACKAGE
SQL>
SQL>
SQL> /*************************************************************************/
SQL> REM VIEW
SQL>
SQL>
SQL> /*************************************************************************/
SQL>
SQL> REM FUNCTION, PROCEDURE, LIBRARY and PACKAGE BODY
SQL>
SQL>
SQL> /*************************************************************************/
SQL>
SQL> REM OTHERS
SQL>
SQL>
SQL>
SQL> /*************************************************************************/
SQL>
SQL> SET ECHO OFF VER OFF PAGES 1000 LIN 80 LONG 8000000 LONGC 800000;

PL/SQL procedure successfully completed.


:VALID_OBJECTS
--------------------------------------------------------------------------------
VALID TABLE TC64661 COUNTRIES
VALID TABLE TC64661 CUSTOMERS
VALID TABLE TC64661 SALES
VALID INDEX TC64661 COUNTRIES_PK
VALID INDEX TC64661 CUSTOMERS_GENDER_BIX
VALID INDEX TC64661 CUSTOMERS_MARITAL_BIX
VALID INDEX TC64661 CUSTOMERS_PK
VALID INDEX TC64661 CUSTOMERS_YOB_BIX
VALID INDEX TC64661 SALES_CHANNEL_BIX
VALID INDEX TC64661 SALES_CUST_BIX
VALID INDEX TC64661 SALES_PROD_BIX
VALID INDEX TC64661 SALES_PROMO_BIX
VALID INDEX TC64661 SALES_TIME_BIX
```

```
:INVALID_OBJECTS
--------------------------------------------------------------------------------


SQL> REM In case of INVALID OBJECTS: review log, fix errors and execute again.
SQL> SPO OFF;
SQL> SET ECHO OFF;

2/7 Press ENTER to purge statement_id 64661 from SQLT repository.
```

In my example all my metadata objects are valid, and I have no packages, views, functions, or procedures. Since there are no invalid objects, I press Return to proceed with step2.

2. In step 2 sqlt_snnnnn_purge.sql is run. This purges the SQLT repository of any data related to the SQL statement that is being analyzed. It is common to reload a test case when needed. The script output looks like this

```
SQL> @@sqlt_s64661_purge.sql
SQL> REM Purges statement_id 64661 from local SQLT repository. Just execute
"@sqlt_s64661_purge.sql" from sqlplu
SQL> SPO sqlt_s64661_purge.log;
SQL> SET SERVEROUT ON;
SQL> EXEC sqltxplain.sqlt$a.purge_repository(64661, 64661);
15:13:56 sqlt$a: purging repository
15:13:57 sqlt$a: TRUNCATE TABLE SQLI$_DB_LINK
15:13:57 sqlt$a: TRUNCATE TABLE SQLI$_FILE
15:13:57 sqlt$a: TRUNCATE TABLE SQLI$_STATTAB_TEMP
15:13:57 sqlt$a: TRUNCATE TABLE SQLI$_STGTAB_SQLPROF
15:13:57 sqlt$a: TRUNCATE TABLE SQLI$_STGTAB_SQLSET
15:13:57 sqlt$a: TRUNCATE TABLE SQLT$_AUX_STATS$
15:13:57 sqlt$a: TRUNCATE TABLE SQLT$_DBA_AUDIT_POLICIES
15:13:57 sqlt$a: TRUNCATE TABLE SQLT$_DBA_AUTOTASK_CLIENT
15:13:57 sqlt$a: TRUNCATE TABLE SQLT$_DBA_COL_STATS_VERSIONS
15:13:58 sqlt$a: TRUNCATE TABLE SQLT$_DBA_COL_USAGE$
15:13:58 sqlt$a: TRUNCATE TABLE SQLT$_DBA_CONSTRAINTS
15:13:58 sqlt$a: TRUNCATE TABLE SQLT$_DBA_DEPENDENCIES
15:13:58 sqlt$a: TRUNCATE TABLE SQLT$_DBA_HISTGRM_STATS_VERSN
15:13:58 sqlt$a: TRUNCATE TABLE SQLT$_DBA_HIST_ACTIVE_SESS_HIS
```

At the end of this section you are prompted to proceed with step 3.

```
15:14:03 sqlt$a: 128 tables were truncated
15:14:03 sqlt$a: -> delete_sqltxplain_stats
15:14:07 sqlt$a: <- delete_sqltxplain_stats

PL/SQL procedure successfully completed.

SQL> SET SERVEROUT OFF;
SQL> SPO OFF;
SQL> SET ECHO OFF;

3/7 Press ENTER to import SQLT repository for statement_id 64661.
```

3. Step 3 imports the data collected from the target system into the SQLT repository. Here is the prompt that you see.

   ```
   SQL> HOS imp sqltxplain FILE=sqlt_s64661_exp.dmp TABLES=sqlt% IGNORE=Y

   Import: Release 11.2.0.1.0 - Production on Sat Dec 15 15:20:18 2012

   Copyright (c) 1982, 2009, Oracle and/or its affiliates.  All rights reserved.
   Password:
   ```

You need to enter the password that you set for SQLTXPLAIN when you installed the utility. When you enter the password and press Enter the import begins. Below is an example of this.

```
Connected to: Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - Production
With the Partitioning, OLAP, Data Mining and Real Application Testing options

Export file created by EXPORT:V11.02.00 via conventional path
import done in WE8MSWIN1252 character set and AL16UTF16 NCHAR character set
. importing SQLTXPLAIN's objects into SQLTXPLAIN
. importing SQLTXPLAIN's objects into SQLTXPLAIN
. . importing table          "SQLT$_SQL_STATEMENT"          1 rows imported
. . importing table             "SQLT$_AUX_STATS$"         13 rows imported
. . importing table    "SQLT$_DBA_AUTOTASK_CLIENT"          1 rows imported
. . importing table "SQLT$_DBA_COL_STATS_VERSIONS"        452 rows imported
. . importing table          "SQLT$_DBA_COL_USAGE$"        14 rows imported
. . importing table         "SQLT$_DBA_CONSTRAINTS"        39 rows imported
. . importing table     "SQLT$_DBA_HIST_PARAMETER"      75307 rows imported
. . importing table   "SQLT$_DBA_HIST_PARAMETER_M"         22 rows imported
. . importing table      "SQLT$_DBA_HIST_SNAPSHOT"        214 rows imported
. . importing table       "SQLT$_DBA_HIST_SQLTEXT"          1 rows imported
. . importing table "SQLT$_DBA_HISTGRM_STATS_VERSN"       2983 rows imported
. . importing table         "SQLT$_DBA_IND_COLUMNS"        10 rows imported
. . importing table     "SQLT$_DBA_IND_PARTITIONS"        140 rows imported
. . importing table     "SQLT$_DBA_IND_STATISTICS"        150 rows imported
. . importing table "SQLT$_DBA_IND_STATS_VERSIONS"        298 rows imported
. . importing table             "SQLT$_DBA_INDEXES"        10 rows imported
. . importing table             "SQLT$_DBA_OBJECTS"       181 rows imported
. . importing table "SQLT$_DBA_OPTSTAT_OPERATIONS"        471 rows imported
. . importing table "SQLT$_DBA_PART_COL_STATISTICS"        196 rows imported
. . importing table    "SQLT$_DBA_PART_HISTOGRAMS"       2942 rows imported
. . importing table   "SQLT$_DBA_PART_KEY_COLUMNS"          6 rows imported
. . importing table            "SQLT$_DBA_SEGMENTS"       175 rows imported
. . importing table            "SQLT$_DBA_TAB_COLS"        40 rows imported
. . importing table     "SQLT$_DBA_TAB_HISTOGRAMS"        524 rows imported
. . importing table     "SQLT$_DBA_TAB_PARTITIONS"         28 rows imported
. . importing table     "SQLT$_DBA_TAB_STATISTICS"         31 rows imported
. . importing table "SQLT$_DBA_TAB_STATS_VERSIONS"         60 rows imported
. . importing table              "SQLT$_DBA_TABLES"         3 rows imported
. . importing table         "SQLT$_DBA_TABLESPACES"         6 rows imported
. . importing table              "SQLT$_DBMS_XPLAN"        66 rows imported
. . importing table      "SQLT$_GV$NLS_PARAMETERS"         19 rows imported
```

```
. . importing table          "SQLT$_GV$PARAMETER2"        345 rows imported
. . importing table       "SQLT$_GV$PARAMETER_CBO"        275 rows imported
. . importing table          "SQLT$_GV$PQ_SYSSTAT"         20 rows imported
. . importing table  "SQLT$_GV$PX_PROCESS_SYSSTAT"         15 rows imported
. . importing table    "SQLT$_GV$SYSTEM_PARAMETER"        344 rows imported
. . importing table                    "SQLT$_LOG"       1522 rows imported
. . importing table               "SQLT$_METADATA"        117 rows imported
. . importing table "SQLT$_NLS_DATABASE_PARAMETERS"         20 rows imported
. . importing table           "SQLT$_OUTLINE_DATA"         26 rows imported
. . importing table         "SQLT$_PLAN_EXTENSION"          9 rows imported
. . importing table              "SQLT$_PLAN_INFO"          6 rows imported
. . importing table         "SQLT$_SQL_PLAN_TABLE"          9 rows imported
. . importing table                "SQLT$_STATTAB"       3626 rows imported
. . importing table  "SQLT$_V$SESSION_FIX_CONTROL"        406 rows imported
. . importing table "SQLT$_WRI$_OPTSTAT_AUX_HISTORY"        243 rows imported
Import terminated successfully without warnings.

SQL> SET ECHO OFF;

4/7 Press ENTER to restore schema object stats for TC64661.
```

As you can see from the list of objects imported, step 3 has imported information captured by SQLT during the SQLTXTRACT and is now storing it in the SQLT repository. Then you are prompted to proceed to step 4, which will restore the statistics for the test case objects.

4. Press Enter to proceed to step 4. In step 4 we replace the data dictionary information for the test case from the SQLT repository. This is why the system you are doing this on has to be one that you can re-create. Here is what you see for step 4.

```
SQL> @@sqlt_s64661_restore.sql
SQL> REM Restores schema object stats for statement_id 64661 from local SQLT repository
into data dictionary. Just execute "@sqlt_s64661_resore.sql" from sqlplus.
SQL> SPO sqlt_s64661_restore.log;
SQL> SET SERVEROUT ON;
SQL> EXEC sqltxplain.sqlt$a.import_cbo_stats(p_statement_id => 's64661', p_schema_owner =>
'&&tc_user.', p_include_bk => 'N');
remapping stats into user TC64661(120)
obtain statistics staging table version for this system
statistics version for this system: 5
+-----+
upgrade/downgrade of sqli$_stattab_temp to version 5 as per this system
restoring cbo stats for table TC64661.COUNTRIES
restoring cbo stats for table TC64661.CUSTOMERS
restoring cbo stats for table TC64661.SALES
deleting conflicting rows from tables:
wri$_optstat_histgrm_history, _histhead_history, and _tab_history
deleting conflicting wri$_optstat_ind_history
restoring wri$_optstat_tab_history
restoring wri$_optstat_ind_history
restoring wri$_optstat_histhead_history
restoring wri$_optstat_histgrm_history
deleted 30 rows from wri$_optstat_tab_history
deleted 18 rows from wri$_optstat_ind_history
deleted 226 rows from wri$_optstat_histhead_history
deleted 0 rows from wri$_optstat_histgrm_history
restored 60 rows into wri$_optstat_tab_history
restored 298 rows into wri$_optstat_ind_history
restored 452 rows into wri$_optstat_histhead_history
restored 2983 rows into wri$_optstat_histgrm_history
+
|
|   Stats from id "s64661_snc1_locutus"
|   have been restored into data dict
|
|           METRIC   IN STATTAB  RESTORED  OK
|     -------------  ----------  --------  --
|       STATS ROWS:        3626      3626  OK
|           TABLES:           3         3  OK
|       TABLE PART:          28        28  OK
|    TABLE SUBPART:           0         0  OK
|          INDEXES:          10        10  OK
|       INDEX PART:         140       140  OK
|    INDEX SUBPART:           0         0  OK
|          COLUMNS:         498       498  OK
|      COLUMN PART:        2947      2947  OK
|   COLUMN SUBPART:           0         0  OK
|     AVG AGE DAYS:        14.5      14.5  OK
|
+

PL/SQL procedure successfully completed.

SQL> SET SERVEROUT OFF;
SQL> SPO OFF;
SQL> SET ECHO OFF;

5/7 Press ENTER to restore system statistics.
```

We just imported object statistics into TC64661 (from the repository) into the system so that they are the statistics for the test schema. At the end of this process we see that each object's statistics were imported successfully, and then we are prompted to proceed to step 5.

5. Now in step 5 we delete the existing system statistics (did I mention that you do this only on a system you can replace with no production data and no other users?). Then the new values are set for the system statistics. Then you are prompted to proceed to step 6.

```
SQL> EXEC DBMS_STATS.DELETE_SYSTEM_STATS;

PL/SQL procedure successfully completed.

SQL> EXEC DBMS_STATS.SET_SYSTEM_STATS('CPUSPEEDNW', 1683.65129195846);

PL/SQL procedure successfully completed.
```

```
SQL> EXEC DBMS_STATS.SET_SYSTEM_STATS('IOSEEKTIM', 10);

PL/SQL procedure successfully completed.

SQL> EXEC DBMS_STATS.SET_SYSTEM_STATS('IOTFRSPEED', 4096);

PL/SQL procedure successfully completed.

SQL>
SQL> SPO OFF;
SQL> SET ECHO OFF;

6/7 Press ENTER to connect as TC64661 and set CBO env.
```

6. In step 6 we connect as the test user. You see this line in the output

```
SQL> CONN &&tc_user./&&tc_user.
Connected.
SQL> @@sqlt_s64661_set_cbo_env.sql
```

The script sqlt_s64661_set_cbo_env.sql will set the CBO environment. It is an important and you will be prompted before you run it.

```
SQL> ALTER SESSION SET optimizer_features_enable = '11.2.0.1';

Session altered.

SQL>
SQL> SET ECHO OFF;

Press ENTER to execute ALTER SYSTEM/SESSION commands to set CBO env.
```

When you press Enter at this point, *all* of the CBO environment settings are set to those from the system where the SQL came from. In my case the log file contains the following at the top:

```
/**************************************************************************************/
SQL>
SQL> REM Non-Default or Modified Parameters
SQL>
SQL> -- enable modification monitoring. isdefault="TRUE" ismodified="SYSTEM_MOD"
issys_modifiable="IMMEDIATE"
SQL> ALTER SYSTEM SET "_dml_monitoring_enabled" = TRUE SCOPE=MEMORY;

System altered.

SQL>
SQL> -- optimizer secure view merging and predicate pushdown/movearound. isdefault="TRUE"
ismodified="SYSTEM_MOD" issys_modifiable="IMMEDIATE"
SQL> ALTER SYSTEM SET optimizer_secure_view_merging = TRUE SCOPE=MEMORY;

System altered.
```

```
SQL>
SQL> -- number of CPUs for this instance. isdefault="TRUE" ismodified="SYSTEM_MOD"
issys_modifiable="IMMEDIATE"
SQL> ALTER SYSTEM SET cpu_count = 2 SCOPE=MEMORY;

System altered.

SQL> -- number of parallel execution threads per CPU. isdefault="TRUE" ismodified="SYSTEM_MOD"
issys_modifiable="IMMEDIATE"
SQL> ALTER SYSTEM SET parallel_threads_per_cpu = 2 SCOPE=MEMORY;

System altered.

SQL>
SQL> -- Maximum size of the PGA memory for one process. isdefault="TRUE" ismodified="SYSTEM_MOD"
issys_modifiable="IMMEDIATE"
SQL> ALTER SYSTEM SET "_pga_max_size" = 209715200 SCOPE=MEMORY;

System altered.

SQL>
SQL> -- optimizer use feedback. isdefault="TRUE" ismodified="SYSTEM_MOD"
SQL> ALTER SESSION SET "_optimizer_use_feedback" = TRUE;

Session altered.

SQL>
SQL> -- optimizer dynamic sampling. isdefault="TRUE" ismodified="SYSTEM_MOD"
SQL> ALTER SESSION SET optimizer_dynamic_sampling = 0;

Session altered.
```

Notice how we are changing system parameters. For example `optimizer_dynamic_sampling` is set 0. This is not the default, as we learned in chapter 8. Apart from non-default system parameters we also set the session parameters. Here is a section from the log file where we set a number of hidden session parameters:

```
SQL>
SQL> -- compute join cardinality using non-rounded input values
SQL> ALTER SESSION SET "_optimizer_new_join_card_computation" = TRUE;

Session altered.

SQL>
SQL> -- null-aware antijoin parameter
SQL> ALTER SESSION SET "_optimizer_null_aware_antijoin" = TRUE;

Session altered.

SQL> -- Use subheap for optimizer or-expansion
SQL> ALTER SESSION SET "_optimizer_or_expansion_subheap" = TRUE;

Session altered.
```

```
SQL>
SQL> -- Eliminates order bys from views before query transformation
SQL> ALTER SESSION SET "_optimizer_order_by_elimination_enabled" = TRUE;

Session altered.
```

Notice how in the preceding example hidden parameters are included. We even set the `fix_control` parameters, in the next example. The `fix_control` parameters control whether certain bug fixes (included with the database) are enabled or disabled. Here is a section from the fix control part of the log file (we'll talk more about fix control in the next chapter).

```
SQL>
SQL> -- remove distribution method optimization for insert/update qbc (ofe 11.2.0.1) (event 0)
SQL> ALTER SESSION SET "_fix_control" = '6376551:1';

Session altered.

SQL>
SQL> -- Convert outer-join to inner-join if single set aggregate functio (ofe 11.1.0.7) (event 0)
SQL> ALTER SESSION SET "_fix_control" = '6377505:1';

Session altered.
At the end of step 6 we are prompted to execute the test case.
SQL> /**********************************
SQL>
SQL> SPO OFF;
SQL> SET ECHO OFF;

7/7 Press ENTER to execute test case.
```

7. In step 7 we finally get to execute the SQL from our test case. Executing the test case as the test case user will output the result of the query and then the execution plan for the query. The result in our example will look something like this.

```
SQL> @@tc.sql
SQL> REM Executes SQL on TC then produces execution plan. Just execute "@tc.sql" from sqlplus.
SQL> SET APPI OFF SERVEROUT OFF;
SQL> @@q.sql
SQL> REM $Header: 215187.1 sqlt_s64661_tc_script.sql 11.4.4.6 2012/12/13 carlos.sierra $
SQL>
SQL> select
  2   /* ^^unique_id */  country_name, sum(AMOUNT_SOLD)
  3  from sh.sales s, sh.customers c, sh.countries co
  4  where
  5    s.cust_id=c.cust_id
  6    and co.country_id=c.country_id
  7    and country_name in (
  8      'Ireland','Denmark','Poland','United Kingdom',
  9       'Germany','France','Spain','The Netherlands','Italy')
 10    group by country_name order by sum(AMOUNT_SOLD);
```

```
COUNTRY_NAME                             SUM(AMOUNT_SOLD)
---------------------------------------- ----------------
Poland                                            8447.14
Denmark                                        1977764.79
Spain                                          2090863.44
France                                         3776270.13
Italy                                          4854505.28
United Kingdom                                 6393762.94
Germany                                        9210129.22

7 rows selected.

SQL> @@plan.sql
SQL> REM Displays plan for most recently executed SQL. Just execute "@plan.sql" from sqlplus.
SQL> SET PAGES 2000 LIN 180;
SQL> SPO plan.txt;
SQL> SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR);

PLAN_TABLE_OUTPUT
----------------------------------------------------------------------------------------
---------------------------------------
SQL_ID  0zskj9bd07jdy, child number 0
-------------------------------------
select  /* ^^unique_id */  country_name, sum(AMOUNT_SOLD) from sh.sales
s, sh.customers c, sh.countries co where   s.cust_id=c.cust_id   and
co.country_id=c.country_id   and country_name in (
'Ireland','Denmark','Poland','United Kingdom',
'Germany','France','Spain','The Netherlands','Italy')   group by
country_name order by sum(AMOUNT_SOLD)

Plan hash value: 2938593747
----------------------------------------------------------------------------------------
|Id |Operation               |Name     |Rows  |Bytes |Cost(%CPU)|Time    |Pstart| Pstop |
----------------------------------------------------------------------------------------
|  0|SELECT STATEMENT        |         |      |      |  947(100)|        |      |       |
|  1| SORT ORDER BY          |         |    9 |  315 |  947  (7)|00:00:12|      |       |
|  2|  HASH GROUP BY         |         |    9 |  315 |  947  (7)|00:00:12|      |       |
|* 3|   HASH JOIN            |         | 435K|   14M|  909  (3)|00:00:11|      |       |
|* 4|    HASH JOIN           |         |26289 | 641K|  409  (1)|00:00:05|      |       |
|* 5|     TABLE ACCESS FULL  |COUNTRIES|    9 |  135 |    3  (0)|00:00:01|      |       |
|  6|     TABLE ACCESS FULL  |CUSTOMERS|55500 | 541K|  406  (1)|00:00:05|      |       |
|  7|    PARTITION RANGE ALL|         | 918K| 8973K|  494  (3)|00:00:06|    1 |    28 |
|  8|     TABLE ACCESS FULL  |SALES    | 918K| 8973K|  494  (3)|00:00:06|    1 |    28 |
----------------------------------------------------------------------------------------

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

   3 - access("S"."CUST_ID"="C"."CUST_ID")
   4 - access("CO"."COUNTRY_ID"="C"."COUNTRY_ID")
   5 - filter(("COUNTRY_NAME"='Denmark' OR "COUNTRY_NAME"='France' OR
              "COUNTRY_NAME"='Germany' OR "COUNTRY_NAME"='Ireland' OR "COUNTRY_NAME"='Italy' OR
              "COUNTRY_NAME"='Poland' OR "COUNTRY_NAME"='Spain' OR "COUNTRY_NAME"=
              'The Netherlands' OR
              "COUNTRY_NAME"='United Kingdom'))
```

Notice how in my case there are results from the query because I retained the schema name in the SQL, and the data happened to exist on the system in the same schema. This would not normally be the case, and the result of the query is usually to return no data but with the same execution plan. Let me repeat that. The test case can work, with the same execution plan and with no data. The CBO only works based on what the statistics say about the tables, and we replaced that statistical data during the import steps. Since normally there is no data in your test case system you will get an output such as this (here I have replaced sh.sales, sh.countries, and sh.customers with sales, countries and customers, as these tables now exist in the test case schema (but with no data in them).

```
SQL> @tc
SQL> REM Executes SQL on TC then produces execution plan. Just execute "@tc.sql" from sqlplus.
SQL> SET APPI OFF SERVEROUT OFF;
SQL> @@q.sql
SQL> REM $Header: 215187.1 sqlt_s64661_tc_script.sql 11.4.4.6 2012/12/13 carlos.sierra $
SQL>
SQL>
SQL> select
  2   /* ^^unique_id */  country_name, sum(AMOUNT_SOLD)
  3  from sales s, customers c, countries co
  4  where
  5    s.cust_id=c.cust_id
  6    and co.country_id=c.country_id
  7    and country_name in (
  8      'Ireland','Denmark','Poland','United Kingdom',
  9       'Germany','France','Spain','The Netherlands','Italy')
 10    group by country_name order by sum(AMOUNT_SOLD);

no rows selected <<<No Data was returned. There is none.

SQL> @@plan.sql
SQL> REM Displays plan for most recently executed SQL. Just execute "@plan.sql" from sqlplus.
SQL> SET PAGES 2000 LIN 180;
SQL> SPO plan.txt;
SQL> SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR);

PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------------------
SQL_ID  f43bszax8xh07, child number 0
-------------------------------------
select  /* ^^unique_id */  country_name, sum(AMOUNT_SOLD) from sales s,
customers c, countries co where   s.cust_id=c.cust_id   and
co.country_id=c.country_id   and country_name in (
'Ireland','Denmark','Poland','United Kingdom',
'Germany','France','Spain','The Netherlands','Italy')   group by
country_name order by sum(AMOUNT_SOLD)
```

```
Plan hash value: 2938593747
---------------------------------------------------------------------------------------
|Id |Operation             |Name     |Rows  |Bytes |Cost(%CPU)|Time     |Pstart| Pstop |
---------------------------------------------------------------------------------------
|  0|SELECT STATEMENT      |         |      |      |  947(100)|         |      |       |
|  1| SORT ORDER BY        |         |    9 |  315 |  947  (7)|00:00:12|      |       |
|  2|  HASH GROUP BY       |         |    9 |  315 |  947  (7)|00:00:12|      |       |
|* 3|   HASH JOIN          |         |  435K|   14M|  909  (3)|00:00:11|      |       |
|* 4|    HASH JOIN         |         |26289 |  641K|  409  (1)|00:00:05|      |       |
|* 5|     TABLE ACCESS FULL |COUNTRIES|    9 |  135 |    3  (0)|00:00:01|      |       |
|  6|     TABLE ACCESS FULL |CUSTOMERS|55500 |  541K|  406  (1)|00:00:05|      |       |
|  7|    PARTITION RANGE ALL|         |  918K| 8973K|  494  (3)|00:00:06|    1 |    28 |
|  8|     TABLE ACCESS FULL |SALES    |  918K| 8973K|  494  (3)|00:00:06|    1 |    28 |
---------------------------------------------------------------------------------------

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

   3 - access("S"."CUST_ID"="C"."CUST_ID")
   4 - access("CO"."COUNTRY_ID"="C"."COUNTRY_ID")
   5 - filter(("COUNTRY_NAME"='Denmark' OR "COUNTRY_NAME"='France' OR
               "COUNTRY_NAME"='Germany' OR "COUNTRY_NAME"='Ireland' OR
               "COUNTRY_NAME"='Italy' OR "COUNTRY_NAME"='Poland' OR
               "COUNTRY_NAME"='Spain' OR "COUNTRY_NAME"='The Netherlands' OR
               "COUNTRY_NAME"='United Kingdom'))
```

Now we have reached that point in the test case where we can do some work on exploring the set up of the environment and changing things to see what we can achieve or to see how the SQL, under the microscope can be influenced to do what we want.

Exploring the Execution Plan

Now that we have a test case, we can explore the execution plan by changing all those things that can affect the execution plan. I've listed some of them below:

- Optimizer parameters
- SQL hints
- Optimizer versions
- Structure of the SQL
- Adding or removing indexes
- System statistics
- Object Statistics

Of these environmental factors, the last two require the setting of object statistics through routines. Setting object statistics can be done but is beyond the scope of this book. System statistics can also be set and tested; but again, this is not generally a test that is carried out because it implies you are planning for a different machine. (I give a short example of this at the end of the section.) The things most commonly explored using the test case are optimizer parameters and the structure of the SQL.

While this kind of testing can be done in other ways (set autotrace, EXPLAIN PLAN, etc.), setting up the environment can be time consuming and tricky. Even if you do set up an environment to be a duplicate of another environment, how will you know you've got everything right? The optimizer works the same way and produces an execution plan when you test. With a SQLT test case everything is brought in, and nothing is missing. You get the environment right every time. If you copy the data you will also have a very similar copy of the source environment. An additional advantage of this method of working is that the system environment can also be changed.

Optimizer Parameters

If a developer comes to you and says my SQL would work fine if `optimizer_index_cost_adj` was set to 10, you can now test the suggestion with a test case in your own test environment. Of course, good (or bad) suggestions for tuning can come from any source; the point here is that you can take a suggestion, put it through the optimizer via the test case you built, and see what the cost and execution plan will be. Below we see such an example. Notice that I am setting the optimizer parameter at the session level, but this would normally be a pre-cursor to setting the parameter at the system level. By the way I would not suggest you make changes like this on a real system without extensive testing, there is too much scope for other SQL to be affected by changes like this. This is an exploration of what the optimizer might be able to do for us if the environment were different. So here's the result.

```
Plan hash value: 2917593948

-----------------------------------------------------------------------------------------------
|Id |Operation                          |Name                 |Rows |Cost(%CPU)|Pstart|Pstop |
-----------------------------------------------------------------------------------------------
|  0|SELECT STATEMENT                   |                     |     |  593(100)|      |      |
|  1| SORT ORDER BY                     |                     |    9|  593 (12)|      |      |
|  2|  HASH GROUP BY                    |                     |    9|  593 (12)|      |      |
|* 3|   HASH JOIN                       |                     | 435K|  555  (6)|      |      |
|* 4|    HASH JOIN                      |                     |26289|  243  (2)|      |      |
|* 5|     TABLE ACCESS FULL             |COUNTRIES            |    9|    3  (0)|      |      |
|  6|     TABLE ACCESS BY INDEX ROWID   |CUSTOMERS            |55500|  239  (1)|      |      |
|  7|      BITMAP CONVERSION TO ROWIDS  |                     |     |          |      |      |
|  8|       BITMAP INDEX FULL SCAN      |CUSTOMERS_GENDER_BIX|     |          |      |      |
|  9|    PARTITION RANGE ALL            |                     | 918K|  306  (7)|     1|    28 |
| 10|     TABLE ACCESS BY LOCAL INDEX ROWID|SALES             | 918K|  306  (7)|     1|    28 |
| 11|      BITMAP CONVERSION TO ROWIDS  |                     |     |          |      |      |
| 12|       BITMAP INDEX FULL SCAN      |SALES_PROMO_BIX      |     |          |     1|    28 |
-----------------------------------------------------------------------------------------------

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

   3 - access("S"."CUST_ID"="C"."CUST_ID")
   4 - access("CO"."COUNTRY_ID"="C"."COUNTRY_ID")
   5 - filter(("COUNTRY_NAME"='Denmark' OR "COUNTRY_NAME"='France' OR
              "COUNTRY_NAME"='Germany' OR "COUNTRY_NAME"='Ireland' OR
              "COUNTRY_NAME"='Italy' OR "COUNTRY_NAME"='Poland' OR
              "COUNTRY_NAME"='Spain' OR "COUNTRY_NAME"='The Netherlands' OR
              "COUNTRY_NAME"='United Kingdom'))
```

In this example, I've removed some of the execution plan columns for clarity. We can see that the cost has gone down from 947 to 593. Let me emphasize this next point, as it is very important. This does not mean that changing `optimizer_index_cost_adj` is a good idea: rather, this means that if the optimizer environment were set correctly the optimizer could choose a better plan. The next step in this scenario would be to investigate why the optimizer is choosing a sub-optimal plan. The important fact to remember here is that the test case allows you to confirm that there is a better plan.

Adding and Removing Hints

In my previous example we saw that setting session parameters can change the execution plan and that on our test system we can see those changes without having all the data. We saw that setting `optimizer_index_cost_adj` made a difference and that maybe we need an index hint to help. So setting `optimizer_index_cost_adj` back to the default value of 100, and then at random I selected the index SALES_CUST_BIX (which we'll see wasn't a good choice). I modify the SQL to include the hint `/*+ INDEX(S SALES_CUST_BIX) */` and run `tc.sql`. The result in my case is:

```
---------------------------------------------------------------------
|Id |Operation                          |Name          |Cost(%CPU)|
---------------------------------------------------------------------
| 0|SELECT STATEMENT                    |              |3787(100) |
| 1| SORT ORDER BY                      |              |3787  (2) |
| 2|  HASH GROUP BY                     |              |3787  (2) |
|*3|   HASH JOIN                        |              |3749  (1) |
|*4|    HASH JOIN                       |              | 409  (1) |
|*5|     TABLE ACCESS FULL              |COUNTRIES     |   3  (0) |
| 6|     TABLE ACCESS FULL              |CUSTOMERS     | 406  (1) |
| 7|    PARTITION RANGE ALL             |              | 3334 (1) |
| 8|    TABLE ACCESS BY LOCAL INDEX ROWID |SALES       | 3334 (1) |
| 9|      BITMAP CONVERSION TO ROWIDS   |              |          |
|10|       BITMAP INDEX FULL SCAN       |SALES_CUST_BIX|          |
---------------------------------------------------------------------

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

   3 - access("S"."CUST_ID"="C"."CUST_ID")
   4 - access("CO"."COUNTRY_ID"="C"."COUNTRY_ID")
   5 - filter(("COUNTRY_NAME"='Denmark' OR "COUNTRY_NAME"='France' OR
               "COUNTRY_NAME"='Germany' OR "COUNTRY_NAME"='Ireland' OR
               "COUNTRY_NAME"='Italy' OR "COUNTRY_NAME"='Poland' OR
               "COUNTRY_NAME"='Spain' OR "COUNTRY_NAME"='The Netherlands' OR
               "COUNTRY_NAME"='United Kingdom'))
```

Again, I've removed some columns from the execution plan display to emphasize the important columns. We see in this example that the cost has increased, and so we decide that this hint is not helpful to us. The steps to check the effect of this hint are to modify `q.sql` (add the hint) and run `tc.sql`. Nothing more is needed.

Versions of the optimizer

Suppose we upgraded or transferred SQL from version 10g to version 11g. We think that maybe the execution plan changed between these two versions. We can test this idea by changing `optimizer_features_enable` to `'10.2.0.5'`. Then we can do that easily and simply by setting the parameter at the session level and retesting.

The steps to carry out this test are:

```
SQL> alter session set optimizer_features_enable='10.2.0.5';
SQL> @tc
```

In this case we see that the execution plan is unchanged, but in other cases we may see some changes, usually the better execution plan is from 11g although not always. Remember that `optimizer_features_enable` is like a super-switch that sets a number of features in the database on or off. Setting this parameter to a value that was there before the feature was introduced can turn off any new features. For example, you could disable SQL Plan management by setting this parameter to 10.2.0.5, a version of the database that existed when this feature was not present. This is not a recommended solution to any particular problem, but will help in giving the developer or DBA a clue, if it improves a particular SQL, that the solution may lie in investigating one of the new features introduced in the later version. As the `optimizer_features_enable` parameter made no difference we set it back to 11.2.0.1.

Structure of the SQL

The test case once built allows you to investigate the target SQL in many ways, including changing the SQL itself, as long as your SQL does not include any new tables (which were not captured by XTRACT) you could test the SQL in the same way. In this example I changed the SQL from

```
select /*+ INDEX(S SALES_CUST_BIX) */
  country_name, sum(AMOUNT_SOLD)
from sales s, customers c, countries co
where
  s.cust_id=c.cust_id
  and co.country_id=c.country_id
  and country_name in ('Ireland','Denmark','Poland',
  'United Kingdom','Germany','France','Spain','The Netherlands','Italy')
  group by country_name order by sum(AMOUNT_SOLD);
```

to

```
select /*+ INDEX(S SALES_CUST_BIX) */
  country_name, sum(AMOUNT_SOLD)
from sales s, customers c, countries co
where
  s.cust_id=c.cust_id
  and co.country_id=c.country_id
  and country_name in ( select country_name from sh.countries where
  country_name in ('Ireland','Denmark','Poland',
  'United Kingdom','Germany','France','Spain','The Netherlands','Italy'))
  group by country_name order by sum(AMOUNT_SOLD);
```

In this case there was no change to the execution plan, the optimizer used a query optimization feature (as we discussed in Chapter 5), to give us the same execution plan. So we can remove this change and try some changes to the indexes.

Indexes

We currently are using an index due to this hint `/*+ INDEX(S SALES_CUST_BIX) */`

We see that the execution plan looks like this:

```
-----------------------------------------------------------------------------------------
|Id |Operation                             |Name          |Rows |Cost(%CPU)|Pstart| Pstop |
-----------------------------------------------------------------------------------------
|  0|SELECT STATEMENT                      |              |     | 3793(100)|      |       |
|  1| SORT ORDER BY                        |              |    9| 3793  (2)|      |       |
|  2|  HASH GROUP BY                       |              |    9| 3793  (2)|      |       |
|* 3|   HASH JOIN RIGHT SEMI               |              | 435K| 3755  (1)|      |       |
|* 4|    TABLE ACCESS FULL                 |COUNTRIES     |    9|    3  (0)|      |       |
|* 5|    HASH JOIN                         |              | 435K| 3749  (1)|      |       |
|* 6|     HASH JOIN                        |              |26289|  409  (1)|      |       |
|* 7|      TABLE ACCESS FULL               |COUNTRIES     |    9|    3  (0)|      |       |
|  8|      TABLE ACCESS FULL               |CUSTOMERS     |55500|  406  (1)|      |       |
|  9|     PARTITION RANGE ALL              |              | 918K| 3334  (1)|    1 |    28 |
| 10|      TABLE ACCESS BY LOCAL INDEX ROWID|SALES         | 918K| 3334  (1)|    1 |    28 |
| 11|       BITMAP CONVERSION TO ROWIDS    |              |     |          |      |       |
| 12|        BITMAP INDEX FULL SCAN        |SALES_CUST_BIX|     |          |    1 |    28 |
-----------------------------------------------------------------------------------------

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

   3 - access("COUNTRY_NAME"="COUNTRY_NAME")
   4 - filter(("COUNTRY_NAME"='Denmark' OR "COUNTRY_NAME"='France' OR
              "COUNTRY_NAME"='Germany' OR "COUNTRY_NAME"='Ireland' OR
              "COUNTRY_NAME"='Italy' OR "COUNTRY_NAME"='Poland'  OR
              "COUNTRY_NAME"='Spain' OR "COUNTRY_NAME"='The Netherlands' OR
              "COUNTRY_NAME"='United Kingdom'))
   5 - access("S"."CUST_ID"="C"."CUST_ID")
   6 - access("CO"."COUNTRY_ID"="C"."COUNTRY_ID")
   7 - filter(("COUNTRY_NAME"='Denmark' OR "COUNTRY_NAME"='France' OR
              "COUNTRY_NAME"='Germany' OR "COUNTRY_NAME"='Ireland' OR
              "COUNTRY_NAME"='Italy' OR "COUNTRY_NAME"='Poland' OR
              "COUNTRY_NAME"='Spain' OR "COUNTRY_NAME"='The Netherlands' OR
              "COUNTRY_NAME"='United Kingdom'))
```

But we suspect that the index use was a bad idea, so we want to disable it without changing the code.

```
SQL> alter index sales_cust_bix invisible;
SQL> @tc
```

```
-------------------------------------------------------------------------------------------
|Id |Operation                          |Name           | Rows  |Cost(%CPU)| Pstart| Pstop |
-------------------------------------------------------------------------------------------
|  0|SELECT STATEMENT                   |               |       | 3347(100)|       |       |
|  1| SORT ORDER BY                     |               |     9 | 3347  (3)|       |       |
|  2|  HASH GROUP BY                    |               |     9 | 3347  (3)|       |       |
|* 3|   HASH JOIN RIGHT SEMI            |               |   435K| 3309  (1)|       |       |
|* 4|    TABLE ACCESS FULL              |COUNTRIES      |     9 |    3  (0)|       |       |
|* 5|    HASH JOIN                      |               |   435K| 3303  (1)|       |       |
|* 6|     HASH JOIN                     |               | 26289 |  409  (1)|       |       |
|* 7|      TABLE ACCESS FULL            |COUNTRIES      |     9 |    3  (0)|       |       |
|  8|      TABLE ACCESS FULL            |CUSTOMERS      | 55500 |  406  (1)|       |       |
|  9|     PARTITION RANGE ALL           |               |   918K| 2889  (1)|     1 |    28 |
| 10|      TABLE ACCESS BY LOCAL INDEX ROWID|SALES      |   918K| 2889  (1)|     1 |    28 |
| 11|       BITMAP CONVERSION TO ROWIDS |               |       |          |       |       |
| 12|        BITMAP INDEX FULL SCAN     |SALES_PROMO_BIX|       |          |     1 |    28 |
-------------------------------------------------------------------------------------------

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

   3 - access("COUNTRY_NAME"="COUNTRY_NAME")
   4 - filter(("COUNTRY_NAME"='Denmark' OR "COUNTRY_NAME"='France' OR "COUNTRY_NAME"='Germany' OR
              "COUNTRY_NAME"='Ireland' OR "COUNTRY_NAME"='Italy' OR "COUNTRY_NAME"='Poland' OR
              "COUNTRY_NAME"='Spain' OR
              "COUNTRY_NAME"='The Netherlands' OR "COUNTRY_NAME"='United Kingdom'))
   5 - access("S"."CUST_ID"="C"."CUST_ID")
   6 - access("CO"."COUNTRY_ID"="C"."COUNTRY_ID")
   7 - filter(("COUNTRY_NAME"='Denmark' OR "COUNTRY_NAME"='France' OR "COUNTRY_NAME"='Germany' OR
              "COUNTRY_NAME"='Ireland' OR "COUNTRY_NAME"='Italy' OR "COUNTRY_NAME"='Poland' OR
              "COUNTRY_NAME"='Spain' OR
              "COUNTRY_NAME"='The Netherlands' OR "COUNTRY_NAME"='United Kingdom'))
```

As expected the index `sales_cust_bix` is no longer being used. We now see a different execution plan, which looks slightly better (cost 3347 compared to the previous 3793). Finally we remove the hint and decide that we need a machine twice as powerful to get a better elapsed time.

Setting System Statistics

As an example of the power of setting system statistics, I'll carry out the steps to do this to my test rig. If you wanted to do something similar to your throwaway test environment to make it similar to another environment, you could get the system statistics and set them manually as I am going to do in this example. From the example above our best cost so far was 947. We also know that our source machine (my ancient laptop) had a CPU speed of 1683 MHz and I/O seek time of 10ms and an I/O transfer speed of 4096. In this example I'll set the CPU speed to 10 times the current value:

```
SQL> EXEC DBMS_STATS.SET_SYSTEM_STATS('CPUSPEEDNW', 16830.00);
```

Then when we next run the test case (after flushing the shared Pool) we get this execution plan:

```
-----------------------------------------------------------------------------------------------
|Id  |Operation              |Name      | Rows  | Bytes | Cost (%CPU)| Time     | Pstart| Pstop |
-----------------------------------------------------------------------------------------------
|  0 |SELECT STATEMENT       |          |       |       |  897 (100)|          |       |       |
|  1 | SORT ORDER BY         |          |     9 |   315 |  897   (2)| 00:00:11 |       |       |
|  2 |  HASH GROUP BY        |          |     9 |   315 |  897   (2)| 00:00:11 |       |       |
|* 3 |   HASH JOIN           |          |  435K|    14M|  891   (1)| 00:00:11 |       |       |
|* 4 |    HASH JOIN          |          | 26289 |  641K|  408   (1)| 00:00:05 |       |       |
|* 5 |     TABLE ACCESS FULL |COUNTRIES|     9 |   135 |    3   (0)| 00:00:01 |       |       |
|  6 |     TABLE ACCESS FULL |CUSTOMERS| 55500 |  541K|  404   (0)| 00:00:05 |       |       |
|  7 |    PARTITION RANGE ALL|          |  918K|  8973K|  482   (1)| 00:00:06 |     1 |    28 |
|  8 |     TABLE ACCESS FULL |SALES     |  918K|  8973K|  482   (1)| 00:00:06 |     1 |    28 |
-----------------------------------------------------------------------------------------------
```

Please note that in this case the plan has not changed; however, what we do see is that the elapsed time has been reduced slightly from 12 seconds to 11 seconds and that the cost has been reduced slightly. Most of the cost of the query is in the I/O, and the CPU is not crucial to the execution time. We could conclude from this that if we wanted a bigger better laptop, that a much more important parameter to pay attention to would be the transfer speed of the disks rather than the speed of the CPU (as far as this query is concerned, of course). Playing around in your test environment with settings like this can allow you to discover what is important in your environment for improving performance and save time by allowing you to concentrate on what is important and ignore what is less important.

Object Statistics

You can set object statistics for all the objects in the query also, but this is a tricky operation and not recommended for this kind of exploration. You are better off looking at the 10053 trace file to see why your costs are the way they are than by changing the costs to see what happens.

Debugging the Optimizer

Creating and using a test case is mostly something that you should expect Oracle support to do on your behalf. They will take the test case files that you supply and explore them sufficiently to be able to solve your problem. You can also use the test case files (as I've described above) to do some testing in a free-standing test environment that nobody else is using. The test case routines are mostly used to explore the behavior of the optimizer. Everything in the test case files is designed to make the optimizer think it is on the original source system. If there is a bug that causes the optimizer to change its execution plan in an inappropriate way, then a test case is what Oracle support will use possibly in conjunction with your data to determine if there is a true bug or some unexpected behavior that is not a bug. Sometimes in rare cases some particular execution plan can cause a bug to be exposed, and in these cases you can sometimes avoid the bug by setting some optimizer environmental factor.

Other Test Case Utilities

As if the ability to explore your SQL in a stand-alone test environment isn't enough, SQLT provides further utilities for understanding what's happening in any particular SQL. Sometimes you want to do more than just change environmental settings and retry your query. Just like a dependable Swiss Army knife of tuning, SQLT has a tool for nearly every job.

What Does sel.sql Do?

Sel is a nice little utility that you would probably write if it wasn't already written for you. Here is the code for `sel.sql`:

```
REM Computes predicate selectivity using CBO. Requires sel_aux.sql.
SPO sel.log;
SET ECHO OFF FEED OFF SHOW OFF VER OFF;
PRO
COL table_rows NEW_V table_rows FOR 999999999999;
COL selectivity FOR 0.000000000000 HEA "Selectivity";
COL e_rows NEW_V e_rows FOR 999999999999 NOPRINT;
ACC table PROMPT 'Table Name: ';
SELECT num_rows table_rows FROM user_tables WHERE table_name = UPPER(TRIM('&&table.'));
@@sel_aux.sql
```

This routine prompts for the table to be accessed after setting up the column formats and then calls `sel_aux.sql`:

```
REM Computes predicate selectivity using CBO. Requires sel.sql.
PRO
ACC predicate PROMPT 'Predicate for &&table.: ';
DELETE plan_table;
EXPLAIN PLAN FOR SELECT /*+ FULL(t) */ COUNT(*) FROM &&table. t WHERE &&predicate.;
SELECT MAX(cardinality) e_rows FROM plan_table;
SELECT &&e_rows. "Comp Card", ROUND(&&e_rows./&&table_rows., 12) selectivity FROM DUAL;
@@sel_aux.sql
```

`sel_aux.sql` uses `explain plan` to determine the cardinality and selectivity by selecting this information from the `plan_table` populated by `explain plan`. It displays the computed cardinality and selectivity for any particular predicate and table in the test case. At the end of the routine `sel_aux` is called again to give a chance to select a different value for a predicate or a different predicate entirely. So if you were looking at problems regarding a particular value you might want to test cardinality by using `sel.sql`. In the example below I choose the SALES table to investigate, and I'm interested in the cardinality of a query against SALES where the CUST_ID is 100 and 200.

```
SQL> @sel

Table Name: sales

   TABLE_ROWS
-------------
       918843

Predicate for sales: cust_id=100

 Comp Card     Selectivity
---------- ---------------
       130  0.000141482277

Predicate for sales: cust_id=200

 Comp Card     Selectivity
---------- ---------------
       130  0.000141482277
```

```
Predicate for sales: cust_id between 100 and 300

 Comp Card     Selectivity
---------- ---------------
      2080  0.002263716435

Predicate for sales:
```

I can see that the statistics estimate that the cardinality for SALES for CUST_ID is 130. The same value is seen for `cust_id=200`. Since I also have the data on this system I can see what the actual values are.

```
SQL> select count(*) From sh.sales where cust_id=100;

  COUNT(*)
----------
        30

SQL> select count(*) From sh.sales where cust_id=200;

  COUNT(*)
----------
        68

SQL> select count(*) From sh.sales where cust_id between 100 and 300;

  COUNT(*)
----------
     18091
```

Notice that each of the estimates is wrong. The estimates are so far off that using the BETWEEN clause gives me an estimate of 2,080, whereas the actual value is 18,091. We can see from this that the statistics on the objects need to be improved. Perhaps a histogram on the CUST_ID column would help here.

What Does sqlt_snnnnn_del_hgrm.sql Do?

Often histograms can cause a problem, if they are not properly sampled or are inappropriately used. In such cases, with the test case utility you can try deleting the histograms to see what the effect on the execution plan is. In this example we delete all the histograms registered against the test schema.

```
SQL> @sqlt_s64661_del_hgrm.sql
Enter value for tc_user: TC64661
delete_column_hgrm: TC64661.CBO_STAT_TAB_4TC.<partname>.C1
delete_column_hgrm: TC64661.CBO_STAT_TAB_4TC.<partname>.C2
delete_column_hgrm: TC64661.CBO_STAT_TAB_4TC.<partname>.C3
delete_column_hgrm: TC64661.CBO_STAT_TAB_4TC.<partname>.C4
delete_column_hgrm: TC64661.CBO_STAT_TAB_4TC.<partname>.C5
delete_column_hgrm: TC64661.CBO_STAT_TAB_4TC.<partname>.CH1
delete_column_hgrm: TC64661.CBO_STAT_TAB_4TC.<partname>.D1
delete_column_hgrm: TC64661.CBO_STAT_TAB_4TC.<partname>.FLAGS
delete_column_hgrm: TC64661.CBO_STAT_TAB_4TC.<partname>.N1
delete_column_hgrm: TC64661.CBO_STAT_TAB_4TC.<partname>.N10
delete_column_hgrm: TC64661.CBO_STAT_TAB_4TC.<partname>.N11
delete_column_hgrm: TC64661.CBO_STAT_TAB_4TC.<partname>.N12
delete_column_hgrm: TC64661.CBO_STAT_TAB_4TC.<partname>.N2
```

```
delete_column_hgrm: TC64661.CBO_STAT_TAB_4TC.<partname>.N3
delete_column_hgrm: TC64661.CBO_STAT_TAB_4TC.<partname>.N4
delete_column_hgrm: TC64661.CBO_STAT_TAB_4TC.<partname>.N5
delete_column_hgrm: TC64661.CBO_STAT_TAB_4TC.<partname>.N6
delete_column_hgrm: TC64661.CBO_STAT_TAB_4TC.<partname>.N7
delete_column_hgrm: TC64661.CBO_STAT_TAB_4TC.<partname>.N8
delete_column_hgrm: TC64661.CBO_STAT_TAB_4TC.<partname>.N9
delete_column_hgrm: TC64661.CBO_STAT_TAB_4TC.<partname>.STATID
delete_column_hgrm: TC64661.CBO_STAT_TAB_4TC.<partname>.TYPE
delete_column_hgrm: TC64661.CBO_STAT_TAB_4TC.<partname>.VERSION
delete_column_hgrm: TC64661.COUNTRIES.<partname>.COUNTRY_ID
delete_column_hgrm: TC64661.COUNTRIES.<partname>.COUNTRY_ISO_CODE
delete_column_hgrm: TC64661.COUNTRIES.<partname>.COUNTRY_NAME
delete_column_hgrm: TC64661.COUNTRIES.<partname>.COUNTRY_NAME_HIST
delete_column_hgrm: TC64661.COUNTRIES.<partname>.COUNTRY_REGION
delete_column_hgrm: TC64661.COUNTRIES.<partname>.COUNTRY_REGION_ID
delete_column_hgrm: TC64661.COUNTRIES.<partname>.COUNTRY_SUBREGION
delete_column_hgrm: TC64661.COUNTRIES.<partname>.COUNTRY_SUBREGION_ID
delete_column_hgrm: TC64661.COUNTRIES.<partname>.COUNTRY_TOTAL
delete_column_hgrm: TC64661.COUNTRIES.<partname>.COUNTRY_TOTAL_ID
delete_column_hgrm: TC64661.CUSTOMERS.<partname>.COUNTRY_ID
delete_column_hgrm: TC64661.CUSTOMERS.<partname>.CUST_CITY
delete_column_hgrm: TC64661.CUSTOMERS.<partname>.CUST_CITY_ID
delete_column_hgrm: TC64661.CUSTOMERS.<partname>.CUST_CREDIT_LIMIT
delete_column_hgrm: TC64661.CUSTOMERS.<partname>.CUST_EFF_FROM
delete_column_hgrm: TC64661.CUSTOMERS.<partname>.CUST_EFF_TO
delete_column_hgrm: TC64661.CUSTOMERS.<partname>.CUST_EMAIL
delete_column_hgrm: TC64661.CUSTOMERS.<partname>.CUST_FIRST_NAME
delete_column_hgrm: TC64661.CUSTOMERS.<partname>.CUST_GENDER
delete_column_hgrm: TC64661.CUSTOMERS.<partname>.CUST_ID
delete_column_hgrm: TC64661.CUSTOMERS.<partname>.CUST_INCOME_LEVEL
delete_column_hgrm: TC64661.CUSTOMERS.<partname>.CUST_LAST_NAME
delete_column_hgrm: TC64661.CUSTOMERS.<partname>.CUST_MAIN_PHONE_NUMBER
delete_column_hgrm: TC64661.CUSTOMERS.<partname>.CUST_MARITAL_STATUS
delete_column_hgrm: TC64661.CUSTOMERS.<partname>.CUST_POSTAL_CODE
delete_column_hgrm: TC64661.CUSTOMERS.<partname>.CUST_SRC_ID
delete_column_hgrm: TC64661.CUSTOMERS.<partname>.CUST_STATE_PROVINCE
delete_column_hgrm: TC64661.CUSTOMERS.<partname>.CUST_STATE_PROVINCE_ID
delete_column_hgrm: TC64661.CUSTOMERS.<partname>.CUST_STREET_ADDRESS
delete_column_hgrm: TC64661.CUSTOMERS.<partname>.CUST_TOTAL
delete_column_hgrm: TC64661.CUSTOMERS.<partname>.CUST_TOTAL_ID
delete_column_hgrm: TC64661.CUSTOMERS.<partname>.CUST_VALID
delete_column_hgrm: TC64661.CUSTOMERS.<partname>.CUST_YEAR_OF_BIRTH
delete_column_hgrm: TC64661.SALES.<partname>.AMOUNT_SOLD
delete_column_hgrm: TC64661.SALES.<partname>.CHANNEL_ID
delete_column_hgrm: TC64661.SALES.<partname>.CUST_ID
delete_column_hgrm: TC64661.SALES.<partname>.PROD_ID
delete_column_hgrm: TC64661.SALES.<partname>.PROMO_ID
delete_column_hgrm: TC64661.SALES.<partname>.QUANTITY_SOLD
delete_column_hgrm: TC64661.SALES.<partname>.TIME_ID

PL/SQL procedure successfully completed.
```

In this case there was no difference in the execution plan, so we know that the histograms make no difference for this query. We cannot of course go back to the target system and delete the histograms because there could be other SQL that relies on those histograms to execute effectively.

What Does sqlt_snnnnn_tcx.zip Do?

In the latest version of SQLT you can create test cases that have no dependence on SQLT on the target platform. In other words, once you have run SQLTXECUTE or SQLTXTRACT you can take the test case (as described in the sections above) and use the `sqlt_snnnnn_tcx.zip` file to create a test case on the target platform and not be reliant on SQLT being installed on that database. In this section I'll use the same SQL as before to create a test case on a new database that has no SQLT installed on it. As usual I have put the files from this zip file into a sub-directory. Now on my new standard database I'll install the test case with the `install.sql` script provided in the list of files. Here is the list of files in the `sqlt_s11996_tcx` directory I created.

```
12/29/2012  09:59 AM               132 10053.sql
12/29/2012  09:59 AM               103 flush.sql
12/29/2012  09:59 AM               118 install.sh
12/29/2012  09:59 AM               960 install.sql
12/29/2012  09:59 AM             2,911 pack_tcx.sql
12/29/2012  10:03 AM             2,209 plan.log
12/29/2012  09:59 AM               279 plan.sql
12/29/2012  09:50 AM               324 q.sql
12/29/2012  09:59 AM               424 sel.sql
12/29/2012  09:59 AM               378 sel_aux.sql
12/29/2012  09:59 AM         1,193,984 sqlt_s11996_exp2.dmp
12/29/2012  10:02 AM               596 sqlt_s11996_imp2.log
12/29/2012  10:02 AM            44,263 sqlt_s11996_metadata1.log
12/29/2012  09:59 AM            29,223 sqlt_s11996_metadata1.sql
12/29/2012  10:02 AM           106,511 sqlt_s11996_metadata2.log
12/29/2012  09:59 AM            95,604 sqlt_s11996_metadata2.sql
12/29/2012  10:02 AM             3,272 sqlt_s11996_schema_stats.log
12/29/2012  09:59 AM             2,174 sqlt_s11996_schema_stats.sql
12/29/2012  10:03 AM           109,835 sqlt_s11996_set_cbo_env.log
12/29/2012  09:59 AM            84,672 sqlt_s11996_set_cbo_env.sql
12/29/2012  10:02 AM             1,658 sqlt_s11996_system_stats.log
12/29/2012  09:59 AM             1,285 sqlt_s11996_system_stats.sql
12/29/2012  09:59 AM           103,448 sqlt_s11996_tcx.zip
12/29/2012  09:59 AM               141 tc.sql
```

All of these files are recognizable from the TC files we saw earlier. The new interesting file for us is `install.sql`, which installs the test case in our new database. If we run this script we see the final page as shown below, where the test case user (in this case TC11996) has been created and the test SQL run.

```
------------------------
EXPLAINED SQL STATEMENT:
------------------------
select   country_name, sum(AMOUNT_SOLD) from sales s, customers c,
countries co where   s.cust_id=c.cust_id   and
co.country_id=c.country_id   and country_name in
('Ireland','Denmark','Poland','United
Kingdom','Germany','France','Spain','The Netherlands','Italy')   group
```

```
by country_name order by sum(AMOUNT_SOLD)
Plan hash value: 2938593747
-----------------------------------------------------------------
| Id  | Operation              | Name      | Rows  | Cost (%CPU)|
-----------------------------------------------------------------
|   0 | SELECT STATEMENT       |           |       |   947 (100)|
|   1 |  SORT ORDER BY         |           |     9 |   947   (7)|
|   2 |   HASH GROUP BY        |           |     9 |   947   (7)|
|*  3 |    HASH JOIN           |           |  435K|   909   (3)|
|*  4 |     HASH JOIN          |           | 26289 |   409   (1)|
|*  5 |      TABLE ACCESS FULL | COUNTRIES |     9 |     3   (0)|
|   6 |      TABLE ACCESS FULL | CUSTOMERS | 55500 |   406   (1)|
|   7 |     PARTITION RANGE ALL|           |  918K|   494   (3)|
|   8 |      TABLE ACCESS FULL | SALES     |  918K|   494   (3)|
-----------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   3 - access("S"."CUST_ID"="C"."CUST_ID")
   4 - access("CO"."COUNTRY_ID"="C"."COUNTRY_ID")
   5 - filter(("COUNTRY_NAME"='Denmark' OR "COUNTRY_NAME"='France' OR
               "COUNTRY_NAME"='Germany' OR "COUNTRY_NAME"='Ireland' OR
               "COUNTRY_NAME"='Italy' OR "COUNTRY_NAME"='Poland' OR
               "COUNTRY_NAME"='Spain' OR "COUNTRY_NAME"='The Netherlands' OR
               "COUNTRY_NAME"='United Kingdom'))
36 rows selected.
SQL> SPO OFF;
SQL> show user
USER is "TC11996"
```

This database is completely fresh. There is no SH schema (I dropped the schema after creation of the database) or SQLT schema (I never installed SQLT on this database). All the schemas are the standard accounts, and yet if I log in as TC11996 I can execute my test case and do all the things to try and improve the execution plan as I would normally be able to do with SQLT. There is one additional wrinkle to TCX that is sometimes useful. There is a file called `pack_tcx.sql` that allows you to pack the test case even more tightly. If you run this file a new zip file is produced called `tcx.zip`, which contains the files shown below (some of these file names will be different in your case).

```
12/29/2012  09:59 AM             132 10053.sql
12/29/2012  09:59 AM             103 flush.sql
12/29/2012  09:59 AM             279 plan.sql
12/29/2012  09:50 AM             324 q.sql
12/29/2012  10:29 AM              74 readme.txt
12/29/2012  10:29 AM           1,447 setup.sql
12/29/2012  09:59 AM          95,604 sqlt_s11996_metadata2.sql
12/29/2012  09:59 AM          84,672 sqlt_s11996_set_cbo_env.sql
12/29/2012  09:59 AM           1,285 sqlt_s11996_system_stats.sql
12/29/2012  10:30 AM       1,136,640 TC11996_expdat.dmp
```

These files are automatically unzipped (by `pack_tcx.sql`). This time we run just `setup.sql` as sys and the test case user is created. While these extra options to create test cases may seem a little redundant when you have SQLT already installed (and I encourage you to do this), these increasingly smaller test cases that show your problem SQL and execution plan are extremely helpful when trying to convince someone (for example, Oracle support) that you have a problem. The fewer extraneous details included in your test case (as long as it still shows the problem) the better it is for trying to solve the problem.

Including Test Case Data

SQLT test cases do not by default include application data. This is because the problem (1) has nothing to do with the application data, (2) there is too much data or (3) the data is privileged in some way and cannot be shown to anyone else. In some rare cases data may be required to show the problem. Luckily SQLT has an option to allow this as long as the other hurdles can be overcome. In the later versions of SQLT you can do it like this:

```
SQL> EXEC SQLTXADMIN.sqlt$a.set_param('tcb_export_data', 'TRUE');

PL/SQL procedure successfully completed.
```

In the older versions of SQLT, find this section of code in the `sqcpkgi.pkb` file in the install directory and change the FALSE to TRUE. This is what the code fragment looks like before it is changed:

```
EXECUTE IMMEDIATE
  'BEGIN DBMS_SQLDIAG.EXPORT_SQL_TESTCASE ( '||
  'directory     => :directory, '||
  'sql_id        => :sql_id, '||
  'exportData    => FALSE , '||  <<<Change this to TRUE
  'timeLimit     => :timeLimit, '||
  'testcase_name => :testcase_name, '||
  'testcase      => :testcase ); END;'

Then rebuild this package
SQL> @sqcpkgi.pkb

Package body created.

No errors.
```

Once you've selected this option you can run `sqltxecute.sql` (or `sqltxtract.sql`) and collect a test case that contains application data. All the set up steps are exactly the same after this, you run `xpress.sql` and follow the steps outlined in previous sections.

Summary

I hope you've enjoyed looking at the features of the test case utility with me. They are powerful and flexible. They are also the beginning of a new world of tuning. Once you have a test case you can change (on a stand-alone system) you can endlessly explore the CBO environment looking for the best performance. As your knowledge of SQL and tuning concepts increases you will be able to try more and more methods to improve the execution plan. Your bedrock in all these endeavors will be SQLTXPLAIN's test case utility. In the next chapter we'll look at a way you can use the test case to find a solution to a problem. With the XPLORE method you'll use a sledgehammer to crack a nut.

CHAPTER 12

Using XPLORE to Investigate Unexpected Plan Changes

I'm sure by now you're thinking that SQLTXPLAIN is a pretty cool tool to be familiar with, if you want to tune Oracle SQL. In Chapter 11 we discussed the test case utility and how that could build an entire environment for testing SQL, system parameter, optimizer settings, objects statistics, enough of the settings in fact to make the optimizer behave as if it was on the source system. Once you achieve that state you have a superb environment to go exploring the optimizer's choices. This intelligent, directed approach is extremely fruitful when you have a rough idea what the problem might be and you suspect that you are only a few changes away from the solution. In those kinds of situations you can create the test case, try your changes, and test the result. This is a good way to solve tuning problems, but what if you had no idea what had caused a regression to some SQL after an upgrade or patch to Oracle or if a SQL plan changed unexpectedly. Then what could you do? There are hundreds of parameter changes you could try, but it would take far too long to go through them all. On the other hand, computers are pretty good at churning through hundreds of choices to find the best one, and this is exactly what XPLORE does.

When Should You Use XPLORE?

The XPLORE method was designed with one purpose in mind: to do a brute force attack on an SQL statement by trying every choice of parameter and then presenting you with the answers. Even though XPLORE can explore many changes to the optimizer environment, it was designed with one specific goal: to find bugs caused by upgrades and patches to the Oracle engine. It is not designed to tune SQL directly, as there are adequate features in other parts of SQLT to deal with that scenario. Nonetheless, it can still be used in a non-upgrade situation if you've run out of ideas and need a hint or if you've hit a bug on a new SQL and you think a fix control change may help. It is designed to deal with situations where an upgrade has occurred and the system works just fine, except for one or two SQLs that for some unspecified reason have regressed. In this scenario it is possible that some feature upgrade of the Oracle engine has caused a specific SQL to no longer be optimal. Generally speaking SQL performance improves from version to version, but with every version upgrade if 1000s of SQL statements improve maybe one or two will be worse off. If those one or two SQL happen to be crucial to your operation, you'll need XPLORE to find the reason. Can you use XPLORE to tune statements that just need some tuning? It's possible that if you ran XPLORE, and it came up with some parameter changes that might be beneficial, those changes might give you some ideas for improvements. But it's generally not the best use of XPLORE.

How Does XPLORE Work?

XPLORE relies on a working test case (as described in Chapter 11). You must have used XECUTE or XTRACT to create a zip file that contains a test case file. Once you have been through the steps to create your test case you will have a test case that XPLORE can work from. Once this test case is in place, XPLORE goes to work. It generates a script with every possible value for every optimizer parameter available, including the hidden ones as well as the fix control settings (we'll discuss those shortly). This script then runs the test case for each possible value for all the parameters and generates an HTML file with the results. The results of all these tests are summarized and hyperlinked in the HTML file to the execution plan and the setting that was changed. It's a real work out for the cost-based optimizer, but remember there's no data (usually).

There are four basic steps to using XPLORE.

1. Get access to the test case
2. Set the baseline environment for the test case
3. Create a script that can run your test case
4. Create a superscript that will run all possible values with your test case.

Let's talk about each of these steps in more detail.

You must be able to run `q.sql` without an error and be able to generate the execution plan. Both of these are requirements, because if the SQL does not run you cannot generate an execution plan, and if you cannot generate an execution plan you cannot generate costs. Without costs you cannot go exploring.

The second step allows you to set some special characteristic of the test case that applies to all the variants tested. Whatever you select here (it could be an optimizer parameter or a fix control or some other environmental setting) it will be done before each iteration of the XPLORE process. You might choose to do this if you were sure that the problem you were investigating had some characteristic you did not want to deviate from. Then you let the brute force method take over.

The third step is to generate a generic script based on your settings, which is used as a framework for the superscript.

The final step is to use this framework script to iterate over all the possible values of the choices you have selected (optimizer parameter, fix control, and Exadata parameters) to create a very big script that does all the work. This big script (the superscript) collects all the information from all the executions and puts all the information into one HTML document. This is the HTML document you then look at manually and apply your intelligence to.

What Can XPLORE Change?

As we mentioned earlier, XPLORE can look at the effect of changes to the optimizer parameters, the fix control settings, and the special Exadata parameters. It cannot be used to test for bad statistics, skewed data, index changes, or similar structural changes to the database objects. If you choose CBO parameters you will only get a report covering the standard CBO parameters (including the hidden ones). This is a much shorter report than including the fix control settings. If you suspect that some optimizer feature related to a bug is the cause of the problem then you may choose to use the fix control settings and ignore the optimizer parameters. If you suspect an Exadata specific problem then you can choose to select only those parameters related to Exadata. There are not many (16) Exadata specific parameters, so I think it would be useful to list them here. You can get this list from `sqlt$_v$parameter_exadata`. You can get a similar list for the "normal" non-Exadata parameters, but the list is too long to put here (there are 275 parameters, I list these in Appendix B)

```
SQL> select name, description from sqlt$_v$parameter_exadata;

NAME                                 DESCRIPTION
-----------------------------------------------------------------------------------------
_kcfis_cell_passthru_enabled         Do not perform smart IO filtering on the cell
_kcfis_rdbms_blockio_enabled         Use block IO instead of smart IO in the smart IO module on
                                     RDBMS
_kcfis_storageidx_disabled           Don't use storage index optimization on the storage cell
_kcfis_control1                      Kcfis control1
_kcfis_control2                      Kcfis control2
cell_offload_processing              enable SQL processing offload to cells
_slave_mapping_enabled               enable slave mapping when TRUE
_projection_pushdown                 projection pushdown
_bloom_filter_enabled                enables or disables bloom filter
_bloom_vector_elements               number of elements in a bloom filter vector
_bloom_predicate_enabled             enables or disables bloom filter predicate pushdown
_bloom_predicate_pushdown_to_storage enables or disables bloom filter predicate pushdown to
                                     storage
_bloom_folding_enabled               Enable folding of bloom filter
_bloom_pushing_max                   bloom filter pushing size upper bound
_bloom_pruning_enabled               Enable partition pruning using bloom filtering
parallel_force_local                 force single instance execution

16 rows selected.
```

Optimizer parameters, the fix control settings and the special Exadata parameters are the only things that you can explore with this method, but remember you can still change the baseline environment and re-run your explore each time. Each XPLORE iteration takes a considerable amount of time, so if you do make changes like these you need to be sure you're on the right track before committing to an XPLORE run.

What XPLORE Cannot Change

XPLORE cannot change the structure of your SQL or suggest an index or remove an index a table. It is purely designed to find the changes that occur when CBO parameters are changed or when fix control changes are made. Even though XPLORE cannot change these things for you you're free to change the query in `q.sql` and re-run XPLORE. For example you might decide that you need to change an index type or remove an index. This is again somewhat beyond the scope of XPLORE and can be done in better ways.

What Is Fix Control?

Oracle is an extremely flexible product, with many features and bug fixes, some of which are turned off by default and some of which are turned on by default. Fix Control then is the facility within Oracle that allows bug fixes to be changed from 0 (FALSE) to 1 (TRUE) or visa versa. As you can imagine enabling or disabling bug fixes in this way should only be done on a throwaway database as it can severely damage your database (open a support call before making any changes like this). Let's look at one of these settings from the many that are used in `xplore_script_1.sql`, the main script of the whole XPLORE process. We'll look at the 6897034 fix control parameter as an example. There are many many other fix control parameters like this that we will not describe, but they are all controlled in a similar way.

This particular fix control parameter controls the bug fix that takes into account NULL rows in estimating cardinality. When "_fix_control" is true the optimizer takes NULLS into account. Here we set it to FALSE, which is not likely to be something you would want to do in a real-life situation.

```
ALTER SESSION SET "_fix_control" = '6897034:0';
```

The script, `xplore_script_1.sql`, which is the main driving script for the XPLORE process will in one of the iterations of XPLORE change this fix control setting from its default to its non-default value. In this case it will be changed from 1 to 0. When it is set to 1 this fix control setting is on. When it is set to 0 this fix control feature is off. We can find out what the feature is that this fix control setting represents by querying `v$system_fix_control`. Below I show a query where I interrogate my local database for the meaning of this particular fix control setting:

```
SQL> select
  BUGNO,
  VALUE,
  DESCRIPTION,
  OPTIMIZER_FEATURE_ENABLE,
  IS_DEFAULT
from
  v$system_fix_control
where
  bugno=6897034;

     BUGNO      VALUE DESCRIPTION          OPTIMIZER_FEATURE_ENABLE  IS_DEFAULT
---------- ---------- -------------------- ------------------------- ----------
   6897034          1 index cardinality est 10.2.0.5                         1
                      imates not taking int
                      o account NULL rows
```

In this particular case bug number 6897034 has to do with index cardinality estimates and those estimates in relation to NULLs in rows. That could be a pretty serious problem, so by default this has been set to 1 (TRUE). The script I am running will set it to 0 to see if that makes a difference to the execution plan. The oldest version of Oracle in which this value was set to 1 was 10.2.0.5. In the previous version, 10.2.0.4 the value was 0 (or this bug fix was not introduced).

Remember I said that `optimizer_features_enable` was like a super switch. Well, we can test that now by setting `optimizer_features_enable` to 10.2.0.4 and seeing what happens to this particular fix control. First I show the current value of `optimizer_features_enable`, and then I show the current value of the fix control. Then I change the super parameter to 10.2.0.4, (first at the session level then at the system level). The value I have to set is the value just before 10.2.0.5, which is the value of the `OPTIMIZER_FEATURE_ENABLE` column for the fix control. (Yes the column is `OPTIMIZER_FEATURE_ENABLE`, and the parameter is `optimizer_features_enable`). After setting the super parameter at the system level I re-query the value of the fix control and I see that the super parameter has done its job and set the fix control as it would have been in that version.

```
SQL> show parameter optimizer_features_enable;

NAME                                 TYPE        VALUE
------------------------------------ ----------- ------------------------------
optimizer_features_enable            string      11.2.0.1

SQL> select
  BUGNO,
  VALUE,
  DESCRIPTION,
  OPTIMIZER_FEATURE_ENABLE,
  IS_DEFAULT
from
  v$system_fix_control
where
  bugno=6897034;

     BUGNO      VALUE DESCRIPTION          OPTIMIZER_FEATURE_ENABLE  IS_DEFAULT
---------- ---------- -------------------- ------------------------- ----------
   6897034          1 index cardinality est 10.2.0.5                           1
                      imates not taking int
                      o account NULL rows

SQL> alter session set optimizer_features_enable='10.2.0.4';

Session altered.

SQL> select
  BUGNO,
  VALUE,
  DESCRIPTION,
  OPTIMIZER_FEATURE_ENABLE,
  IS_DEFAULT
from
  v$system_fix_control
where
  bugno=6897034;

     BUGNO      VALUE DESCRIPTION          OPTIMIZER_FEATURE_ENABLE  IS_DEFAULT
---------- ---------- -------------------- ------------------------- ----------
   6897034          1 index cardinality est 10.2.0.5                           1
                      imates not taking int
                      o account NULL rows


SQL> alter system set optimizer_features_enable='10.2.0.4' scope=memory;

System altered.

SQL> select
  BUGNO,
  VALUE,
  DESCRIPTION,
  OPTIMIZER_FEATURE_ENABLE,
  IS_DEFAULT
from
  v$system_fix_control
where
  bugno=6897034;
```

```
     BUGNO      VALUE DESCRIPTION          OPTIMIZER_FEATURE_ENABLE  IS_DEFAULT
---------- ---------- -------------------- ------------------------- ----------
   6897034          0 index cardinality est 10.2.0.5                          1
                      imates not taking int
                      o account NULL rows
```

If we run the super script it will do this step so that this parameter is set. The result will be that during that step of the XPLORE script we have a non-default set up for the session. It seems unlikely that this parameter is going to improve your query, but it's not impossible. Imagine a case where some SQL is reliant on a faulty calculation for cardinality, which is then “fixed” by applying the fix for bug 6897034. Then the SQL could regress (perform less well), and it could then appear that the SQL was broken by an upgrade to 10.2.0.5. However, the XPLORE method is only applying a brute force approach; it just powers through every single option and lets you supply the intelligence to determine what is important and what is not. Now we know how XPLORE works, when to use it, and how to use it. We also know XPLORE is not the panacea for tuning problems but just one more tool on the Swiss Army knife of tuning. Before we look at an example XPLORE session let me first explain about SQL monitor.

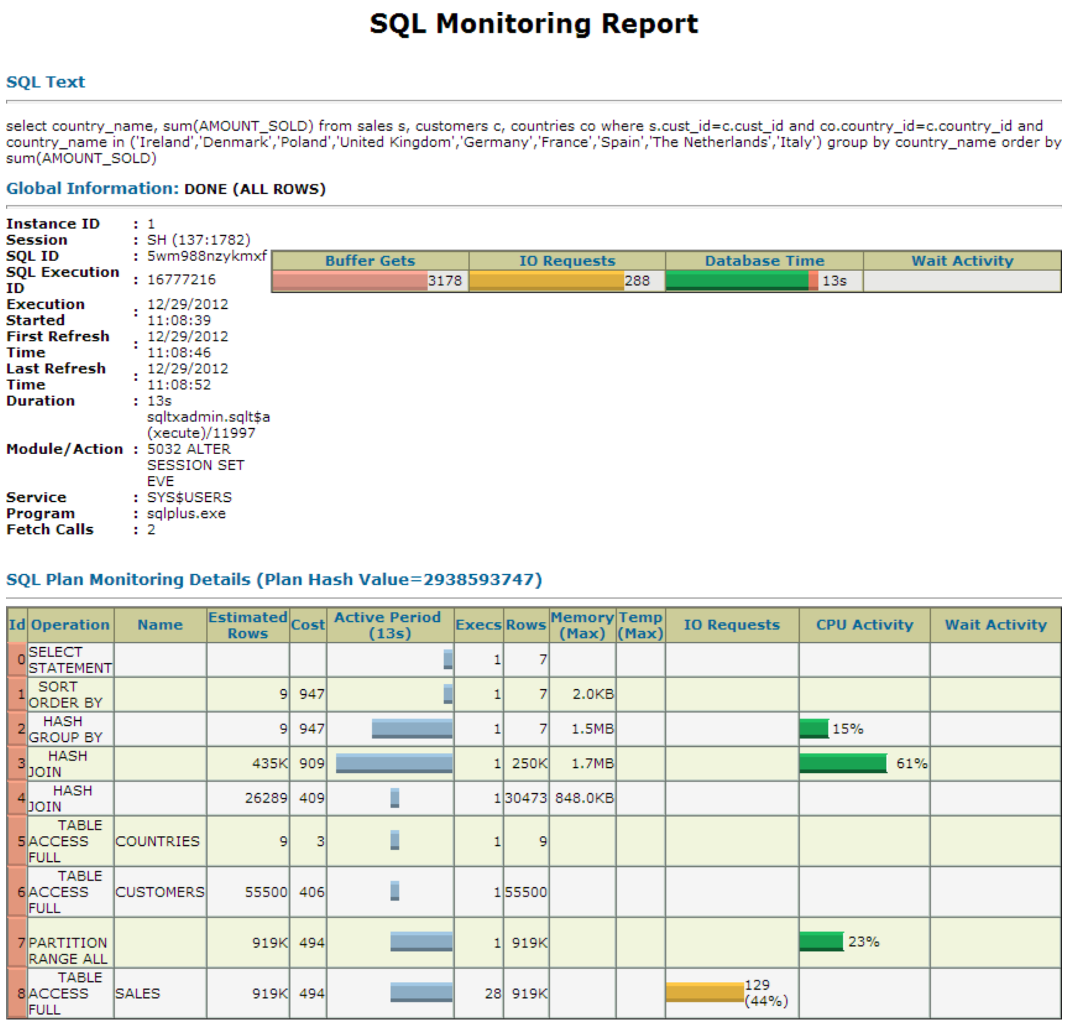
What Is a SQL Monitor Report?

The real-time SQL monitoring feature was introduced in 11g and is by default turned off. If the SQL uses more than five seconds of CPU or runs in parallel the real-time monitoring will be turned on. If this feature is enabled statistics metrics are collected as the SQL executes and are stored for a few minutes. During that time the view `v$sql_monitor` and `v$sql_plan_monitor` contain the performance information. The type of information collected can be controlled by the `type` parameter and I've selected `active` which produces an HTML report type display (This is new in 11g Release 2). This detailed information is particularly useful because it shows each step's use of CPU and I/O resources. This can be very helpful in determining which parts of your query need to be investigated, and if you are comparing SQL, which parts of your query have changed significantly. Below I list a typical set of commands you would issue to get a SQL monitor HTML report.

1. Add the hint `/*+ monitor */` in the query. This causes the CBO to monitor the query even if it runs for less than five seconds or is not parallel.
2. Run your query.
3. Get the SQL_ID (`select sql_id from v$sql_text where sql_text like '%<Your SQL Text here>%';`)
4. Then execute this code:

```
set trimspool on
set trim on
set pages 0
set linesize 1000
set long 1000000
set longchunksize 1000000
spool sqlmon_active_1st_run.html
select dbms_sqltune.report_sql_monitor(sql_id=>'&sqlid',type=>'active') from
dual;
spool off
```

This produces an HTML file just like the one in Figure 12-1.



SQL Monitoring Report

SQL Text

select country_name, sum(AMOUNT_SOLD) from sales s, customers c, countries co where s.cust_id=c.cust_id and co.country_id=c.country_id and country_name in ('Ireland','Denmark','Poland','United Kingdom','Germany','France','Spain','The Netherlands','Italy') group by country_name order by sum(AMOUNT_SOLD)

Global Information: DONE (ALL ROWS)

Instance ID	: 1
Session	: SH (137:1782)
SQL ID	: 5wm988nzykmxf
SQL Execution ID	: 16777216
Execution Started	: 12/29/2012 11:08:39
First Refresh Time	: 12/29/2012 11:08:46
Last Refresh Time	: 12/29/2012 11:08:52
Duration	: 13s
Module/Action	: sqltxadmin.sqlt$a (xecute)/11997 5032 ALTER SESSION SET EVE
Service	: SYS$USERS
Program	: sqlplus.exe
Fetch Calls	: 2

Buffer Gets	IO Requests	Database Time	Wait Activity
3178	288	13s	

SQL Plan Monitoring Details (Plan Hash Value=2938593747)

Id	Operation	Name	Estimated Rows	Cost	Active Period (13s)	Execs	Rows	Memory (Max)	Temp (Max)	IO Requests	CPU Activity	Wait Activity
0	SELECT STATEMENT					1	7					
1	SORT ORDER BY		9	947		1	7	2.0KB				
2	HASH GROUP BY		9	947		1	7	1.5MB			15%	
3	HASH JOIN		435K	909		1	250K	1.7MB			61%	
4	HASH JOIN		26289	409		1	30473	848.0KB				
5	TABLE ACCESS FULL	COUNTRIES	9	3		1	9					
6	TABLE ACCESS FULL	CUSTOMERS	55500	406		1	55500					
7	PARTITION RANGE ALL		919K	494		1	919K				23%	
8	TABLE ACCESS FULL	SALES	919K	494		28	919K			129 (44%)		

Figure 12-1. *A typical SQL Monitor report*

Now that we know what SQL monitor HTML reports are, we'll see that the XPLORE session creates them for us by default. Let's see the steps required to create an XPLORE report.

An Example XPLORE Session

In this XPLORE session we will go through every single step, including looking at the results and explaining what they tell us about the SQL in relation to the parameters which were changed. In this example test case I am using an XTRACT test case with no data, it is case S64661. This is the SQL being explored:

```
select country_name, sum(AMOUNT_SOLD)
from sales s, customers c, countries co
where
  s.cust_id=c.cust_id
  and co.country_id=c.country_id
  and country_name in (
    'Ireland','Denmark','Poland','United Kingdom',
    'Germany','France','Spain','The Netherlands','Italy')
  group by country_name order by sum(AMOUNT_SOLD);
```

It is being run on 11.2.0.1, and I'm curious to know if this SQL ran better in previous versions of Oracle. I also want to know if I can set up my throwaway database to run this more quickly. I could also be interested in running XPLORE for this SQL because I recently wrote it, it's not performing as I expected, and I think this may be because of a bug. It's a long shot, but I'm willing to give XPLORE a shot.

Getting Your Test Case Built

As I mentioned earlier there is no way to run XPLORE without first building a test case. In this example I'm using the same test case as I used in Chapter 11. By navigating to the xplore directory under utl I am able to easily run XPLORE's install script.

```
C:\Documents and Settings\Stelios\Desktop\SQLT 11.4.4.6\sqlt\utl\xplore>sqlplus / as sysdba
SQL*Plus: Release 11.2.0.1.0 Production on Mon Dec 17 19:37:00 2012
Copyright (c) 1982, 2010, Oracle.  All rights reserved.
Connected to:
Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - Production
With the Partitioning, OLAP, Data Mining and Real Application Testing options
SQL> @install
Test Case User: TC64661
Password: TC64661
```

This install script grants the test case user enough privileges (DBA) to run the scripts that are to follow. Remember this can only be run on a stand-alone system that can be replaced easily. The script finishes with

```
Package created.
No errors.
Package body created.
No errors.
Installation completed.
You are now connected as TC64661.
1. Set CBO env if needed
2. Execute @create_xplore_script.sql
```

The first step “Set CBO env if needed” is reminding you that if your test case needs to set any environmental settings that you want to be constant for all the runs of your test case, then you must set it here. This is then applied to the baseline and is executed before each execution of the scripts for each test. By giving the install script the test case user you have changed the user into a DBA of the system and allowed it to change all system parameters and fix control settings. The test case user also knows the script that needs to be run and will then proceed to build the template script and then the superscript.

Building the First Script

This first script does nothing more than collect your settings for your XPLORE session and then calls `xplore.create_xplore_script` with those parameters. Here is the line from the script:

```
EXEC xplore.create_xplore_script(
  '&&xplore_method.',         <<<This can take a value of XECUTE or XPLAIN
  '&&include_cbo_parameters.',<<<Include CBO parameters or not
  '&&include_exadata_parameters.',<<<Include Exadata specific parameters or not
  '&&include_fix_control.',   <<<Include Fix control or not
  '&&generate_sql_monitor_reports.'); <<<Generate SQL monitor reports or not.
```

These lines all make a difference to the final report that is created. The `xplore_method` controls XPLORE to allow it to run in XPLAIN mode or in XECUTE mode. In XPLAIN mode there is no execution of the SQL. XPLAIN mode is generally fast enough for most cases. XECUTE executes the SQL, and if there is also data present then this can take a very long time to complete. To clarify, the XECUTE option here is not related to the XECUTE method: this is an option of XPLORE utility we are talking about here. If the SQL can be executed relatively quickly, or if you have enough time to wait for the result then XECUTE will give you more information.

The next two parameters control the groups of parameters that you wish to explore. You can choose to explore all of them, in other words try all values for all CBO parameters, and all Exadata-specific parameters. If your test case is from an Exadata machine then you may be interested in choosing this option also. If your test case is not from Exadata, then there is no point in choosing this option because these tests will only have an effect on an Exadata machine. Then you can choose to set fix control parameters (we mentioned these in a previous section of this chapter). You may be interested in checking these parameters if you suspect you are suffering from an optimizer bug. When you run this script and enter the required parameters a new bigger script will be generated.

Building the Superscript

Now we come to the last step before running the XPLORE session. The first thing we need to do is copy the test SQL script to the `utl\xplore` directory. This is to allow the XPLORE session to access the script. In my case this is the command I issued:

```
>copy "C:\Documents and Settings\Stelios\Desktop\SQLT 11.4.4.6\sqlt\run\sqlt_s64661\TC\q.sql"
"C:\Documents and Settings\Stelios\Desktop\SQLT 11.4.4.6\sqlt\utl\xplore\q.sql"
```

We run the `create_xplore_script.sql`, which will prompt for the parameters described in the previous section. Here we see a run of the script where we have chosen XPLAIN mode and chosen to explore CBO parameters, Exadata parameters, and fix control parameters and to produce SQL monitor reports.

```
SQL> @create_xplore_script.sql
Parameter 1:
XPLORE Method: XECUTE (default) or XPLAIN
"XECUTE" requires /* ^^unique_id */ token in SQL
```

```
"XPLAIN" uses "EXPLAIN PLAN FOR" command
Enter "XPLORE Method" [XECUTE]: XPLAIN
Parameter 2:
Include CBO Parameters: Y (default) or N
Enter "CBO Parameters" [Y]: Y
Parameter 3:
Include Exadata Parameters: Y (default) or N
Enter "EXADATA Parameters" [Y]: Y
Parameter 4:
Include Fix Control: Y (default) or N
Enter "Fix Control" [Y]: Y
Parameter 5:
Generate SQL Monitor Reports: N (default) or Y
Only applicable when XPLORE Method is XECUTE
Enter "SQL Monitor" [N]: Y
Review and execute @xplore_script_1.sql
```

The script created now is extremely long. We can get a general feeling for what's in this script by looking at a few entries near the top of the file.

```
SET DEF ON ECHO OFF TERM ON APPI OFF SERVEROUT ON SIZE 1000000 NUMF "" SQLP SQL>;
SET SERVEROUT ON SIZE UNL;
SET ESC ON SQLBL ON;
SPO xplore_script_1.log;
COL connected_user NEW_V connected_user FOR A30;
SELECT user connected_user FROM DUAL;
PRO
PRO Parameter 1:
PRO Name of SCRIPT file that contains SQL to be xplored (required)
PRO
SET DEF ^ ECHO OFF;
DEF script_with_sql = '^1';
PRO
PRO Parameter 2:
PRO Password for ^^connected_user. (required)
PRO
DEF user_password = '^2';
PRO
PRO Value passed to xplore_script.sql:
PRO ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
PRO SCRIPT_WITH_SQL: ^^script_with_sql
PRO
PRO -- begin common
PRO DEF _SQLPLUS_RELEASE
PRO SELECT USER FROM DUAL;
PRO SELECT TO_CHAR(SYSDATE, 'YYYY-MM-DD HH24:MI:SS') current_time FROM DUAL;
PRO SELECT * FROM v$version;
PRO SELECT * FROM v$instance;
PRO SELECT name, value FROM v$parameter2 WHERE name LIKE '%dump_dest';
PRO SELECT directory_name||' '||directory_path directories FROM dba_directories WHERE
    directory_name LIKE 'SQLT$%' OR directory_name LIKE 'TRCA$%' ORDER BY 1;
```

```
31. PRO -- end common
32. PRO
33. SET VER ON HEA ON LIN 2000 PAGES 1000 TRIMS ON TI OFF TIMI OFF;
```

I've numbered the lines in the code, as it is worth explaining what's happening in detail here for some of these lines. Some lines are just PROMPT, which is just a blank line.

1. **SET DEF ON ECHO OFF TERM ON APPI OFF SERVEROUT ON SIZE 1000000 NUMF "" SQLP SQL>;** - Sets up the default format for output as the script relies on the format to produce a workable HTML file as it runs the script
2. **SET SERVEROUT ON SIZE UNL;** - Makes sure the size of output is unlimited
3. **SET ESC ON SQLBL ON;** - Sets ESCAPE Mode on and blank line mode
4. **SPO xplore_script_1.log;** - The script we are spooling to
5. **COL connected_user NEW_V connected_user FOR A30;** - Sets the format for the `connected_user` column
6. **SELECT user connected_user FROM DUAL;** - Gets the connected user
8. **PRO Parameter 1:** - Gets the script name
9. **PRO Name of SCRIPT file that contains SQL to be explored (required)** - Prompts for information
11. **SET DEF ^ ECHO OFF;** - Sets up more environmental settings
12. **DEF script_with_sql = '^1';** - Sets up the variable name *(*`script_with_sql`*)* to `^1`
15. **PRO Password for ^^connected_user. (required)** - Gets the password for the test case user
17. **DEF user_password = '^2';** - Sets the password for the test case user to the variable `user_password`
21. **PRO SCRIPT_WITH_SQL: ^^script_with_sql** - Gets the script file name and sets the variable `script_with_sql`
25. **PRO SELECT USER FROM DUAL;** - Selects the current connected user
26. **PRO SELECT TO_CHAR(SYSDATE, 'YYYY-MM-DD HH24:MI:SS') current_time FROM DUAL;** - Selects the date and time
27. **PRO SELECT * FROM v$version;** - Selects the current version
28. **PRO SELECT * FROM v$instance;** - Selects the instance information
29. **PRO SELECT name, value FROM v$parameter2 WHERE name LIKE '%dump_dest';** - Shows the dump destinations
30. **PRO SELECT directory_name||' '||directory_path directories FROM dba_directories WHERE directory_name LIKE 'SQLT$%' OR directory_name LIKE 'TRCA$%' ORDER BY 1;** - Checks the Oracle Directories are set up
33. **SET VER ON HEA ON LIN 2000 PAGES 1000 TRIMS ON TI OFF TIMI OFF;** - At the end of the common section, sets the required format environment.

This is the end of the common code. Now we have code that is generated in a loop. I show the first section of this script:

```
--
 1. SET ECHO ON;
 2. --in case of disconnects, suspect 6356566 and un-comment workaround in line below if needed
 3. --ALTER SESSION SET "_cursor_plan_unparse_enabled" = FALSE;
 4. WHENEVER SQLERROR EXIT SQL.SQLCODE;
 5. --
 6. COL run_id NEW_V run_id FOR A4;
 7. SELECT LPAD((NVL(MAX(run_id), 0) + 1), 4, '0') run_id FROM xplore_test;
 8. --
 9. DELETE plan_table_all WHERE statement_id LIKE 'xplore_{001}_[^^run_id.]_(%)';
10. EXEC xplore.set_baseline(1);
11. --
12. SET BLO .
13. GET ^^script_with_sql.
14. .
15. C/;/
16. 0 EXPLAIN PLAN SET statement_id = 'xplore_{001}_[^^run_id.]_(00000)' INTO plan_table_all FOR
17. L
18. /
19. EXEC xplore.snapshot_plan('xplore_{001}_[^^run_id.]_(00000)', 'XPLAIN', 'Y');
20. WHENEVER SQLERROR CONTINUE;
```

These 20 lines are the core of XPLORE. I'll explain what is happening here.

> **Lines 1–4.** Ensures that echo is on so that we get some output and ensure that an error stops the SQL
>
> **Lines 5–8.** Sets the format for the column `run_id`, and selects the current `run_id` from the table `xplore_test`
>
> **Line 9.** Deletes the entries in the `PLAN_TABLE` where there are any matching statements to the about to be executed statement.
>
> **Line 10.** Sets the baseline that we decided as users to set before the execution of every iteration of the XPLORE
>
> **Lines 11–14.** Sets the block termination to “.”, gets the script (in our case q.sql) and stops the block
>
> **Lines 15–17.** Removes the “;” from the script you just got from the file and replaces it with a blank (we are going to execute this with “/” not “;”); then sets line `0` as `EXPLAIN PLAN` etc., appends the SQL you got from the file with `L`, and executes the combination statement. The complete line should look like this:

```
EXPLAIN PLAN SET statement_id = 'xplore_{001}_[0001]_(00001)' INTO plan_table_all FOR
select /* ^^001 */  country_name, sum(AMOUNT_SOLD) from sh.sales s, sh.customers c,
sh.countries co where s.cust_id=c.cust_id and co.country_id=c.country_id and country_
name in ('Ireland','Denmark','Poland','United Kingdom','Germany','France','Spain','The
Netherlands','Italy') group by country_name order by sum(AMOUNT_SOLD);
```

Lines 18–20. Executes the above statement and uses the snapshot_plan routine that captures all the information in the plan table and other information and stores it all in the internal SQLT repository ready to be output at the end in the final report.

Once you've reviewed this script and are happy that it will produce the right results, then you can run the xplore_script_1.sql. I know I've gone into more detail here than is needed to use the script, but these few lines are the very core of the XPLORE method. We could go into more detail and look at the procedure snapshot_plan in xplore.pkb, but that is beyond the scope of this book.

Running the Script

Running the script is just a matter of executing xplore_script_1.sql. I show below the first run through of the script, with the collection of the parameters:

```
SQL>@xplore_script_1.sql
TC64661

Parameter 1:
Name of SCRIPT file that contains SQL to be xplored (required)
Note: SCRIPT must contain comment /* ^^unique_id */
Enter value for 1: q.sql
Parameter 2:
Password for TC64661 (required)
Enter value for 2: TC64661
Value passed to xplore_script.sql:
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
SCRIPT_WITH_SQL: q.sql
-- begin common
DEF _SQLPLUS_RELEASE
SELECT USER FROM DUAL
SELECT TO_CHAR(SYSDATE, 'YYYY-MM-DD HH24:MI:SS') current_time FROM DUAL
SELECT * FROM v$version
SELECT * FROM v$instance
SELECT name, value FROM v$parameter2 WHERE name LIKE '%dump_dest'
SELECT directory_name||' '||directory_path directories FROM dba_directories WHERE directory_name
LIKE 'SQLT' ORDER BY 1
-- end common
SQL>--in case of disconnects, suspect 6356566 and un-comment workaround in line below if needed
SQL>--ALTER SESSION SET "_cursor_plan_unparse_enabled" = FALSE;
SQL>WHENEVER SQLERROR EXIT SQL.SQLCODE;
SQL>--
SQL>COL run_id NEW_V run_id FOR A4;
SQL>SELECT LPAD((NVL(MAX(run_id), 0) + 1), 4, '0') run_id FROM xplore_test;
RUN_
----
0001
SQL>--
SQL>DELETE plan_table_all WHERE statement_id LIKE 'xplore_{001}_[^^run_id.]_(%)';
old   1: DELETE plan_table_all WHERE statement_id LIKE 'xplore_{001}_[^^run_id.]_(%)'
new   1: DELETE plan_table_all WHERE statement_id LIKE 'xplore_{001}_[0001]_(%)'
```

We've set the environment now and have to set any baseline set up before running the script we are investigating.

```
SQL>EXEC xplore.set_baseline(1);
--
-- begin set_baseline
--
--
-- end set_baseline
--
SQL>--
SQL>ALTER SESSION SET STATISTICS_LEVEL = ALL;
SQL>DEF unique_id = "xplore_{001}_[^^run_id.]_(00000)"
SQL>@^^script_with_sql.
SQL>REM $Header: 215187.1 sqlt_s64661_tc_script.sql 11.4.4.6 2012/12/13 carlos.sierra $
SQL>
SQL>
SQL>select
  2   /* ^^unique_id */  country_name, sum(AMOUNT_SOLD)
  3  from sh.sales s, sh.customers c, sh.countries co
  4  where
  5    s.cust_id=c.cust_id
  6    and co.country_id=c.country_id
  7    and country_name in (
  8      'Ireland','Denmark','Poland','United Kingdom',
  9       'Germany','France','Spain','The Netherlands','Italy')
 10    group by country_name order by sum(AMOUNT_SOLD);
old   2:  /* ^^unique_id */  country_name, sum(AMOUNT_SOLD)
new   2:  /* xplore_{001}_[0001]_(00000) */  country_name, sum(AMOUNT_SOLD)
COUNTRY_NAME                             SUM(AMOUNT_SOLD)
---------------------------------------- ----------------
Poland                                            8447.14
Denmark                                        1977764.79
Spain                                          2090863.44
France                                         3776270.13
Italy                                          4854505.28
United Kingdom                                 6393762.94
Germany                                        9210129.22
SQL>EXEC xplore.snapshot_plan('xplore_{001}_[^^run_id.]_(00000)', 'XECUTE', 'Y');
SQL>WHENEVER SQLERROR CONTINUE;
SQL>--
```

After this first run we see that we do get results (as I selected to have data in my XPLORE). The data from this run will be collected, along with all the other runs into the SQLT repository ready to produce the report at the end of the run.

Reviewing the Results

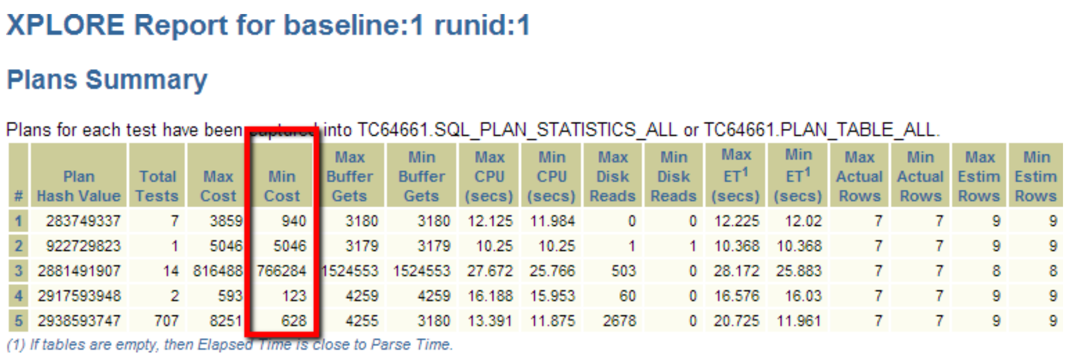
When the script finally finishes (it can take hours if you have data) you will see a messages indicating that the HTML files are being compressed and the main report is being created. The XPLORE Completed message is a big clue that we are now ready to read the report. Here is the output at the end of the XPLORE run.

```
  adding: xplore_sql_monitor_report_1_00728.html (164 bytes security) (deflated 81%)
  adding: xplore_sql_monitor_report_1_00729.html (164 bytes security) (deflated 80%)
  adding: xplore_sql_monitor_report_1_00730.html (164 bytes security) (deflated 80%)
  adding: xplore_sql_monitor_report_1_00731.html (164 bytes security) (deflated 80%)
test of xplore_sql_monitor_report_1.zip OK
  adding: xplore_report_1.html (164 bytes security) (deflated 96%)
  adding: xplore_script_1.log (164 bytes security) (deflated 98%)
  adding: xplore_script_1.sql (164 bytes security) (deflated 95%)
  adding: xplore_sql_monitor_report_1.zip (164 bytes security) (stored 0%)
test of xplore_1.zip OK
        zip warning: error deleting xplore_script_1.sql
XPLORE Completed.
Disconnected from Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - Production
With the Partitioning, OLAP, Data Mining and Real Application Testing options
```

A new file has been created called xplore_1.zip, which is in the directory where we ran the XPLORE superscript.

```
C:\Documents and Settings\Stelios\Desktop\SQLT 11.4.4.6\sqlt\utl\xplore>dir
 Volume in drive C has no label.
 Volume Serial Number is 77E9-80B4

 Directory of C:\Documents and Settings\Stelios\Desktop\SQLT 11.4.4.6\sqlt\utl\xplore

12/22/2012  12:28 PM    <DIR>          .
12/22/2012  12:28 PM    <DIR>          ..
08/11/2011  01:19 AM             1,696 create_xplore_script.sql
07/09/2011  07:01 PM               325 drop_sys_views.sql
08/11/2011  12:46 AM               499 drop_user_objects.sql
08/11/2011  12:46 AM               537 install.sql
12/13/2012  08:53 PM               452 q.sql
08/11/2011  12:46 AM             2,622 readme.txt
07/09/2011  07:01 PM             3,972 sys_views.sql
08/11/2011  12:46 AM               184 uninstall.sql
08/11/2011  12:46 AM             6,844 user_objects.sql
04/02/2012  10:43 AM            58,807 xplore.pkb
10/11/2011  06:29 AM             2,843 xplore.pks
12/22/2012  12:28 PM         3,839,878 xplore_1.zip
12/22/2012  09:45 AM           301,600 xplore_script_1.sql
12/22/2012  09:31 AM             3,239 xplore_script_2.log
12/22/2012  09:29 AM           301,600 xplore_script_2.sql
12/22/2012  09:33 AM             2,341 xplore_script_3.log
12/22/2012  09:32 AM           301,600 xplore_script_3.sql
              17 File(s)      4,829,039 bytes
               2 Dir(s)   6,701,252,608 bytes free
```

To use this report it is best to create a sub-directory and unzip the files in `xplore_1.zip`. In the zip file is an HTML file (the main report), a log file of the run, and the `xplore` script that was run. The only file of interest to us now is the HTML file. If we open this file with a browser we see it is much less fussy than a normal XECUTE or XTRACT report. It starts with a simple title "XPLORE Report for baseline:1 runid:1", and we are straight into endless numbers. This "Plans Summary" part of the report shows all the discovered plans for our one SQL. Let's look at Figure 12-2, which shows this part of the report. This is the jumping off point to all the other parts of the report.



XPLORE Report for baseline:1 runid:1

Plans Summary

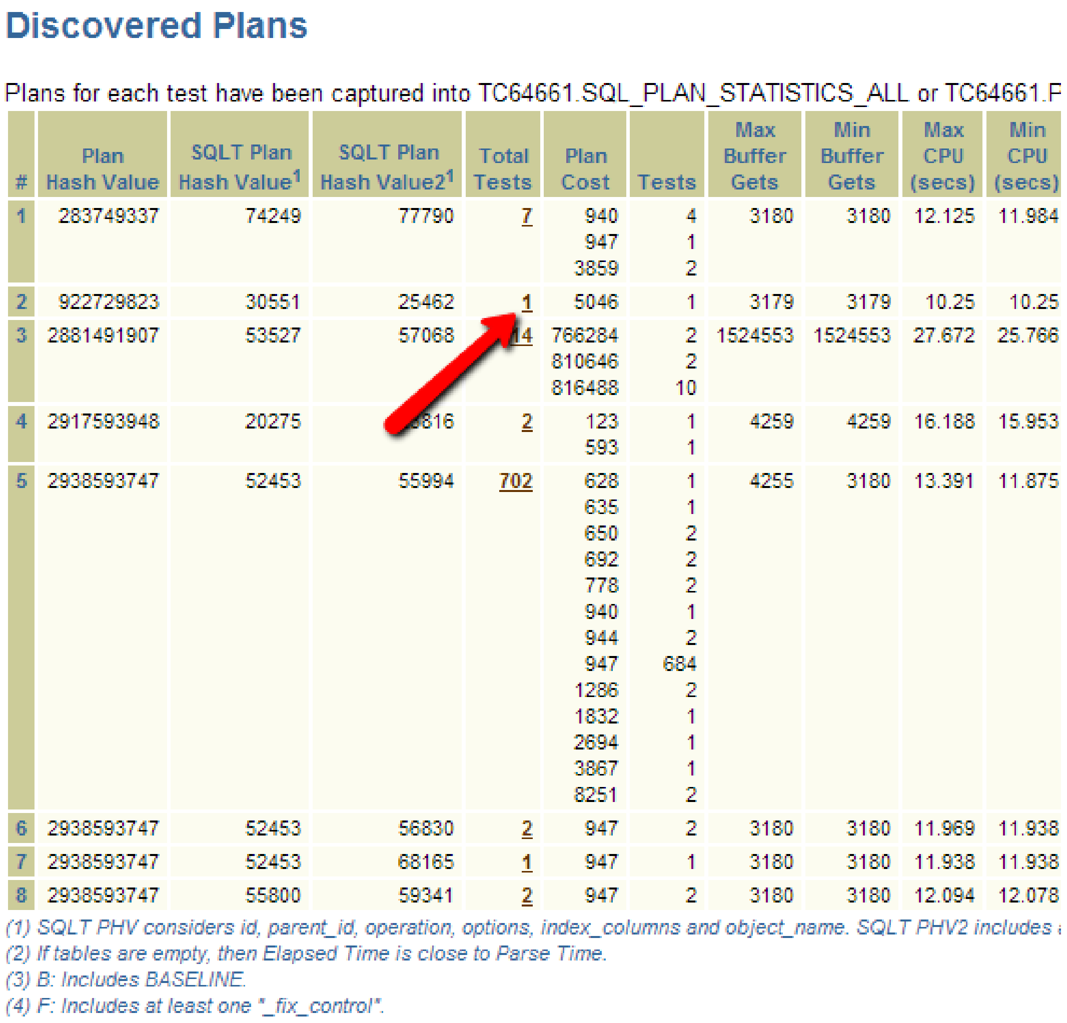
Plans for each test have been captured into TC64661.SQL_PLAN_STATISTICS_ALL or TC64661.PLAN_TABLE_ALL.

#	Plan Hash Value	Total Tests	Max Cost	Min Cost	Max Buffer Gets	Min Buffer Gets	Max CPU (secs)	Min CPU (secs)	Max Disk Reads	Min Disk Reads	Max ET[1] (secs)	Min ET[1] (secs)	Max Actual Rows	Min Actual Rows	Max Estim Rows	Min Estim Rows
1	283749337	7	3859	940	3180	3180	12.125	11.984	0	0	12.225	12.02	7	7	9	9
2	922729823	1	5046	5046	3179	3179	10.25	10.25	1	1	10.368	10.368	7	7	9	9
3	2881491907	14	816488	766284	1524553	1524553	27.672	25.766	503	0	28.172	25.883	7	7	8	8
4	2917593948	2	593	123	4259	4259	16.188	15.953	60	0	16.576	16.03	7	7	9	9
5	2938593747	707	8251	628	4255	3180	13.391	11.875	2678	0	20.725	11.961	7	7	9	9

(1) If tables are empty, then Elapsed Time is close to Parse Time.

Figure 12-2. *The top part of the XPLORE report*

There were only 5 different plans in this case (it was a very simple piece of SQL). For example, PHV 922729823 only had one test so it is not surprising that the maximum and minimum cost was the same at 5046. The original plan had a cost of 947, so this option clearly is not a good one. If we look at what that test was we'll be able to see why it didn't turn out so well. Scroll to the section show in Figure 12-3, which shows the "Discovered Plans," just below the "Plans Summary." Here I want to know why my original cost of 947 increased to 5,046. The arrow is pointing to the hyperlink that takes me to a different section of the same report, which details what happened on this test. See Figure 12-3, which shows the "Discovered Plans" section of the report.



Discovered Plans

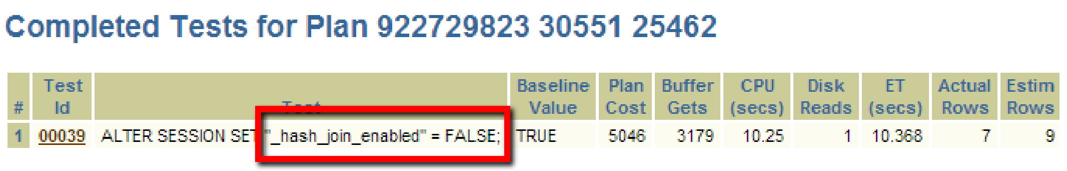
Plans for each test have been captured into TC64661.SQL_PLAN_STATISTICS_ALL or TC64661.F

#	Plan Hash Value	SQLT Plan Hash Value[1]	SQLT Plan Hash Value2[1]	Total Tests	Plan Cost	Tests	Max Buffer Gets	Min Buffer Gets	Max CPU (secs)	Min CPU (secs)
1	283749337	74249	77790	7	940 947 3859	4 1 2	3180	3180	12.125	11.984
2	922729823	30551	25462	1	5046	1	3179	3179	10.25	10.25
3	2881491907	53527	57068	14	766284 810646 816488	2 2 10	1524553	1524553	27.672	25.766
4	2917593948	20275	3816	2	123 593	1 1	4259	4259	16.188	15.953
5	2938593747	52453	55994	702	628 635 650 692 778 940 944 947 1286 1832 2694 3867 8251	1 1 2 2 2 1 2 684 2 1 1 1 2	4255	3180	13.391	11.875
6	2938593747	52453	56830	2	947	2	3180	3180	11.969	11.938
7	2938593747	52453	68165	1	947	1	3180	3180	11.938	11.938
8	2938593747	55800	59341	2	947	2	3180	3180	12.094	12.078

(1) SQLT PHV considers id, parent_id, operation, options, index_columns and object_name. SQLT PHV2 includes
(2) If tables are empty, then Elapsed Time is close to Parse Time.
(3) B: Includes BASELINE.
(4) F: Includes at least one "_fix_control".

***Figure 12-3.** You can get to the test details by clicking on the hyperlink under the "Total Tests" column*

Once we click on this hyperlink we are taken to section of the report showing what was done and what happened. Figure 12-4 shows the details for this particular plan hash value.

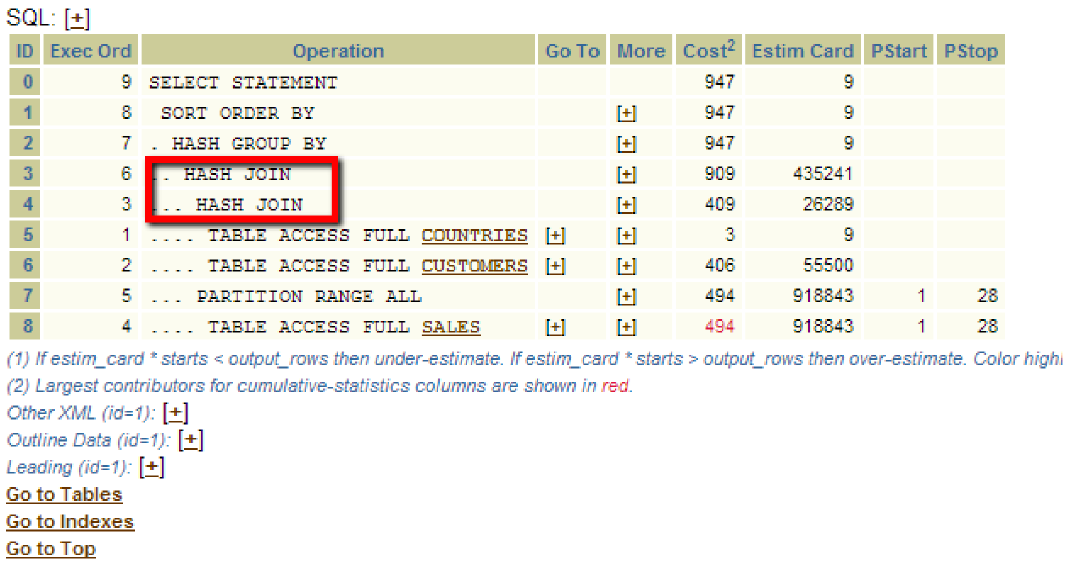


Completed Tests for Plan 922729823 30551 25462

#	Test Id	Test	Baseline Value	Plan Cost	Buffer Gets	CPU (secs)	Disk Reads	ET (secs)	Actual Rows	Estim Rows
1	00039	ALTER SESSION SET "_hash_join_enabled" = FALSE;	TRUE	5046	3179	10.25	1	10.368	7	9

***Figure 12-4.** The details for the PHV 922729823*

We can see that the test for this PHV was to set `"_hash_join_enabled"=FALSE` if we look at the original execution plan from the XTRACT report for this SQL (see Figure 12-5)



SQL: [+]

ID	Exec Ord	Operation	Go To	More	Cost²	Estim Card	PStart	PStop
0	9	SELECT STATEMENT			947	9		
1	8	SORT ORDER BY		[+]	947	9		
2	7	. HASH GROUP BY		[+]	947	9		
3	6	.. HASH JOIN		[+]	909	435241		
4	3	... HASH JOIN		[+]	409	26289		
5	1	.... TABLE ACCESS FULL COUNTRIES	[+]	[+]	3	9		
6	2	.... TABLE ACCESS FULL CUSTOMERS	[+]	[+]	406	55500		
7	5	... PARTITION RANGE ALL		[+]	494	918843	1	28
8	4	.... TABLE ACCESS FULL SALES	[+]	[+]	494	918843	1	28

(1) If estim_card * starts < output_rows then under-estimate. If estim_card * starts > output_rows then over-estimate. Color highl

(2) Largest contributors for cumulative-statistics columns are shown in red.

Other XML (id=1): [+]

Outline Data (id=1): [+]

Leading (id=1): [+]

Go to Tables

Go to Indexes

Go to Top

Figure 12-5. *The original execution plan as shown in the XTRACT report for the SQL*

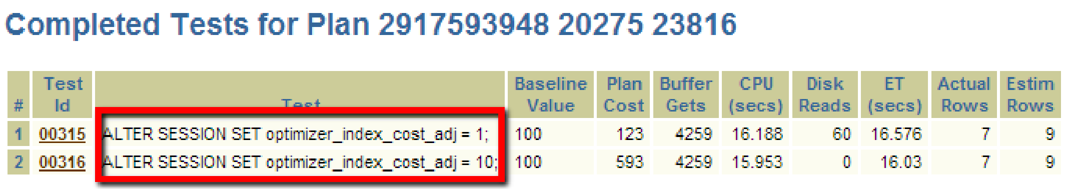
The original plan was to use "TABLE ACCESS FULL" for COUNTRIES and CUSTOMERS and then use a hash join,followed by another hash join of that with the result of the "TABLE ACCESS FULL" of SALES. The main thrust of this plan is to use hash joins. Now our XPLORE has disabled this option by setting the hidden parameter, not too surprising the plan has become more costly. If we click on the number under the "Test Id" column (as shown in Figure 12-4) we will see which plan was actually chosen. This is the plan we see.

```
-----------------------------------------------------------------------------------------
|Id |Operation                         |Name          | Cost (%CPU)| Buffers |  1Mem |  O/1/M   |
-----------------------------------------------------------------------------------------
|  0|SELECT STATEMENT                  |              |  5046 (100)|    3179 |       |          |
|  1| SORT ORDER BY                    |              |  5046   (3)|    3179 |  2048 |     1/0/0|
|  2|  HASH GROUP BY                   |              |  5046   (3)|    3179 |  770K|     1/0/0|
|  3|   MERGE JOIN                     |              |  5008   (2)|    3179 |       |          |
|  4|    SORT JOIN                     |              |   825   (1)|    1461 |  546K|     1/0/0|
|  5|     MERGE JOIN                   |              |   632   (1)|    1461 |       |          |
|* 6|      TABLE ACCESS BY INDEX ROWID|COUNTRIES     |     2   (0)|       2 |       |          |
|  7|       INDEX FULL SCAN            |COUNTRIES_PK |     1   (0)|       1 |       |          |
|* 8|      SORT JOIN                   |              |   630   (1) |    1459 |  615K|     1/0/0|
|  9|       TABLE ACCESS FULL          |CUSTOMERS     |   406   (1)|    1459 |       |          |
|*10|    SORT JOIN                     |              |  4183   (2)|    1718 | 1763K|     1/0/0|
| 11|     PARTITION RANGE ALL          |              |   494   (3)|    1718 |       |          |
| 12|      TABLE ACCESS FULL           |SALES         |   494   (3)|    1718 |       |          |
```

No sign of a hash join, as expected. The optimizer has honored our requirement and avoided this, but it has been detrimental to our plan, so we know not to force the optimizer to not use a hash join. This is where the intelligence part comes in. We know that a hash join is a good idea here, but the brute force approach of XPLORE does not.

Finding the Best Execution Plan

So enough of how the plan can be made worse. Let's go back to the plan summaries (see Figure 12-2) and now look to see if there have been any improvements. We see that there was a big improvement for PHV 2917593948. Here we see a plan that shows a minimum cost of 123 (much less than our original plan cost of 947. If we then look at the "Discovered Plans" section we see that there were two tests done. If we now click on the hyperlinked "2" we see the completed tests for this PHV (see Figure 12-6)



Completed Tests for Plan 2917593948 20275 23816

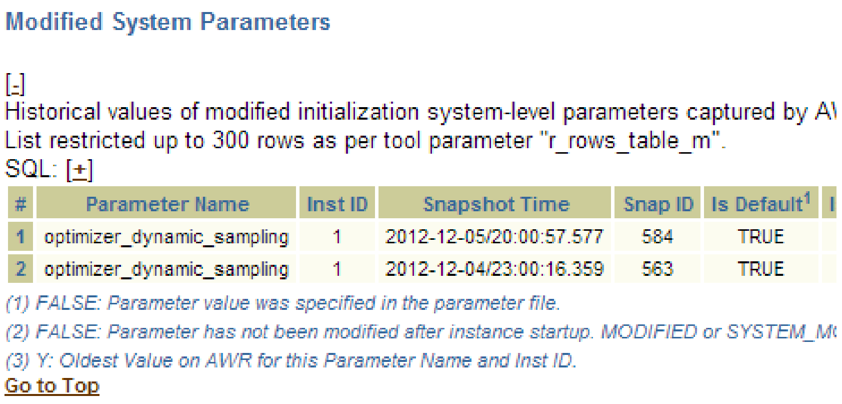
#	Test Id	Test	Baseline Value	Plan Cost	Buffer Gets	CPU (secs)	Disk Reads	ET (secs)	Actual Rows	Estim Rows
1	00315	ALTER SESSION SET optimizer_index_cost_adj = 1;	100	123	4259	16.188	60	16.576	7	9
2	00316	ALTER SESSION SET optimizer_index_cost_adj = 10;	100	593	4259	15.953	0	16.03	7	9

Figure 12-6. *The test details for PHV 2917593948*

We see in this already that the reason the cost was so low in this case was that we set `optimizer_index_cost_adj` to 1. In other words, we forced indexes to be used. We can see the plan used for test 315 by clicking on the hyperlinked 315. This is the plan we see (I've removed some columns for clarity).

```
-------------------------------------------------------------------------------------------
|Id  |Operation                            | Name                   | Cost (%CPU)| Reads  |
-------------------------------------------------------------------------------------------
|  0 |SELECT STATEMENT                     |                        |  123 (100)|    60  |
|  1 | SORT ORDER BY                       |                        |  123  (54)|    60  |
|  2 |  HASH GROUP BY                      |                        |  123  (54)|    60  |
|* 3 |   HASH JOIN                         |                        |   84  (33)|    60  |
|* 4 |    HASH JOIN                        |                        |   31  (10)|     4  |
|* 5 |     TABLE ACCESS FULL               | COUNTRIES              |    3   (0)|     0  |
|  6 |     TABLE ACCESS BY INDEX ROWID     | CUSTOMERS              |   27   (8)|     4  |
|  7 |      BITMAP CONVERSION TO ROWIDS    |                        |           |     4  |
|  8 |       BITMAP INDEX FULL SCAN        | CUSTOMERS_GENDER_BIX   |           |     4  |
|  9 |    PARTITION RANGE ALL              |                        |   49  (41)|    56  |
| 10 |     TABLE ACCESS BY LOCAL INDEX ROWID| SALES                 |   49  (41)|    56  |
| 11 |      BITMAP CONVERSION TO ROWIDS    |                        |           |    56  |
| 12 |       BITMAP INDEX FULL SCAN        | SALES_PROMO_BIX        |           |    56  |
```

It looks like using the bitmap indexes is useful in this case, as the cost is so low. We have to ask why the optimizer did not choose the index in the first place, and for this we would need to look at the statistics in the original XTRACT report. We see from the system observations that `optimizer_dynamic_sampling` is set to 1 (so in this case no dynamic sampling would have been done). See Figure 12-7, which shows this from the XTRACT report.


Modified System Parameters

[-]
Historical values of modified initialization system-level parameters captured by A\
List restricted up to 300 rows as per tool parameter "r_rows_table_m".
SQL: [+]

#	Parameter Name	Inst ID	Snapshot Time	Snap ID	Is Default[1]	I
1	optimizer_dynamic_sampling	1	2012-12-05/20:00:57.577	584	TRUE	
2	optimizer_dynamic_sampling	1	2012-12-04/23:00:16.359	563	TRUE	

(1) FALSE: Parameter value was specified in the parameter file.
(2) FALSE: Parameter has not been modified after instance startup. MODIFIED or SYSTEM_M
(3) Y: Oldest Value on AWR for this Parameter Name and Inst ID.
Go to Top

Figure 12-7. *Optimizer dynamic sampling is set to 1*

As we have a value of 1 for dynamic sampling we know that dynamic sampling could take place but in this case did not because the tables involved are not unindexed (see Chapter 8 for details on cardinality feedback and dynamic sampling). Still not using dynamic sampling is not an excuse for getting the plan wrong, but it could explain why the plan was not saved by dynamic sampling if statistics are missing.

Reviewing the Original Test Case

If we look at the observations in the original report we see that the many of the partition statistics are missing and are out of date (which doesn't matter in my case as the data is static). However as it looks like the statistics are the cause of the wrong plan being selected I'll update them and retry the SQL.

The SQL I used to collect the new up to date statistics was:

```
SQL> exec dbms_stats.gather_Table_stats(ownname=>'TC64661',
  tabname=>'SALES', estimate_percent=>dbms_stats.auto_sample_size,cascade=>TRUE)

PL/SQL procedure successfully completed.

SQL> exec dbms_stats.gather_Table_stats(ownname=>'TC64661',
  tabname=>'COUNTRIES', estimate_percent=>dbms_stats.auto_sample_size,cascade=>TRUE);

PL/SQL procedure successfully completed.

SQL> exec dbms_stats.gather_Table_stats(ownname=>'TC64661',
  tabname=>'CUSTOMERS', estimate_percent=>dbms_stats.auto_sample_size,cascade=>TRUE);

PL/SQL procedure successfully completed.
```

Remember, in this case I also used data from the source tables (which is not always the case). If I collect fresh statistics on all of the relevant tables (as shown above) and re-run my test case I get this execution plan

```
SQL> @tc

COUNTRY_NAME                             SUM(AMOUNT_SOLD)
---------------------------------------- ----------------
Poland                                            8447.14
Denmark                                        1977764.79
Spain                                          2090863.44
France                                         3776270.13
Italy                                          4854505.28
United Kingdom                                 6393762.94
Germany                                        9210129.22

7 rows selected.

PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------------------
-------------------------------------
SQL_ID  f43bszax8xh07, child number 0
-------------------------------------
select  /* ^^unique_id */  country_name, sum(AMOUNT_SOLD) from sales s,
customers c, countries co where   s.cust_id=c.cust_id   and
co.country_id=c.country_id   and country_name in (
'Ireland','Denmark','Poland','United Kingdom',
'Germany','France','Spain','The Netherlands','Italy')   group by
country_name order by sum(AMOUNT_SOLD)

Plan hash value: 1235134607

------------------------------------------------------------------------------------------------
| Id  | Operation                         | Name         | Rows  | Bytes | Cost (%CPU)| Time     |
------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                  |              |       |       |     4 (100)|          |
|   1 |  SORT ORDER BY                    |              |     1 |    87 |     4  (50)| 00:00:01 |
|   2 |   HASH GROUP BY                   |              |     1 |    87 |     4  (50)| 00:00:01 |
|   3 |    NESTED LOOPS                   |              |     1 |    87 |     2   (0)| 00:00:01 |
|   4 |     NESTED LOOPS                  |              |     1 |    52 |     2   (0)| 00:00:01 |
|   5 |      PARTITION RANGE ALL          |              |     1 |    26 |     2   (0)| 00:00:01 |
|   6 |       TABLE ACCESS FULL           | SALES        |     1 |    26 |     2   (0)| 00:00:01 |
|   7 |      TABLE ACCESS BY INDEX ROWID| CUSTOMERS    |     1 |    26 |     0   (0)|          |
|*  8 |       INDEX UNIQUE SCAN           | CUSTOMERS_PK |     1 |       |     0   (0)|          |
|*  9 |     TABLE ACCESS BY INDEX ROWID | COUNTRIES    |     1 |    35 |     0   (0)|          |
|* 10 |      INDEX UNIQUE SCAN            | COUNTRIES_PK |     1 |       |     0   (0)|          |
------------------------------------------------------------------------------------------------
```

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

   8 - access("S"."CUST_ID"="C"."CUST_ID")
   9 - filter(("COUNTRY_NAME"='Denmark' OR "COUNTRY_NAME"='France' OR "COUNTRY_NAME"='Germany' OR
              "COUNTRY_NAME"='Ireland' OR "COUNTRY_NAME"='Italy' OR "COUNTRY_NAME"='Poland'
              OR "COUNTRY_NAME"='Spain'
              OR "COUNTRY_NAME"='The Netherlands' OR "COUNTRY_NAME"='United Kingdom'))
  10 - access("CO"."COUNTRY_ID"="C"."COUNTRY_ID")

36 rows selected.
```

This plan is even better than the one found by XPLORE. It has a cost of 4. We found this plan because we used XPLORE to give us a hint. The hint was that using indexes would be a good idea for everything except `SALES.` We need to do a `TABLE ACCESS FULL` on `SALES` as we are summing sales for a big group of countries. However, using `TABLE ACCESS FULL` on the other tables didn't make sense. It's not surprising that the example came out this way (I planned it that way), but real life examples are exactly like this. The steps in the discovery usually go:

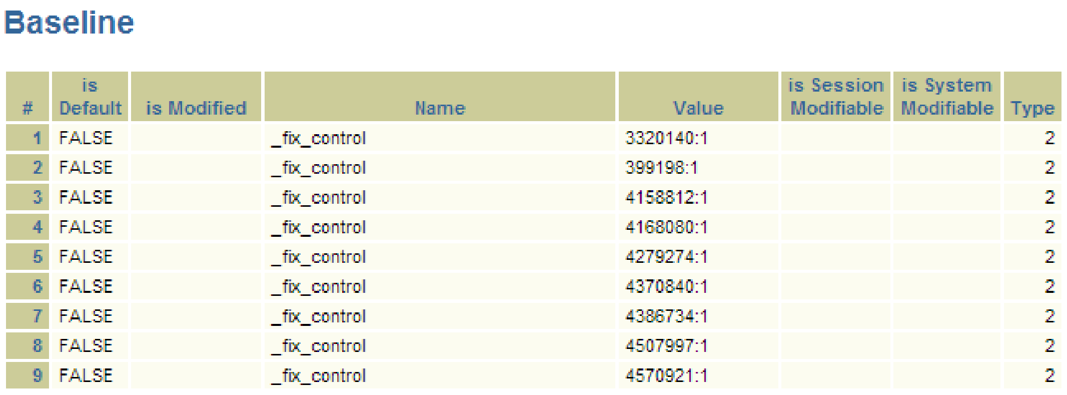
1. Something unexpected happens to some SQL (slows down invariably).
2. Get the XTRACT test case.
3. Discover what is wrong with the SQL or its environment by studying the report.
4. If the previous step fails, run an XPLORE, which sometimes finds an interesting change to consider (such as the `optimizer_index_cost_adj` in our example above).
5. We compare the "good" plan from the XPLORE with the "bad" plan from the XTRACT and figure out what the difference is in terms of optimizer steps. In our case it was lack of index use.
6. Review the XTRACT report again to see if you can determine why the action that should have taken place did not (in our case it was "why were the indexes not used").
7. Amend the original test environment to make the optimizer action take place and compare the execution plan costs again.
8. If step 7 has the desired effect, do more testing, and if everything goes as expected implement the improvement.

These steps are an example of the standard tuning methodology: test, make single changes, and test again. The test case allows you to do this quickly and efficiently. As you get more proficient with XTRACT you will find that using XPLORE is not needed. It's a bit like weather forecasting: "The accuracy of a weather forecast is in direct proportion to the number of gray hairs on the forecaster's head." The lesson to remember from this chapter is that although we found that setting `optimizer_index_cost_adj` to 1 made our execution plan much better, that was not the solution. That only prompted us to find the real solution, which was to fix the statistics.

In other words, we found a problem that was not related to an upgrade (i.e., the statistics were wrong), but XPLORE hinted at the solution. This is the surprising thing about XPLORE. Again, it is primarily used to determine what changes (due to upgrades) were made to the optimizer and may have regressed execution plans. However, if we use XPLORE *in extremis*, we may find in one of the execution plans a kernel of an idea that suggests a solution to our tuning problem, even though the original problem may not be related to changes in the optimizer behavior.

Other Information in XPLORE

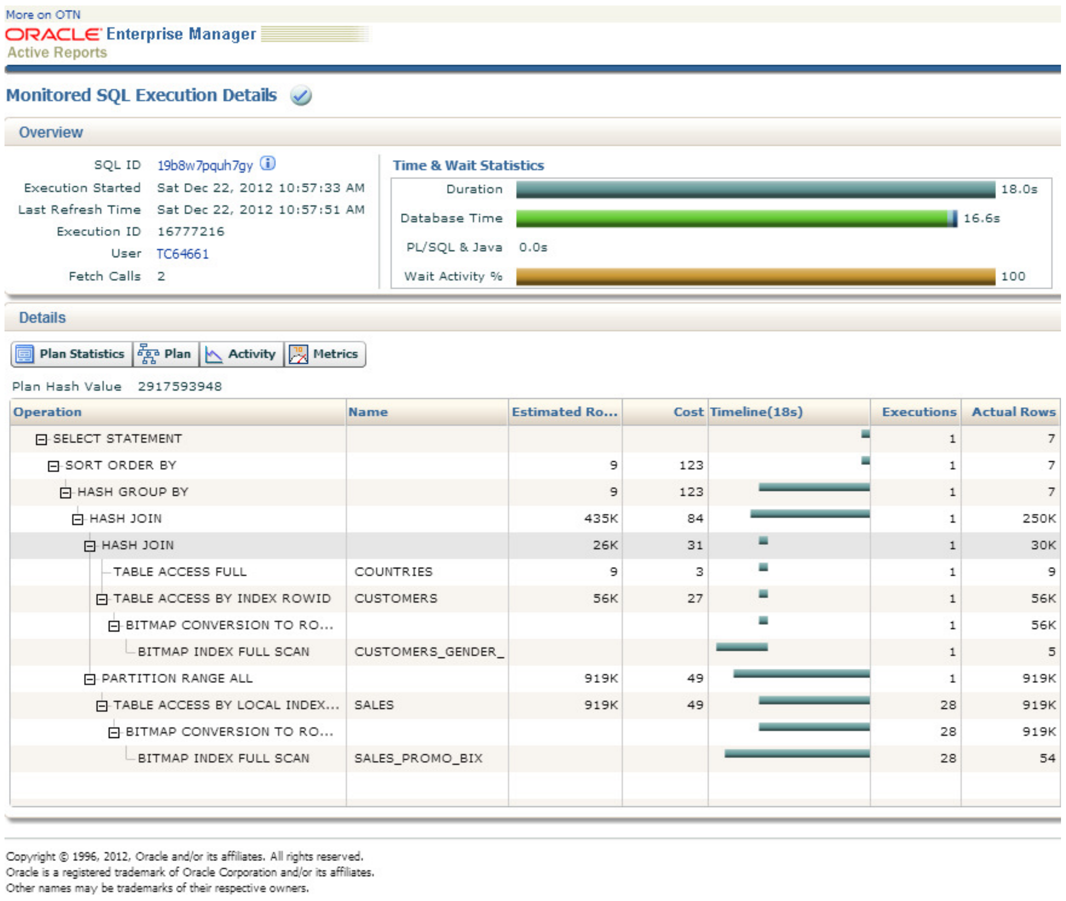
In our example XPLORE we focused directly on finding the best plan and didn't stop to look at all the other information presented by XPLORE. Here we'll briefly look at some other sections that need to be mentioned. The first one is the "Baseline" section, which shows the settings for each of the tests. See Figure 12-8, which shows the top part of the baseline section of the XPLORE report.


Baseline

#	is Default	is Modified	Name	Value	is Session Modifiable	is System Modifiable	Type
1	FALSE		_fix_control	3320140:1			2
2	FALSE		_fix_control	399198:1			2
3	FALSE		_fix_control	4158812:1			2
4	FALSE		_fix_control	4168080:1			2
5	FALSE		_fix_control	4279274:1			2
6	FALSE		_fix_control	4370840:1			2
7	FALSE		_fix_control	4386734:1			2
8	FALSE		_fix_control	4507997:1			2
9	FALSE		_fix_control	4570921:1			2

Figure 12-8. *The top part of the baselines report in XPLORE*

This part of the report shows the starting position of the tests. It starts by listing all the fix control settings as they are on the test database before we start and then goes on to list all the optimizer parameters (including the hidden ones). Along with all this information in the main report there is also a zip file created in the XPLORE zip file (yes, a zip file inside a zip file). This zip file called `xplore_sql_monitor_report_1.zip` contains the SQL monitor HTML output for every single execution of the script. This is a vast amount of information. You would never look at every single one of these execution reports, but if you have data and you have finally settled on a plan that you like, such as our test Id 315 from above, then we can investigate the SQL monitor report for this one test more closely. As usual you should create a directory and put the zip file in there. Then unzip the file. In this directory you will end up with 732 HTML reports: one for each test and one SQL file that shows how the scripts were created. Let's look at one of these monitor reports, specifically report 315. Look at Figure 12-9, which shows the left hand side of the monitor report for test Id 315.


More on OTN
ORACLE Enterprise Manager
Active Reports

Monitored SQL Execution Details

Overview

SQL ID	19b8w7pquh7gy
Execution Started	Sat Dec 22, 2012 10:57:33 AM
Last Refresh Time	Sat Dec 22, 2012 10:57:51 AM
Execution ID	16777216
User	TC64661
Fetch Calls	2

Time & Wait Statistics

Duration	18.0s
Database Time	16.6s
PL/SQL & Java	0.0s
Wait Activity %	100

Details

Plan Statistics | Plan | Activity | Metrics

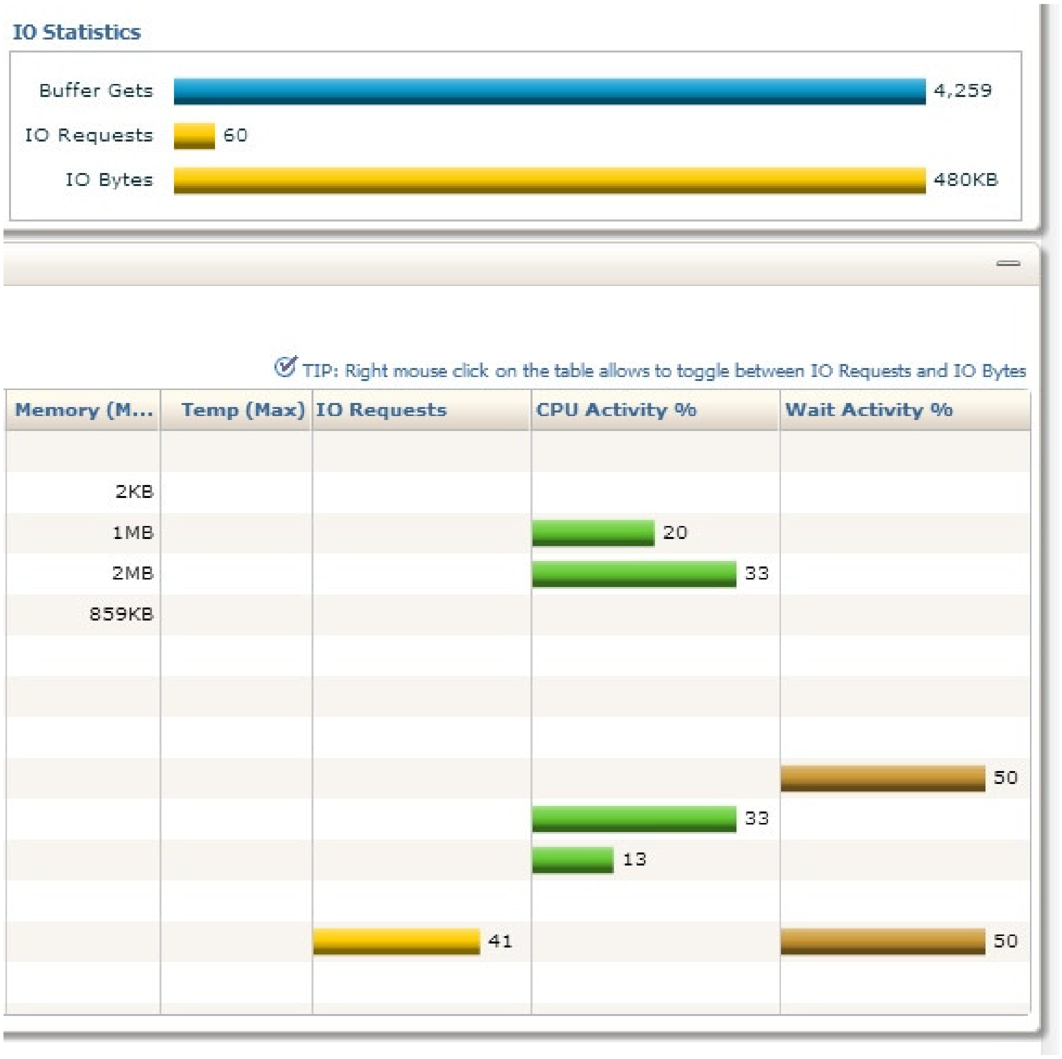
Plan Hash Value 2917593948

Operation	Name	Estimated Ro...	Cost	Timeline(18s)	Executions	Actual Rows
⊟ SELECT STATEMENT					1	7
⊟ SORT ORDER BY		9	123		1	7
⊟ HASH GROUP BY		9	123		1	7
⊟ HASH JOIN		435K	84		1	250K
⊟ HASH JOIN		26K	31		1	30K
TABLE ACCESS FULL	COUNTRIES	9	3		1	9
⊟ TABLE ACCESS BY INDEX ROWID	CUSTOMERS	56K	27		1	56K
⊟ BITMAP CONVERSION TO RO...					1	56K
BITMAP INDEX FULL SCAN	CUSTOMERS_GENDER_				1	5
⊟ PARTITION RANGE ALL		919K	49		1	919K
⊟ TABLE ACCESS BY LOCAL INDEX...	SALES	919K	49		28	919K
⊟ BITMAP CONVERSION TO RO...					28	919K
BITMAP INDEX FULL SCAN	SALES_PROMO_BIX				28	54

Copyright © 1996, 2012, Oracle and/or its affiliates. All rights reserved.
Oracle is a registered trademark of Oracle Corporation and/or its affiliates.
Other names may be trademarks of their respective owners.

Figure 12-9. *The left hand side of the SQL monitor report for test ID 315*

We can see the execution plan and the cost for each step, but we can also see the amount of time spent on each step of the plan. We can see that much of the time was spent on BITMAP INDEX FULL SCAN of SALES_PROMO_BIX. In fact if we hover the mouse over the duration bar for this bitmap index we will see the duration in seconds for that step. If we look at the right hand side of the same report (as shown in Figure 12-10).



IO Statistics

Buffer Gets	4,259
IO Requests	60
IO Bytes	480KB

TIP: Right mouse click on the table allows to toggle between IO Requests and IO Bytes

Memory (M...	Temp (Max)	IO Requests	CPU Activity %	Wait Activity %
2KB				
1MB			20	
2MB			33	
859KB				
				50
			33	
			13	
		41		50

Figure 12-10. *The right hand side of the SQL Monitor report for plan ID 315*

This shows that 50 percent of the wait activity was for BITMAP INDEX FULL SCAN on CUSTOMERS and the other 50 percent was for a BITMAP INDEX FULL SCAN on SALES. Because this plan was obtained by setting `optimizer_index_cost_adj` to 1, we can see that indexes were used inappropriately and the change has resulted in a better plan than the original bad plan. With this information we know that working on these two indexes is the way to improve the execution time because they are the biggest contributors to the wait time.

Summary

I hope you are impressed with what XPLORE can do for you, especially in conjunction with XTRACT and XECUTE and of course with your knowledge of how the optimizer works. Although the example in this chapter was a nice simple piece of SQL, this same methodology can work with very complex SQL and execution plans. Large pieces of SQL can be broken down into smaller steps and attacked in order of importance, the importance being governed by the cost of each step. XPLORE can investigate all the changes that are likely to have affected the optimizer's calculations, and this is most often the case when the optimizer changes when the versions of the database change. Use XPLORE wisely and you will capture more of those pesky SQLs that go awry.

I must emphasize that all of the XPLORE activity must be run on a disposable database. It bears repeating that setting environmental variables and system parameters and statistics that XPLORE does could cause havoc for any system that you might be sharing with somebody else.

For all its power and flexibility XPLORE is not the ideal tool for general tuning but is extremely useful in those cases where something has changed unexpectedly. In the next chapter we'll look at the more advanced methods available in SQLT.

CHAPTER 13

Trace Files, TRCANLZR and Modifying SQLT behavior

Even with SQLT helping you sometimes you need to look at 10046 trace files and analyze these to determine what's happening. How you analyze and interpret 10046 files is outside the scope of this book. The standard guide for this is Cary Millsap's book *Optimizing Oracle Performance* (O'Reilly 2003). (This book is a little dated now, but still contains useful information on 10046 trace files.) How to more easily collect these trace files, however, and format them so they are easier to interpret is not beyond the scope of this book, as SQLT provides a number of methods for collecting this information more quickly and more easily.

The 10046 trace files are log files of the Oracle engine's activity as it executes SQL. These files are difficult to interpret, full of obscure codes, and also very long. The 10053 trace files (as I mentioned in Chapter 5) are equally obscure, difficult to understand, and also very long. When SQL statements are executed in parallel and 10046 tracing is enabled, each slave process that is helping with the execution creates its own trace file. This makes the interpretation and understanding of what is happening with parallel execution even more difficult. You can have hundreds of trace files all relating to the same SQL all working in parallel and communicating with each other to achieve a common goal. When things go wrong with these complex cases it can be hard enough just collecting the right trace files let alone analyzing those trace files and getting a coherent picture from potentially hundreds of trace files.

In this chapter I'll describe the following methods building from the simplest to the more complex. I'll try building on the simpler cases to explain the more complex cases.

- 10046 is the raw trace file or files of information.
- TKPROF provides a high-level view of the 10046 trace file. The input to this is the 10046 trace file, the output is the TKPROF report.
- TRCASPLIT takes the raw 10046 and 10053 trace file or files and splits off the 10046 trace file information. This data can then be fed into TKPROF or any other utility that needs 10046 trace.
- 10053 is the raw trace file of the optimizer's choices during parsing.
- TRCANLZR takes the raw 10046 and produces a graphical report that makes understanding the data easier.
- TRCAXTR takes the raw 10046 trace and produces a SQLTXTRACT report.

Each of the above tools takes 10046 or 10046 and 10053 mixed together and produces some simpler version of the information.

We'll start with methods to collect 10046 trace, then look at the oldest tool available, TKPROF (which has been around since Oracle 7, and not part of SQLT) , then we'll look at TRCASPLIT, which is used to separate 10046 and (usually) 10053 trace information. Consider this method the simplest of the advanced tools. Then we'll look at collecting 10053 trace. TRCANLZR is the next step, and it works from a single or multiple files to produce an extended version of TKPROF. TRCANLZR is part of SQLT unlike TKPROF.

Finally we'll talk about TRCAXTR, which combines TRCANLZR and XTRACT to produce a report on multiple SQLs. Each of these sections will be explained with the aid of an example SQL.

10046 Trace

10046 is essentially a low-level logging facility of the activity of the Oracle engine while it is processing your SQL statement. It produces a file called a "trace" file, which is a text file, with details of the low-level step of the execution. This trace can be configured (depending on the trace level selected) to collect wait events, bind variables, and bind values (see below for the different levels that can be selected). There is an overhead in collecting this information, and it can consume a considerable amount of disk space and take resources away from the executing statement in producing the trace file.

Why Collect 10046 Trace?

Collecting 10046 trace is not an activity to take lightly. As mentioned above it can take considerable resource (especially disk space). Considering the overhead in collecting 10046 trace you would not lightly choose to collect this information. It would not normally be collected unless there was a good reason. To investigate tuning problems that are not solvable by looking at the high-level aggregated information, 10046 trace is specifically collected. In these more difficult cases some low-level piece of information (which is lost in the aggregation) is needed as a vital clue in the puzzle we are solving. This is because it contains very detailed information of what is happening during the SQL execution.

The 10046 trace can also be produced by enabling trace in other ways, such as at system level or by tracing another session instead of your own. The information collected is the same and the decode is the same; only the method of initiating the trace will change. In each case the trace file is a stand-alone file found in the `user_dump_dest` area (controlled by the parameter `diagnostic_dest` for 11g Release 1 and above).

10046 Decode

Some people will read 10046 directly; this can be useful, as it contains low-level information that is sometimes not included in an aggregation (this is the nature of aggregation). This is a typical snippet of a SQL 10046 trace file.

```
EXEC #1:c=0,e=162,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=1,plh=2938593747,tim=6801557607
WAIT #1: nam='SQL*Net message to client' ela= 6 driver id=1111838976 #bytes=1 p3=0 obj#=-1
tim=6801557680
WAIT #1: nam='Disk file operations I/O' ela= 821 FileOperation=2 fileno=5 filetype=2 obj#=-1
tim=6801559064
*** 2012-12-26 09:59:43.625
WAIT #1: nam='asynch descriptor resize' ela= 4 outstanding #aio=0 current aio limit=4294967295 new
aio limit=257 obj#=-1 tim=6802577972
WAIT #1: nam='asynch descriptor resize' ela= 2 outstanding #aio=0 current aio limit=4294967295 new
aio limit=257 obj#=-1 tim=6802578401
FETCH #1:c=0,e=1020838,p=0,cr=3180,cu=0,mis=0,r=1,dep=0,og=1,plh=2938593747,tim=6802578585
WAIT #1: nam='SQL*Net message from client' ela= 302 driver id=1111838976 #bytes=1 p3=0 obj#=-1
tim=6802578974
WAIT #1: nam='SQL*Net message to client' ela= 3 driver id=1111838976 #bytes=1 p3=0 obj#=-1
tim=6802579032
FETCH #1:c=0,e=45,p=0,cr=0,cu=0,mis=0,r=6,dep=0,og=1,plh=2938593747,tim=6802579064
STAT #1 id=1 cnt=7 pid=0 pos=1 obj=0 op='SORT ORDER BY (cr=3180 pr=0 pw=0 time=0 us cost=947
size=315 card=9)'
```

```
STAT #1 id=2 cnt=7 pid=1 pos=1 obj=0 op='HASH GROUP BY (cr=3180 pr=0 pw=0 time=18 us cost=947
size=315 card=9)'
STAT #1 id=3 cnt=250069 pid=2 pos=1 obj=0 op='HASH JOIN  (cr=3180 pr=0 pw=0 time=1261090 us cost=909
size=15233435 card=435241)'
```

Easy to understand, right? In the subsections below I have produced a 10046 trace file we can use to see the sort of sections that appear in a trace file and do at least a partial decode. Here is the SQL example:

```
SQL> alter session set events '10046 trace name context forever, level 64';

Session altered.

SQL> select count(*) from sh.sales;

  COUNT(*)
----------
    918843
```

SQL> exit

I've broken the decode of a 10046 trace file into three main sections. The header, the main section record format and the decode of the details on each line.

The Header

Although the header itself is not part of the SQL execution it is important that you look at the header. Here is an example header of the SQL I have executed.

```
Trace file f:\app\stelios\diag\rdbms\snc1\snc1\trace\snc1_ora_7892.trc
Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - Production
With the Partitioning, OLAP, Data Mining and Real Application Testing options
Windows XP Version V5.1 Service Pack 3
CPU                 : 2 - type 586, 2 Physical Cores
Process Affinity    : 0x0x00000000
Memory (Avail/Total): Ph:988M/3455M, Ph+PgF:2515M/5337M, VA:1251M/2047M
Instance name: snc1
Redo thread mounted by this instance: 1
Oracle process number: 20
Windows thread id: 7892, image: ORACLE.EXE (SHAD)

*** 2013-02-03 12:41:50.754
*** SESSION ID:(16.15175) 2013-02-03 12:41:50.754
*** CLIENT ID:() 2013-02-03 12:41:50.754
*** SERVICE NAME:(SYS$USERS) 2013-02-03 12:41:50.754
*** MODULE NAME:(sqlplus.exe) 2013-02-03 12:41:50.754
*** ACTION NAME:() 2013-02-03 12:41:50.754
```

The header section sets the stage for the information to follow. This should be checked to make sure you are looking at the correct trace file. For example the right instance, and the right time. This section also gives you some basic information about the resources on the node you are on such as the memory available see the "Memory (Avail/Total)" line and the number of CPUs. The 10046 trace starts immediately after this section.

The Main 10046 Tracing Section

Once we get into the main 10046 trace section we see a single line for each piece of information bounded by a divider. An example is shown. I've truncated the lines on the right because we are concentrating on the text at the beginning of the line and emphasising the trace file has a logical structure. We see at the beginning of each line the code words:

```
==============
PARSING IN CURSOR - Gives you the cursor number
select  - This is statement being issued on your behalf
END OF STMT - The end of the SQL statement is marked
PARSE - Now we parse the statement
EXEC - Now we execute the statement
FETCH - Now we fetch rows for the statement
STAT - Status information
WAIT - Wait for something
XCTEND - Transaction ends
CLOSE - Close the cursor we are finished with this statement.
==============
```

Here is what it looks like in the raw form (I've truncated the lines on the right to makes it less confusing).

```
CLOSE #2:c=0,e=19,dep=0,type=1,tim=839550052037
=====================
PARSING IN CURSOR #2 len=29 dep=0 uid=0 oct=3 lid=0 tim=843704111499 hv=3864810328
select count(*) from sh.sales
END OF STMT
PARSE #2:c=0,e=88,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=1,plh=1123225294,tim=843704111495
EXEC #2:c=0,e=134,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=1,plh=1123225294,tim=843704111781
WAIT #2: nam='SQL*Net message to client' ela= 10 driver id=1111838976 #bytes=1 p3=0
FETCH #2:c=0,e=2502,p=0,cr=140,cu=0,mis=0,r=1,dep=0,og=1,plh=1123225294,tim=843704114422
STAT #2 id=1 cnt=1 pid=0 pos=1 obj=0 op='SORT AGGREGATE (cr=140 pr=0 pw=0 time=0 us)'
STAT #2 id=2 cnt=54 pid=1 pos=1 obj=0 op='PARTITION RANGE ALL PARTITION: 1 28 (cr=140
STAT #2 id=3 cnt=54 pid=2 pos=1 obj=0 op='BITMAP CONVERSION COUNT (cr=140 pr=0 pw=0 t
STAT #2 id=4 cnt=54 pid=3 pos=1 obj=74275 op='BITMAP INDEX FAST FULL SCAN SALES_PROMO
WAIT #2: nam='SQL*Net message from client' ela= 301 driver id=1111838976 #bytes=1 p3=0
FETCH #2:c=0,e=5,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=0,plh=1123225294,tim=843704115083
WAIT #2: nam='SQL*Net message to client' ela= 5 driver id=1111838976 #bytes=1 p3=0 ob

*** 2013-02-03 13:51:07.597
WAIT #2: nam='SQL*Net message from client' ela= 2865946 driver id=1111838976 #bytes=1
XCTEND rlbk=0, rd_only=1, tim=843706981465
CLOSE #2:c=0,e=31,dep=0,type=0,tim=843706981612
```

Although the raw 10046 trace can look intimidating, it has the huge advantage that it is logical in its layout. This makes it relatively easy to decode with utilities. For example TKPROF (which we'll mention later) can read this raw file and produce an aggregated file. The downside of such facilities is that the interpretation of the file may leave out details you need or the decode can be wrong, causing you to see the wrong information and possibly come to the wrong conclusion.

Decoding Records and Keywords

The main 10046 tracing section consists of record keywords (at the beginning of a line) and keywords within the record (detail keywords), which give detailed information about what is happening. Table 13-1 provides the decodes for the record keywords.

Table 13-1. *The 10046 Record Keywords*

Record Keyword	Description
APPNAME	Application name setting
PARSING IN CURSOR #n	The cursor number which is currently being parsed
PARSE ERROR	Seen if there is a parsing error
ERROR	Seen if there is an error
END OF STMT	The end of the SQL statement is marked
PARSE #n	The cursor number being parsed
BINDS #n	Shows the binds information if selected
EXEC #n	The cursor number being executed
FETCH #n	The cursor number for which we are fetching rows
RPC	Remote procedure call
SORT UNMAP	Closing operating system temporary files
STAT #n	Status information for the cursor number
UNMAP	Closing of a temporary segment
CLOSE #n	Close the cursor, we are finished with this statement
WAIT #n	A wait event
XCTEND	Transaction end

The detail keywords occur within the record itself and are preceded by record keywords. The table below covers the majority of the keywords used; however, there may be others.

Detail Keyword	Description
act	Action
ad	Address of SQL
c	The number of CPU seconds used
card	Estimated cardinality
cnt	Number of rows
cost	Optimizer cost
cr	The number of consistent reads
cu	Current mode consistent reads
dep	The depth of the cursor. 0 represents a top level statement. dep=1 and 2 indicate trigger involvement and dep=3 represents a trigger called from a trigger.

(*continued*)

Detail Keyword	Description
dty	Data Type
e	Elapsed time in microseconds
err	Standard Error code
flg	Flag indicating bind status
hv	Hash value
len	The character count of the string representing the SQL statement
lid	Privilege user ID
mis	Number of shared pool misses
mod	Module name
mx1	Maximum length of bind variables
oct	The Oracle command type (2=insert, 3=select, 6=update, 7=delete, 26=lock table, 35=alter database, 42=alter session, 44=commit, 47=anonymous block, 45=rollback)
nam	Name of the wait
oacflg	Bind options
obj	Object ID
og	Optimizer goal. 1=ALL_ROWS, 2=FIRST_ROWS, 3=RULE, 4=CHOOSE
op	The operation being done. Examples are PARTITION RANGE ALL, SORT AGGREGATE etc.
p	Physical blocks read from disk
p1, p2, p3	Parameter for a given wait
pr	Physical reads
pre	Precision
pid	Parent ID of the row source
pw	Physical writes
r	Number of rows returned
rd_only	No data changed in the database on commit
rlbk	Rollback. 0=Commit, 1=Rollback
size	Estimated size in bytes
sqlid	The SQL ID inside single quotes
tim	The time stamp. Measured in 1 millionths of a second.
time	Elapsed time in microseconds
uid	The user Id of the schema doing the parsing
value	The value of a bind variable

So, for example, given this line in the raw 10046 trace file:

```
PARSE #2:c=0,e=88,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=1,plh=1123225294,tim=843704111495
```

We could translate this into:

"We are parsing in cursor number2, which was issued directly from the application, the elapsed time was 88 microseconds and no physical reads were done, nor any consistent reads or current mode reads. The cursor was not hard parsed and no rows were returned, the depth of the cursor was top level, and the optimizer goal was ALL_ROWS for the plan hash value 1123225294, the time stamp was 843704111495."

As you can see, this dense tracing information has a lot of information in it. A number of tools have grown up around interpreting this and managing the information. But why do we need this trace file in the first place?

How Do We Collect 10046 Trace

There are different ways 10046 can be collected and at different levels. If you have the luxury of being able to execute the SQL statement directly, then you can collect SQL trace with:

```
SQL> alter session set sql_trace=true
```

To turn this off

```
SQL> alter session set sql_trace=false
```

Or if you want to adjust the level of trace (discussed below):

```
SQL> alter session set events '10046 trace name context forever, level n'
```

To turn this off:

```
SQL> alter session set events '10046 trace name context off'
```

You can also enable trace for another session (with pid n) with an oradebug command.

```
SQL> oradebug setorapid n
SQL> oradebug event 10046 trace name contect forever, level m;
```

This issues the trace for the pid n to 10046 trace level m.

You can even trace a session with a log on trigger, that is a trigger, which is fired when a session logs into the database. Before the user session begins some commands are carried out which enable tracing for that session. The "triggering" code usually begins with something like this:

```
Create or replace trigger Start_10046_trace after logon on database
  begin
  execute immediate 'alter session set timed_statistics=true';
  execute immediate 'alter session set events "10046 trace name context forever, level 4" '
end;
```

The above trigger will trace every session that logs into the database, you may want to restrict the collection of trace information by checking on the user or some other user context to limit the amount of trace collected. You may also want to identify each trace file and set the file size to unlimited. In these cases you should add code such as this before the 10046 trace command:

```
execute immediate 'alter session set max_dump_file_size=unlimited';
```

and/or

```
execute immediate 'alter session set tracefile_identifier="My_trace"';
```

Different Levels of Trace

Different types of information are collected for different investigations. Here are the levels and what they do as of 11g. The default collects the least information, thus protecting you from excessive size of trace files. With option 4 you also collect information on bind variables

- 01 – Default
- 04 – Standard plus binds
- With this level of parsing we see a BINDS section

  ```
  BINDS #1:
   Bind#0
    oacdty=02 mxl=22(22) mxlc=00 mal=00 scl=00 pre=00
    oacflg=03 fl2=1000000 frm=00 csi=00 siz=24 off=0
    kxsbbbfp=0e90cbb8  bln=22  avl=02  flg=05
    value=100
  ```

- 8 – Standard plus waits
- With this level of tracing we see the waits indicating where we spend the time in the statement
- 12 – Standard plus waits and binds
- With this level we collect not only the binds but also the waits
- 16 – Generate STAT line dumps for each execution
- 32 – Never dump execution statistics
- 64 – Adaptive dump of STAT lines (11.2.0.2+)

A typical example of generating this trace would be as follows:

1. Find the location of `USER_DUMP_DEST`
2. Set the file size to unlimited if possible
3. Set the statistics level to all
4. Select the tracing level based on the values above

Here is an example set of steps, beginning with setting the identifier for the trace file name, which will allow you to easily find the trace file.

1. Set the identifier:

   ```
   SQL> alter session set tracefile_identifier='10046_STELIOS';
   Session altered.
   ```

2. Now I want to make sure I don't lose information if the trace file is too long, so I set the size to unlimited.

   ```
   SQL> alter session set max_dump_file_size=unlimited;
   Session altered.
   ```

3. Set the statistics_level parameter to all to collect as much information as possible.

   ```
   SQL> alter session set statistics_level=all;
   Session altered.
   ```

4. Now set the 10046 trace event to collect tracing information at level 12 (which includes binds and waits).

   ```
   SQL> alter session set events '10046 trace name context forever, level 12';
   Session altered.
   ```

5. Now I execute my SQL

   ```
   SQL> @q3

   COUNTRY_NAME                             SUM(AMOUNT_SOLD)
   ---------------------------------------- ----------------
   Poland                                            8447.14
   Denmark                                        1977764.79
   Spain                                          2090863.44
   France                                         3776270.13
   Italy                                          4854505.28
   United Kingdom                                 6393762.94
   Germany                                        9210129.22

   7 rows selected.
   ```

6. Then turn off the tracing

   ```
   SQL> alter session set events '10046 trace name context off';
   ```

   This is the SQL that ran:

   ```
   select
     country_name, sum(AMOUNT_SOLD)
   from sh.sales s, sh.customers c, sh.countries co
   where
     s.cust_id=c.cust_id
     and co.country_id=c.country_id
     and country_name in (
       'Ireland','Denmark','Poland','United Kingdom',
        'Germany','France','Spain','The Netherlands','Italy')
     group by country_name order by sum(AMOUNT_SOLD);
   ```

■ **NOTE** If you want to follow along with these examples and collect your own trace files and view them yourself you can easily do so by using all the code in these examples. All of the data and examples used rely on the standard example schemas shipped with every database installation. If you do not have the schemas SH and HR you can either select to install the example schemas at installation time or manually add them afterward with the instructions here http://docs.oracle.com/cd/E14072_01/server.112/e10831/installation.htm#sthref33.

---

7. Finally I want to find the file so I look at the value of user_dump_dest

```
SQL> show parameter user_dump_dest

NAME                                 TYPE        VALUE
------------------------------------ ----------- ------------------------------
user_dump_dest                       string      f:\app\stelios\diag\rdbms\snc1
                                                 \snc1\trace
```

This is all very well if you want to look at raw 10046 trace files, but what if you wanted to interpret what was going on more quickly? First we'll look at one of the oldest utilities around for procesing 10046 trace, TKPROF, which is still used today to get a good overview of what has been happening. TKPROF is not part of SQLT; but as it is commonly used, we'll give it a brief mention.

TKPROF

No doubt some people can read 10046 trace files in the raw (and it wouldn't be too difficult to do once you gather the translation of the short codes), but this is a pretty inefficient way of going about things. It would be like listing the position, velocity, and direction of every atom in a football, when in fact you could aggregate the information to describe the ball's trajectory with just a few parameters. TKPROF aggregates information and presents a summary of the high-level view of what's going on. TKPROF is not part of SQLT and has been in existence since early versions of Oracle. TKPROF is best explained through the use of an example: we'll take a trace file produced as desribed above and generate the TKPROF output. To use TKPROF just run the TKPROF command with the input of the trace file.

```
>tkprof trca_e68572_10046.trc output = trca_e68572_10046.txt
TKPROF: Release 11.2.0.1.0 - Development on Wed Dec 26 11:08:49 2012
Copyright (c) 1982, 2009, Oracle and/or its affiliates.  All rights reserved.
```

If we look at the TKPROF output file and find our SQL, we then see this kind of information:

```
********************************************************************************
select
  country_name, sum(AMOUNT_SOLD)
from sh.sales s, sh.customers c, sh.countries co
where
  s.cust_id=c.cust_id
  and co.country_id=c.country_id
  and country_name in (
    'Ireland','Denmark','Poland','United Kingdom',
     'Germany','France','Spain','The Netherlands','Italy')
  group by country_name order by sum(AMOUNT_SOLD)
```

```
call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.09       0.09          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        2     12.20      12.24          0       3180          0           7
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        4     12.29      12.34          0       3180          0           7

Misses in library cache during parse: 1
Optimizer mode: ALL_ROWS
Parsing user id: SYS

Rows     Row Source Operation
-------  ---------------------------------------------------
 7  SORT ORDER BY (cr=3180 pr=0 pw=0 time=15 us cost=947 size=315 card=9)
 7   HASH GROUP BY (cr=3180 pr=0 pw=0 time=21 us cost=947 size=315 card=9)
250069  HASH JOIN  (cr=3180 pr=0 pw=0 time=11059983 us cost=909 size=15233435 card=435241)
30473   HASH JOIN  (cr=1462 pr=0 pw=0 time=453228 us cost=409 size=657225 card=26289)
 9     TABLE ACCESS FULL COUNTRIES (cr=3 pr=0 pw=0 time=20 us cost=3 size=135 card=9)
55500 TABLE ACCESS FULL CUSTOMERS (cr=1459 pr=0 pw=0 time=112536 us cost=406 size=555000 card=55500)
918843 PARTITION RANGE ALL PARTITION: 1 28 (cr=1718 pr=0 pw=0 time=5428724 us cost=494 size=9188430
card=918843)
918843 TABLE ACCESS FULL SALES PARTITION: 1 28 (cr=1718 pr=0 pw=0 time=1892572 us cost=494
size=9188430 card=918843)

Elapsed times include waiting on following events:
  Event waited on                             Times   Max. Wait  Total Waited
  ----------------------------------------   Waited  ----------  ------------
  SQL*Net message to client                       2        0.00          0.00
  asynch descriptor resize                        3        0.00          0.00
  SQL*Net message from client                     2      574.61        574.61
********************************************************************************
```

In the TKPROF of the SQL we see the SQL statement, a summary section that describes the execution times for "Parse", "Execute", and "Fetch", as well as an execution plan and a brief description of the wait times. Compare this kind of information with the sections on 10046 trace file interpretation. Which do you think would be faster? I know which one I prefer to look at. That kind of aggregation of information makes interpretation of what is happening much simpler and faster. Now that we've briefly mentioned 10046 tracing (and we mentioned 10053 tracing in chapter 5), we look at what TRCASPLIT can do for us.

TRCASPLIT

TRCASPLIT the first of the trace utilities available in SQLT simply separates 10046 trace from other trace. While this may be a simple task in theory it would be very difficult and error prone to do manually. TRCASPLIT does the job in no time and presents you with the results in a zip file. First let me show you what we are going to collect, then we will collect it. Then we will split it using TRCASPLIT.

In this example we are going to collect both 10046 and 10053 trace file information. In other words we are turning on debugging information for both running the SQL and for parsing it. As before we will also ensure that the dump file is not truncated by setting the dump file size to unlimited, and we'll also set the statistics collection level to the highest possible level of all.

```
SQL> alter session set tracefile_identifier='10046_10053';
Session altered.
SQL> alter session set max_dump_file_size=unlimited;
Session altered.
SQL> alter session set statistics_level=all;
Session altered.
SQL> alter session set events '10046 trace name context forever, level 12';
Session altered.
SQL> alter session set events '10053 trace name context forever, level 1';
Session altered.
SQL> alter system flush shared_pool;
System altered.
SQL> @q3
```

I'm sure we don't need to see the results of this query again. What we do need to look at is the trace file created. In the trace file we see sections which are clearly 10046 type lines such as:

```
STAT #1 id=6 cnt=55500 pid=4 pos=2 obj=74151 op='TABLE ACCESS FULL CUSTOMERS (cr=1459 pr=0 pw=0
time=112536 us cost=406 size=555000 card=55500)'
STAT #1 id=7 cnt=918843 pid=3 pos=2 obj=0 op='PARTITION RANGE ALL PARTITION: 1 28 (cr=1718 pr=0 pw=0
time=5428724 us cost=494 size=9188430 card=918843)'
STAT #1 id=8 cnt=918843 pid=7 pos=1 obj=74083 op='TABLE ACCESS FULL SALES PARTITION: 1 28 (cr=1718
pr=0 pw=0 time=1892572 us cost=494 size=9188430 card=918843)'
```

These are related to 10046 tracing, but we also see:

```
***************************************
PARAMETERS USED BY THE OPTIMIZER
********************************
  *************************************
  PARAMETERS WITH ALTERED VALUES
  ******************************
Compilation Environment Dump
sqlstat_enabled                    = true
optimizer_dynamic_sampling         = 4
statistics_level                   = all
```

These lines are recognizable as 10053 tracing. We don't see any of the logical 10046 tracing information related to the execution lines of code and waits. There are no codes at the beginning of lines related to what is happening. Any program designed to decode 10046 would have to stop and ask for help at this point, but TRACSPLIT helps us by sifting the two types of information. To split these we need only run `sqltrcasplit.sql`. In this example I've copied the trace file produced in the trace directory to the local directory. There is only one parameter to pass to `sqltrcasplit.sql`, the name of the trace file that needs to be split.

```
C:\Documents and Settings\Stelios\Desktop\SQLT 11.4.5.1\sqlt\run>sqlplus stelios/password
SQL*Plus: Release 11.2.0.1.0 Production on Wed Dec 26 10:50:43 2012
Copyright (c) 1982, 2010, Oracle.  All rights reserved.
Connected to:
Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - Production
With the Partitioning, OLAP, Data Mining and Real Application Testing options
SQL> @sqltrcasplit.sql
```

```
PL/SQL procedure successfully completed.
Parameter 1:
Trace Filename (required)
Enter value for 1:
Enter value for 1: snc1_ora_2796_10046_10053.trc
Value passed to sqltrcasplit.sql:
TRACE_FILENAME: snc1_ora_2796_10046_10053.trc
PL/SQL procedure successfully completed.
Splitting snc1_ora_2796_10046_10053.trc
***
*** NOTE:
*** If you get error below it means SQLTXPLAIN is not installed:
***   PLS-00201: identifier 'SQLTXADMIN.SQLT$A' must be declared.
*** In such case look for errors in NN_*.log files created during install.
***
SQLT_VERSION
----------------------------------------
SQLT version number: 11.4.5.1
SQLT version date  : 2012-11-27
Installation date  : 2012-12-26/08:20:26
... please wait ...
To monitor progress, login into another session and execute:
SQL> SELECT * FROM SQLTXADMIN.trca$_log_v;
... splitting trace(s) ...
Execution ID: 68572 started at 2012-12-26 10:57:15
In case of premature termination, read trcanlzr_error.log located in SQL*Plus default directory
/*************************************************************************************/
10:57:15 => trcanlzr
10:57:15 file_name:"snc1_ora_2796_10046_10053.trc"
10:57:15 analyze:"NO"
10:57:15 split:"YES"
10:57:15 tool_execution_id:"68572"
10:57:15 directory_alias_in:"SQLT$STAGE"
10:57:15 file_name_log:""
10:57:15 file_name_html:""
10:57:15 file_name_txt:""
10:57:15 file_name_10046:""
10:57:15 file_name_10053:""
10:57:15 out_file_identifier:""
10:57:15 calling trca$p.parse_main
10:57:15 => parse_main
10:57:15 analyzing input file snc1_ora_2796_10046_10053.trc in f:\app\stelios\diag\rdbms\snc1\snc1\
trace (SQLT$STAGE)
10:57:15 -> parse_file
10:57:15 parsing file snc1_ora_2796_10046_10053.trc in f:\app\stelios\diag\rdbms\snc1\snc1\trace
10:57:32 parsed snc1_ora_2796_10046_10053.trc (input 11167513 bytes, parsed as 11167513 bytes)
10:57:32 <- parse_file
10:57:32 parsed 1 file(s) (input 11167513 bytes)
10:57:32 first trace: f:\app\stelios\diag\rdbms\snc1\snc1\trace\snc1_ora_2796_10046_10053.trc
10:57:32 <= parse_main
10:57:32 <= trcanlzr
/*************************************************************************************/
```

```
Trace Analyzer executed successfully.
There are no fatal errors in this log file.
Execution ID: 68572 completed at 2012-12-26 10:57:32
Trace Split completed.
Review first sqltrcasplit_error.log file for possible fatal errors.
Review next trca_e68572.log for parsing messages and totals.
Copying now generated files into local directory
  adding: trca_e68572.log (164 bytes security) (deflated 65%)
  adding: trca_e68572_10046.trc (164 bytes security) (deflated 93%)
  adding: trca_e68572_not_10046.trc (164 bytes security) (deflated 91%)
  adding: sqltrcasplit_error.log (164 bytes security) (deflated 79%)
deleting: sqltrcasplit_error.log
File trca_e68572.zip has been created.
SQLTRCASPLIT completed.
```

Once the script has finished, I have a zip file in the local directory called `trca_e68572.zip` (in my example) containing the results of the split. If I create a directory and put the files from the zip file in this directory I see that there are two files. One is named `trca_e68572_10046.trc` and one is named `trca_e68572_not_10046.trc`. Now we can look at the 10046 trace file without the 10053 trace information making the interpretation less difficult.

TRCANLZR

TRCANLZR is used to analyze multiple trace files and generate one aggregated form of information. In the example below I have added a parallel hint to force parallel execution and multiple trace files. This is the SQL I used with the hint in place.

```
select /*+ parallel (s, 2) */
  country_name, sum(AMOUNT_SOLD)
from sh.sales s, sh.customers c, sh.countries co
where
  s.cust_id=c.cust_id
  and co.country_id=c.country_id
  and country_name in (
    'Ireland','Denmark','Poland','United Kingdom',
     'Germany','France','Spain','The Netherlands','Italy')
  group by country_name order by sum(AMOUNT_SOLD);
```

Once I have enabled tracing for 10046 (see the earlier section “How Do We Collect 10046 Trace”) I have a number of trace files in the trace directory. To allow TRCANLZR to know which files to analyze I created a text file called `control.txt` that lists the file names in the trace directory that relate to the execution of the SQL. This is what `control.txt` contains.

```
snc1_p003_3480_10046.trc
snc1_p003_3480.trc
snc1_p002_5284_10046.trc
snc1_p002_5284.trc
snc1_p001_4320_10046.trc
snc1_p001_4320.trc
snc1_p000_3600_10046.trc
snc1_p000_3600.trc
snc1_j000_4656.trc
```

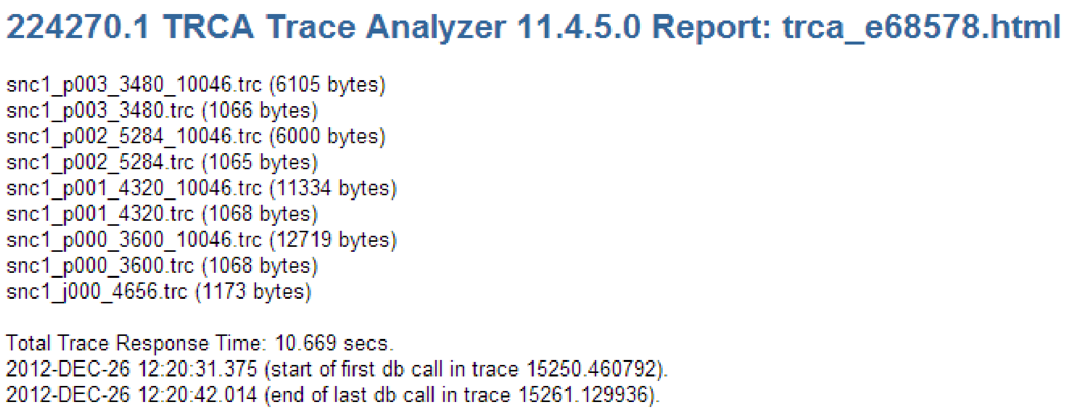
By the way the file name `control.txt` is hard-coded and part of SQLT. You have to use this file name. Also notice in this case that some of the files contain the text “10046” and some do not. Remember that 10046 trace files will not usually have the text 10046 in the name unless you specify it with:

```
alter session set tracefile_identifier='10046'
```

When I run `sqltracnlzr.sql` I am prompted for a control file name or a trace file to analyze. In my case I have a number of trace files so I entered `control.txt`. This file is located in the same directory as the trace files. The execution of `sqltrcanlzr` finishes with the following screen.

```
  adding: trca_e68578.html (164 bytes security) (deflated 92%)
  adding: trca_e68578.log (164 bytes security) (deflated 90%)
  adding: trca_e68578.txt (164 bytes security) (deflated 87%)
  adding: trca_e68578_nosort.tkprof (164 bytes security) (deflated 78%)
  adding: trca_e68578_sort.tkprof (164 bytes security) (deflated 78%)
  adding: sqltrcanlzr_error.log (164 bytes security) (deflated 86%)
deleting: sqltrcanlzr_error.log

File trca_e68578.zip has been created.

SQLTRCANLZR completed.
```

In the run directory of SQLT I now have a zip file, which as usual I create a directory for and unzip the files into that directory. The HTML file in this directory is the Trace Analyzer file. It starts with the list of trace files it analyzed. See Figure 13-1.

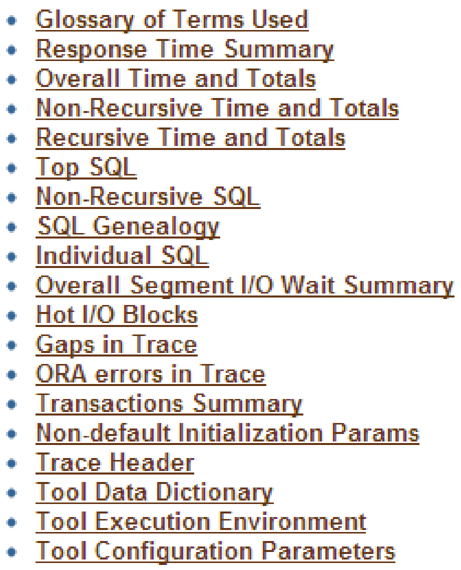

224270.1 TRCA Trace Analyzer 11.4.5.0 Report: trca_e68578.html

snc1_p003_3480_10046.trc (6105 bytes)
snc1_p003_3480.trc (1066 bytes)
snc1_p002_5284_10046.trc (6000 bytes)
snc1_p002_5284.trc (1065 bytes)
snc1_p001_4320_10046.trc (11334 bytes)
snc1_p001_4320.trc (1068 bytes)
snc1_p000_3600_10046.trc (12719 bytes)
snc1_p000_3600.trc (1068 bytes)
snc1_j000_4656.trc (1173 bytes)

Total Trace Response Time: 10.669 secs.
2012-DEC-26 12:20:31.375 (start of first db call in trace 15250.460792).
2012-DEC-26 12:20:42.014 (end of last db call in trace 15261.129936).

Figure 13-1. *The top of the TRCANLZR report*

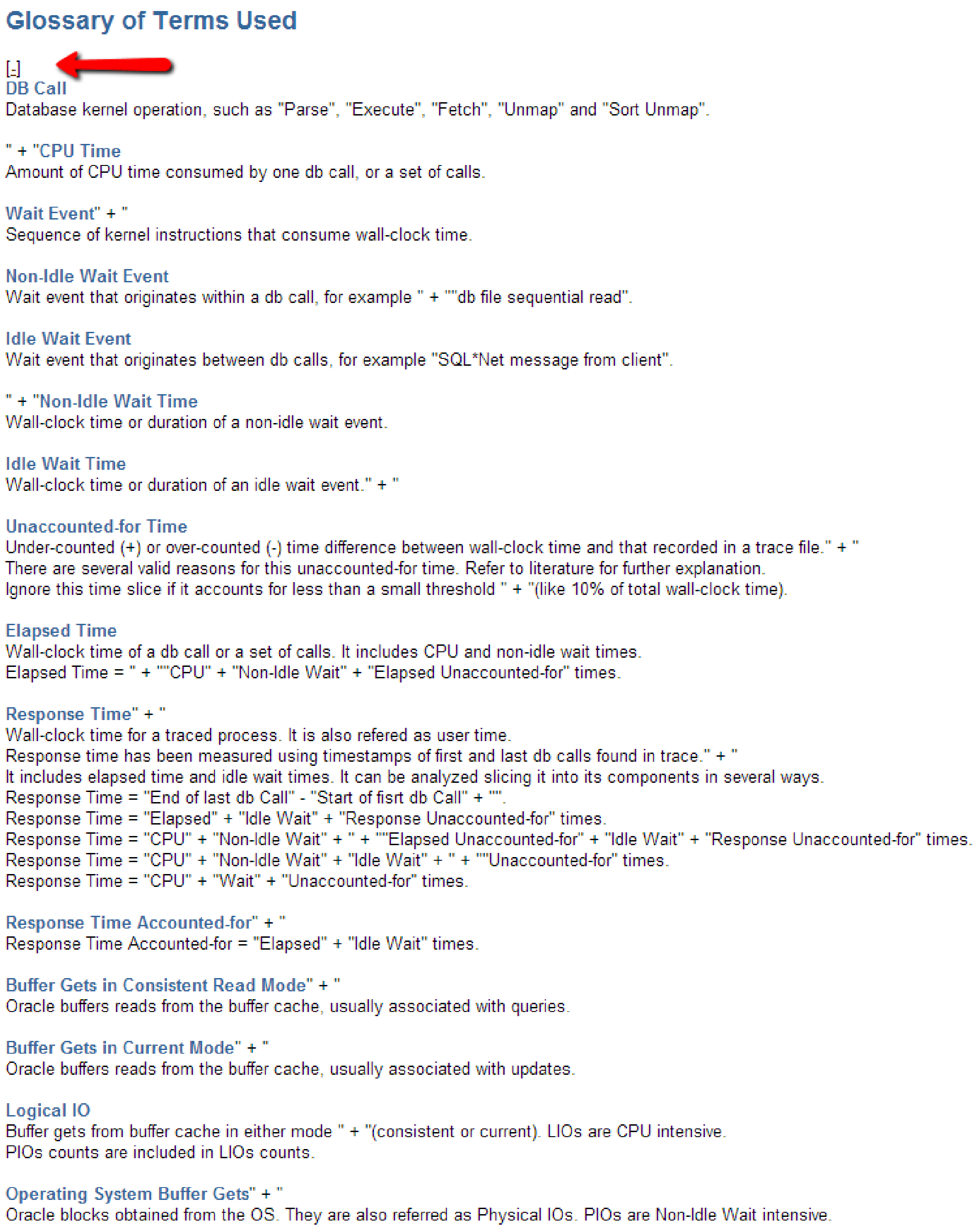
This list at the top of the file ensures that we have the right TRCANLZR output. Below this is a summary of the sections of the report that can be reached. See Figure 13-2.



- Glossary of Terms Used
- Response Time Summary
- Overall Time and Totals
- Non-Recursive Time and Totals
- Recursive Time and Totals
- Top SQL
- Non-Recursive SQL
- SQL Genealogy
- Individual SQL
- Overall Segment I/O Wait Summary
- Hot I/O Blocks
- Gaps in Trace
- ORA errors in Trace
- Transactions Summary
- Non-default Initialization Params
- Trace Header
- Tool Data Dictionary
- Tool Execution Environment
- Tool Configuration Parameters

Figure 13-2. *The header section includes the links to other parts of the report*

Look at Figure 13-2. There are many sections here all with useful information. The first is the “Glossary of Terms Used”. You might be tempted to go straight to the “Response Time Summary” as the “Glossary of Terms Used” sounds boring. If you did go to the “Glossary of Terms Used,” on first glance it looks like there is not much there. See Figure 13-3. Understanding exactly what each of the terms used is crucial to your understanding of the report and so the “Glossary of Terms Used” should be examined. You just need to click on the plus under the heading to get the details. See Figure 13-3, which shows the screen after I expand the section.



Glossary of Terms Used

[-]
DB Call
Database kernel operation, such as "Parse", "Execute", "Fetch", "Unmap" and "Sort Unmap".

" + "CPU Time
Amount of CPU time consumed by one db call, or a set of calls.

Wait Event" + "
Sequence of kernel instructions that consume wall-clock time.

Non-Idle Wait Event
Wait event that originates within a db call, for example " + ""db file sequential read".

Idle Wait Event
Wait event that originates between db calls, for example "SQL*Net message from client".

" + "Non-Idle Wait Time
Wall-clock time or duration of a non-idle wait event.

Idle Wait Time
Wall-clock time or duration of an idle wait event." + "

Unaccounted-for Time
Under-counted (+) or over-counted (-) time difference between wall-clock time and that recorded in a trace file." + "
There are several valid reasons for this unaccounted-for time. Refer to literature for further explanation.
Ignore this time slice if it accounts for less than a small threshold " + "(like 10% of total wall-clock time).

Elapsed Time
Wall-clock time of a db call or a set of calls. It includes CPU and non-idle wait times.
Elapsed Time = " + ""CPU" + "Non-Idle Wait" + "Elapsed Unaccounted-for" times.

Response Time" + "
Wall-clock time for a traced process. It is also refered as user time.
Response time has been measured using timestamps of first and last db calls found in trace." + "
It includes elapsed time and idle wait times. It can be analyzed slicing it into its components in several ways.
Response Time = "End of last db Call" - "Start of fisrt db Call" + "".
Response Time = "Elapsed" + "Idle Wait" + "Response Unaccounted-for" times.
Response Time = "CPU" + "Non-Idle Wait" + " + ""Elapsed Unaccounted-for" + "Idle Wait" + "Response Unaccounted-for" times.
Response Time = "CPU" + "Non-Idle Wait" + "Idle Wait" + " + ""Unaccounted-for" times.
Response Time = "CPU" + "Wait" + "Unaccounted-for" times.

Response Time Accounted-for" + "
Response Time Accounted-for = "Elapsed" + "Idle Wait" times.

Buffer Gets in Consistent Read Mode" + "
Oracle buffers reads from the buffer cache, usually associated with queries.

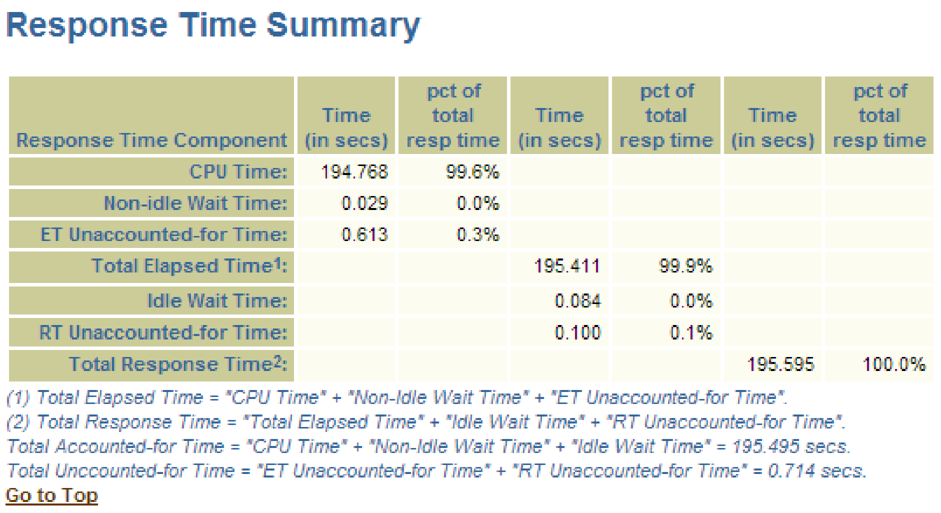
Buffer Gets in Current Mode" + "
Oracle buffers reads from the buffer cache, usually associated with updates.

Logical IO
Buffer gets from buffer cache in either mode " + "(consistent or current). LIOs are CPU intensive.
PIOs counts are included in LIOs counts.

Operating System Buffer Gets" + "
Oracle blocks obtained from the OS. They are also referred as Physical IOs. PIOs are Non-Idle Wait intensive.

Figure 13-3. *The glossary of terms used is not normally displayed*

The reason I mention the glossary of terms is because TRCANLZR needs to be interpreted correctly. There are many measures of time and each one needs to be understood for you to get the right picture of what is happening. For example the “response time” is the measure of time that the user perceives if they were sitting at a terminal running the SQL. That time is made up of actual work (Elapsed) and non-idle wait times. Idle wait times can be ignored as they are due the end user not responding (normally shown as `SQL*Net message from client`). Non-idle wait times are usually made up of activities that happen during steps in the SQL, for example fetching data from a disk. Once we have these definitions clear we can look at the “Response Time Summary”. Look at Figure 13-4, which shows an example response time summary for the SQL (this is for a simple SQL with no complications).



Response Time Summary

Response Time Component	Time (in secs)	pct of total resp time	Time (in secs)	pct of total resp time	Time (in secs)	pct of total resp time
CPU Time:	194.768	99.6%				
Non-idle Wait Time:	0.029	0.0%				
ET Unaccounted-for Time:	0.613	0.3%				
Total Elapsed Time[1]:			195.411	99.9%		
Idle Wait Time:			0.084	0.0%		
RT Unaccounted-for Time:			0.100	0.1%		
Total Response Time[2]:					195.595	100.0%

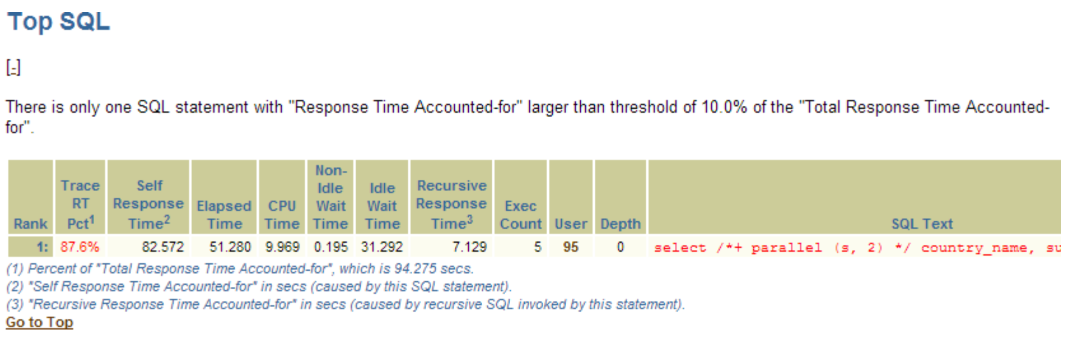
(1) Total Elapsed Time = "CPU Time" + "Non-Idle Wait Time" + "ET Unaccounted-for Time".
(2) Total Response Time = "Total Elapsed Time" + "Idle Wait Time" + "RT Unaccounted-for Time".
Total Accounted-for Time = "CPU Time" + "Non-Idle Wait Time" + "Idle Wait Time" = 195.495 secs.
Total Unccounted-for Time = "ET Unaccounted-for Time" + "RT Unaccounted-for Time" = 0.714 secs.
Go to Top

Figure 13-4. *The response time summary shown*

This response time summary shows the following

- The CPU time was 194.768 seconds
- Non-idle wait time was 0.029 seconds
- Unaccounted for elapsed time was 0.613 seconds
- The percentage of each of the above times
- The total elapsed time which is a sum of the CPU Time, Non-idle Wait time and The unaccounted for Time in this example 195.411
- Idle wait time was 0.084 seconds
- Unaccounted for time was 0.1 seconds
- The total time for everything was 195.595 seconds

This is an aggregation of all the information from all trace files. This gives you an idea of where your time is going and which item in the list above is worth attacking. In this case 99.6% of all time is spent on the CPU so if there is any improvement that is where it will be made. This kind of report also lists all the related SQL (if it is recursive for example), so you will see a “Top SQL” section of this report (see Figure 13-5).



Top SQL

[-]

There is only one SQL statement with "Response Time Accounted-for" larger than threshold of 10.0% of the "Total Response Time Accounted-for".

Rank	Trace RT Pct[1]	Self Response Time[2]	Elapsed Time	CPU Time	Non-Idle Wait Time	Idle Wait Time	Recursive Response Time[3]	Exec Count	User	Depth	SQL Text
1:	87.6%	82.572	51.280	9.969	0.195	31.292	7.129	5	95	0	select /*+ parallel (s, 2) */ country_name, su

(1) Percent of "Total Response Time Accounted-for", which is 94.275 secs.
(2) "Self Response Time Accounted-for" in secs (caused by this SQL statement).
(3) "Recursive Response Time Accounted-for" in secs (caused by recursive SQL invoked by this statement).
Go to Top

Figure 13-5. *The "Top SQL" section of the TRCANLZR report*

TRCAXTR

TRCAXTR is a combination report, which takes the same parameters are TRCANLZR (as described in the section above) but then determines the top SQL from the report and runs that through XTRACT. In the case above the top SQL would be the one under the heading "SQL Text" in Figure 13-5. You run `sqltrcxtr.sql` by entering the following on the command line

```
SQL> @sqltrcxtr.sql
```

You will be prompted for the same parameters as for TRCANLZR but after that ends you will be prompted for the parameters to run XTRACT.

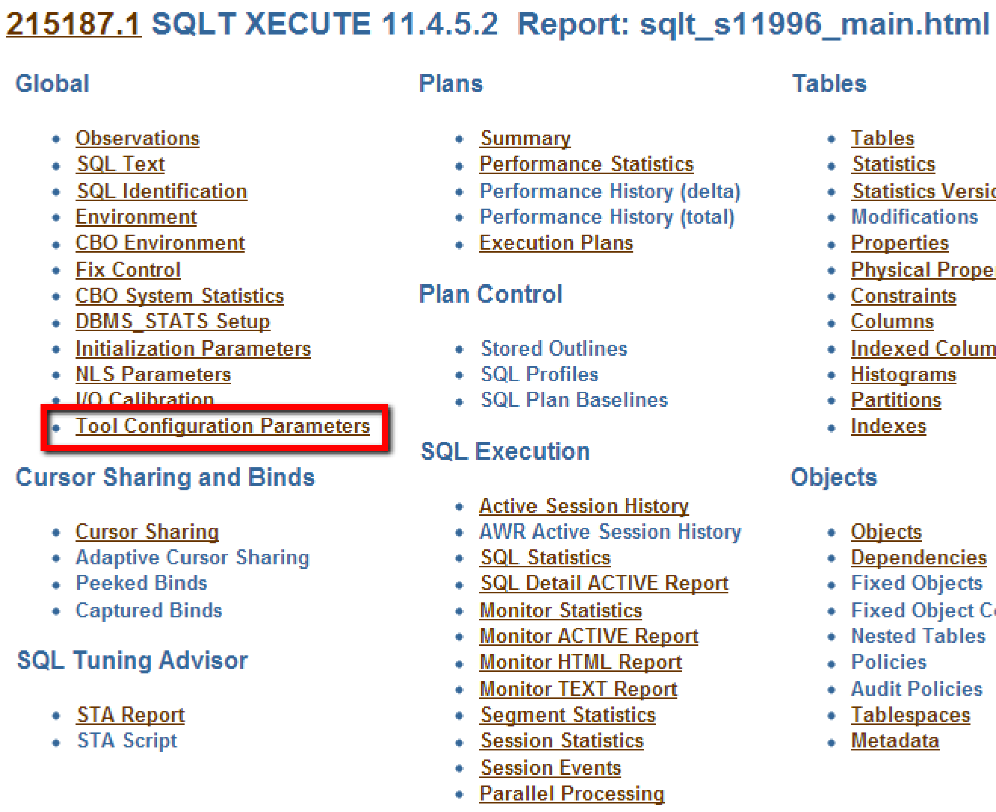
Here are the parameters that you will be prompted for

- The file containing the trace file or control file containing the trace files (as was mentioned earlier `control.txt` is the 'control' file and can contain multiple trace file names)
- The SQLTXPLAIN password

The process is very long, because it goes through all the trace files generating XTRACTs for each SQL. The top level zip file is called `sqlt_snnnnn_set.zip`. This files contains all the XTRACTs for each of the SQLs in the trace file. This simple routine, with very few parameters can produce many many files to analyze. It can be a quick way to collect information for a whole session, but be warned the the zip files produced can be very big.

Modifying SQLT Behavior

SQLT has many aspects to it and many tools that can be used. Despite this there are still situations where SQLT could do better or be modified to help with some situation. Luckily this is covered by the parameters which can be set to change SQLT's behavior. If we look at the now familiar first page of the main report (see Figure 13-6), we see the "Tool Configuration Parameters" link.



215187.1 SQLT XECUTE 11.4.5.2 Report: sqlt_s11996_main.html

Global

- Observations
- SQL Text
- SQL Identification
- Environment
- CBO Environment
- Fix Control
- CBO System Statistics
- DBMS_STATS Setup
- Initialization Parameters
- NLS Parameters
- I/O Calibration
- Tool Configuration Parameters

Cursor Sharing and Binds

- Cursor Sharing
- Adaptive Cursor Sharing
- Peeked Binds
- Captured Binds

SQL Tuning Advisor

- STA Report
- STA Script

Plans

- Summary
- Performance Statistics
- Performance History (delta)
- Performance History (total)
- Execution Plans

Plan Control

- Stored Outlines
- SQL Profiles
- SQL Plan Baselines

SQL Execution

- Active Session History
- AWR Active Session History
- SQL Statistics
- SQL Detail ACTIVE Report
- Monitor Statistics
- Monitor ACTIVE Report
- Monitor HTML Report
- Monitor TEXT Report
- Segment Statistics
- Session Statistics
- Session Events
- Parallel Processing

Tables

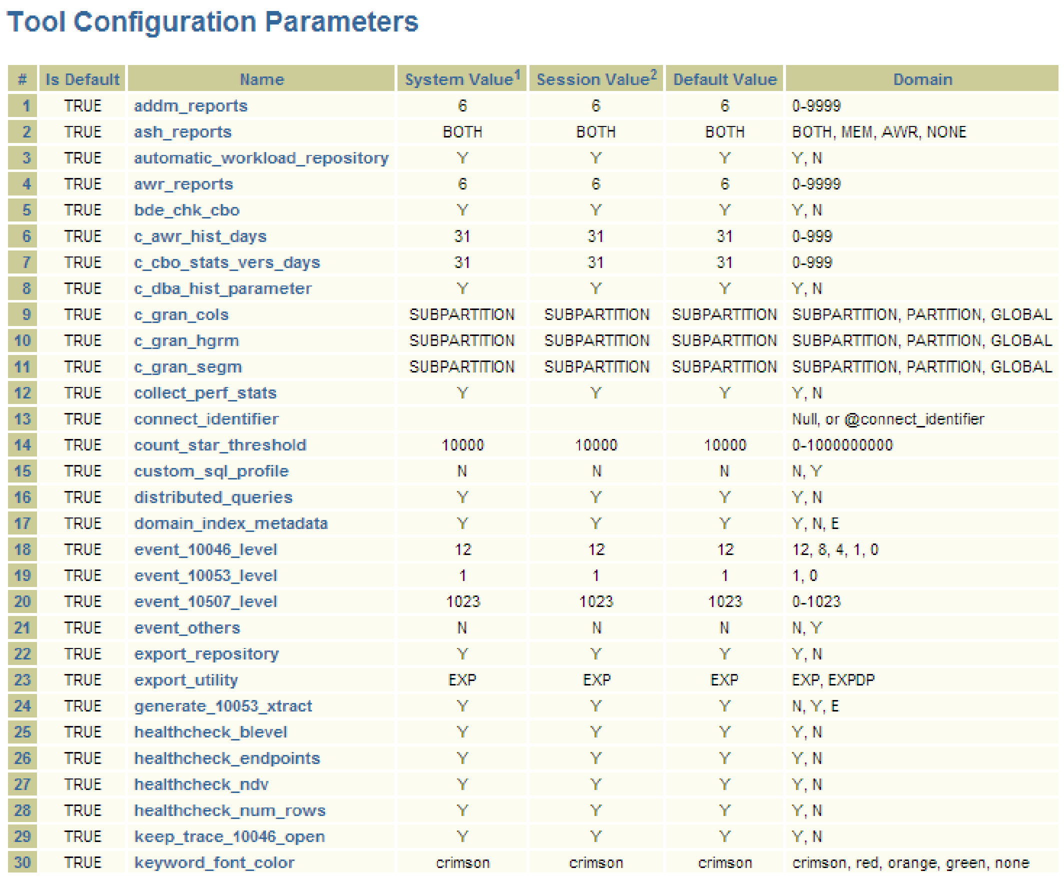
- Tables
- Statistics
- Statistics Versions
- Modifications
- Properties
- Physical Properties
- Constraints
- Columns
- Indexed Columns
- Histograms
- Partitions
- Indexes

Objects

- Objects
- Dependencies
- Fixed Objects
- Fixed Object Columns
- Nested Tables
- Policies
- Audit Policies
- Tablespaces
- Metadata

***Figure 13-6.** The Tool configuration option is under the Global section*

If we click on this link we see the list of parameters that can be set within SQLT to change behavior or options. Many of the options here turn on features or turn off features, such as `ash_reports`. The “Domain” column lists the possible values, in this case “Y” or “N”.

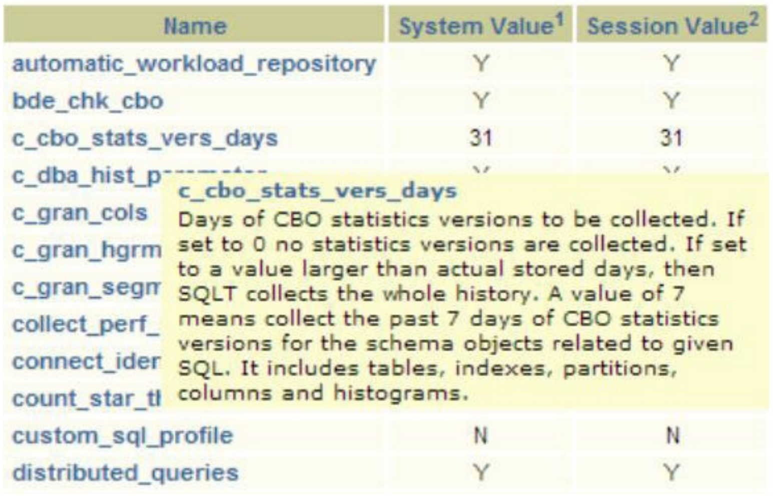


Tool Configuration Parameters

#	Is Default	Name	System Value[1]	Session Value[2]	Default Value	Domain
1	TRUE	addm_reports	6	6	6	0-9999
2	TRUE	ash_reports	BOTH	BOTH	BOTH	BOTH, MEM, AWR, NONE
3	TRUE	automatic_workload_repository	Y	Y	Y	Y, N
4	TRUE	awr_reports	6	6	6	0-9999
5	TRUE	bde_chk_cbo	Y	Y	Y	Y, N
6	TRUE	c_awr_hist_days	31	31	31	0-999
7	TRUE	c_cbo_stats_vers_days	31	31	31	0-999
8	TRUE	c_dba_hist_parameter	Y	Y	Y	Y, N
9	TRUE	c_gran_cols	SUBPARTITION	SUBPARTITION	SUBPARTITION	SUBPARTITION, PARTITION, GLOBAL
10	TRUE	c_gran_hgrm	SUBPARTITION	SUBPARTITION	SUBPARTITION	SUBPARTITION, PARTITION, GLOBAL
11	TRUE	c_gran_segm	SUBPARTITION	SUBPARTITION	SUBPARTITION	SUBPARTITION, PARTITION, GLOBAL
12	TRUE	collect_perf_stats	Y	Y	Y	Y, N
13	TRUE	connect_identifier				Null, or @connect_identifier
14	TRUE	count_star_threshold	10000	10000	10000	0-1000000000
15	TRUE	custom_sql_profile	N	N	N	N, Y
16	TRUE	distributed_queries	Y	Y	Y	Y, N
17	TRUE	domain_index_metadata	Y	Y	Y	Y, N, E
18	TRUE	event_10046_level	12	12	12	12, 8, 4, 1, 0
19	TRUE	event_10053_level	1	1	1	1, 0
20	TRUE	event_10507_level	1023	1023	1023	0-1023
21	TRUE	event_others	N	N	N	N, Y
22	TRUE	export_repository	Y	Y	Y	Y, N
23	TRUE	export_utility	EXP	EXP	EXP	EXP, EXPDP
24	TRUE	generate_10053_xtract	Y	Y	Y	N, Y, E
25	TRUE	healthcheck_blevel	Y	Y	Y	Y, N
26	TRUE	healthcheck_endpoints	Y	Y	Y	Y, N
27	TRUE	healthcheck_ndv	Y	Y	Y	Y, N
28	TRUE	healthcheck_num_rows	Y	Y	Y	Y, N
29	TRUE	keep_trace_10046_open	Y	Y	Y	Y, N
30	TRUE	keyword_font_color	crimson	crimson	crimson	crimson, red, orange, green, none

Figure 13-7. *The first page of configuration parameters*

If you are in doubt of the meaning of these parameters then you can just hover your mouse over the parameter and the descriptive comment will be shown. You can see that in Figure 13-8. (For your convenience, Appendix C includes a full list of these parameters and their descriptions.)



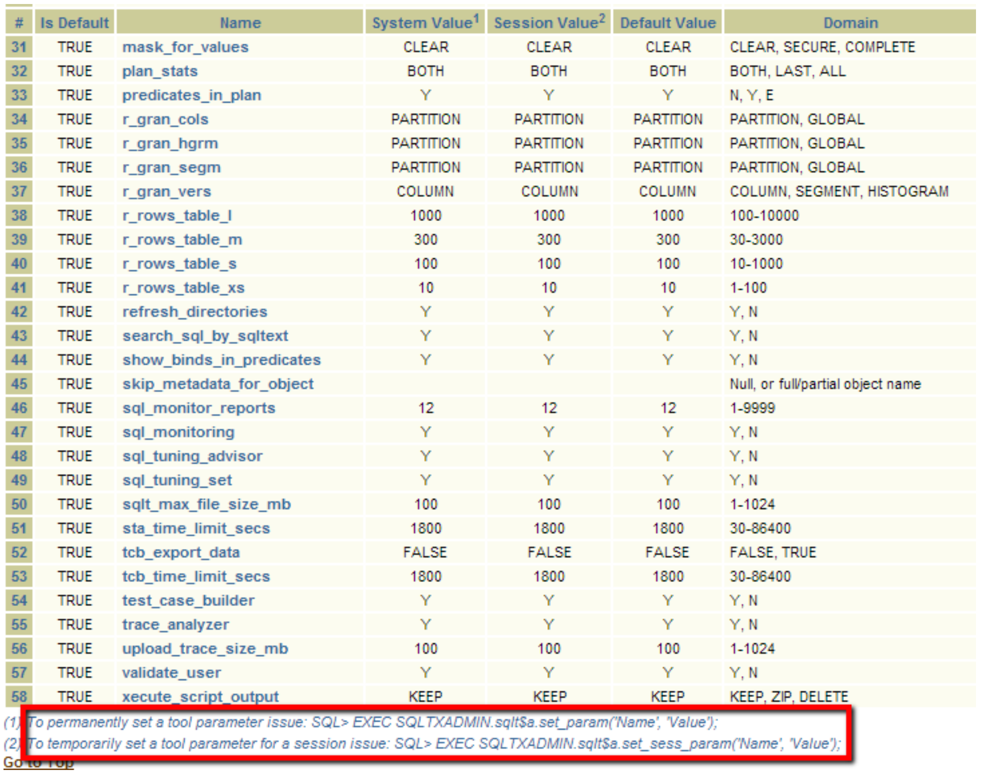
Name	System Value[1]	Session Value[2]
automatic_workload_repository	Y	Y
bde_chk_cbo	Y	Y
c_cbo_stats_vers_days	31	31
c_dba_hist_p		
c_gran_cols		
c_gran_hgrm		
c_gran_segm		
collect_perf_		
connect_ider		
count_star_tl		
custom_sql_profile	N	N
distributed_queries	Y	Y

c_cbo_stats_vers_days

Days of CBO statistics versions to be collected. If set to 0 no statistics versions are collected. If set to a value larger than actual stored days, then SQLT collects the whole history. A value of 7 means collect the past 7 days of CBO statistics versions for the schema objects related to given SQL. It includes tables, indexes, partitions, columns and histograms.

Figure 13-8. *Hover the mouse over the parameter names to get detailed descriptions*

These parameters make the SQLT utility even more flexible and useful than it would otherwise be. At the end of the second page (see Figure 13-9) we see that the way to set these parameters is described:



#	Is Default	Name	System Value[1]	Session Value[2]	Default Value	Domain
31	TRUE	mask_for_values	CLEAR	CLEAR	CLEAR	CLEAR, SECURE, COMPLETE
32	TRUE	plan_stats	BOTH	BOTH	BOTH	BOTH, LAST, ALL
33	TRUE	predicates_in_plan	Y	Y	Y	N, Y, E
34	TRUE	r_gran_cols	PARTITION	PARTITION	PARTITION	PARTITION, GLOBAL
35	TRUE	r_gran_hgrm	PARTITION	PARTITION	PARTITION	PARTITION, GLOBAL
36	TRUE	r_gran_segm	PARTITION	PARTITION	PARTITION	PARTITION, GLOBAL
37	TRUE	r_gran_vers	COLUMN	COLUMN	COLUMN	COLUMN, SEGMENT, HISTOGRAM
38	TRUE	r_rows_table_l	1000	1000	1000	100-10000
39	TRUE	r_rows_table_m	300	300	300	30-3000
40	TRUE	r_rows_table_s	100	100	100	10-1000
41	TRUE	r_rows_table_xs	10	10	10	1-100
42	TRUE	refresh_directories	Y	Y	Y	Y, N
43	TRUE	search_sql_by_sqltext	Y	Y	Y	Y, N
44	TRUE	show_binds_in_predicates	Y	Y	Y	Y, N
45	TRUE	skip_metadata_for_object				Null, or full/partial object name
46	TRUE	sql_monitor_reports	12	12	12	1-9999
47	TRUE	sql_monitoring	Y	Y	Y	Y, N
48	TRUE	sql_tuning_advisor	Y	Y	Y	Y, N
49	TRUE	sql_tuning_set	Y	Y	Y	Y, N
50	TRUE	sqlt_max_file_size_mb	100	100	100	1-1024
51	TRUE	sta_time_limit_secs	1800	1800	1800	30-86400
52	TRUE	tcb_export_data	FALSE	FALSE	FALSE	FALSE, TRUE
53	TRUE	tcb_time_limit_secs	1800	1800	1800	30-86400
54	TRUE	test_case_builder	Y	Y	Y	Y, N
55	TRUE	trace_analyzer	Y	Y	Y	Y, N
56	TRUE	upload_trace_size_mb	100	100	100	1-1024
57	TRUE	validate_user	Y	Y	Y	Y, N
58	TRUE	xecute_script_output	KEEP	KEEP	KEEP	KEEP, ZIP, DELETE

(1) To permanently set a tool parameter issue: SQL> EXEC SQLTXADMIN.sqlt$a.set_param('Name', 'Value');
(2) To temporarily set a tool parameter for a session issue: SQL> EXEC SQLTXADMIN.sqlt$a.set_sess_param('Name', 'Value');
Go to Top

Figure 13-9. *The second page of configuration parameters*

To set the value for the tool for all sessions:

```
SQL> EXEC SQLTXADMIN.sqlt$a.set_param('Name', 'Value');
```

And to set the value for one session only:

```
SQL> EXEC SQLTXADMIN.sqlt$a.set_sess_param('Name', 'Value');
```

With these options available we can, for example, make smaller test cases, or limit the number of AWR records reviewed or exclude the test case builder.

Summary

In this chapter we covered the use of the tools to analyze trace files which are available in SQLT. These are special tools over and above what TKPROF can offer and are useful in cases where you have 10046 trace files to analyze. SQLT XTRACT is still the best tool for helping with tuning in most cases, but sometimes you have a busy system which does not show a specific SQL to investigate. In cases such as these TRCANLZR can help to find the SQL which is the problem. If you use TRCAXTR as well then you can combine the best of both worlds.

CHAPTER 14

Running a Health Check

The SQL health check script is not SQLT. It is a completely separate utility downloaded from a different Metalink note and used in a different way. Why is it even mentioned in this book? SQLHC is considered a backup option when SQLT cannot be installed. This does not happen very often, but when it does it is usually because of site restrictions. There are many reasons for this:

- Not allowed to install anything on production (security reasons).
- Not allowed to install anything on production (fear of affecting production).
- Installation and testing procedures take too long and you have a problem on production.
- Don't trust SQLT.
- SQLT is too complicated.
- Not enough space to install SQLT.

Let me start by saying that SQLT has a small footprint and has been tried and tested on many different systems. Business reasons may preclude the installation of new packages in a timely manner: for example, if you have to test on development and QA before you can get to production. If you have those rules in place you should honor those business rules or local IT standards and consider using the health check script instead. However, I hope I've shown that SQLT is a trustworthy tool with many benefits for the user. If you come across a situation in which the use of SQLT is prohibited, then you can still choose the second best option, which is to run a SQL health check script against the target SQL.

What Is SQL Health Check?

The SQL health check (SQLHC) script is available as a free download from Oracle Metalink note 1366133.1. It is a zip file containing three simple SQL scripts (`sqldx.sql`, `sqlhc.sql` and `sqlhcxec.sql`) that you run as sys on the target system. Until recently `sqlhc.sql` was only script, so we'll talk about that script mostly and cover the other scripts later in this chapter. With all of these scripts nothing is installed: they are just SQL scripts and are very simple to use. These scripts were developed in response to the requirements on some sites that precluded the installation of SQLT (which has a schema, schema objects, tables and indexes, packages and functions). Because a script such as `sqlhc.sql` is just a simple script, more sites will accept it for installation so more sites can take advantage of the recommendations that come from SQLHC. SQLHC was developed from SQLT (it provides a subset of SQLT's functionality) and so discussions of what SQLHC can do cover many of the topics already covered in this book.

Here are some of the sections covered in a SQL health check report:

- An observation report, similar to SQLT but limited because of the limited access to the database.
- The SQL text.
- A summary of the table information including the number of rows and statistical sampling size.

- The number of columns and histograms for each table.
- A summary of all the index information for each table.
- A description of SQL plan baselines and profiles that are used for the SQL.
- The status of cursor sharing and the reasons for sharing.
- A history of the SQL Plans with associated statistics.
- A history of the execution plans.
- Active session history information.
- Column statistics and histogram types.
- System parameters and their values and default or non-default nature.
- A detailed description of all the execution plans.
- A SQL monitor report of the SQL.

The above functionality is produced by the main script of SQLHC, which is `sqlhc.sql`: in fact, the previous versions of SQLHC consisted of only this script. Later versions also included `sqldx.sql` and `sqlhcxec.sql`. The sqldx routine is a superb utility that produces a test case with no dependence on SQLT. We'll talk more about that in the section "The sqldx.sql Script." We'll also cover sqlhcxec. For now let's look at the main script `sqlhc.sql`. The first step in any tuning exercise should be to look for the obviously wrong things. This is called a checklist: why go into a long tuning exercise when you're missing an obvious index?

The list of things that can be done incorrectly on a database or on any particular SQL is very long. The health check script and SQLT have these error checks built in. SQLHC has approximately a hundred checks that it carries out (SQLT has two to three times that many). With these built in checks you can find and fix those problems before you start any tuning exercise. In SQLT these errors are pointed out in the "Observations" section of the main report (see Chapter 1). In SQLHC there is also an observations section of the report in the first page of the SQLHC HTML file. If only a few of your SQLs are caught before they go to production with some "obvious" mistakes, it will be well worth your time to download and use the SQL health check scripts.

The sqlhc.sql Script

The `sqlhc.sql` script is a single SQL script that produces one zip file, which contains four HTML files, three zip files and one trace file. If you do nothing else but run `sqlhc.sql` on each new SQL you introduce to your system, the health check script will have been worth the effort. The HTML files are reports in their own right and are well worth a careful read. The `sqlhc.sql` script is easy to use and takes only two parameters (we'll look at an example run later in this chapter in the section "Running the SQLHC Script.")

Remember that these files can be reviewed in any order, and one does not depend on another. If you have a particular area you want to check, you can look at the appropriate file directly without looking at files in between. However, if you want to be thorough and check everything, the main HTML files are numbered for your convenience. Example file names are shown below:

- `sqlhcxec_20130103_205245_81s67vj4pjqm8_1_health_check.html`
- `sqlhcxec_20130103_205245_81s67vj4pjqm8_2_diagnostics.html`
- `sqlhcxec_20130103_205245_81s67vj4pjqm8_3_execution_plans.html`
- `sqlhcxec_20130103_205245_81s67vj4pjqm8_4_sql_detail.html`

We'll look at all the files produced by `sqlhc.sql` in the next section.

What sqlhc.sql Produces

This section describes the main report files. The first four files mentioned are the HTML pages created by `sqlhc.sql`. They all contain the SQL text so you can make sure you're looking at the right file and view these in order from one to four.

- The first HTML report, the main health check script, gives you a good overview of observations and the general checks on the SQL. It covers observations and table and index summaries. The observations should be carefully reviewed for any problems.
- The second HTML report is more detailed and covers many miscellaneous areas, but most importantly it covers the current statistics (in the Current SQL Statistics section) and instance and system parameters. It also covers cursor sharing information and execution plan summaries. Historical information is also included, on tables, columns, indexes, instance, and system parameters. This report also covers SQL Plan baselines and profiles, as well as the new 11g feature SQL_Patches.
- Report three is useful if you think there is some issue with the execution plan or the execution plan has changed unexpectedly. Both current and historical execution plans are shown.
- The fourth report will give you some timings and shows you where you should concentrate your tuning effort.

Other files include the following:

- A zip file with the SQL Monitor execution details for each of the execution plans.
- A 10046 and 10053 trace file for debugging optimizer issues.
- A text file containing the results from the execution of the SQL. This is to check that we are running the right SQL and that the results are what is expected.
- A zip file containing the log of the run and some ancillary SQL.
- Lastly, a SQLDX zip file that contains all of the discovered information in a CSV format. This will allow further post processing using spreadsheets.

There are many useful files here, but the most useful parts of this output are the four HTML files and the SQL Monitor output. I'll show some examples of these after running an example script. You can see that although we have not run SQLT, we still have a lot of information to work with.

Running the SQLHC Script

Before I start the example, I have already logged in to Metalink (or My Oracle Support) and downloaded the one zip file that constitutes the SQLHC script files. The file is called `sqlhc.zip`. It contains `sqldx.sql`, `sqlhc.sql` and `sqlhcxec.sql`. This one simple zip file is all that is required to get the information listed in the example below. When considering the endless HTML reports and historical information gathering and execution plans reports, why would you not want to run this script as a check for each deployed SQL?

If you consider how much information you get from the SQL health check script and how little effort you have to put in to get this information it's a wonder that this script is free to supported customers. In the example below I will log in as sys, run my `q3.sql` test script, and get the SQL ID. We'll follow through all the required steps and show the example output.

```
>sqlplus / as sysdba
SQL*Plus: Release 11.2.0.1.0 Production on Thu Dec 27 09:29:45 2012
Copyright (c) 1982, 2010, Oracle.  All rights reserved.
```

```
Connected to:
Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - Production
With the Partitioning, OLAP, Data Mining and Real Application Testing options

SQL> @q3

COUNTRY_NAME                             SUM(AMOUNT_SOLD)
---------------------------------------- ----------------
Poland                                           8447.14
Denmark                                       1977764.79
Spain                                         2090863.44
France                                        3776270.13
Italy                                         4854505.28
United Kingdom                                6393762.94
Germany                                       9210129.22

7 rows selected.

SQL> select sql_id, sql_text from v$sqlarea where sql_text like 'select%parallel%Poland%';

SQL_ID
-------------
SQL_TEXT
--------------------------------------------------------------------------------
81s67vj4pjqm8

select /*+ parallel (s, 2) */
  country_name,
  sum(AMOUNT_SOLD)
from
  sh.sales s,
  sh.customers c,
  sh.countries co
where
  s.cust_id=c.cust_id
  and co.country_id= c.country_id
  and country_name in (     'Ireland','Denmark','Poland',
  'United Kingdom',
  'Germany','France','Spain','The Netherlands','Italy')
  group by country_name order by sum(AMOUNT_SOLD);
```

Now we have the SQL ID of the SQL we want to investigate, we need only run the sqlhc.sql script. We will be prompted for the license level of the database (I enter “T” in my case) and the SQL ID. That’s all there is to it. This is much simpler than SQLT.

```
SQL> @sqlhc.sql
Parameter 1:
Oracle Pack License (Tuning, Diagnostics or None) [T|D|N] (required)

Enter value for 1: T
PL/SQL procedure successfully completed.
```

```
Parameter 2:
SQL_ID of the SQL to be analyzed (required)
Enter value for 2: 81s67vj4pjqm8
```

Then the SQL script runs and produces an output file. The final page of the output looks like this

```
  adding: sql_shared_cursor_cur_81s67vj4pjqm8.sql (164 bytes security) (deflated 18%)
test of sqlhc_snc1_locutus_11.2.0.1.0_81s67vj4pjqm8_20121227_093856_9_log.zip OK

        zip warning: name not matched: sqlhc_snc1_locutus_11.2.0.1.0_81s67vj4pjqm8_20121227_09385
        6_tkprof_from_tool_exec.txt
test of sqlhc_snc1_locutus_11.2.0.1.0_81s67vj4pjqm8_20121227_093856_9_log.zip OK

  adding: sqlhc_snc1_locutus_11.2.0.1.0_81s67vj4pjqm8_20121227_093856_5_sql_monitor.sql
  (164 bytes security) (deflated 71%)
test of sqlhc_snc1_locutus_11.2.0.1.0_81s67vj4pjqm8_20121227_093856_9_log.zip OK

  adding: sqlhc_snc1_locutus_11.2.0.1.0_81s67vj4pjqm8_20121227_093856_9_log.zip (164 bytes security)
  (stored 0%)
test of sqlhc_snc1_locutus_11.2.0.1.0_81s67vj4pjqm8_20121227_093856.zip OK

  adding: sqlhc_snc1_locutus_11.2.0.1.0_81s67vj4pjqm8_16777216_3572724195_1_20121227_093856_5_sql_
  monitor.html (164 bytes security) (deflate 86%)
  adding: sqlhc_snc1_locutus_11.2.0.1.0_81s67vj4pjqm8_16777217_3572724195_1_20121227_093856_5_sql_
  monitor.html (164 bytes security) (deflate 87%)
test of sqlhc_snc1_locutus_11.2.0.1.0_81s67vj4pjqm8_20121227_093856_5_sql_monitor.zip OK

  adding: sqlhc_snc1_locutus_11.2.0.1.0_81s67vj4pjqm8_20121227_093856_5_sql_monitor.zip
  (164 bytes security) (stored 0%)
test of sqlhc_snc1_locutus_11.2.0.1.0_81s67vj4pjqm8_20121227_093856.zip OK

Ignore CP or COPY error below
'cp' is not recognized as an internal or external command,
operable program or batch file.
f:\app\stelios\diag\rdbms\snc1\snc1\trace\snc1_ora_4012_DBMS_SQLDIAG_10053_20121227_093815.trc
        1 file(s) copied.
  adding: sqlhc_snc1_locutus_11.2.0.1.0_81s67vj4pjqm8_20121227_093856_6_10053_trace_from_cursor.trc
  (164 bytes security) (deflated 82%)
test of sqlhc_snc1_locutus_11.2.0.1.0_81s67vj4pjqm8_20121227_093856.zip OK

SQLHC files have been created.
```

As usual we see messages regarding linux commands that do not run on my Windows system, but they do not cause a problem. A directory listing of the local directory shows a zip file that is produced.

```
sqlhc_snc1_locutus_11.2.0.1.0_81s67vj4pjqm8_20121227_093856.zip
```

Now I put the zip file in a separate directory I created and unzip it.

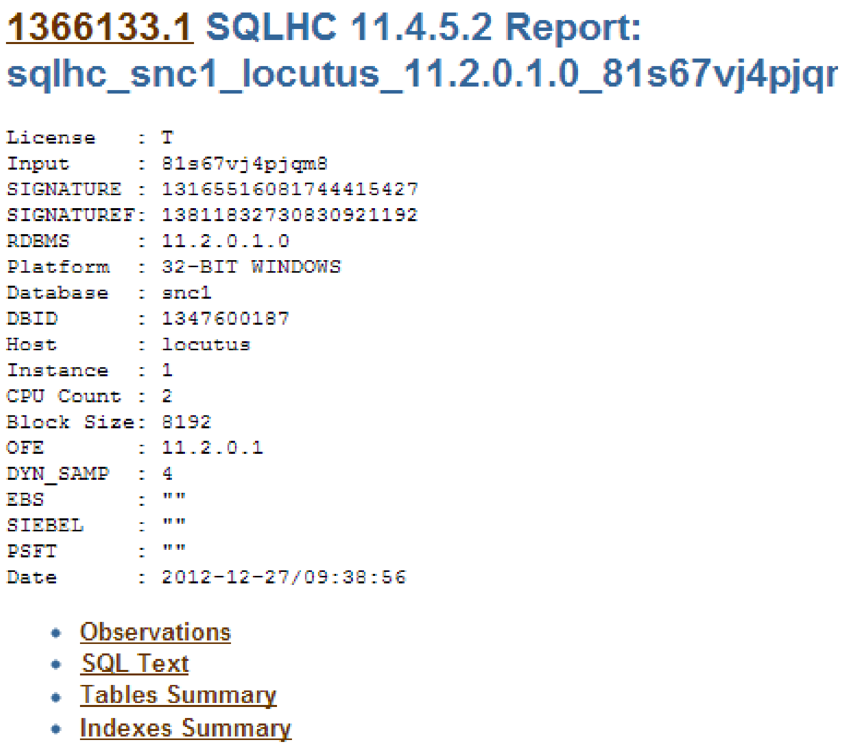
```
>mkdir sqlhc
>copy sqlhc_snc1_locutus_11.2.0.1.0_81s67vj4pjqm8_20121227_093856.zip sqlhc\
        1 file(s) copied.
```

```
>cd sqlhc
>unzip sqlhc_snc1_locutus_11.2.0.1.0_81s67vj4pjqm8_20121227_093856.zip
Archive:  sqlhc_snc1_locutus_11.2.0.1.0_81s67vj4pjqm8_20121227_093856.zip
  inflating: sqlhc_snc1_locutus_11.2.0.1.0_81s67vj4pjqm8_20121227_093856_1_health_check.html
  inflating: sqlhc_snc1_locutus_11.2.0.1.0_81s67vj4pjqm8_20121227_093856_2_diagnostics.html
  inflating: sqlhc_snc1_locutus_11.2.0.1.0_81s67vj4pjqm8_20121227_093856_3_execution_plans.html
  inflating: sqlhc_snc1_locutus_11.2.0.1.0_81s67vj4pjqm8_20121227_093856_4_sql_detail.html
 extracting: sqlhc_snc1_locutus_11.2.0.1.0_81s67vj4pjqm8_20121227_093856_9_log.zip
 extracting: sqlhc_snc1_locutus_11.2.0.1.0_81s67vj4pjqm8_20121227_093856_5_sql_monitor.zip
  inflating: sqlhc_snc1_locutus_11.2.0.1.0_81s67vj4pjqm8_20121227_093856_6_10053_trace_from_cursor.trc
```

And that's all there is in the running of the health check script. Now we need only look at each of the individual files.

The Main Health Check Report

The first HTML file contains the main report, which covers the observations, the SQL text (so we can make sure we are looking at the right page), and the table and index summary information. Figure 14-1 shows the top part of page one.



1366133.1 SQLHC 11.4.5.2 Report: sqlhc_snc1_locutus_11.2.0.1.0_81s67vj4pjqr

```
License    : T
Input      : 81s67vj4pjqm8
SIGNATURE  : 13165516081744415427
SIGNATUREF: 13811832730830921192
RDBMS      : 11.2.0.1.0
Platform   : 32-BIT WINDOWS
Database   : snc1
DBID       : 1347600187
Host       : locutus
Instance   : 1
CPU Count  : 2
Block Size: 8192
OFE        : 11.2.0.1
DYN_SAMP   : 4
EBS        : ""
SIEBEL     : ""
PSFT       : ""
Date       : 2012-12-27/09:38:56
```

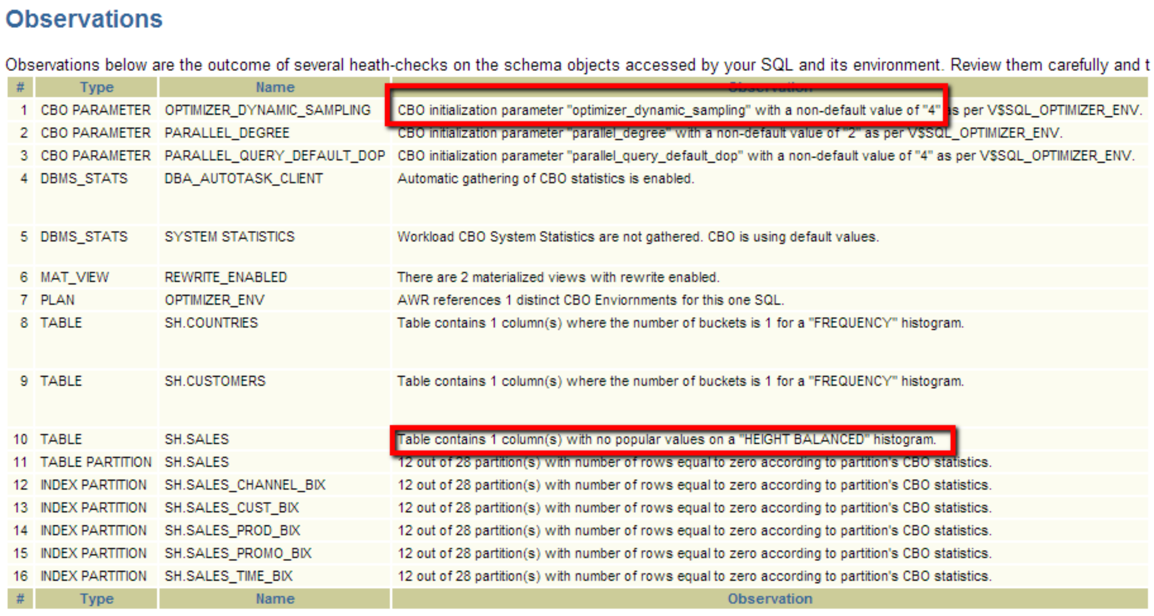
- Observations
- SQL Text
- Tables Summary
- Indexes Summary

Figure 14-1. *The top of the first SQLHC HTML file*

Each of the sections in the HTML report can be reached by use of the hyperlinks in the section shown above in the figure. I'll look at three of the four sections of this report in the following sections, omitting the SQL Text section, which is self-explanatory.

The Observations Section

The most interesting one is the "Observations" section, which lists what it considers observations of interest. Usually these are complaints about non-standard settings or architectural items that do not conform to best practices. Let's look at an example "Observations" section as shown in Figure 14-2.



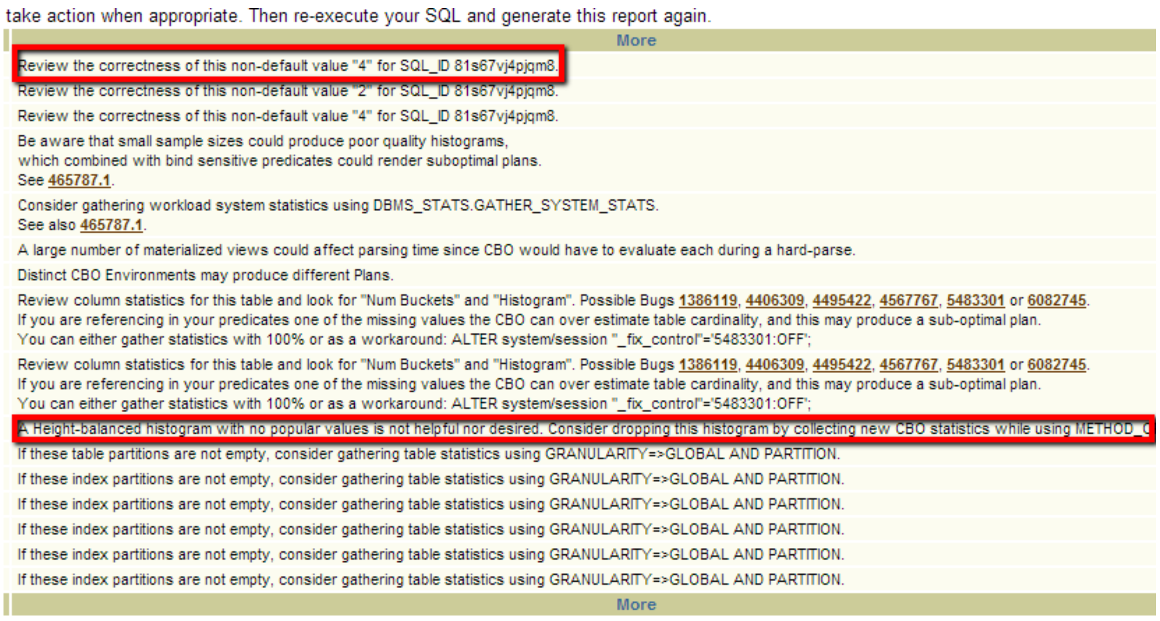
Observations

Observations below are the outcome of several heath-checks on the schema objects accessed by your SQL and its environment. Review them carefully and t

#	Type	Name	Observation
1	CBO PARAMETER	OPTIMIZER_DYNAMIC_SAMPLING	CBO initialization parameter "optimizer_dynamic_sampling" with a non-default value of "4" as per V$SQL_OPTIMIZER_ENV.
2	CBO PARAMETER	PARALLEL_DEGREE	CBO initialization parameter "parallel_degree" with a non-default value of "2" as per V$SQL_OPTIMIZER_ENV.
3	CBO PARAMETER	PARALLEL_QUERY_DEFAULT_DOP	CBO initialization parameter "parallel_query_default_dop" with a non-default value of "4" as per V$SQL_OPTIMIZER_ENV.
4	DBMS_STATS	DBA_AUTOTASK_CLIENT	Automatic gathering of CBO statistics is enabled.
5	DBMS_STATS	SYSTEM STATISTICS	Workload CBO System Statistics are not gathered. CBO is using default values.
6	MAT_VIEW	REWRITE_ENABLED	There are 2 materialized views with rewrite enabled.
7	PLAN	OPTIMIZER_ENV	AWR references 1 distinct CBO Enviornments for this one SQL.
8	TABLE	SH.COUNTRIES	Table contains 1 column(s) where the number of buckets is 1 for a "FREQUENCY" histogram.
9	TABLE	SH.CUSTOMERS	Table contains 1 column(s) where the number of buckets is 1 for a "FREQUENCY" histogram.
10	TABLE	SH.SALES	Table contains 1 column(s) with no popular values on a "HEIGHT BALANCED" histogram.
11	TABLE PARTITION	SH.SALES	12 out of 28 partition(s) with number of rows equal to zero according to partition's CBO statistics.
12	INDEX PARTITION	SH.SALES_CHANNEL_BIX	12 out of 28 partition(s) with number of rows equal to zero according to partition's CBO statistics.
13	INDEX PARTITION	SH.SALES_CUST_BIX	12 out of 28 partition(s) with number of rows equal to zero according to partition's CBO statistics.
14	INDEX PARTITION	SH.SALES_PROD_BIX	12 out of 28 partition(s) with number of rows equal to zero according to partition's CBO statistics.
15	INDEX PARTITION	SH.SALES_PROMO_BIX	12 out of 28 partition(s) with number of rows equal to zero according to partition's CBO statistics.
16	INDEX PARTITION	SH.SALES_TIME_BIX	12 out of 28 partition(s) with number of rows equal to zero according to partition's CBO statistics.
#	Type	Name	Observation

***Figure 14-2.** The Observations section of the health check*

The Observations section has many interesting suggestions for ways to improve your SQL. I have only shown the left side of the screen. In Figure 14-3 I show the corresponding right side of the screen, which tells us what needs to be done to remedy or investigate the observations. I have highlighted two observations in Figure 14-2, which could be useful to investigate. The first one related to the value of the parameter `optimizer_dynamic_sampling` having a non-default value of 4 (the default is 2). The second one is related to a height-balanced histogram that has no popular values. Let's look at the right hand side of the display (Figure 14-3) to see the column labeled "More" (changing to "Details" in subsequent versions of SQLT).


take action when appropriate. Then re-execute your SQL and generate this report again.

More

Review the correctness of this non-default value "4" for SQL_ID 81s67vj4pjqm8.

Review the correctness of this non-default value "2" for SQL_ID 81s67vj4pjqm8.

Review the correctness of this non-default value "4" for SQL_ID 81s67vj4pjqm8.

Be aware that small sample sizes could produce poor quality histograms,
which combined with bind sensitive predicates could render suboptimal plans.
See 465787.1.

Consider gathering workload system statistics using DBMS_STATS.GATHER_SYSTEM_STATS.
See also 465787.1.

A large number of materialized views could affect parsing time since CBO would have to evaluate each during a hard-parse.

Distinct CBO Environments may produce different Plans.

Review column statistics for this table and look for "Num Buckets" and "Histogram". Possible Bugs 1386119, 4406309, 4495422, 4567767, 5483301 or 6082745.
If you are referencing in your predicates one of the missing values the CBO can over estimate table cardinality, and this may produce a sub-optimal plan.
You can either gather statistics with 100% or as a workaround: ALTER system/session "_fix_control"='5483301:OFF';

Review column statistics for this table and look for "Num Buckets" and "Histogram". Possible Bugs 1386119, 4406309, 4495422, 4567767, 5483301 or 6082745.
If you are referencing in your predicates one of the missing values the CBO can over estimate table cardinality, and this may produce a sub-optimal plan.
You can either gather statistics with 100% or as a workaround: ALTER system/session "_fix_control"='5483301:OFF';

A Height-balanced histogram with no popular values is not helpful nor desired. Consider dropping this histogram by collecting new CBO statistics while using METHOD_O

If these table partitions are not empty, consider gathering table statistics using GRANULARITY=>GLOBAL AND PARTITION.

If these index partitions are not empty, consider gathering table statistics using GRANULARITY=>GLOBAL AND PARTITION.

If these index partitions are not empty, consider gathering table statistics using GRANULARITY=>GLOBAL AND PARTITION.

If these index partitions are not empty, consider gathering table statistics using GRANULARITY=>GLOBAL AND PARTITION.

If these index partitions are not empty, consider gathering table statistics using GRANULARITY=>GLOBAL AND PARTITION.

If these index partitions are not empty, consider gathering table statistics using GRANULARITY=>GLOBAL AND PARTITION.

More

Figure 14-3. *The right hand side of the observations section*

The "More" column should really be called the "suggestion" column. For example, in the case of the non-default `optimizer_dynamic_sampling` parameter the suggestion is "Review the correctness of this non-default value "4" for SQL_ID 81s67vj4pjqm8." This means that the value 4 was found, and it was "suggested" that we should think long and hard before sticking to the current value of 4. The suggested value is 2. If this were a SQL going into production there would have to be a discussion between DBAs and developers about why the value was set to 4 and whether 2 would be better.

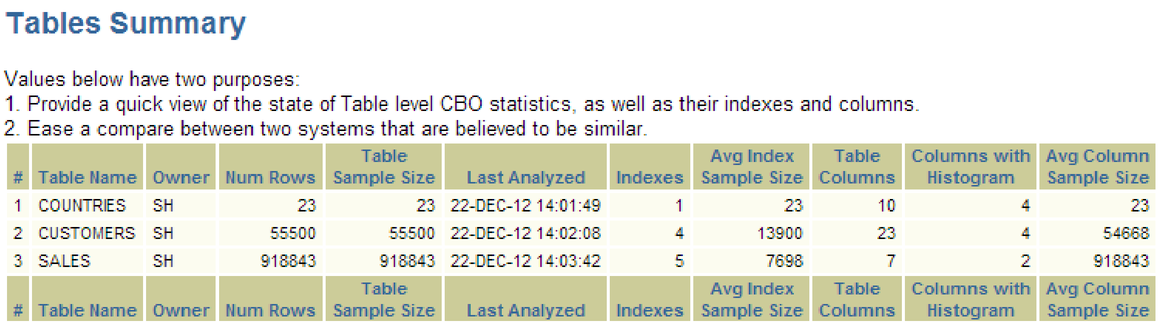
In the case of the histogram for SH.SALES, the suggestion is "A Height-balanced histogram with no popular values is not helpful nor desired. Consider dropping this histogram by collecting new CBO statistics while using METHOD_OPT with SIZE 1." This suggestion is saying that the histogram on this column is not useful and should be dropped. It will only waste time. The statistics collection option is even suggested to you. Again, if this appeared in a pre-production test there would have to be some discussion about this histogram and whether it was useful or not.

I've only mentioned two of the observations as examples. There are many suggestions (even for my simple SQL), and there are many other observation possibilities that do not appear because I did not violate the rules that cause them to appear. When you see the suggestions for your SQL you should carefully review them. Sometimes they are important (for example, the two mentioned above would be considered important), and in such cases you should either change your system (after careful testing) or have a good reason why those suggestions can be ignored. In most cases, observations are not generated when you run SQLHC because you did not trigger the rule by doing (or not doing) whatever it was that triggers the rule. This means that if you have a short list of observations you are doing well. If you have a long list of observations you may be doing badly.

In the case of this example I would consider "Automatic gathering of CBO statistics is enabled." as a relatively unimportant observation (in this case), since I am aware I am gathering statistics in that way and I know that if needed I will collect specific statistics to cover certain objects if needed. The warning is about small sample sizes in some cases causing non-optimal plans. It is not always easy to distinguish important observations from unimportant ones. How important an observation is will depend to some extent on your circumstances. Remember SQLHC is only following rules, it doesn't "know" your environment. You have to judge if an observation is important or not. With practice and a good knowledge of your environment you can develop the skill to quickly separate the wheat from the chaff. Each of the four HTML reports list the SQL text, but there's nothing to say about the SQL text, it's just the text of your SQL. If you've run the SQL many times with many different parameters or hints, it may be worth giving it a quick glance to make sure you're looking at the right report.

The Tables Summary Section

The "Tables Summary" section is the usual collection of information that you should be familiar with from a SQLT XTRACT or XECUTE report. See Figure 14-4.



Tables Summary

Values below have two purposes:
1. Provide a quick view of the state of Table level CBO statistics, as well as their indexes and columns.
2. Ease a compare between two systems that are believed to be similar.

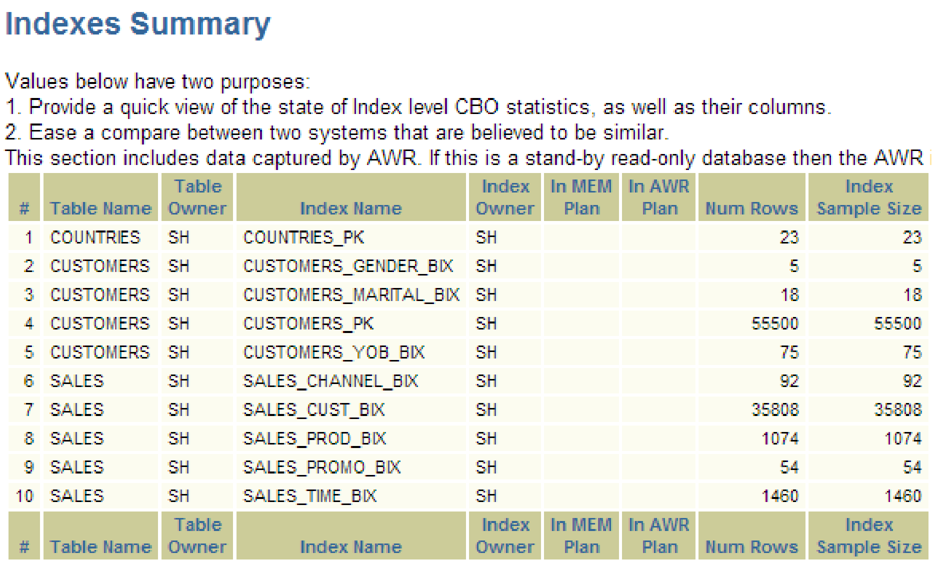
#	Table Name	Owner	Num Rows	Table Sample Size	Last Analyzed	Indexes	Avg Index Sample Size	Table Columns	Columns with Histogram	Avg Column Sample Size
1	COUNTRIES	SH	23	23	22-DEC-12 14:01:49	1	23	10	4	23
2	CUSTOMERS	SH	55500	55500	22-DEC-12 14:02:08	4	13900	23	4	54668
3	SALES	SH	918843	918843	22-DEC-12 14:03:42	5	7698	7	2	918843
#	Table Name	Owner	Num Rows	Table Sample Size	Last Analyzed	Indexes	Avg Index Sample Size	Table Columns	Columns with Histogram	Avg Column Sample Size

Figure 14-4. *The "Tables Summary" section of the first page of the SQLHC report*

In this report we see the tables involved in the query and the time of the statistics gathering. We also see some information about the number of indexes and columns, but notice no hyperlinks are present to guide us quickly through the report.

The Indexes Summary Section

The indexes are shown in Figure 14-5 and can be reached by clicking on "Indexes Summary" at the top of the page. There are no sophisticated links from the Tables section like there are in SQLT reports. This is because the SQLHC script is much simpler and cannot link in this way so easily.


Indexes Summary

Values below have two purposes:
1. Provide a quick view of the state of Index level CBO statistics, as well as their columns.
2. Ease a compare between two systems that are believed to be similar.
This section includes data captured by AWR. If this is a stand-by read-only database then the AWR i

#	Table Name	Table Owner	Index Name	Index Owner	In MEM Plan	In AWR Plan	Num Rows	Index Sample Size
1	COUNTRIES	SH	COUNTRIES_PK	SH			23	23
2	CUSTOMERS	SH	CUSTOMERS_GENDER_BIX	SH			5	5
3	CUSTOMERS	SH	CUSTOMERS_MARITAL_BIX	SH			18	18
4	CUSTOMERS	SH	CUSTOMERS_PK	SH			55500	55500
5	CUSTOMERS	SH	CUSTOMERS_YOB_BIX	SH			75	75
6	SALES	SH	SALES_CHANNEL_BIX	SH			92	92
7	SALES	SH	SALES_CUST_BIX	SH			35808	35808
8	SALES	SH	SALES_PROD_BIX	SH			1074	1074
9	SALES	SH	SALES_PROMO_BIX	SH			54	54
10	SALES	SH	SALES_TIME_BIX	SH			1460	1460
#	Table Name	Table Owner	Index Name	Index Owner	In MEM Plan	In AWR Plan	Num Rows	Index Sample Size

Figure 14-5. *The index report from SQLHC*

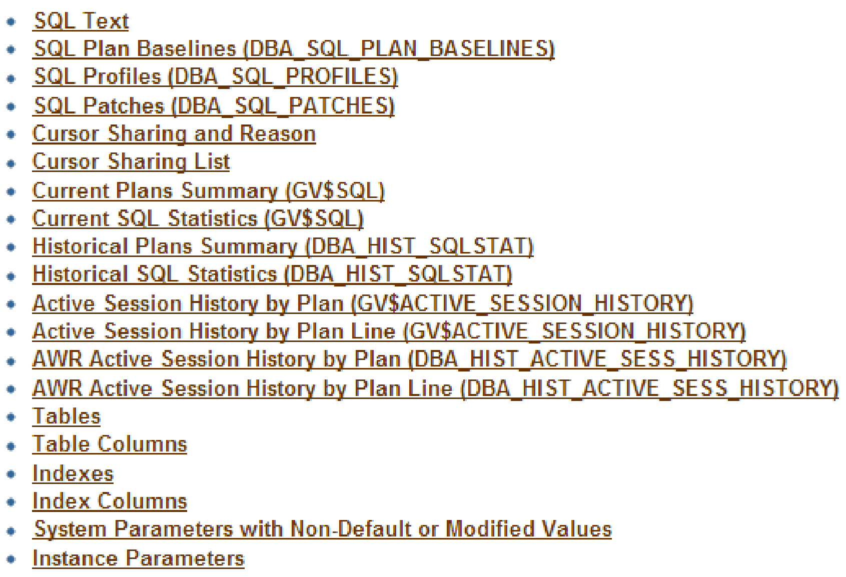
A similar page is made available for the indexes involved in the query. So the first HTML report from SQLHC is a fairly simple collection of data on the SQL tables and indexes and any observations related to these. The next SQLHC HTML report has more sections in it, including some historical information.

The Diagnostics Report

The second page of the SQLHC report can potentially show many pieces of information: for example, instance parameters. Some sections may contain no information: for example, in my case there are no profiles in use. In fact, if you go to that section in the report you will see the text

```
Available in 10g or higher. If this section is empty that means there are no profiles for this SQL.
```

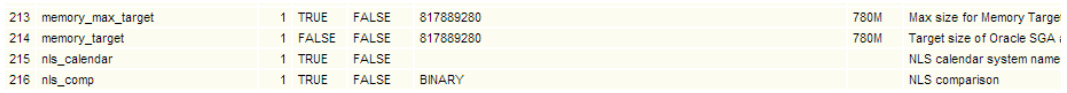
(We covered profiles in more detail in Chapter 6.) Figure 14-6 shows the header part of the HTML report.



- SQL Text
- SQL Plan Baselines (DBA_SQL_PLAN_BASELINES)
- SQL Profiles (DBA_SQL_PROFILES)
- SQL Patches (DBA_SQL_PATCHES)
- Cursor Sharing and Reason
- Cursor Sharing List
- Current Plans Summary (GV$SQL)
- Current SQL Statistics (GV$SQL)
- Historical Plans Summary (DBA_HIST_SQLSTAT)
- Historical SQL Statistics (DBA_HIST_SQLSTAT)
- Active Session History by Plan (GV$ACTIVE_SESSION_HISTORY)
- Active Session History by Plan Line (GV$ACTIVE_SESSION_HISTORY)
- AWR Active Session History by Plan (DBA_HIST_ACTIVE_SESS_HISTORY)
- AWR Active Session History by Plan Line (DBA_HIST_ACTIVE_SESS_HISTORY)
- Tables
- Table Columns
- Indexes
- Index Columns
- System Parameters with Non-Default or Modified Values
- Instance Parameters

Figure 14-6. *We see here the header section of page 2 of the SQLHC report*

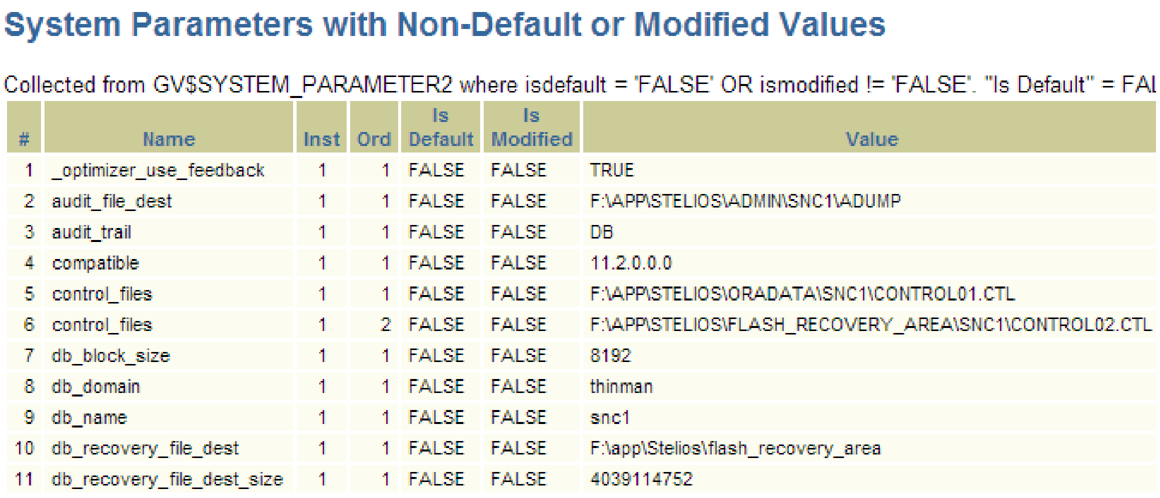
From the header you can link to all the sections mentioned through hyperlinks. There is much more detailed statistical information in this report. For example, we have all the system parameters (not just the default ones) along with the description of the parameter. This section should be scanned by the DBA to ensure that the understanding of the system matches with what you see in this section. For example, I see that `memory_target` is set to 780M (see Figure 14-7).



213	memory_max_target	1	TRUE	FALSE	817889280	780M	Max size for Memory Targe
214	memory_target	1	FALSE	FALSE	817889280	780M	Target size of Oracle SGA a
215	nls_calendar	1	TRUE	FALSE			NLS calendar system name
216	nls_comp	1	TRUE	FALSE	BINARY		NLS comparison

Figure 14-7. *A snippet from the Instance Parameter section of the report*

Is this what I expected? In my case, yes. This is to check your instance settings against what you are expecting for the instance. Another example might be `optimizer_mode` is set to ALL_ROWS. Luckily I know this is the default, and this is the value I expect. For every parameter on the system you should have a good understanding of why it is set that way. In Figure 14-8 we see the non-default parameters. You should pay even more attention to these because SQLHC is telling you that these are out of the ordinary.

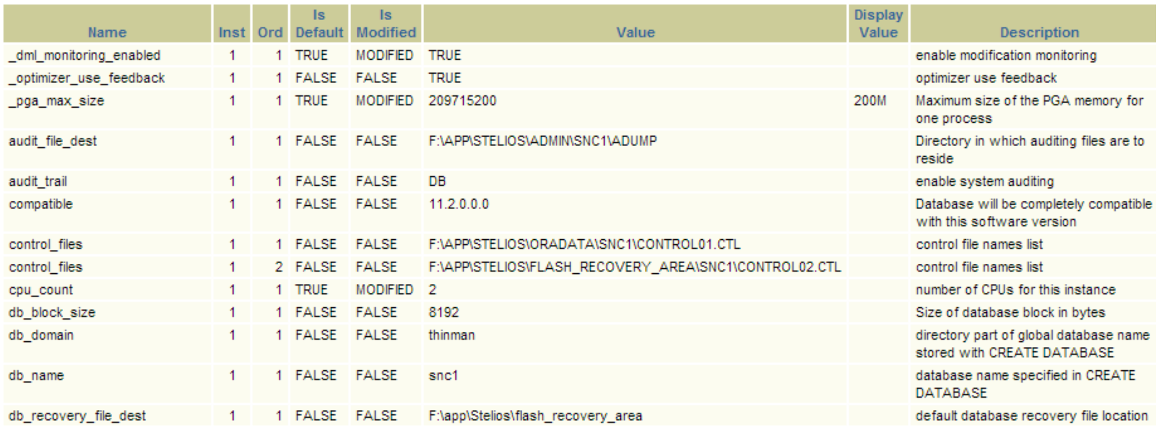


System Parameters with Non-Default or Modified Values

Collected from GV$SYSTEM_PARAMETER2 where isdefault = 'FALSE' OR ismodified != 'FALSE'. "Is Default" = FAI

#	Name	Inst	Ord	Is Default	Is Modified	Value
1	_optimizer_use_feedback	1	1	FALSE	FALSE	TRUE
2	audit_file_dest	1	1	FALSE	FALSE	F:\APP\STELIOS\ADMIN\SNC1\ADUMP
3	audit_trail	1	1	FALSE	FALSE	DB
4	compatible	1	1	FALSE	FALSE	11.2.0.0.0
5	control_files	1	1	FALSE	FALSE	F:\APP\STELIOS\ORADATA\SNC1\CONTROL01.CTL
6	control_files	1	2	FALSE	FALSE	F:\APP\STELIOS\FLASH_RECOVERY_AREA\SNC1\CONTROL02.CTL
7	db_block_size	1	1	FALSE	FALSE	8192
8	db_domain	1	1	FALSE	FALSE	thinman
9	db_name	1	1	FALSE	FALSE	snc1
10	db_recovery_file_dest	1	1	FALSE	FALSE	F:\app\Stelios\flash_recovery_area
11	db_recovery_file_dest_size	1	1	FALSE	FALSE	4039114752

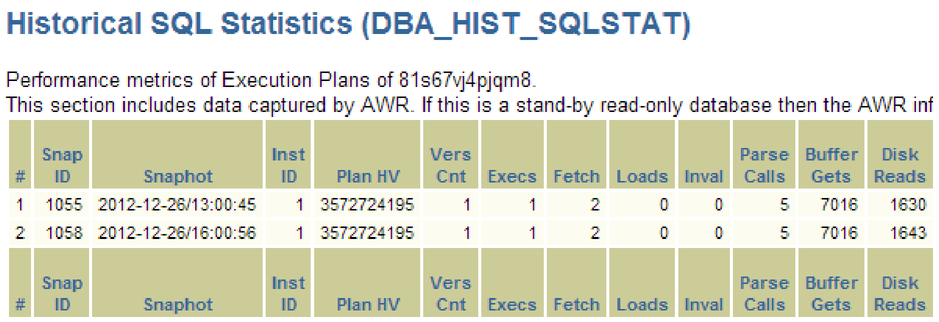
Figure 14-8. *Part of the Non-default System Parameter section of the report*



Name	Inst	Ord	Is Default	Is Modified	Value	Display Value	Description
_dml_monitoring_enabled	1	1	TRUE	MODIFIED	TRUE		enable modification monitoring
_optimizer_use_feedback	1	1	FALSE	FALSE	TRUE		optimizer use feedback
_pga_max_size	1	1	TRUE	MODIFIED	209715200	200M	Maximum size of the PGA memory for one process
audit_file_dest	1	1	FALSE	FALSE	F:\APP\STELIOS\ADMIN\SNC1\ADUMP		Directory in which auditing files are to reside
audit_trail	1	1	FALSE	FALSE	DB		enable system auditing
compatible	1	1	FALSE	FALSE	11.2.0.0.0		Database will be completely compatible with this software version
control_files	1	1	FALSE	FALSE	F:\APP\STELIOS\ORADATA\SNC1\CONTROL01.CTL		control file names list
control_files	1	2	FALSE	FALSE	F:\APP\STELIOS\FLASH_RECOVERY_AREA\SNC1\CONTROL02.CTL		control file names list
cpu_count	1	1	TRUE	MODIFIED	2		number of CPUs for this instance
db_block_size	1	1	FALSE	FALSE	8192		Size of database block in bytes
db_domain	1	1	FALSE	FALSE	thinman		directory part of global database name stored with CREATE DATABASE
db_name	1	1	FALSE	FALSE	snc1		database name specified in CREATE DATABASE
db_recovery_file_dest	1	1	FALSE	FALSE	F:\app\Stelios\flash_recovery_area		default database recovery file location

Figure 14-9. *The descriptions of the intialization parameters*

We see in the figure that the default values of parameters are highlighted and their current value, if manually set. For example `_optimizer_use_feedback` is set to `TRUE` (which is the default value, but its value was set in the `spfile`). If we look at this page (see Figure 14-10) we see the descriptions of the parameters.



Historical SQL Statistics (DBA_HIST_SQLSTAT)

Performance metrics of Execution Plans of 81s67vj4pjqm8.
This section includes data captured by AWR. If this is a stand-by read-only database then the AWR inf

#	Snap ID	Snaphot	Inst ID	Plan HV	Vers Cnt	Execs	Fetch	Loads	Inval	Parse Calls	Buffer Gets	Disk Reads
1	1055	2012-12-26/13:00:45	1	3572724195	1	1	2	0	0	5	7016	1630
2	1058	2012-12-26/16:00:56	1	3572724195	1	1	2	0	0	5	7016	1643
#	Snap ID	Snaphot	Inst ID	Plan HV	Vers Cnt	Execs	Fetch	Loads	Inval	Parse Calls	Buffer Gets	Disk Reads

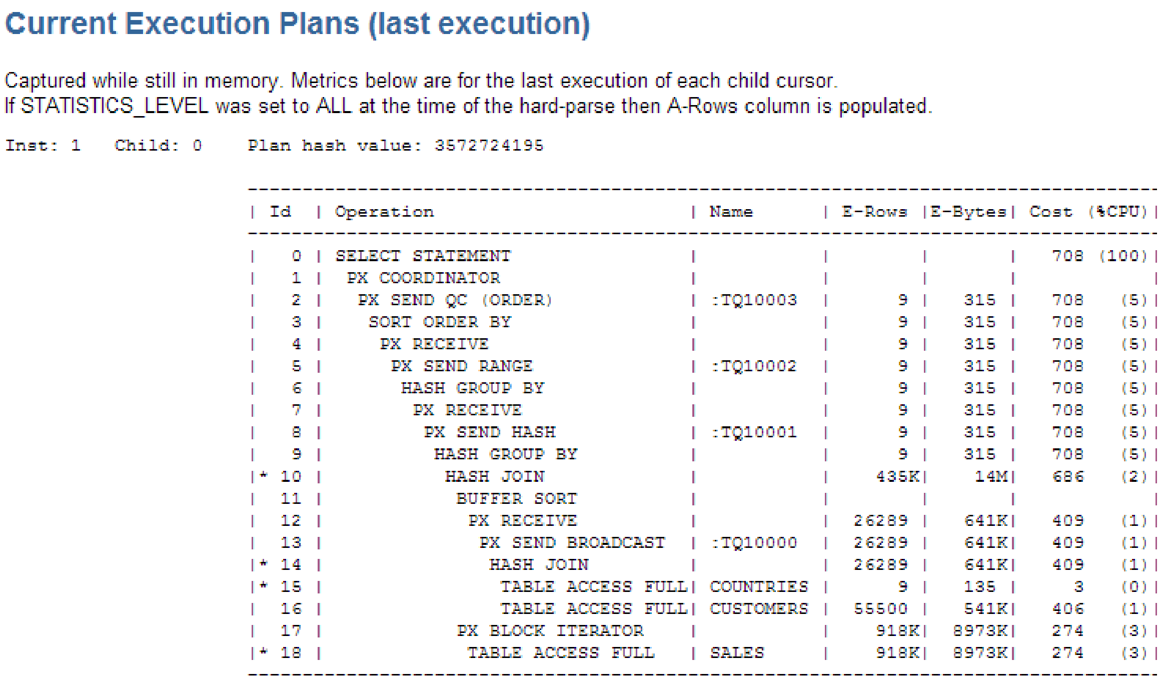
***Figure 14-10.** The historical information from our example SQL. (Only the left hand side of the report is shown)*

Finally we can look at the historical information that SQHC can provide. All the usual information is there: disk reads, buffer gets, etc. Just as in SQLT we can use this information to relate the history to any change in the execution plan.

As you can see, the second report from the SQLHC report is useful because it shows the metrics related to the statistics execution, as well as reports on non-default parameters and a historical view of the execution of your SQL. The historical information in conjunction with the plan hash value (the unique identifier for a specific execution plan [PHV for short]), and the system initialization parameters can be very useful in working out what is happening with your SQL.

The Execution Plan Report

The next report shows the execution plan for all the SQLs captured. See Figure 14-11, which shows the execution plan for the latest execution.



Current Execution Plans (last execution)

Captured while still in memory. Metrics below are for the last execution of each child cursor.
If STATISTICS_LEVEL was set to ALL at the time of the hard-parse then A-Rows column is populated.

```
Inst: 1   Child: 0    Plan hash value: 3572724195

                         ------------------------------------------------------------------------------------
                         | Id  | Operation                  | Name       | E-Rows |E-Bytes| Cost  (%CPU)|
                         ------------------------------------------------------------------------------------
                         |   0 | SELECT STATEMENT           |            |        |       |   708  (100)|
                         |   1 |  PX COORDINATOR            |            |        |       |             |
                         |   2 |   PX SEND QC (ORDER)       | :TQ10003   |      9 |   315 |   708    (5)|
                         |   3 |    SORT ORDER BY           |            |      9 |   315 |   708    (5)|
                         |   4 |     PX RECEIVE             |            |      9 |   315 |   708    (5)|
                         |   5 |      PX SEND RANGE         | :TQ10002   |      9 |   315 |   708    (5)|
                         |   6 |       HASH GROUP BY        |            |      9 |   315 |   708    (5)|
                         |   7 |        PX RECEIVE          |            |      9 |   315 |   708    (5)|
                         |   8 |         PX SEND HASH       | :TQ10001   |      9 |   315 |   708    (5)|
                         |   9 |          HASH GROUP BY     |            |      9 |   315 |   708    (5)|
                         |* 10 |           HASH JOIN        |            |   435K|    14M|   686    (2)|
                         |  11 |            BUFFER SORT     |            |        |       |             |
                         |  12 |             PX RECEIVE     |            |  26289 |   641K|   409    (1)|
                         |  13 |              PX SEND BROADCAST | :TQ10000 |  26289 |   641K|   409    (1)|
                         |* 14 |               HASH JOIN    |            |  26289 |   641K|   409    (1)|
                         |* 15 |                TABLE ACCESS FULL| COUNTRIES |      9 |   135 |     3    (0)|
                         |  16 |                TABLE ACCESS FULL| CUSTOMERS |  55500 |   541K|   406    (1)|
                         |  17 |            PX BLOCK ITERATOR |          |   918K|  8973K|   274    (3)|
                         |* 18 |             TABLE ACCESS FULL | SALES    |   918K|  8973K|   274    (3)|
                         ------------------------------------------------------------------------------------
```

***Figure 14-11.** The left hand side of the report showing the execution plan of the last execution*

I've shown only the left hand side of the report, but all the usual columns are present for the execution plan. This execution plan shows a parallelized execution with a hash join between SALES and the hash join of COUNTRIES and CUSTOMERS. As you would for an XTRACT report look at the steps and determine if they are appropriate, later in this chapter we'll also see a SQL monitor report that shows where time was spent during the execution. Look at the expected rows, look at the cost of each line and see if they meet your expectations. Additionally you can see the historical execution plans for the same SQL. Remember all this information is available from a system where you have NOT installed SQLT.

The Monitor Reports

There are two types of monitor reports. The first one is the previously mentioned HTML file that provides a summary of the activity of all the SQL being monitored. The second type of monitor report consists of the details of execution plans. A zip file includes a separate report with the details for each execution plan.

The SQL Monitor Summary Report

The purpose of the first report is to put the SQL under investigation into some sort of context. After all, if you are tuning an SQL statement, you want to make sure it's having a significant effect on the system. In other words, there's no point in tuning an SQL statement so that it takes half the time (if it's only run once), and it uses only 1 percent of the system resources when there's another statement that takes 50 percent of the system resources.

This summary view has three panes: "SQL Text", "Top Activity", and "Details". You can expand and minimize these sections by clicking the minimize icons on the right hand side of the pane. The "Detail" pane has four icons, which cover "Activity", "Statistics", "Plan", and "SQL Monitoring".

- Activity: This shows the active sessions in a graphic form and shows what the active sessions were doing. For example in Figure 14-12 we see the "Activity" button in the "Details" section, which shows a spike in "direct path read" and "db file sequential read" near the end of the sampling interval.

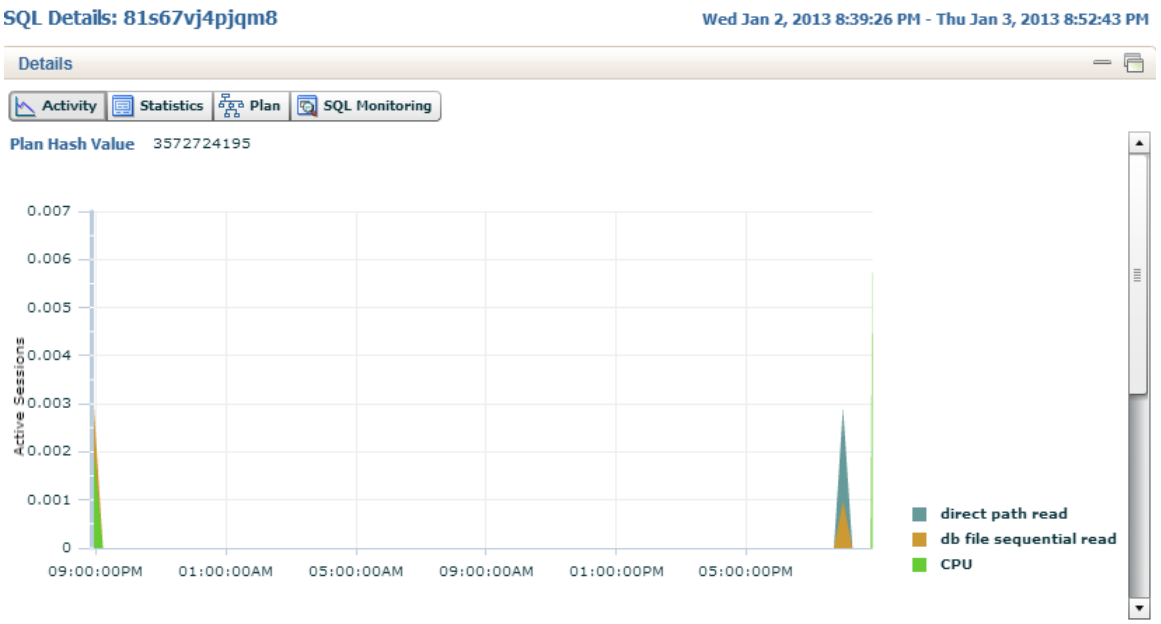
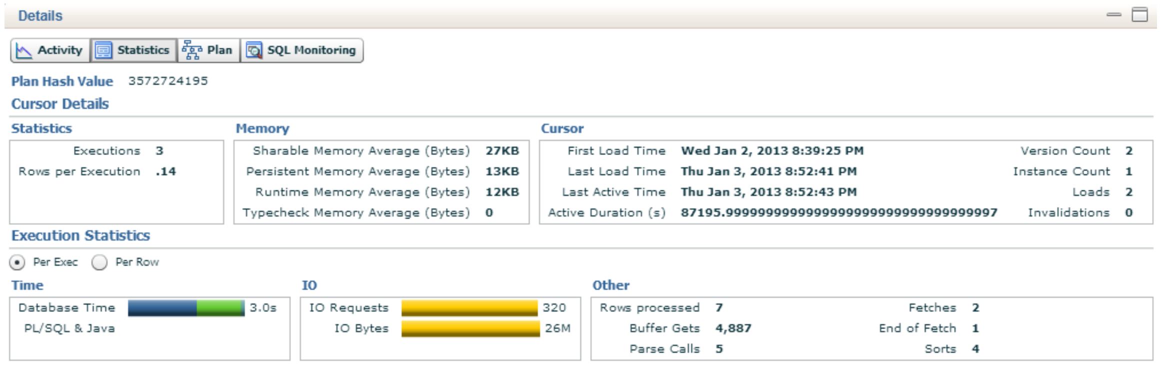


Figure 14-12. *The Detail section activity chart*

- Statistics: Shows the cursor details, including such details as number of executions, memory usage, cursor load times, number of versions of the cursor and database time used as well as I/O requests and many other details as shown in Figure 14-13.



Plan Hash Value 3572724195

Cursor Details

Statistics

Executions	3
Rows per Execution	.14

Memory

Sharable Memory Average (Bytes)	27KB
Persistent Memory Average (Bytes)	13KB
Runtime Memory Average (Bytes)	12KB
Typecheck Memory Average (Bytes)	0

Cursor

First Load Time	Wed Jan 2, 2013 8:39:25 PM	Version Count	2
Last Load Time	Thu Jan 3, 2013 8:52:41 PM	Instance Count	1
Last Active Time	Thu Jan 3, 2013 8:52:43 PM	Loads	2
Active Duration (s)	87195.99999999999999999999999999999997	Invalidations	0

Execution Statistics

[● Per Exec] ○ Per Row

Time

Database Time	3.0s
PL/SQL & Java	

IO

IO Requests	320
IO Bytes	26M

Other

Rows processed	7	Fetches	2
Buffer Gets	4,887	End of Fetch	1
Parse Calls	5	Sorts	4

Figure 14-13. *Showing an example statistics page from the Details section of the report*

- Plan: Shows the execution plan in graphic form. An example of this is shown in Figure 14-14, showing the plan in left to right style.

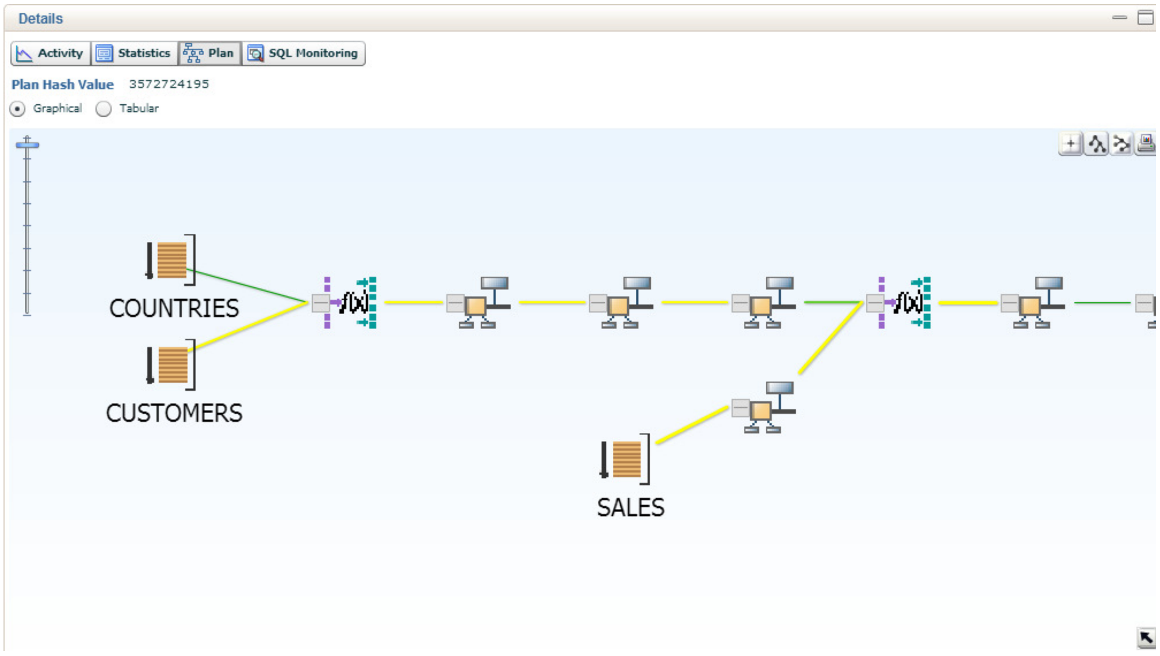
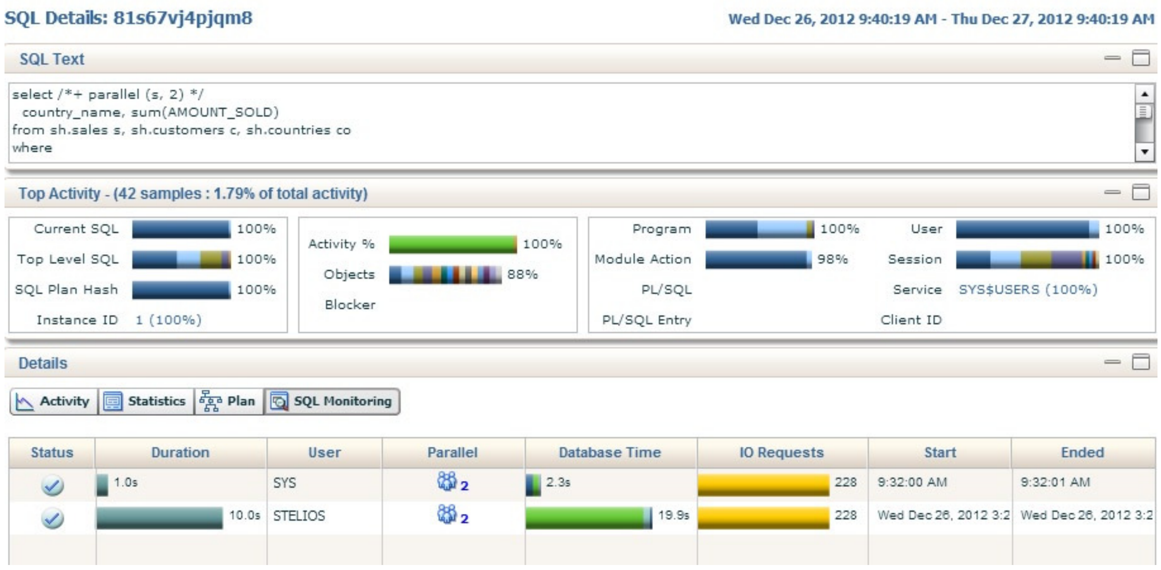


Figure 14-14. *Shows the plan section of the Detail tab*

- SQL Monitoring: Shows the duration, users, serial/parallel status, and other statistics related to the SQL ID. An example of this is shown in Figure 14-15.



SQL Details: 81s67vj4pjqm8 Wed Dec 26, 2012 9:40:19 AM - Thu Dec 27, 2012 9:40:19 AM

SQL Text

```
select /*+ parallel (s, 2) */
  country_name, sum(AMOUNT_SOLD)
from sh.sales s, sh.customers c, sh.countries co
where
```

Top Activity - (42 samples : 1.79% of total activity)

Current SQL	100%	Activity %	100%	Program	100%	User	100%
Top Level SQL	100%	Objects	88%	Module Action	98%	Session	100%
SQL Plan Hash	100%	Blocker		PL/SQL		Service	SYS$USERS (100%)
Instance ID	1 (100%)			PL/SQL Entry		Client ID	

Details

Activity | Statistics | Plan | SQL Monitoring

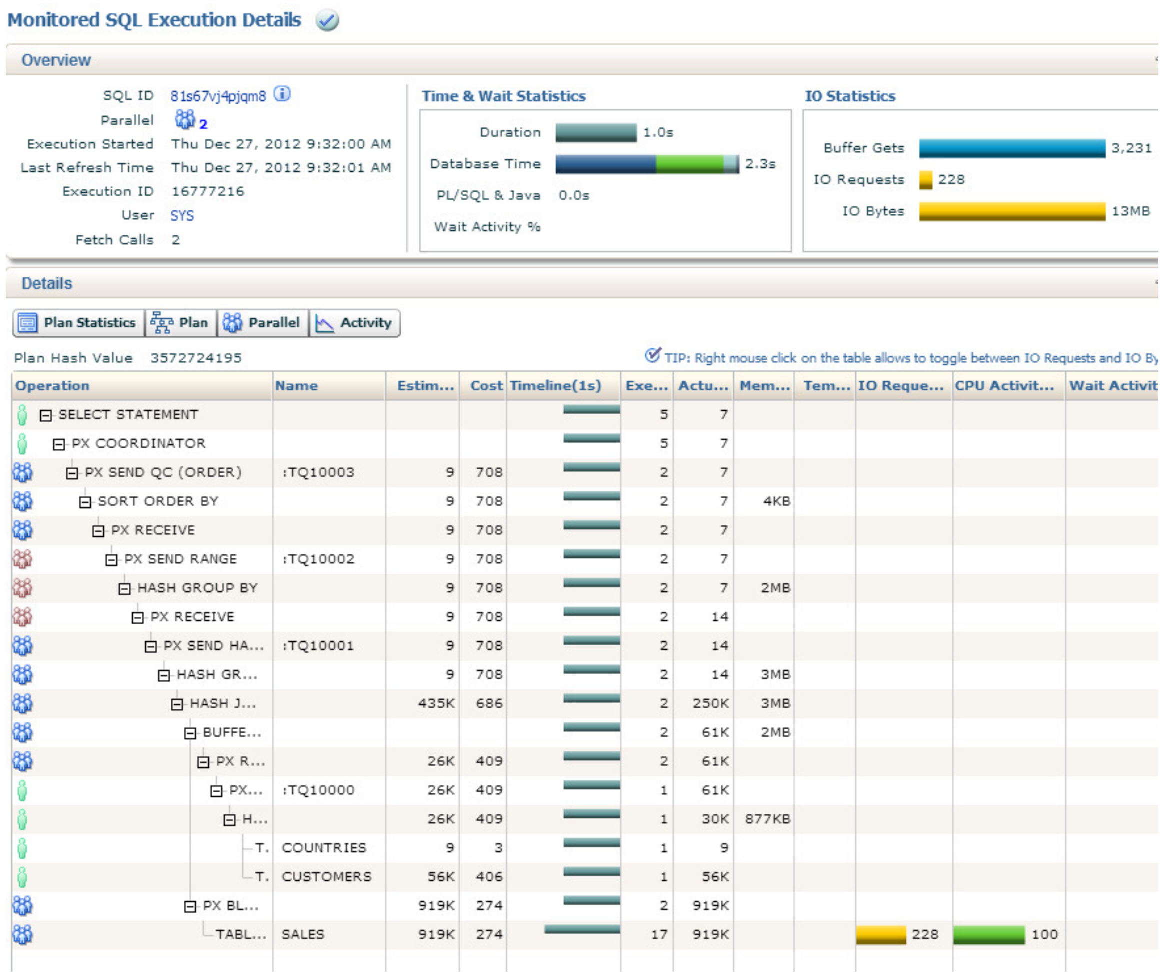
Status	Duration	User	Parallel	Database Time	IO Requests	Start	Ended
✓	1.0s	SYS	2	2.3s	228	9:32:00 AM	9:32:01 AM
✓	10.0s	STELIOS	2	19.9s	228	Wed Dec 26, 2012 3:2	Wed Dec 26, 2012 3:2

Figure 14-15. *The SQL Monitor tab on the Details page*

This report shows that for the current SQL text, almost 100% of the database activity was that SQL. We see in the bottom half of the report that my SQL took 10 seconds to run and that it ran in parallel. The result of this was that it used almost 20 seconds of CPU time.

The SQL Monitor Detail Reports

The other SQL monitor reports show the details of the execution plan. All of the SQL monitor reports for this are zipped up into one file, and each one represents one execution. Figure 14-16 shows the monitored SQL execution plan for the SQL in detail.



Monitored SQL Execution Details

Overview

SQL ID	81s67vj4pjqm8
Parallel	2
Execution Started	Thu Dec 27, 2012 9:32:00 AM
Last Refresh Time	Thu Dec 27, 2012 9:32:01 AM
Execution ID	16777216
User	SYS
Fetch Calls	2

Time & Wait Statistics

Duration	1.0s
Database Time	2.3s
PL/SQL & Java	0.0s
Wait Activity %	

IO Statistics

Buffer Gets	3,231
IO Requests	228
IO Bytes	13MB

Details

Plan Statistics | Plan | Parallel | Activity

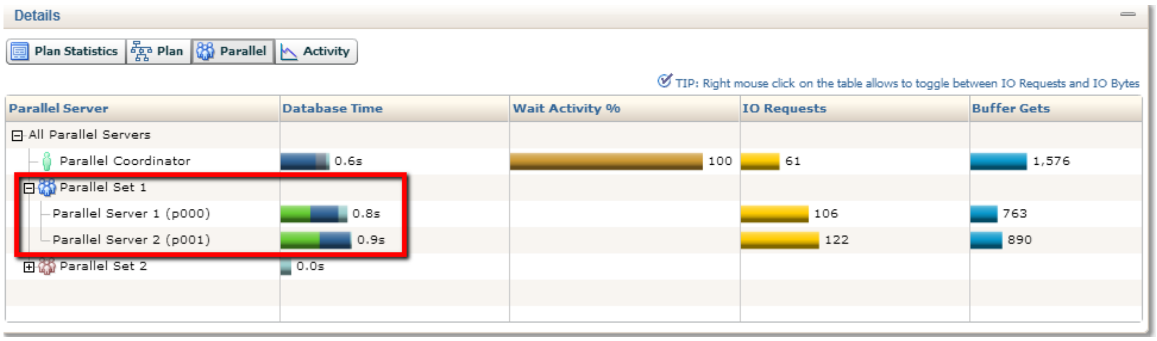
Plan Hash Value 3572724195

TIP: Right mouse click on the table allows to toggle between IO Requests and IO By

Operation	Name	Estim...	Cost	Timeline(1s)	Exe...	Actu...	Mem...	Tem...	IO Reque...	CPU Activit...	Wait Activit
SELECT STATEMENT					5	7					
PX COORDINATOR					5	7					
PX SEND QC (ORDER)	:TQ10003	9	708		2	7					
SORT ORDER BY		9	708		2	7	4KB				
PX RECEIVE		9	708		2	7					
PX SEND RANGE	:TQ10002	9	708		2	7					
HASH GROUP BY		9	708		2	7	2MB				
PX RECEIVE		9	708		2	14					
PX SEND HA...	:TQ10001	9	708		2	14					
HASH GR...		9	708		2	14	3MB				
HASH J...		435K	686		2	250K	3MB				
BUFFE...					2	61K	2MB				
PX R...		26K	409		2	61K					
PX...	:TQ10000	26K	409		1	61K					
H...		26K	409		1	30K	877KB				
T.	COUNTRIES	9	3		1	9					
T.	CUSTOMERS	56K	406		1	56K					
PX BL...		919K	274		2	919K					
TABL...	SALES	919K	274		17	919K			228	100	

Figure 14-16. *One of the SQL Monitor reports in the Detail pane for the Plan Statistics button*

We covered this kind of execution plan in Chapter 9. The Plan button of the Details pane shows a graphical plan layout just like in Figure 14-14, shown earlier. The Parallel button shows some parallel specific information (see Figure 14-17). This shows the I/O requests, buffer gets, and CPU time spent by each parallel set. For example, in my SQL I see that Parallel server p000 used 0.8 seconds of database time while P001 used 0.9 seconds. If I hover my mouse over the bar for those I get even more detail on this, broken down into CPU, I/O, and other. This kind of report should allow you to investigate particular branches of your complex code quickly and easily.



TIP: Right mouse click on the table allows to toggle between IO Requests and IO Bytes

Parallel Server	Database Time	Wait Activity %	IO Requests	Buffer Gets
All Parallel Servers				
Parallel Coordinator	0.6s	100	61	1,576
Parallel Set 1				
Parallel Server 1 (p000)	0.8s		106	763
Parallel Server 2 (p001)	0.9s		122	890
Parallel Set 2	0.0s			

Figure 14-17. *Showing the Parallel button page on the Details pane. Parallel Set 1 has been expanded*

The rest of the files inside the main zip file, produced by the `sqlhc.sql` script are text files. One is a 10053 trace file of the SQL (covered in Chapter 5) and the other is the log of the `sqlhc.sql` script itself (which you can check for errors) and the temporary scripts that were generated to produce the HTML files.

The sqldx.sql Script

Sqldx was written to collect detailed information about one SQL statement. It does not need SQLT and produces many CSV formatted files (comma separated values) to take away from a further analysis. It also produces over 20 HTML formatted files.

In the example below I've used the same SQL, `q3.sql`, which I have used throughout this chapter. To generate a report from `sqldx.sql`, we need only run the script on the system where the SQL has run. The SQL_ID in this case is `81s67vj4pjqm8`. When we run `sqldx.sql` we get a prompt to confirm our license level. Select the appropriate license level. In my case this is `T`.

```
SQL> @sqldx
Parameter 1:
Oracle Pack License (Tuning or Diagnostics) [T|D] (required)
Enter value for 1: T
PL/SQL procedure successfully completed.
```

Now I need to enter the format of the report. `H` stands for HTML and `C` stands for a CSV report and `B` stands for both. I'm going to choose `B` in this case to create the maximum number of files. Generally speaking the HTML files are the most useful. The CSVs are a nice to have. They contain information I have not used thus far.

```
Parameter 2:
Output Type (HTML or CSV or Both) [H|C|B] (required)
Enter value for 2: B
PL/SQL procedure successfully completed.
```

```
Parameter 3:
SQL_ID of the SQL to be analyzed (required)
Enter value for 3: 81s67vj4pjqm8
```

Before the main execution starts we see the values passed as a confirmation that we are doing what we wanted.

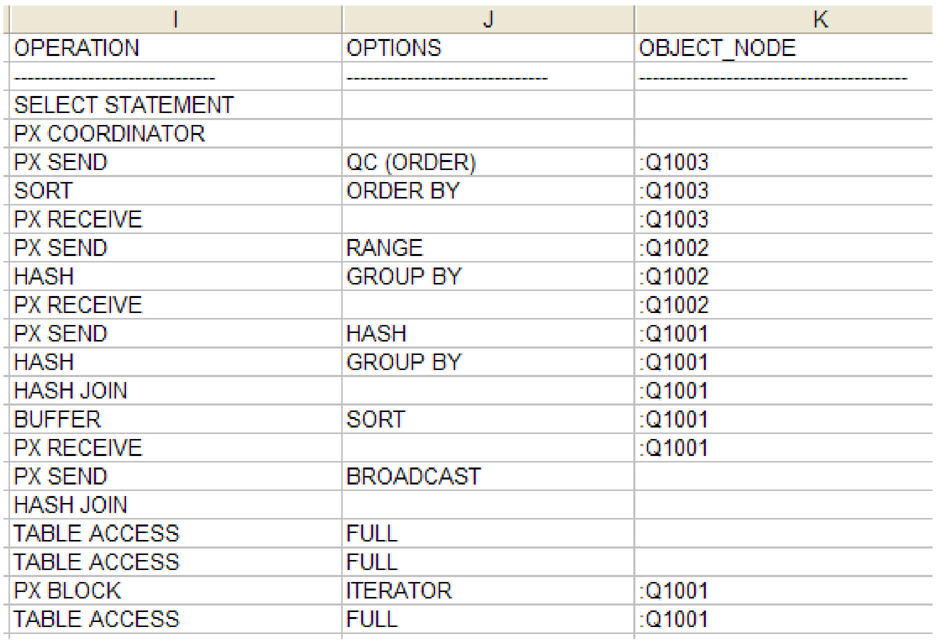
```
Values passed:
~~~~~~~~~~~~~
License: "T"
Output : "B"
SQL_ID : "81s67vj4pjqm8"
```

The execution proceeds, it takes just a few minutes usually, but may take longer if there is more data in your database to analyze. After a couple of pages of output we see the final result.

```
SQLDX files have been created.
Archive:  sqldx_20130103_195515.zip
  Length      Date    Time    Name
---------  ---------- -----   ----
    39708  01/03/2013 19:55   sqldx_20130103_195515_81s67vj4pjqm8_csv.zip
    67897  01/03/2013 19:55   sqldx_20130103_195515_81s67vj4pjqm8_html.zip
    15118  01/03/2013 19:55   sqldx_20130103_195515_13811832730830921192_force_csv.zip
    28024  01/03/2013 19:55   sqldx_20130103_195515_13811832730830921192_force_html.zip
    26717  01/03/2013 19:55   sqldx_20130103_195515_global_csv.zip
    21463  01/03/2013 19:55   sqldx_20130103_195515_global_html.zip
     6976  01/03/2013 19:55   sqldx_20130103_195515_81s67vj4pjqm8_log.zip
---------                     -------
   205903                     7 files
```

The zip file produced contains seven more zip files. Yes, that's what I said: zip files within zip files. If you want to keep track of which file came from which, create a directory for each zip file and unpack the zip file into that directory. If that zip file contains other zip files then create a sub-directory to keep those and so on.

- Three CSV zip files (Normal, Force, and Global). These contain comma-separated files of the information gathered. They are in a form suitable to import into a spreadsheet. I don't think this is as useful as the HTML format, but it may be useful for further analysis: for example, you could add up shared memory use or number of executions if you were investigating a complex parallel statement. See Figure 14-18 below, which shows an example spreadsheet created from `sqldx_20130103_195515_81s67vj4pjqm8_GVsSQL_PLAN.csv`. There are 20 other CSV files in total in here with various forms of information all in CSV format. (After the figure is a hierarchical listing of the files in the various zip, which I show to make the layout clear.)



I	J	K
OPERATION	OPTIONS	OBJECT_NODE
------------------------------	------------------------------	--
SELECT STATEMENT		
PX COORDINATOR		
PX SEND	QC (ORDER)	:Q1003
SORT	ORDER BY	:Q1003
PX RECEIVE		:Q1003
PX SEND	RANGE	:Q1002
HASH	GROUP BY	:Q1002
PX RECEIVE		:Q1002
PX SEND	HASH	:Q1001
HASH	GROUP BY	:Q1001
HASH JOIN		:Q1001
BUFFER	SORT	:Q1001
PX RECEIVE		:Q1001
PX SEND	BROADCAST	
HASH JOIN		
TABLE ACCESS	FULL	
TABLE ACCESS	FULL	
PX BLOCK	ITERATOR	:Q1001
TABLE ACCESS	FULL	:Q1001

Figure 14-18. *A section of an import CSV result*

- Three HTML zip files (Normal, Force, and Global). These are the files we'll look at to see what sqldx has collected.
- A Log zip file

Here's a hierarchical listing of the files starting from the main zip file: `sqldx_20130103_195515.zip`. `sqldx_20130103_195515.zip` contains:

- The CSV directory, which contains:

```
sqldx_20130103_195515_81s67vj4pjqm8_DBA_HIST_ACTIVE_SESS_HISTORY.csv
sqldx_20130103_195515_81s67vj4pjqm8_DBA_HIST_SQLSTAT.csv
sqldx_20130103_195515_81s67vj4pjqm8_DBA_HIST_SQLTEXT.csv
sqldx_20130103_195515_81s67vj4pjqm8_DBA_HIST_SQL_PLAN.csv
sqldx_20130103_195515_81s67vj4pjqm8_GVsACTIVE_SESSION_HISTORY.csv
sqldx_20130103_195515_81s67vj4pjqm8_GVsSQL.csv
sqldx_20130103_195515_81s67vj4pjqm8_GVsSQLAREA.csv
sqldx_20130103_195515_81s67vj4pjqm8_GVsSQLAREA_PLAN_HASH.csv
sqldx_20130103_195515_81s67vj4pjqm8_GVsSQLSTATS.csv
sqldx_20130103_195515_81s67vj4pjqm8_GVsSQLSTATS_PLAN_HASH.csv
sqldx_20130103_195515_81s67vj4pjqm8_GVsSQLTEXT.csv
sqldx_20130103_195515_81s67vj4pjqm8_GVsSQLTEXT_WITH_NEWLINES.csv
sqldx_20130103_195515_81s67vj4pjqm8_GVsSQL_MONITOR.csv
sqldx_20130103_195515_81s67vj4pjqm8_GVsSQL_OPTIMIZER_ENV.csv
sqldx_20130103_195515_81s67vj4pjqm8_GVsSQL_PLAN.csv
sqldx_20130103_195515_81s67vj4pjqm8_GVsSQL_PLAN_MONITOR.csv
sqldx_20130103_195515_81s67vj4pjqm8_GVsSQL_PLAN_STATISTICS_ALL.csv
sqldx_20130103_195515_81s67vj4pjqm8_GVsSQL_REDIRECTION.csv
```

```
sqldx_20130103_195515_81s67vj4pjqm8_GVsSQL_SHARED_CURSOR.csv
sqldx_20130103_195515_81s67vj4pjqm8_GVsSQL_SHARED_MEMORY.csv
sqldx_20130103_195515_81s67vj4pjqm8_GVsSQL_WORKAREA.csv
```

- The FCSV directory, which contains:

```
sqldx_20130103_195515_13811832730830921192_force_DBA_HIST_ACTIVE_SESS_HISTORY.csv
sqldx_20130103_195515_13811832730830921192_force_DBA_HIST_SQLSTAT.csv
sqldx_20130103_195515_13811832730830921192_force_GVsACTIVE_SESSION_HISTORY.csv
sqldx_20130103_195515_13811832730830921192_force_GVsSQL.csv
sqldx_20130103_195515_13811832730830921192_force_GVsSQLAREA.csv
sqldx_20130103_195515_13811832730830921192_force_GVsSQLAREA_PLAN_HASH.csv
sqldx_20130103_195515_13811832730830921192_force_GVsSQLSTATS.csv
sqldx_20130103_195515_13811832730830921192_force_GVsSQLSTATS_PLAN_HASH.csv
sqldx_20130103_195515_13811832730830921192_force_GVsSQL_MONITOR.csv
```

- The FHTML directory, which contains:

```
sqldx_20130103_195515_13811832730830921192_force_csv.zip
sqldx_20130103_195515_13811832730830921192_force_DBA_HIST_ACTIVE_SESS_HISTORY.csv
sqldx_20130103_195515_13811832730830921192_force_DBA_HIST_SQLSTAT.csv
sqldx_20130103_195515_13811832730830921192_force_GVsACTIVE_SESSION_HISTORY.csv
sqldx_20130103_195515_13811832730830921192_force_GVsSQL.csv
sqldx_20130103_195515_13811832730830921192_force_GVsSQLAREA.csv
sqldx_20130103_195515_13811832730830921192_force_GVsSQLAREA_PLAN_HASH.csv
sqldx_20130103_195515_13811832730830921192_force_GVsSQLSTATS.csv
sqldx_20130103_195515_13811832730830921192_force_GVsSQLSTATS_PLAN_HASH.csv
sqldx_20130103_195515_13811832730830921192_force_GVsSQL_MONITOR.csv
```

- The GCSV directory, which contains:

```
sqldx_20130103_195515_global_csv.zip
sqldx_20130103_195515_global_DBA_HIST_SNAPSHOT.csv
sqldx_20130103_195515_global_GVsPARAMETER2.csv
GHTML directory which contains
sqldx_20130103_195515_global_DBA_HIST_SNAPSHOT.html
sqldx_20130103_195515_global_GVsPARAMETER2.html
sqldx_20130103_195515_global_html.zip
```

- The HTML directory, which contains:

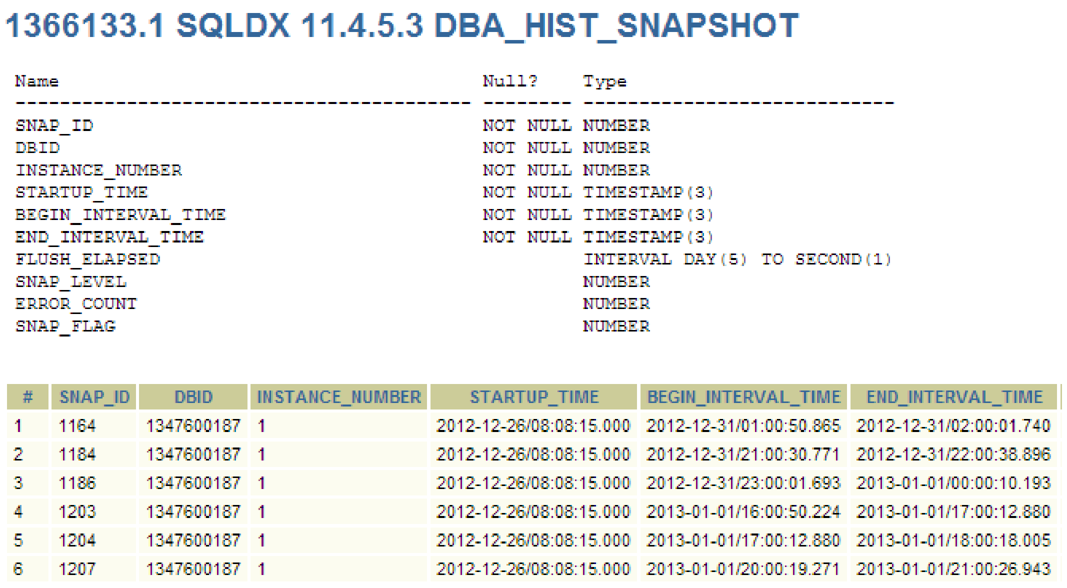
```
sqldx_20130103_195515_81s67vj4pjqm8_DBA_HIST_ACTIVE_SESS_HISTORY.html
sqldx_20130103_195515_81s67vj4pjqm8_DBA_HIST_SQLSTAT.html
sqldx_20130103_195515_81s67vj4pjqm8_DBA_HIST_SQLTEXT.html
sqldx_20130103_195515_81s67vj4pjqm8_DBA_HIST_SQL_PLAN.html
sqldx_20130103_195515_81s67vj4pjqm8_GVsACTIVE_SESSION_HISTORY.html
sqldx_20130103_195515_81s67vj4pjqm8_GVsSQL.html
sqldx_20130103_195515_81s67vj4pjqm8_GVsSQLAREA.html
sqldx_20130103_195515_81s67vj4pjqm8_GVsSQLAREA_PLAN_HASH.html
sqldx_20130103_195515_81s67vj4pjqm8_GVsSQLSTATS.html
sqldx_20130103_195515_81s67vj4pjqm8_GVsSQLSTATS_PLAN_HASH.html
```

```
sqldx_20130103_195515_81s67vj4pjqm8_GVsSQLTEXT.html
sqldx_20130103_195515_81s67vj4pjqm8_GVsSQLTEXT_WITH_NEWLINES.html
sqldx_20130103_195515_81s67vj4pjqm8_GVsSQL_MONITOR.html
sqldx_20130103_195515_81s67vj4pjqm8_GVsSQL_OPTIMIZER_ENV.html
sqldx_20130103_195515_81s67vj4pjqm8_GVsSQL_PLAN.html
sqldx_20130103_195515_81s67vj4pjqm8_GVsSQL_PLAN_MONITOR.html
sqldx_20130103_195515_81s67vj4pjqm8_GVsSQL_PLAN_STATISTICS_ALL.html
sqldx_20130103_195515_81s67vj4pjqm8_GVsSQL_REDIRECTION.html
sqldx_20130103_195515_81s67vj4pjqm8_GVsSQL_SHARED_CURSOR.html
sqldx_20130103_195515_81s67vj4pjqm8_GVsSQL_SHARED_MEMORY.html
sqldx_20130103_195515_81s67vj4pjqm8_GVsSQL_WORKAREA.html
sqldx_20130103_195515_81s67vj4pjqm8_html.zip
```

- The LOG directory, which contains:

```
sqldx_20130103_195515_81s67vj4pjqm8_driver.sql
sqldx_20130103_195515_81s67vj4pjqm8_log.zip
```

In total there are over 80 different files in HTML, CSV, and plain text format, too many to go into detail on each one. I'll just mention the DBA_HIST_SNAPSHOT HTML file, which shows the history of the SQL in question, when it started, and which AWR snapshot it can be found in. In Figure 14-19, I show part of this page.



1366133.1 SQLDX 11.4.5.3 DBA_HIST_SNAPSHOT

```
Name                                       Null?    Type
------------------------------------------ -------- ----------------------------
SNAP_ID                                    NOT NULL NUMBER
DBID                                       NOT NULL NUMBER
INSTANCE_NUMBER                            NOT NULL NUMBER
STARTUP_TIME                               NOT NULL TIMESTAMP(3)
BEGIN_INTERVAL_TIME                        NOT NULL TIMESTAMP(3)
END_INTERVAL_TIME                          NOT NULL TIMESTAMP(3)
FLUSH_ELAPSED                                       INTERVAL DAY(5) TO SECOND(1)
SNAP_LEVEL                                          NUMBER
ERROR_COUNT                                         NUMBER
SNAP_FLAG                                           NUMBER
```

#	SNAP_ID	DBID	INSTANCE_NUMBER	STARTUP_TIME	BEGIN_INTERVAL_TIME	END_INTERVAL_TIME
1	1164	1347600187	1	2012-12-26/08:08:15.000	2012-12-31/01:00:50.865	2012-12-31/02:00:01.740
2	1184	1347600187	1	2012-12-26/08:08:15.000	2012-12-31/21:00:30.771	2012-12-31/22:00:38.896
3	1186	1347600187	1	2012-12-26/08:08:15.000	2012-12-31/23:00:01.693	2013-01-01/00:00:10.193
4	1203	1347600187	1	2012-12-26/08:08:15.000	2013-01-01/16:00:50.224	2013-01-01/17:00:12.880
5	1204	1347600187	1	2012-12-26/08:08:15.000	2013-01-01/17:00:12.880	2013-01-01/18:00:18.005
6	1207	1347600187	1	2012-12-26/08:08:15.000	2013-01-01/20:00:19.271	2013-01-01/21:00:26.943

***Figure 14-19.** The HTML DBA_HIST_SNAPSHOT report from sqldx*

The sqlhcxec.sql Script

By now of course you're a SQLT pro and can probably take a good guess at what this routine does. This routine takes as parameter the license level and the name of a file containing the SQL you want to analyze. Remember this is all working without SQLT being installed on the database. The output files consist of two main files: a result file, showing the result of the query, and a zip file containing all the reports.

The last few lines of this report's output looks like this:

```
SQL> @sqlhcxec
Parameter 1:
Oracle Pack License (Tuning, Diagnostics or None) [T|D|N] (required)
Enter value for 1: T
PL/SQL procedure successfully completed.
Parameter 2:
SCRIPT name which contains SQL and its binds (required)
Enter value for 2: q3.sql
Values passed:
~~~~~~~~~~~~~
License: "T"
Script : "q3.sql"
In case of a disconnect review sqlhcxec_20130209_122156_error.log
SQL> PRO Ignore MOVE or MV error below
Ignore MOVE or MV error below
SQL> SET TERM OFF;
'mv' is not recognized as an internal or external command,
operable program or batch file.
SQL> WHENEVER SQLERROR EXIT SQL.SQLCODE;
SQL>
SQL> BEGIN
  2    IF '^^sql_id.' IS NULL THEN
  3      RAISE_APPLICATION_ERROR(-20200, 'SQL_ID "^^sql_id." not found in memory.');
  4    END IF;
  5  END;
  6  /
PL/SQL procedure successfully completed.
SQL>
SQL> WHENEVER SQLERROR CONTINUE;
SQL> SET ECHO ON TIMI ON;
SQL>
SQL> /**********************************************
SQL>  *
SQL>  * begin_common: from begin_common to end_common sqlhc.sql and sqlhcxec.sql are identical
SQL>  *
SQL> ***************************************************************************************/
SQL> SELECT 'BEGIN: '||TO_CHAR(SYSDATE, 'YYYY-MM-DD/HH24:MI:SS') FROM dual;
```

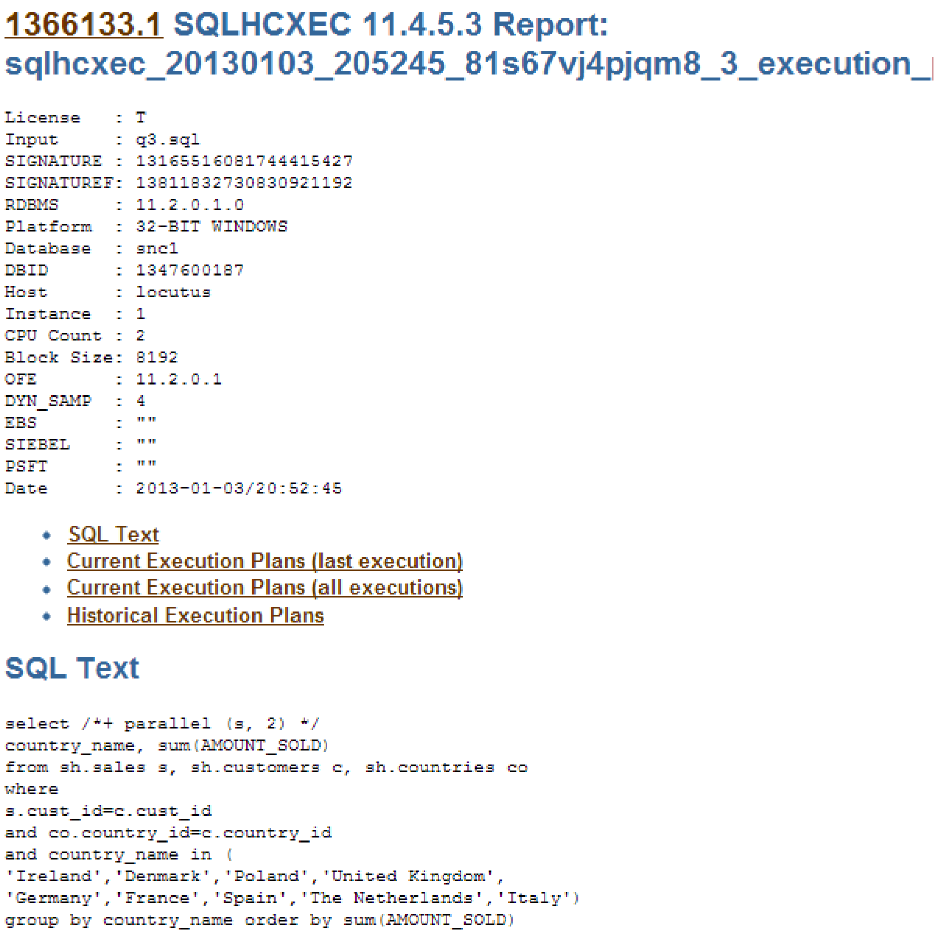
I've removed many lines for clarity, and the script finishes with these lines:

```
SQLHCXEC files have been created.
Archive:  sqlhcxec_20130103_205245_81s67vj4pjqm8.zip
```

```
  Length      Date    Time    Name
---------  ---------- -----   ----
    16784  01/03/2013 20:53   sqlhcxec_20130103_205245_81s67vj4pjqm8_1_health_check.html
   146959  01/03/2013 20:54   sqlhcxec_20130103_205245_81s67vj4pjqm8_2_diagnostics.html
    34947  01/03/2013 20:54   sqlhcxec_20130103_205245_81s67vj4pjqm8_3_execution_plans.html
    81269  01/03/2013 20:54   sqlhcxec_20130103_205245_81s67vj4pjqm8_4_sql_detail.html
   327092  01/03/2013 20:54   sqlhcxec_20130103_205245_81s67vj4pjqm8_6_10046_10053_trace_from_user_
                              script_exec.trc
    23968  01/03/2013 20:54   sqlhcxec_20130103_205245_81s67vj4pjqm8_9_log.zip
    15367  01/03/2013 20:54   sqlhcxec_20130103_205245_81s67vj4pjqm8_5_sql_monitor.zip
   247417  01/03/2013 20:54   sqlhcxec_20130103_205245_81s67vj4pjqm8_8_sqldx.zip
---------                     -------
   893803                     8 files
```

The zip file for sqlhcxec contains eight files:

- A health check HTML file that is identical to the `sqlhc.sql` output (mentioned above).
- A diagnostics HTML file, which is also the same as the `sqlhc.sql` output mentioned above as the Diagnostics and profiles page.
- An execution plan HTML file that shows the current execution plan and the historical execution plans (which `sqlhc.sql` did not). See Figure 14-20 below.



1366133.1 SQLHCXEC 11.4.5.3 Report:
sqlhcxec_20130103_205245_81s67vj4pjqm8_3_execution_

```
License   : T
Input     : q3.sql
SIGNATURE : 13165516081744415427
SIGNATUREF: 13811832730830921192
RDBMS     : 11.2.0.1.0
Platform  : 32-BIT WINDOWS
Database  : snc1
DBID      : 1347600187
Host      : locutus
Instance  : 1
CPU Count : 2
Block Size: 8192
OFE       : 11.2.0.1
DYN_SAMP  : 4
EBS       : ""
SIEBEL    : ""
PSFT      : ""
Date      : 2013-01-03/20:52:45
```

- SQL Text
- Current Execution Plans (last execution)
- Current Execution Plans (all executions)
- Historical Execution Plans

SQL Text

```
select /*+ parallel (s, 2) */
country_name, sum(AMOUNT_SOLD)
from sh.sales s, sh.customers c, sh.countries co
where
s.cust_id=c.cust_id
and co.country_id=c.country_id
and country_name in (
'Ireland','Denmark','Poland','United Kingdom',
'Germany','France','Spain','The Netherlands','Italy')
group by country_name order by sum(AMOUNT_SOLD)
```

Figure 14-20. *The sqlhcxec execution plans report, showing historical execution plans*

- A SQL detail HTML file just like the one produced by `sqlhc.sql`.
- SQL Monitor HTML files inside a zip file for each of the known historical executions of the SQL.
- A 10046 / 10053 HTML file for the execution of the SQL.
- A sqldx zip file, containing all of the same reports as the results of `sqldx.sql`.
- A zipped up log file.

Summary

The SQL health check utility, although it not directly linked to SQLT and does not rely on it, has many of the same elements in it as SQLT. It was created so that sites that do not want to or cannot install SQLT can get some benefit from using `sqlhc.sql` and its associated scripts instead. SQLHC is not as interlinked as SQLT nor does it have as many potential observations (SQLHC has about a hundred possible observations, while SQLT has 200 to 300 observations). SQLHC is sufficiently useful that it should be considered a minimum requirement for SQLHC to be run against a new production SQL to check against observations and expected execution plans. We are nearing the end of our journey with SQLT, and so in the next chapter I'll pull back and give you a bigger perspective on tuning.

CHAPTER 15

The Final Word

The key to successful tuning with SQLT is to use SQLT regularly and for real-life problems. If you've reached the final chapter, you should now consider yourself a card-carrying member of the SQLT supporters club. You've learned a lot about what SQLT can do, and along the way you've probably learned some things about the cost-based optimizer and the Oracle engine. Let me remind you of some of the features we came across on our journey:

- The effect of statistics on execution plans
- The effect of skewness on execution plans
- How the optimizer transforms SQL during parsing
- How profiles can help you temporarily freeze an execution plan
- How adaptive cursor sharing works
- How dynamic sampling works
- How cardinality feedback works
- How you can use SQLT with Data Guard
- How test cases can be built with SQLT to allow exploration of the execution plans
- How to use the brute force of XPLORE to look for unexpected effects on the CBO from upgrades and other changes.
- How we can use the COMPARE method to investigate two SQLs
- Last but not least we talked about the health check script, which is a good second choice if SQLT is not available.

In this chapter I'll try and give you a quick overview of a methodology I use to approach a tuning problem. Naturally all methodologies have exceptions, but it's better to have a default plan than having to determine a new one for every occasion. I'll also give my opinion as to why SQLT is the best tool available for tuning (apart from being free). Then I'll discuss some platform issues and assure you that SQLT and the examples we've covered do not just work on one platform. Finally I'll mention a few resources you should be aware of as you continue your greater journey into the world of tuning.

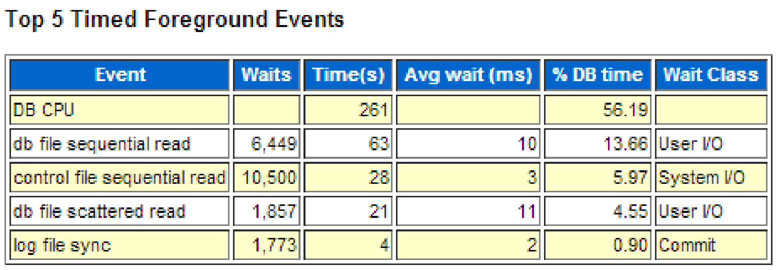
Tuning Methodology

Tuning methodology is not this book's main theme, but I feel I need to say something about it because SQLT can be a central element to a good strategy. The lack of a central methodology to attack SQL tuning problems has always, in my opinion, been the main problem for DBAs especially but also for developers who have to create efficient code. When you have a tuning problem, where do you start? Usually it all depends on what kind of problem you have. To help you here's my five-step method.

1. Get an AWR report for the problem time. If there is a "significant" problem in the "Top 5 waits" section of the report, deal with that first. If there is nothing obvious there then check the "SQL Report" section of the AWR report. If there is an SQL using more resources than other SQLs, then get an appropriate SQLT report. Always be led by the evidence presented in the AWR report and not by your own hunches or guesswork.
2. If it's an individual SQL problem, start with SQLT XTRACT or XEXCUTE (depending on whether you can reliably run the SQL) and then use the information to go from there. If you can't use SQLT, then use SQLHC.
3. Evaluate the information collected and scrutinize any information that looks out of the ordinary (this is where the constant practice helps, because you begin to recognize out-of-the-ordinary behavior on your system).
4. Investigate any anomalies and make sure you understand them. They may be benign. If the anomalies cannot be explained then try and assess if they could be the cause of your problems.
5. If you do end up investigating an individual SQL that has changed performance, remember there are many SQL tools that can be deployed to get more information. Build a test case and use COMPARE or use XPLORE in desperation (if you have the time).

This high-level methodology has a few key elements. The first is to recognize things that are out of the ordinary. To do this you must first understand what is normal, just like our alien visitor back in Chapter 2. The second element is the knowledge of how things work in the optimizer. This takes practice and some reading (hopefully this book helped).

It's important to recognize that SQLT is not the first step in this methodology. Although SQLT is useful for many tuning problems, the first step should be to assess the overall system performance, which can be best done from an AWR report against the appropriate database. If the problem is related to the operating system, you may find your solution there and need never look at a SQLT report or indeed any SQL. If your problem is with the database then the AWR report is a good starting point for memory requirements, unexpected waits (seen in the top five events). See Figure 15-1, which shows the section of the AWR report with the top five waits on the system. This report is not

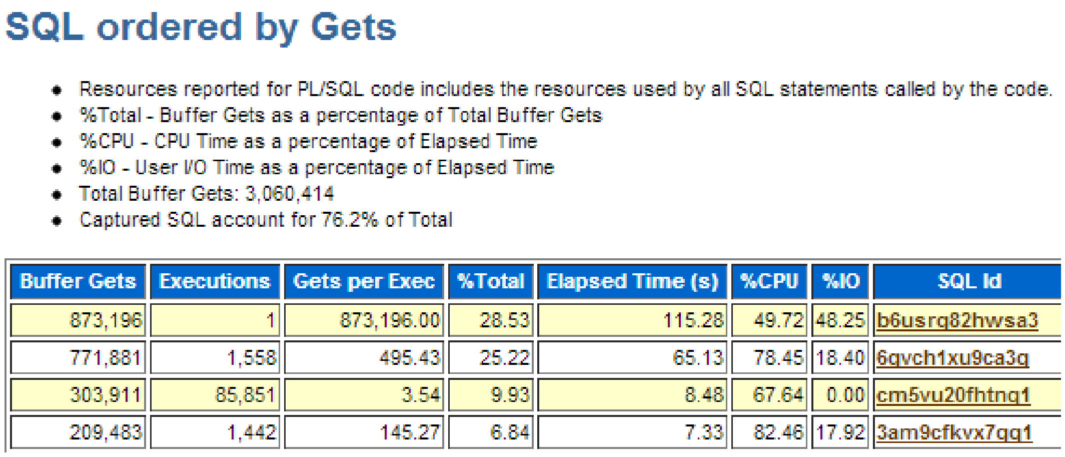

Top 5 Timed Foreground Events

Event	Waits	Time(s)	Avg wait (ms)	% DB time	Wait Class
DB CPU		261		56.19	
db file sequential read	6,449	63	10	13.66	User I/O
control file sequential read	10,500	28	3	5.97	System I/O
db file scattered read	1,857	21	11	4.55	User I/O
log file sync	1,773	4	2	0.90	Commit

Figure 15-1. *The top 5 waits in AWR*

very typical but at least shows no high percentages for unusual waits.

If your system is heavily loaded you should see waits in this section of the report that may require investigation. Depending on what these waits are, you may be led to the SQL section of the report, which then may suggest an investigation of a particular SQL or perhaps one or two. Then SQLT can be used to good effect (as long as the SQL is not some internal Oracle code). See below in Figure 15-2 for the top SQL sorted by buffer gets.



SQL ordered by Gets

- Resources reported for PL/SQL code includes the resources used by all SQL statements called by the code.
- %Total - Buffer Gets as a percentage of Total Buffer Gets
- %CPU - CPU Time as a percentage of Elapsed Time
- %IO - User I/O Time as a percentage of Elapsed Time
- Total Buffer Gets: 3,060,414
- Captured SQL account for 76.2% of Total

Buffer Gets	Executions	Gets per Exec	%Total	Elapsed Time (s)	%CPU	%IO	SQL Id
873,196	1	873,196.00	28.53	115.28	49.72	48.25	b6usrq82hwsa3
771,881	1,558	495.43	25.22	65.13	78.45	18.40	6qvch1xu9ca3q
303,911	85,851	3.54	9.93	8.48	67.64	0.00	cm5vu20fhtnq1
209,483	1,442	145.27	6.84	7.33	82.46	17.92	3am9cfkvx7gq1

Figure 15-2. *The SQL ordered by gets, part of the AWR report*

In the atypical report above we see that one SQL is taking 28 percent of all the gets, that's pretty unusual on a production system, and you should probably investigate this (unless you know what it is and why it's using that much of the system). Sometimes if many SQL have regressed you can use SQLT to look at one SQL in detail: this is to assess the problem afflicting that SQL in the hope that whatever has affected that SQL will be the same solution for all the SQLs.

Why SQLT Is, Hands Down, the Best Tuning Utility

Now that we've put SQLT in its proper place in the strategy of tuning a system, we should acknowledge that for tuning individual SQLs, SQLT is hands down the best starting tool for the job. It's true that there are many tuning utilities out there: some of them on the Oracle MOS site and some of them paid-for products. For example, TRCANLZR is available as a stand-alone utility, TKPROF comes with Oracle as a utility, 10046 and 10053 tracing can be collected from a standard Oracle installation. All of them, however, are focusing on one particular aspect or are going into great detail on a problem without giving a bigger picture. SQLT is the main tool for focusing on one SQL and gives you the big picture in an easily digestible form. SQLT does require you to do some work of course, and analysis of the reports and building the overall picture of the SQL takes some time and some expertise. Sometimes you will need more information than a particular run of SQLT has supplied (for example, the shared pool was flushed, or the system was rebooted, so much information was lost). Depending on what you learn by looking at the SQL history and statistics, you could look at the COMPARE method, for example (if you have a good run of the SQL). If you need to experiment you can use the test case on a stand-alone system to try out a few things and if you know changes have come about because of optimizer version changes, you can use XPLORE. SQLT is like a helpful assistant at a shopping mall:

"I think you need the statistics department today, sir. Try floor two, just two doors along."

A Word About Platforms

A platform, in IT jargon, is the hardware and operating system on which other applications reside. In the case of the Oracle product itself, it can be loaded onto many different platforms: Unix, Linux, OpenVMS, Solaris, Windows, and others. As far as SQL commands are concerned, the platform makes very little difference. A "select" is still a "select." SQLT can be used on Unix (including Linux) and Windows. Other platforms will not allow the full functionality of SQLT. SQLHC, the health check script will run on as many platforms as Oracle runs on, as it relies only on Oracle.

In this book almost all my examples are based on a Windows platform, but on a day-to-day basis I use SQLT on both Linux and Windows, and I have noticed no difference in behavior. I also regularly use SQLT on an Exadata platform and again everything works as you would expect. Every example shown in the book for Windows can just as easily work on Linux and vice versa. The interaction of SQLT with the operating system is usually only during the end phases of reports where files are being collected and being zipped up. In these phases commands such as `cp` (`copy` on Windows) and `ls` (`DIR` on windows) will generate messages indicating that those commands do not exist on the current platform. This is not a serious error, and the alternative command will have been issued for the appropriate platform.

Other Resources

This may be the final word in this book, but there are many other good resources available on SQLT. There are many Metalink notes: I haven't counted them but there are many more than the ones listed here. Luckily many of these notes are linked to each other so these are good starting points:

- 215187.1 – The main SQLT note where you can download SQLT
- 1454160.1 – FAQ about SQLT
- 1465741.1 – How to create test cases using SQLT with test data.
- 1455583.1 – Gives you access to a SQLHC video
- 1366133.1 – SQL Health Check Script

There are even webcasts (look at Note 740964.1), which leads to the Oracle Webcast Program. Select "Oracle Database" and then browse the archived recordings. For SQLT here are some interesting topics for 2012:

- "Resolve – Troubleshooting Performance Issues Using SQLT & SQLHealthCheck"
- "Understanding SQLTXPLAIN Main Report by Navigating Through Some Samples"
- "What is SQLTXPLAIN tool and how do I use it?"
- "How to create in 5 minutes a SQL Tuning Test Case using SQLTXPLAIN"

Blogs on SQL and SQLT can be found at:

- `www.carlos-sierra.net`
- `www.SteliosCharalambides.com`

Summary

The SQLTXPLAIN utility, one of the most useful free utilities available to Oracle customers, has evolved over many versions and many generations of the Oracle product. The latest version 11.4.5.6 (March 5, 2013) has come a long way, and there are many new features. It is still evolving and no doubt will continue to evolve, but at least for 2013 there will be a pseudo-freeze so this is a good time to get up to speed. SQLT is a deceptively complex utility with many features hidden away in innocuous-looking scripts. I hope I've cast some light on those features, so that maybe you'll be inspired to look at other scripts I have not mentioned. Get the latest SQLT version from the MOS site: `http://support.oracle.com`.

I hope you've enjoyed the journey, and I sincerely hope you use SQLT to learn more about your system and the SQL running on it. Remember, the key is to use SQLT regularly and investigate anything you don't understand using reliable sources; soon, you will be tuning with the best of them.

APPENDIX A

Installing SQLTXPLAIN

You may ask, why show the installation log for a utility that installs in five minutes and only has at most five parameters as inputs. The plain fact of the matter is that despite the simplicity of the inputs and that most installs can be quick and easy, there are situations where the installation can fail: either because the inputs were wrong or because the installation steps were not done from suitably privileged accounts. You occasionally get questions about an installation that may have worked, but the user is not clear. This is another reason to show what a "normal" installation looks like. I also look at alternative ways in which SQLT can be installed, including a "silent" mode and a remote install mode. There are also ways to change the setup of SQLT after it has installed, and I mention some of the options available there. Finally I also mention how to de-install SQLT, in case you want to install a later version, for example. By showing these options and describing the installation, I hope to convince you that the installation is simple and robust and should be considered an asset to any system rather than a liability.

A Standard SQLT Installation

As an assistance to anyone installing SQLT I have supplied a partial install log of the SQLT utility. I have highlighted and documented those areas of the installation that are of interest and note. New comers to SQLT may find this useful, but regular users will find the section on other ways to install SQLT more interesting. In the example SQLT installation below I have bolded my responses to make it clear where I am entering data, and I have removed blank lines for brevity. In this example the SQLT zip file has been downloaded to a local directory and unzipped. Inside the zip file we find the sqlt directory and the install directory (as we saw in Chapter 1). Now we connect as SYS into SQL*Plus and start the main installation script `sqcreate.sql`

```
SQL> @sqcreate
PL/SQL procedure successfully completed.
PL/SQL procedure successfully completed.
RDBMS_RELEASE
-------------
         11.2
RDBMS_VERSION
--------------------------------
11.2.0.1.0
no rows selected
```

So far the installation has initialized and discovered the environment it is installing itself into. The next steps are to gather information so that the installation can be properly targeted. For most circumstances installing into the local database is the easiest and simplest option. You only need a tablespace in which SQLT will store its information, packages, functions, and the data repository. The size of this is generally very small (in my small installation this was about 2 Mbytes). If you specify a remote connection here the data will be stored elsewhere, such as on a remote database.

```
Specify optional Connect Identifier (as per Oracle Net)
Include "@" symbol, ie. @PROD
If not applicable, enter nothing and hit the "Enter" key.
This connect identifier is only used while exporting SQLT
repository everytime you execute one of the main methods.
Optional Connect Identifier (ie: @PROD):
```

The optional Connect Identifier is not often used, as you are normally installing SQLT locally. We cover the case of a remote install later, in the section "A Remote SQLT Installation." In this case, however, we just press Return.

```
PL/SQL procedure successfully completed.
Define SQLTXPLAIN password (hidden and case sensitive).
Password for user SQLTXPLAIN:oracle
Re-enter password:oracle
PL/SQL procedure successfully completed.

... please wait
TABLESPACE                     FREE_SPACE_MB
------------------------------ -------------
USERS                                    246
Specify PERMANENT tablespace to be used by SQLTXPLAIN.
Tablespace name is case sensitive.
Default tablespace [UNKNOWN]:USERS
PL/SQL procedure successfully completed.
... please wait
TABLESPACE
------------------------------
TEMP
Specify TEMPORARY tablespace to be used by SQLTXPLAIN.
Tablespace name is case sensitive.
Temporary tablespace [UNKNOWN]:TEMP
PL/SQL procedure successfully completed.
```

The next section is the part of the installation that most often causes confusion. The "main application user of SQLT" is the schema name of the user that actually executes the SQL to be analyzed. This is not SQLTXPLAIN. Throughout most of this book my example schema is called STELIOS, so I enter **STELIOS** here. If for some reason you want to change this or add another schema you only need to grant the SQLT_USER_ROLE role to the user in question. This would be done with `grant SQLT_USER_ROLE to <username>;`. Normally this username should be connectable from the SQL*Plus prompt so that you can run the SQLT methods.

■ **Note** In some systems SQL is executed via a remote connection from another system (for example through JDBC connections), and the account used for these connections cannot be locally connected through SQL*Plus due to security restrictions. In these cases the next best next option is to create a schema or use another schema that can execute the SQL and that has access to the same data and objects as the target schema. Be wary in this case that you are not creating a different environment that does not show your problem.

```
The main application user of SQLT is the schema
owner that issued the SQL to be analyzed.
For example, on an EBS application you would
enter APPS.
You will not be asked to enter its password.
To add more SQLT users after this installation
is completed simply grant them the SQLT_USER_ROLE
role.
Main application user of SQLT:STELIOS
PL/SQL procedure successfully completed.
SQLT can make extensive use of licensed features
provided by the Oracle Diagnostic and the Oracle
Tuning Packs, including SQL Tuning Advisor (STA),
SQL Monitoring and Automatic Workload Repository
(AWR).
To enable or disable access to these features
from the SQLT tool enter one of the following
values when asked:
```

The following is another section of the installation where there is much confusion because quite often the installer of SQLT is not aware of the license level for the database in question. There is no short cut to this, unfortunately.

```
"T" if you have license for Diagnostic and Tuning
"D" if you have license only for Oracle Diagnostic
"N" if you do not have these two licenses
Oracle Pack license [T]:
PL/SQL procedure successfully completed.
PL/SQL procedure successfully completed.
PL/SQL procedure successfully completed.
PL/SQL procedure successfully completed.
PL/SQL procedure successfully completed.
PL/SQL procedure successfully completed.
PL/SQL procedure successfully completed.
TADOBJ completed.
PL/SQL procedure successfully completed.
PL/SQL procedure successfully completed.
RDBMS_RELEASE
-------------
         11.2
RDBMS_VERSION
--------------------------------
11.2.0.1.0
no rows selected
```

The main installation of SQLT starts here; and as it mentions in the prompts, there are some errors generated during a fresh installation of SQLT. These are normal Oracle errors generated during attempts to drop objects in the SQLT schemas that do not yet exist.

```
SQDOLD completed. Ignore errors from this script
SQCUSR completed. Some errors are expected.
Procedure created.
```

```
No errors.
TAUTLTEST completed.
SQUTLTEST completed.
no rows selected
TACOBJ completed.
```

No more errors should be seen from the installation until we get to the section on privileges being revoked. These errors are normal and should be ignored.

```
... creating package specs for SQLT$S
No errors.
... creating package specs for SQLT$T
No errors.
... creating views
PL/SQL procedure successfully completed.
PL/SQL procedure successfully completed.
RDBMS_RELEASE
-------------
         11.2
Synonym created.
REVOKE SELECT, UPDATE ON sys.optstat_hist_control$ FROM SQLTXPLAIN
*
ERROR at line 1:
ORA-01927: cannot REVOKE privileges you did not grant
Grant succeeded.
Synonym created.
View created.
REVOKE SELECT ON sys.sqlt$_dba_tab_stats_vers_v FROM SQLTXPLAIN
*
ERROR at line 1:
ORA-01927: cannot REVOKE privileges you did not grant
Grant succeeded.
Synonym created.
```

There are several more error messages like this until we finally get to the end of the installation.

```
VALID   PACKAGE BODY 11.4.5.0 TRCA$R
VALID   PACKAGE BODY 11.4.5.0 TRCA$T
VALID   PACKAGE BODY 11.4.5.0 TRCA$X
Deleting CBO statistics for SQLTXPLAIN objects ...
13:42:58 sqlt$a: -> delete_sqltxplain_stats
13:43:01 sqlt$a: <- delete_sqltxplain_stats
PL/SQL procedure successfully completed.
SQCPKG completed.
TAUTLTEST completed.
SQUTLTEST completed.
SQLT users must be granted SQLT_USER_ROLE before using this tool.
SQCREATE completed. Installation completed successfully.
```

This last message `SQCREATE completed` is a good sign that all has gone well. Although the amount of work that SQLT does for you during the installation may seem daunting, the installation itself usually takes no more than five minutes. Even if you get something wrong during the installation it takes a very short time to correct the error by simply removing SQLT and re-installing it with the correct settings or if you prefer you can change the setting after the installation. For example if you get the license level wrong during installation you can correct it with one of the following four routines:

```
disable_tuning_pack_access;

enable_tuning_pack_access;

disable_diagnostic_pack_access;

enable_diagnostic_pack_access;
```

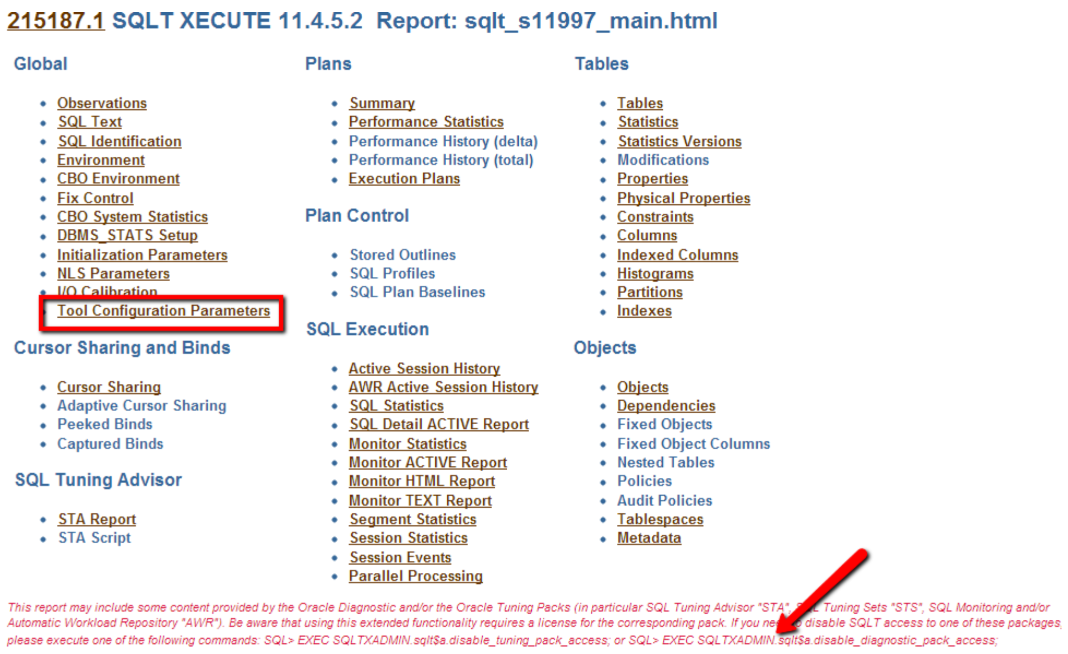
I've given more details on the use of these routines in the section "How to Change the Licensing Level after a SQLT Installation" below.

How to Change the Licensing Level after a SQLT Installation

The default settings for a SQLT installation work for most situations, yet sometimes you may want to change them. For example, after installing SQLT you buy a diagnostic pack or tuning pack license. The tuning pack and diagnostic pack can be bought separately or as a bundle from Oracle. If you have neither of these packs you should use `disable_tuning_pack_access` and `disable_diagnostic_pack_access` if you installed with option "T" in a normal installation. Here are the example steps

```
SQL> exec sqltxadmin.sqlt$a.disable_tuning_pack_access;
PL/SQL procedure successfully completed.
SQL> exec sqltxadmin.sqlt$a.disable_diagnostic_pack_access;
PL/SQL procedure successfully completed.
```

No parameters are needed for these routines, just execute these from the SQLT account on the databases where SQLT is installed. You could change the license level by just re-installing SQLT, but this would not be a good idea if you had many records in the SQLT repository that you wanted to keep. The list of configurable settings can be found under the Global section of the main SQLTXECUTE or SQLTXTRACT report (as covered in Chapter 13). See Figure A-1.



215187.1 SQLT XECUTE 11.4.5.2 Report: sqlt_s11997_main.html

Global

- Observations
- SQL Text
- SQL Identification
- Environment
- CBO Environment
- Fix Control
- CBO System Statistics
- DBMS_STATS Setup
- Initialization Parameters
- NLS Parameters
- I/O Calibration
- Tool Configuration Parameters

Cursor Sharing and Binds

- Cursor Sharing
- Adaptive Cursor Sharing
- Peeked Binds
- Captured Binds

SQL Tuning Advisor

- STA Report
- STA Script

Plans

- Summary
- Performance Statistics
- Performance History (delta)
- Performance History (total)
- Execution Plans

Plan Control

- Stored Outlines
- SQL Profiles
- SQL Plan Baselines

SQL Execution

- Active Session History
- AWR Active Session History
- SQL Statistics
- SQL Detail ACTIVE Report
- Monitor Statistics
- Monitor ACTIVE Report
- Monitor HTML Report
- Monitor TEXT Report
- Segment Statistics
- Session Statistics
- Session Events
- Parallel Processing

Tables

- Tables
- Statistics
- Statistics Versions
- Modifications
- Properties
- Physical Properties
- Constraints
- Columns
- Indexed Columns
- Histograms
- Partitions
- Indexes

Objects

- Objects
- Dependencies
- Fixed Objects
- Fixed Object Columns
- Nested Tables
- Policies
- Audit Policies
- Tablespaces
- Metadata

This report may include some content provided by the Oracle Diagnostic and/or the Oracle Tuning Packs (in particular SQL Tuning Advisor "STA", SQL Tuning Sets "STS", SQL Monitoring and/or Automatic Workload Repository "AWR"). Be aware that using this extended functionality requires a license for the corresponding pack. If you need to disable SQLT access to one of these packages, please execute one of the following commands: SQL> EXEC SQLTXADMIN.sqlt$a.disable_tuning_pack_access; or SQL> EXEC SQLTXADMIN.sqlt$a.disable_diagnostic_pack_access;

Figure A-1. *The top of a SQLTXECUTE report highlighting the configuration section*

The license levels are more important and there are special routines for these. If we installed the product with the wrong license level then we can change this with the `sqltxadmin.sqlt$a` packages, which has the previously mentioned four procedures related to the licensing levels:

```
disable_tuning_pack_access;
```

```
enable_tuning_pack_access;
```

```
disable_diagnostic_pack_access;
```

```
enable_diagnostic_pack_access;
```

These four procedures disable and enable the tuning pack and diagnostic pack functionality in SQLT. The arrow in Figure A-1 points to the prompt at the top of the SQLT main report that shows the example code. Below I've disabled and enabled the tuning pack and then disabled and enabled the diagnostic pack to end up with the same license level I started with.

```
SQL> SQL> exec sqltxadmin.sqlt$a.disable_tuning_pack_access;
PL/SQL procedure successfully completed.
SQL> exec sqltxadmin.sqlt$a.enable_tuning_pack_access;
PL/SQL procedure successfully completed.
SQL> exec sqltxadmin.sqlt$a.disable_diagnostic_pack_access;
PL/SQL procedure successfully completed.
SQL> exec sqltxadmin.sqlt$a.enable_diagnostic_pack_access;
PL/SQL procedure successfully completed.
```

Please note that in earlier versions of SQLT (before 11.4.4.6 and older) the routines to change the license level are in the SQLTXPLAIN schema, and so the command to change the license level is

```
SQL> SQL> exec sqlt$a.disable_tuning_pack_access;
PL/SQL procedure successfully completed.
SQL> exec sqlt$a.enable_tuning_pack_access;
PL/SQL procedure successfully completed.
SQL> exec sqlt$a.disable_diagnostic_pack_access;
PL/SQL procedure successfully completed.
SQL> exec sqlt$a.enable_diagnostic_pack_access;
PL/SQL procedure successfully completed.
```

A Remote SQLT Installation

During the installation you will see the optional connect identifier prompt. Normally this is ignored as you are installing locally. If, however, you want to install SQLT in remote mode you can specify a remote link. In these examples the local system is where you are connected to SQL Plus and the remote system is where the SQLT repository can be found.

```
Specify optional Connect Identifier (as per Oracle Net)
Include "@" symbol, ie. @PROD
If not applicable, enter nothing and hit the "Enter" key.
This connect identifier is only used while exporting SQLT
repository everytime you execute one of the main methods.
Optional Connect Identifier (ie: @PROD):@REMOTE
```

This appends **`@REMOTE`** to all SQL operations so that if you run `@sqltxtract` on the local database it will reach over to the remote database to store SQLT information.

In a remote installation the sequence of steps to run a report are slightly different than running everything locally.

1. Install SQLT on the remote node from the local node.
2. Run the SQL on the remote node
3. Run the SQLTXTRACT or SQLTXECUTE report on the local node but that runs on the remote node.
4. The reports are produced on the local node, but the repository data is stored on the remote node.

Other Ways to Install SQLT

If you want to deploy SQLT to many systems (and who wouldn't?) you might want to do a non-interactive installation. In this case you can populate a number of variables and then run `sqcsilent.sql`. An example variable definition file is provided in the installation directory of SQLT called `sqdefparams.sql`.

This is what it contains. These are all the values we've supplied in the interactive installation.

```
DEF connect_identifier      = '';
DEF enter_tool_password     = 'sqltxplain';
DEF re_enter_password       = 'sqltxplain';
DEF default_tablespace      = 'USERS';
DEF temporary_tablespace    = 'TEMP';
DEF main_application_schema = '';
DEF pack_license            = 'T';
```

These variables should be changed to suit your environment: for example, the password should be changed, and the tablespace and main application user could be different. When you have your own values in the file you can run `sqcsilent.sql`, which will then execute in "silent" mode with no prompts for parameters.

```
SQL> @sqcsilent.sql
```

You can also run the installation with all the parameters on the line as in the example below

```
SQL> @sqcsilent2.sql '' sqltxplain USERS TEMP '' T
```

This will also execute a normal installation with no prompts for information. This could be a quick way to do a standard install on a number of different systems.

How to Remove SQLT

If for some reason you wish to de-install SQLT (perhaps a new version has become available), you can use the routine in the `/install` directory called `sqdrop.sql`. There are no parameters to this routine

```
SQL> @sqdrop
PL/SQL procedure successfully completed.
PL/SQL procedure successfully completed.
RDBMS_RELEASE
-------------
         11.2
RDBMS_VERSION
--------------------------------
11.2.0.1.0
no rows selected
... uninstalling SQLT, please wait
TADOBJ completed.
PL/SQL procedure successfully completed.
PL/SQL procedure successfully completed.
RDBMS_RELEASE
-------------
         11.2
RDBMS_VERSION
--------------------------------
11.2.0.1.0
no rows selected
SQDOLD completed. Ignore errors from this script
PL/SQL procedure successfully completed.
PL/SQL procedure successfully completed.
RDBMS_RELEASE
-------------
         11.2
RDBMS_VERSION
--------------------------------
```

```
11.2.0.1.0
no rows selected
SQDOBJ completed. Ignore errors from this script
PL/SQL procedure successfully completed.
PL/SQL procedure successfully completed.
RDBMS_RELEASE
-------------
         11.2
RDBMS_VERSION
--------------------------------
11.2.0.1.0
no rows selected
SQL>
SQL> DECLARE
  2    my_count INTEGER;
  3
  4  BEGIN
  5    SELECT COUNT(*)
  6      INTO my_count
  7      FROM sys.dba_users
  8     WHERE username = 'TRCADMIN';
  9
 10    IF my_count = 0 THEN
 11      BEGIN
 12        EXECUTE IMMEDIATE 'DROP PROCEDURE sys.sqlt$_trca$_dir_set';
 13      EXCEPTION
 14        WHEN OTHERS THEN
 15          DBMS_OUTPUT.PUT_LINE('Cannot drop procedure sys.sqlt$_trca$_dir_set. '||SQLERRM);
 16      END;
 17
 18      FOR i IN (SELECT directory_name
 19                  FROM sys.dba_directories
WHERE directory_name IN (
'SQLT$UDUMP', 'SQLT$BDUMP', 'SQLT$STAGE', 'TRCA$INPUT1', 'TRCA$INPUT2', 'TRCA$STAGE'))
 21      LOOP
 22        BEGIN
 23          EXECUTE IMMEDIATE 'DROP DIRECTORY '||i.directory_name;
 24          DBMS_OUTPUT.PUT_LINE('Dropped directory '||i.directory_name||'.');
 25        EXCEPTION
 26          WHEN OTHERS THEN
 27            DBMS_OUTPUT.PUT_LINE('Cannot drop directory '||i.directory_name||'. '||SQLERRM);
 28        END;
 29      END LOOP;
 30    END IF;
 31  END;
 32  /
Dropped directory TRCA$INPUT2.
Dropped directory TRCA$INPUT1.
Dropped directory SQLT$BDUMP.
Dropped directory SQLT$UDUMP.
Dropped directory TRCA$STAGE.
Dropped directory SQLT$STAGE.
PL/SQL procedure successfully completed.
SQL>
SQL> WHENEVER SQLERROR CONTINUE;
SQL>
SQL> PAU About to DROP users &&tool_repository_schema. and &&tool_administer_schema.. Press RETURN
to continue.
```

At this point you are prompted to confirm that you want to do a `drop user SQLTXADMIN cascade;` and a `drop user SQLTXPLAIN cascade;`:

```
About to DROP users SQLTXPLAIN and SQLTXADMIN. Press RETURN to continue.
SQL>
SQL> DROP USER &&tool_administer_schema. CASCADE;
old   1: DROP USER &&tool_administer_schema. CASCADE
new   1: DROP USER SQLTXADMIN CASCADE
User dropped.
SQL> DROP USER &&tool_repository_schema. CASCADE;
old   1: DROP USER &&tool_repository_schema. CASCADE
new   1: DROP USER SQLTXPLAIN CASCADE
User dropped.
SQL> DROP ROLE &&role_name.;
old   1: DROP ROLE &&role_name.
new   1: DROP ROLE SQLT_USER_ROLE
Role dropped.
SQL>
SQL> SET ECHO OFF;
SQDUSR completed.
SQDROP completed.
```

And that's it. Now the schemas are dropped along with all related objects.

APPENDIX B

The CBO Parameters (11.2.0.1)

SQLT has within it the entire list of parameters that affect your query performance. It uses this list for the XPLORE method. Some parameters are hidden (only sporadically documented in various places) and some are non-hidden (documented in Oracle's online documentation: database reference manual `http://docs.oracle.com/cd/B28359_01/server.111/b28320/initparams.htm#i102439`). You can get this list by issuing the following query from SQLTXPLAIN:

```
SQL> select name, description from sqlt$_v$parameter_cbo order by name;
```

The parameter list included in the "Full List of Parameters" section later in this chapter is from a database I used previously, which happened to be 11.2.0.1. I could have put a different listing here from previous versions or later versions, but this list is not meant to be exhaustive or definitive. It is here to act as a partial reference and to point out that you can get the latest set from SQLT queries. This list is however a good starting point and many of these parameters are still in use in 12c. You should not, however, consider this a list of parameters that you think might improve your SQL: the default parameter settings are there for a reason: they give the best overall results. But if your XPLORE report highlights some parameter that might make a difference, this list can give you a clue, along with the description as to what is happening in your query. Remember that just knowing the name of the hidden parameter is useful, as it gives you the chance to search My Oracle Support for even more clues.

Dealing with hidden parameters (they are hidden for a reason) can be a tricky business. The purpose of the list in this appendix is to inform, not to give a template for experimentation. I'll give some guidelines, caveats, and more detailed descriptions of certain important hidden parameters, but the general rule is this: talk to support. The names and effects of these parameters can change with any release. Oracle support will have the latest set of parameters and their effects.

General Approach to Dealing with Hidden Parameters

In the general scheme of things, hidden parameters are needed only rarely or under special environments. For example Exadata installations are recommended to use a small number of hidden parameters. Apart from these special circumstances you will sometimes be requested to set some hidden parameters by Oracle support. Hidden parameters when suggested by support will be attempts to confirm some bug that has affected performance (wrong execution plan) or has resulted in wrong results or has even caused errors to be generated (ORA-00600's or ORA-07445's). The setting of these parameters usually follows these steps:

1. A specific error occurs (wrong result, bad plan, ORA-00600, ORA-07445)
2. The characteristics of the wrong result, bad plan steps or trace file of the ORA-00600 or trace file of the ORA-07445 suggest a course of action.

3. Often the suggested bug will have a workaround which consists of setting some parameter, sometimes a hidden parameter, to either mitigate the problem (avoid the error), change the execution plan back to the correct one or to ensure the retrieval of correct data.
4. If the suggested changes have the desired effect and the bug is confirmed, using this technique and other techniques, you then have the options of setting the hidden parameter or parameters at the system level or at the session level if possible, as a short term workaround or tolerating the error while a bug fix is produced. This will depend on many factors including your special circumstances and the effect of the parameters on your system generally.

This is why I specify with all of these parameters that you should be working with Oracle support. Changing hidden parameters without knowing the full implications can be disastrous (corrupted database springs to mind). The following descriptions and list give you the understanding to allow you to work more knowledgeably with Oracle support. If you have a database that is expendable, then you can do some experimentation with these parameters; but they should never be applied to production, development, QA, or testing databases without checking with Oracle support. I think that's enough of a warning.

More Detailed Descriptions of Some Hidden Parameters

For general information I have listed below some of the hidden parameters that may be used to alleviate problems under certain circumstances. None of this information should be used without the assistance of Oracle support. There are sometimes unforeseen effects of setting these parameters. If Oracle support suggests some of these parameters for exploratory investigation you will usually be asked to do this at the session level to determine the effect and also to determine if you are being affected by a particular bug or to assess the efficacy of a bug fix. The defaults are as of 11.2, and these may change in future versions of Oracle, or indeed may be removed completely.

_and_pruning_enabled

With composite partitioned tables you may be affected by some bugs which if they involve "AND" partition pruning can be fixed by setting this parameter to FALSE. The default is TRUE.

_bloom_filters_enabled

Bloom filters are a memory efficient way of determining the membership of a set for a particular value. This can be very effective with Oracle queries and large data sets. If your execution plan shows a bloom filter (BF) in the plan and you get wrong results or ORA-00600's, you may want to open a Service Request with Oracle support and discuss setting this parameter to FALSE. The default is TRUE.

_complex_view_merging

Under some circumstances the process of merging complex views can cause problems. By default this parameter is set to TRUE (as of 8i of Oracle) to allow the view query transformation to take place. If you have wrong results or ORA-00600s you may want to consult with Oracle support to check if setting this to FALSE might help. The default is TRUE.

_disable_function_based_index

A function-based index contains a function (obviously). An example would be

```
sql> create index fbi_1 on t (upper(col1));
```

Here the function is `upper`. In some rare cases you may get wrong results, which can be temporarily fixed (until you get a bug fix applied) by setting this value to the non-default value of FALSE.

_hash_join_enabled

There are a number of bugs that can be worked around by setting this hidden parameter to FALSE. The default value is TRUE. Just because you see a hash join in a statement and your statement crashes doesn't mean setting this parameter to false is the solution. Open a service request and give support the information. There may be other factors at work. Setting this value to FALSE will disable hash joins.

_optimizer_extended_cursor_sharing_rel

We already discussed disabling adaptive cursor sharing in chapter 7. Where we disabled ACS with

```
SQL> alter system set "_optimizer_extended_cursor_sharing_rel"=NONE scope=both;
SQL> alter system set "_optimizer_extended_cursor_sharing"=none scope=both;
```

Both parameters are needed to disable ACS. The default values are SIMPLE for _optimizer_extended_cursor_sharing_rel and UDO for _optimizer_extended_cursor_sharing.

_optimizer_cartesian_enabled

Disable the Cartesian join if set to FALSE. This may be a useful way of debugging a failing statement, in conjunction with Oracle supports help. The default value is TRUE.

_optimizer_cost_model

This changed the basis for costing of activity by the CBO. It can be set to `cpu`, `io` or `choose`. The name is a little misleading, by setting `cpu` we do not optimize for reducing CPU usage, we optimize for reducing cost as before but are now taking CPU into account for I/O operations. The value `choose` allows the optimizer to make a choice based on statistics available in `sys.aux_stats$`. The default value is `choose`.

_optimizer_ignore_hints

This hidden parameter allows the optimizer to ignore embedded hints. The default value is FALSE. You might want to try this if you felt the hints were not producing the best plan and wanted to disable these at the session level with

```
SQL> alter session set "_optimizer_ignore_hints"=TRUE;
```

_optimizer_max_permutations

If you feel somehow that the optimizer is not working hard enough, there is always the option to set `_optimizer_max_permutations` to something other than the default value of 2,000. The parameter controls the number of different permutations that the optimizer will try per query block when joining a number of tables. There may be circumstances where this value can be as high as 80,000. The value is overridden by `_optimizer_search_limit`, where `_optimizer_search_limit` is the number factorial of join permutations.

_optimizer_use_feedback

Controls the cardinality feedback feature as discussed in chapter 8. The default is TRUE. Set to FALSE to disable this feature.

_optimizer_search_limit

The default value for this parameter is 5. This is the factorial number of maximum Cartesian joins that will be considered. 5! (read as "five factorial") is equivalent to 5x4x3x2x1 which equals 120.

Full List of Parameters

Why include a full list of parameters that we should not as DBAs change unless directed to by Oracle support? There are two possible answers.

- Oracle support may suggest some parameter for some investigation and it is useful to be able to have at least a brief description of what the parameter does. You should ask Oracle support for this description before applying to any database anyway. This list is your backup for this information.
- Read and research each of these parameters carefully or try things out on a disposable database to give you insight in to how the optimizer works and what it is doing for us. This greater understanding becomes a story that cements clear knowledge about some aspect of the optimizer. An example of this is `_optimizer_max_permutations`. Before I came across the parameter, I happily assumed that the optimizer tried *all* permutations of joins. A moment's thought would have disabused me of that opinion. But now I know there is a limit to the number of join choices and how to control it.

The following list of CBO parameters (version 11.2.0.1) is a sample output of the SQLT query discussed at the beginning of this appendix:

```
SQL> select name, description from sqlt$_v$parameter_cbo order by name;

NAME                                     DESCRIPTION
---------------------------------------- -------------------------------------------------------
_add_stale_mv_to_dependency_list         Add stale mv to dependency list
_aggregation_optimization_settings       Settings for aggregation optimizations
_always_anti_join                        Always use this method for anti-join when possible
_always_semi_join                        Always use this method for semi-join when possible
_always_star_transformation              Always favor use of star transformation
_and_pruning_enabled                     Allow partition pruning based on multiple mechanisms
_b_tree_bitmap_plans                     Enable the use of bitmap plans for tables w. only
                                         B-tree indexes
```

```
NAME                                   DESCRIPTION
-------------------------------------- -----------------------------------------------------
_bloom_filter_enabled                  Enables or disables bloom filter
_bloom_folding_enabled                 Enable folding of bloom filter
_bloom_predicate_enabled               Enables or disables bloom filter predicate pushdown
_bloom_predicate_pushdown_to_storage   Enables or disables bloom filter predicate pushdown to
                                       storage
_bloom_pruning_enabled                 Enable partition pruning using bloom filtering
_bloom_pushing_max                     Bloom filter pushing size upper bound
_bloom_vector_elements                 Number of elements in a bloom filter vector
_bt_mmv_query_rewrite_enabled          Allow rewrites with multiple MVs and base tables
_complex_view_merging                  Enable complex view merging
_connect_by_use_union_all              Use union all for connect by
_convert_set_to_join                   Enables conversion of set operator to join
_cost_equality_semi_join               Enables costing of equality semi-join
_cpu_to_io                             Divisor for converting CPU cost to I/O cost
_db_file_optimizer_read_count          Multiblock read count for regular clients
_default_non_equality_sel_check        Sanity check on default selectivity for like/
                                       range predicate
_deferred_constant_folding_mode        Deferred constant folding mode
_dimension_skip_null                   Control dimension skip when null feature
_direct_path_insert_features           Disable direct path insert features
_disable_datalayer_sampling            Disable datalayer sampling
_disable_function_based_index          Disable function-based index matching
_disable_parallel_conventional_load    Disable parallel conventional loads
_distinct_view_unnesting               Enables unnesting of in subquery into distinct view
_dm_max_shared_pool_pct                Max percentage of the shared pool to use for a mining
                                       model
_dml_monitoring_enabled                Enable modification monitoring
_eliminate_common_subexpr              Enables elimination of common sub-expressions
_enable_dml_lock_escalation            Enable dml lock escalation against partitioned tables
                                       if TRUE
_enable_query_rewrite_on_remote_objs   mv rewrite on remote table/view
_enable_row_shipping                   Use the row shipping optimization for wide table selects
_enable_type_dep_selectivity           Enable type dependent selectivity estimates
_extended_pruning_enabled              Do runtime pruning in iterator if set to TRUE
_fast_full_scan_enabled                Enable/disable index fast full scan
_fic_area_size                         Size of Frequent Itemset Counting work area
_first_k_rows_dynamic_proration        Enable the use of dynamic proration of join
                                       Cardinalities
_force_datefold_trunc                  Force use of trunc for datefolding rewrite
_force_rewrite_enable                  Control new query rewrite features
_force_slave_mapping_intra_part_loads  Force slave mapping for intra partition loads
_force_temptables_for_gsets            Executes concatenation of rollups using temp tables
_force_tmp_segment_loads               Force tmp segment loads
_full_pwise_join_enabled               Enable full partition-wise join when TRUE
_gby_hash_aggregation_enabled          Enable group-by and aggregation using hash scheme
_generalized_pruning_enabled           Controls extensions to partition pruning for
                                       general predicates
_globalindex_pnum_filter_enabled       Enables filter for global index with
                                       partition extended syntax
_gs_anti_semi_join_allowed             Enable anti/semi join for the GS query
```

```
NAME                                     DESCRIPTION
---------------------------------------- -----------------------------------------------------
_hash_join_enabled                       Enable/disable hash join
_hash_multiblock_io_count                Number of blocks hash join will read/write at once
_improved_outerjoin_card                 Improved outer-join cardinality calculation
_improved_row_length_enabled             Enable the improvements for computing the
                                         average row length
_index_join_enabled                      Enable the use of index joins
_kdt_buffering                           Control kdt buffering for conventional inserts
_left_nested_loops_random                Enable random distribution method for left of
                                         nestedloops
_like_with_bind_as_equality              Treat LIKE predicate with bind as an equality predicate
_local_communication_costing_enabled     Enable local communication costing when TRUE
_local_communication_ratio               Set the ratio between global and local communication
                                         (0..100)
_minimal_stats_aggregation               Prohibit stats aggregation at compile/partition
                                         maintenance time
_mmv_query_rewrite_enabled               Allow rewrites with multiple MVs and/or base tables
_mv_generalized_oj_refresh_opt           Enable/disable new algorithm for MJV with generalized
                                         outer joins
_nested_loop_fudge                       Nested loop fudge
_new_initial_join_orders                 Enable initial join orders based on new ordering
                                         heuristics
_new_sort_cost_estimate                  Enables the use of new cost estimate for sort
_nlj_batching_enabled                    Enable batching of the RHS IO in NLJ
_no_or_expansion                         OR expansion during optimization disabled
_oneside_colstat_for_equijoins           Sanity check on default selectivity for like/
                                         range predicate
_optim_adjust_for_part_skews             Adjust stats for skews across partitions
_optim_enhance_nnull_detection           TRUE to enable index [fast] full scan more often
_optim_new_default_join_sel              Improves the way default equijoin selectivity
                                         are computed
_optim_peek_user_binds                   Enable peeking of user binds
_optimizer_adaptive_cursor_sharing       Optimizer adaptive cursor sharing
_optimizer_adjust_for_nulls              Adjust selectivity for null values
_optimizer_aw_join_push_enabled          Enables AW Join Push optimization
_optimizer_aw_stats_enabled              Enables statistcs on AW olap_table table function
_optimizer_better_inlist_costing         Enable improved costing of index access using in-list(s)
_optimizer_block_size                    Standard block size used by optimizer
_optimizer_cache_stats                   Cost with cache statistics
_optimizer_cartesian_enabled             Optimizer cartesian join enabled
_optimizer_cbqt_factor                   Cost factor for cost-based query transformation
_optimizer_cbqt_no_size_restriction      Disable cost based transformation query size restriction
_optimizer_coalesce_subqueries           Consider coalescing of subqueries optimization
_optimizer_complex_pred_selectivity      Enable selectivity estimation for built-in functions
_optimizer_compute_index_stats           Force index stats collection on index creation/rebuild
_optimizer_connect_by_cb_whr_only        Use cost-based transformation for whr clause
                                         in connect by
_optimizer_connect_by_combine_sw         Combine no filtering connect by and start with
_optimizer_connect_by_cost_based         Use cost-based transformation for connect by
_optimizer_connect_by_elim_dups          Allow connect by to eliminate duplicates from input
_optimizer_correct_sq_selectivity        Force correct computation of subquery selectivity
```

```
NAME                                    DESCRIPTION
--------------------------------------- -----------------------------------------------------
_optimizer_cost_based_transformation    Enables cost-based query transformation
_optimizer_cost_filter_pred             Enables  costing of filter predicates in IO cost model
_optimizer_cost_hjsmj_multimatch        Add cost of generating result set when #rows per key > 1
_optimizer_cost_model                   Optimizer cost model
_optimizer_degree                       Force the optimizer to use the same degree of
                                        parallelism
_optimizer_dim_subq_join_sel            Use join selectivity in choosing star transformation
                                        dimensions
_optimizer_disable_strans_sanity_checks Disable star transformation sanity checks
_optimizer_distinct_agg_transform       Transforms Distinct Aggregates to non-distinct
                                        aggregates
_optimizer_distinct_elimination         Eliminates redundant SELECT DISTNCT's
_optimizer_distinct_placement           Consider distinct placement optimization
_optimizer_eliminate_filtering_join     Optimizer filtering join elimination enabled
_optimizer_enable_density_improvements  Use improved density computation for selectivity
                                        estimation
_optimizer_enable_extended_stats        Use extended statistics for selectivity estimation
_optimizer_enhanced_filter_push         Push filters before trying cost-based query
                                        transformation
_optimizer_extend_jppd_view_types       Join pred pushdown on group-by, distinct,
                                        semi-/anti-joined view
_optimizer_extended_cursor_sharing      Optimizer extended cursor sharing
_optimizer_extended_cursor_sharing_rel  Optimizer extended cursor sharing for relational
                                        operators
_optimizer_extended_stats_usage_control Controls the optimizer usage of extended stats
_optimizer_fast_access_pred_analysis    Use fast algorithm to traverse predicates for
                                        physical optimizer
_optimizer_fast_pred_transitivity       Use fast algorithm to generate transitive predicates
_optimizer_filter_pred_pullup           Use cost-based flter predicate pull up transformation
_optimizer_fkr_index_cost_bias          Optimizer index bias over FTS/IFFS under first K rows
                                        mode
_optimizer_free_transformation_heap     Free transformation subheap after each transformation
_optimizer_group_by_placement           Consider group-by placement optimization
_optimizer_ignore_hints                 Enables the embedded hints to be ignored
_optimizer_improve_selectivity          Improve table and partial overlap join selectivity
                                        computation
_optimizer_instance_count               Force the optimizer to use the specified number of
                                        instances
_optimizer_join_elimination_enabled     Optimizer join elimination enabled
_optimizer_join_factorization           Use join factorization transformation
_optimizer_join_order_control           Controls the optimizer join order search algorithm
_optimizer_join_sel_sanity_check        Enable/disable sanity check for multi-column
                                        join selectivity
_optimizer_max_permutations             Optimizer maximum join permutations per query block
_optimizer_min_cache_blocks             Set minimum cached blocks
_optimizer_mjc_enabled                  Enable merge join cartesian
_optimizer_mode_force                   Force setting of optimizer mode for user
                                        recursive SQL also
_optimizer_multi_level_push_pred        Consider join-predicate pushdown that requires
                                        multi-level pushdown to base table
```

```
NAME                                    DESCRIPTION
--------------------------------------- -----------------------------------------------------
_optimizer_native_full_outer_join       Execute full outer join using native implementation
_optimizer_nested_rollup_for_gset       Number of groups above which we use nested
                                        rollup exec for gset
_optimizer_new_join_card_computation    Compute join cardinality using non-rounded in put values
_optimizer_null_aware_antijoin          Null-aware antijoin parameter
_optimizer_or_expansion                 Control or expansion approach used
_optimizer_or_expansion_subheap         Use subheap for optimizer or-expansion
_optimizer_order_by_elimination_enabled Eliminates order bys from views before query
                                        transformation
_optimizer_outer_to_anti_enabled        Enable transformation of outer-join to anti-join
                                        if possible
_optimizer_percent_parallel             Optimizer percent parallel
_optimizer_push_down_distinct           Push down distinct from query block to table
_optimizer_push_pred_cost_based         Use cost-based query transformation for push
                                        pred optimization
_optimizer_random_plan                  Optimizer seed value for random plans
_optimizer_reuse_cost_annotations       Reuse cost annotations during cost-based
                                        query transformation
_optimizer_rownum_bind_default          Default value to use for rownum bind
_optimizer_rownum_pred_based_fkr        Enable the use of first K rows due to rownum predicate
_optimizer_search_limit                 Optimizer search limit
_optimizer_self_induced_cache_cost      Account for self-induced caching
_optimizer_skip_scan_enabled            Anable/disable index skip scan
_optimizer_skip_scan_guess              Consider index skip scan for predicates with
                                        guessed selectivity
_optimizer_sortmerge_join_enabled       Enable/disable sort-merge join method
_optimizer_sortmerge_join_inequality    Enable/disable sort-merge join using inequality
                                        predicates
_optimizer_squ_bottomup                 Enables unnesting of subquery in a bottom-up manner
_optimizer_star_tran_in_with_clause     Enable/disable star transformation in with
                                        clause queries
_optimizer_star_trans_min_cost          Optimizer star transformation minimum cost
_optimizer_star_trans_min_ratio         Optimizer star transformation minimum ratio
_optimizer_starplan_enabled             Optimizer star plan enabled
_optimizer_system_stats_usage           System statistics usage
_optimizer_table_expansion              Consider table expansion transformation
_optimizer_transitivity_retain          Retain equi-join pred upon transitive equality
                                        pred generation
_optimizer_try_st_before_jppd           Try Star Transformation before Join Predicate Push Down
_optimizer_undo_changes                 Undo changes to query optimizer
_optimizer_undo_cost_change             Optimizer undo cost change
_optimizer_unnest_all_subqueries        Enables unnesting of every type of subquery
_optimizer_unnest_corr_set_subq         Unnesting of correlated set subqueries (TRUE/FALSE)
_optimizer_unnest_disjunctive_subq      Unnesting of disjunctive subqueries (TRUE/FALSE)
_optimizer_use_cbqt_star_transformation Use rewritten star transformation using cbqt framework
_optimizer_use_feedback                 Optimizer use feedback
_optimizer_use_subheap                  Enables physical optimizer subheap
_or_expand_nvl_predicate                Enable OR expanded plan for NVL/DECODE predicate
_ordered_nested_loop                    Enable ordered nested loop costing
```

```
NAME                                     DESCRIPTION
---------------------------------------- -----------------------------------------------------
_parallel_broadcast_enabled              Enable broadcasting of small inputs to hash
                                         and sort merge joins
_parallel_cluster_cache_policy           Policy used for parallel execution on cluster
                                         (ADAPTIVE/CACHED)
_parallel_scalability                    Parallel scalability criterion for parallel execution
_parallel_syspls_obey_force              TRUE to obey force parallel query/dml/ddl
                                         under System PL/SQL
_parallel_time_unit                      Unit of work used to derive the degree of
                                         parallelism (in seconds)
_partial_pwise_join_enabled              Enable partial partition-wise join when TRUE
_partition_view_enabled                  Enable/disable partitioned views
_pga_max_size                            Maximum size of the PGA memory for one process
_pivot_implementation_method             Pivot implementation method
_pre_rewrite_push_pred                   Push predicates into views before rewrite
_pred_move_around                        Enables predicate move-around
_predicate_elimination_enabled           Allow predicate elimination if set to TRUE
_project_view_columns                    Enable projecting out unreferenced columns of a view
_push_join_predicate                     Enable pushing join predicate inside a view
_push_join_union_view                    Enable pushing join predicate inside a union all view
_push_join_union_view2                   Enable pushing join predicate inside a union view
_px_broadcast_fudge_factor               Set the tq broadcasting fudge factor percentage
_px_minus_intersect                      Enables pq for minus/interect operators
_px_pwg_enabled                          Parallel partition wise group by enabled
_px_ual_serial_input                     Enables new pq for UNION operators
_query_cost_rewrite                      Perform the cost based rewrite with materialized views
_query_mmvrewrite_maxcmaps               Query mmv rewrite maximum number of cmaps per
                                         dmap in query disjunct
_query_mmvrewrite_maxdmaps               Query mmv rewrite maximum number of dmaps per
                                         query disjunct
_query_mmvrewrite_maxinlists             Query mmv rewrite maximum number of in-lists
                                         per disjunct
_query_mmvrewrite_maxintervals           Query mmv rewrite maximum number of intervals
                                         per disjunct
_query_mmvrewrite_maxmergedcmaps         Query mmv rewrite maximum number of merged cmaps
_query_mmvrewrite_maxpreds               Query mmv rewrite maximum number of
                                         predicates per disjunct
_query_mmvrewrite_maxqryinlistvals       Query mmv rewrite maximum number of query in-list values
_query_mmvrewrite_maxregperm             Query mmv rewrite maximum number of region permutations
_query_rewrite_1                         Perform query rewrite before&after or only before
                                         view merging
_query_rewrite_2                         Perform query rewrite before&after or only
                                         after view merging
_query_rewrite_drj                       mv rewrite and drop redundant joins
_query_rewrite_expression                Rewrite with cannonical form for expressions
_query_rewrite_fpc                       mv rewrite fresh partition containment
_query_rewrite_fudge                     Cost based query rewrite with MVs fudge factor
_query_rewrite_jgmigrate                 mv rewrite with jg migration
_query_rewrite_maxdisjunct               Query rewrite max disjuncts
_query_rewrite_or_error                  Allow query rewrite, if referenced tables are
                                         not dataless
```

```
NAME                                    DESCRIPTION
--------------------------------------- -----------------------------------------------------
_query_rewrite_setopgrw_enable          Perform general rewrite using set operator summaries
                                        summaries
_query_rewrite_vop_cleanup              Prune frocol chain before rewrite after view-merging
_rdbms_internal_fplib_enabled           Enable CELL FPLIB filtering within rdbms
_remove_aggr_subquery                   Enables removal of subsumed aggregated subquery
_replace_virtual_columns                Replace expressions with virtual columns
_result_cache_auto_size_threshold       Result cache auto max size allowed
_result_cache_auto_time_threshold       Result cache auto time threshold
_right_outer_hash_enable                Right Outer/Semi/Anti Hash Enabled
_row_shipping_explain                   Enable row shipping explain plan support
_row_shipping_threshold                 Row shipping column selection threshold
_rowsrc_trace_level                     Row source tree tracing level
_selfjoin_mv_duplicates                 Control rewrite self-join algorithm
_simple_view_merging                    Control simple view merging performed by the optimizer
_slave_mapping_enabled                  Enable slave mapping when TRUE
_smm_auto_cost_enabled                  If TRUE, use the AUTO size policy cost functions
_smm_auto_max_io_size                   Maximum IO size (in KB) used by sort/hash-join
                                        in auto mode
_smm_auto_min_io_size                   Minimum IO size (in KB) used by sort/ hash-join
                                        in auto mode
_smm_max_size                           Maximum work area size in auto mode (serial)
_smm_min_size                           Minimum work area size in auto mode
_smm_px_max_size                        Maximum work area size in auto mode (global)
_sort_elimination_cost_ratio            Cost ratio for sort eimination under first_rows mode
_sort_multiblock_read_count             Multi-block read count for sort
_spr_push_pred_refspr                   Push predicates through reference spreadsheet
_sql_compatibility                      sql compatability bit vector
_sql_model_unfold_forloops              Specifies compile-time unfolding of sql model forloops
_subquery_pruning_enabled               Enable the use of subquery predicates to perform pruning
_subquery_pruning_mv_enabled            Enable the use of subquery predicates with MVs
                                        to perform pruning
_system_index_caching                   Optimizer percent system index caching
_table_scan_cost_plus_one               Bump estimated full table scan and index ffs cost by one
_trace_virtual_columns                  Trace virtual columns exprs
_union_rewrite_for_gs                   Expand queries with GSets into UNIONs for rewrite
_unnest_subquery                        Enables unnesting of complex subqueries
_use_column_stats_for_function          Enable the use of column statistics for DDP functions
_virtual_column_overload_allowed        Overload virtual columns expression
_with_subquery                          WITH subquery transformation
active_instance_count                   Number of active instances in the cluster database
bitmap_merge_area_size                  Maximum memory allow for BITMAP MERGE
cell_offload_compaction                 Cell packet compaction strategy
cell_offload_plan_display               Cell offload explain plan display
cell_offload_processing                 Enable SQL processing offload to cells
cpu_count                               Number of CPUs for this instance
cursor_sharing                          Cursor sharing mode
db_file_multiblock_read_count           db block to be read each IO
dst_upgrade_insert_conv                 Enables/disables internal conversions during
                                        DST upgrade
hash_area_size                          Size of in-memory hash work area
```

```
NAME                                    DESCRIPTION
--------------------------------------- -----------------------------------------------------
optimizer_capture_sql_plan_baselines    Automatic capture of SQL plan baselines for
                                        repeatable statements
optimizer_dynamic_sampling              Optimizer dynamic sampling
optimizer_features_enable               Optimizer plan compatibility parameter
optimizer_index_caching                 Optimizer percent index caching
optimizer_index_cost_adj                Optimizer index cost adjustment
optimizer_mode                          Optimizer mode
optimizer_secure_view_merging           Optimizer secure view merging and predicate
                                        pushdown/movearound
optimizer_use_invisible_indexes         Usage of invisible indexes (TRUE/FALSE)
optimizer_use_pending_statistics        Control whether to use optimizer pending statistics
optimizer_use_sql_plan_baselines        Use of SQL plan baselines for captured sql statements
parallel_degree_limit                   Limit placed on degree of parallelism
parallel_degree_policy                  Policy used to compute the degree of parallelism
                                        (MANUAL/LIMITED/AUTO)
parallel_force_local                    Force single instance execution
parallel_min_time_threshold             Threshold above which a plan is a candidate
                                        for parallelization (in seconds)
parallel_threads_per_cpu                Number of parallel execution threads per CPU
pga_aggregate_target                    Target size for the aggregate PGA memory
                                        consumed by the instance
query_rewrite_enabled                   Allow rewrite of queries using materialized
                                        views if enabled
query_rewrite_integrity                 Perform rewrite using materialized views with
                                        desired integrity
result_cache_mode                       Result cache operator usage mode
skip_unusable_indexes                   Skip unusable indexes if set to TRUE
sort_area_retained_size                 Size of in-memory sort work area retained
                                        between fetch calls
sort_area_size                          Size of in-memory sort work area
star_transformation_enabled             Enable the use of star transformation
statistics_level                        Statistics level
workarea_size_policy                    Policy used to size SQL working areas (MANUAL/AUTO)
```

APPENDIX C

Tool Configuration Parameters

This appendix includes all the tool configuration parameters and their descriptions.

Parameter	Description
`automatic_workload_repository`	Access to the Automatic Workload Repository (AWR) requires a license for the Oracle Diagnostic Pack. If you don't have it you can set this parameter to `N`.
`bde_chk_cbo`	On EBS applications SQLT automatically executes `bde_chk_cbo.sql` from Note:174605.1.
`c_cbo_stats_vers_days`	Days of CBO statistics versions to be collected. If set to 0 no statistics versions are collected. If set to a value larger than actual stored days, then SQLT collects the whole history. A value of 7 means collect the past 7 days of CBO statistics versions for the schema objects related to given SQL. It includes tables, indexes, partitions, columns, and histograms.
`c_dba_hist_parameter`	Collects relevant entries out of DBA_HIST_PARAMETER. If `automatic_workload_repository` and `c_dba_hist_parameter` are both set to `Y` then SQLT collects relevant rows out of view DBA_HIST_PARAMETER.
`c_gran_cols`	Collection Granularity for Columns. Default value of "SUBPARTITION" allows SQLT to collect into its repository CBO statistics for columns at all levels: table, partitions and subpartitions. All related to the one SQL being analyzed.
`c_gran_hgrm`	Collection Granularity for Histograms. Default value of "SUBPARTITION" allows SQLT to collect into its repository CBO statistics for histograms at all levels: table, partitions and subpartitions. All related to the one SQL being analyzed.
`c_gran_segm`	Collection Granularity for Segments (Tables and Indexes). Default value of "SUBPARTITION" allows SQLT to collect into its repository CBO statistics for tables, indexes, partitions and subpartitions. All related to the one SQL being analyzed.
`collect_perf_stats`	Collects performance statistics on XECUTE method.
`connect_identifier`	Optional Connect Identifier (as per Oracle Net). This is used during export of SQLT repository. Include "@" symbol, i.e., @PROD. You can also set this parameter to `NULL`.

(*continued*)

Parameter	Description
`count_star_threshold`	Limits the number or rows to count while doing a `SELECT COUNT(*)` in set of tables accessed by SQL passed. If you want to disable this functionality set this parameter to `0`.
`custom_sql_profile`	Controls if a script with a Custom SQL Profile is generated with every execution of SQLT main methods.
`distributed_queries`	SQLT can use DB links referenced by the SQL being analyzed. It connects to those remote systems to get 10053 and 10046 traces for the SQL being distributed.
`domain_index_metadata`	This parameter controls if domain index metadata is included in main report and metadata script. If you get an ORA-07445, and the `alert.log` shows the error is caused by CTXSYS.CTX_REPORT.CREATE_INDEX_SCRIPT, then you want to set this parameter to `N`.
`event_10046_level`	SQLT XECUTE turns event 10046 level 12 on by default. You can set a different level or turn this event 10046 off using this parameter. It only affects the execution of the script passed to SQLT XECUTE. Level 0 means no trace, level 1 is standard SQL Trace, level 4 includes bind variable values, level 8 includes waits and level 12 both binds and waits.
`event_10053_level`	SQLT XECUTE, XTRACT and XPLAIN turn event 10053 level 1 on by default. You can turn this event 10053 off using this parameter. It only affects the SQL passed to SQLT. Level 0 means no trace, level 1 traces the CBO.
`event_10507_level`	SQLT XECUTE uses this event on 11g to trace Cardinality Feedback CFB. You can turn this event 10507 off using this parameter. It only affects the SQL passed to SQLT. Level 0 means no trace, for meaning of other levels see MOS Doc ID 740052.1.
`event_others`	This parameter controls the use of events 10241, 10032, 10033, 10104, 10730, 46049, but only if 10046 is turned on (any level but 0). It only affects the execution of the script passed to SQLT XECUTE.
`export_repository`	Methods XTRACT, XECUTE and XPLAIN automatically perform an export of corresponding entries in the SQLT repository. This parameter controls this automatic repository export.
`export_utility`	SQLT repository can be exported automatically using one of two available utilities: traditional export “exp” or data pump “expdp”. With this parameter you can specify which of the two should be used by SQLT.
`generate_10053_xtract`	Generation of 10053 using DBMS_SQLDIAG.DUMP_TRACE on XTRACT can be eliminated as a workaround to a disconnect ORA-07445 on SYS.DBMS_SQLTUNE_INTERNAL. SQLT detects an ORA-07445 and disables the call to DBMS_SQLDIAG.DUMP_TRACE (and SYS.DBMS_SQLTUNE_INTERNAL) in next execution. If this parameter has a value of E or N, then you may have a low-impact bug in your system.
`healthcheck_blevel`	Compute index/partition/subpartition blevel and check if they change more than 10 percent from one statistics gathering to the next.
`healthcheck_endpoints`	Compute histogram endpoints count and check if they change more than 10 percent from one statistics gathering to the next.

(*continued*)

Parameter	Description
`healthcheck_ndv`	Review if number of distinct values for columns changes more than 10 percent from one statistics gathering to the next.
`healthcheck_num_rows`	Review table/partition/subpartition number of rows and check if they change more than 10 percent from one statistics gathering to the next.
`keep_trace_10046_open`	If you need to trace an execution of SQLT XECUTE, XTRACT or XPLAIN, this parameter allows you to keep trace 10046 active even after a custom SCRIPT completes. It is used by XECUTE, XTRACT, and XPLAIN. When set to its default value of `N`, event 10046 is turned off right after the execution of the custom SCRIPT or when 10053 is turned off.
`mask_for_values`	Endpoint values for table columns are part of the CBO statistics. They include column low/high values as well as histograms. If for privacy reasons these endpoints must be removed from SQLT reports, you can set this parameter to SECURE or COMPLETE. SECURE displays only the year for dates, and one character for strings and numbers. COMPLETE blocks completely the display of endpoints, and it also disables the automatic export of the SQLT repository. The default is CLEAR, which shows the values of endpoints. If considering changing to a non-default value, bear in mind that selectivity and cardinality verification requires some knowledge of the values of these column endpoints. Be also aware that 10053 traces also contain some low/high values that are not affected by this parameter.
`plan_stats`	Execution plans from GV$SQL_PLAN may contain statistics for the last execution of a cursor and for all executions of it (if parameter `statistics_level` was set to `ALL` when the cursor was hard-parsed). This parameter controls the display of the statistics of both (last execution as well as all executions).
`predicates_in_plan`	Predicates in plan can be eliminated as a workaround to bug 6356566. SQLT detects an ORA-07445 and disables predicates in next execution. If this parameter has a value of E or N, then you may have bug 6356566 in your system. You may want to apply a fix for bug 6356566, then reset this parameter to its default value.
`r_gran_cols`	Report Granularity for Columns. Default value of “PARTITION” reports table partition columns. All related to the one SQL being analyzed.
`r_gran_hgrm`	Report Granularity for Table Histograms. Default value of “PARTITION” reports table and partition histograms. All related to the one SQL being analyzed.
`r_gran_segm`	Report Granularity for Segments (Tables and Indexes). Default value of “PARTITION” reports tables, indexes, and partitions. All related to the one SQL being analyzed.
`r_gran_vers`	Report CBO Statistics Version Granularity for Tables. Default value of “COLUMN” reports statistics versions for segments and their columns. All related to the one SQL being analyzed.
`r_rows_table_l`	Restricts number of elements for large HTML tables or lists.
`r_rows_table_m`	Restricts number of elements for medium HTML tables or lists.

(*continued*)

Parameter	Description
`r_rows_table_s`	Restricts number of elements for small HTML tables or lists.
`r_rows_table_xs`	Restricts number of elements for extra-small HTML tables or lists.
`refresh_directories`	Controls if SQLT and TRCA directories for UDUMP/BDUMP should be reviewed and refreshed every time SQLT is executed.
`search_sql_by_sqltext`	XPLAIN method uses the SQL text to search in memory and AWR for known executions of SQL being analyzed. If prior executions of this SQL text are found, corresponding plans are extracted and reported.
`skip_metadata_for_object`	This case-sensitive parameter allows you to specify an object name to be skipped from metadata extraction. It is used in cases where DBMS_METADATA errors with ORA-7445. You can specify a full or a partial object name to be skipped (examples: "CUSTOMERS" or "CUSTOMER%" or "CUST%" or "%"). To find the object name where metadata errored out you can use: `SELECT * FROM sqlt$_log WHERE statement_id = 99999 ORDER BY line_id;` You have to replace `99999` with the correct statement_id. To actually fix an error behind ORA-7445, you can use `alert.log` and the trace referenced by it.
`sqlt_max_file_size_mb`	Maximum size of individual SQLT files in megabytes.
`sql_monitoring`	Be aware that using SQL Monitoring (V$SQL_MONITOR and V$SQL_PLAN_MONITOR) requires a license for the Oracle Tuning Pack. If you don't have it you can set this parameter to `N`.
`sql_tuning_advisor`	Be aware that using SQL Tuning Advisor (STA) DBMS_SQLTUNE requires a license for the Oracle Tuning Pack. If you don't have it you can set this parameter to `N`.
`sql_tuning_set`	Generates a SQL Tuning Set for each plan when using XTRACT.
`sta_time_limit_secs`	STA time limit in seconds. See `sql_tuning_advisor`. Be aware that using SQL Tuning Advisor (STA) DBMS_SQLTUNE requires a license for the Oracle Tuning Pack.
`tcb_time_limit_secs`	TCB (test case builder) time limit in seconds. See `test_case_builder`.
`test_case_builder`	11g offers the capability to build a test case for a SQL. TCB is implemented using the API DBMS_SQLDIAG.EXPORT_SQL_TESTCASE. SQLT invokes this API whenever possible. When TCB is invoked by SQLT, the parameter `exportData` gets passed a value of `FALSE`, thus no application data is exported into the test case created by TCB.
`trace_analyzer`	SQLT XECUTE invokes Trace Analyzer - TRCA (Note:224270.1). TRCA analyzes the 10046_10053 trace created by SQLT. It also splits the trace into two stand-alone files 10046 and 10053.
`upload_trace_size_mb`	SQLT uploads to its repository traces generated by events 10046 and 10053. This parameter controls the maximum amount of megabytes to upload per trace.
`validate_user`	Validates that user of main methods has been granted the SQLT_USER_ROLE or DBA roles; or that user is SQLTXPLAIN or SYS.
`xecute_script_output`	SQLT XECUTE generates a spool file with the output of the SQL being analyzed (passed within input script). This file can be kept in the local directory, or included in the zip file, or simply removed.

Index

Oracle SQL Tuning with Oracle SQLTXPLAIN

Stelios Charalambides

Apress®

Oracle SQL Tuning with Oracle SQLTXPLAIN

Copyright © 2013 by Stelios Charalambides

ISBN-13 (pbk): 978-1-4302-4809-5

ISBN-13 (electronic): 978-1-4302-4810-1

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a Delaware corporation.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at www.apress.com/bulk-sales.

Any source code or other supplementary materials referenced by the author in this text is available to readers at www.apress.com. For detailed information about how to locate your book's source code, go to www.apress.com/source-code/.

I dedicate this book to my beautiful family, who put up with my strange working hours and endless nights strapped to a laptop. The journey is nearly at an end. As always, Lesley is my core. Without her I would achieve nothing. Thank you for helping me to achieve this.

Contents

About the Author

Stelios Charalambides has more than 20 years experience working with Oracle databases. He is OCP certified from 7 to 11g and has worked as a Senior Consultant DBA on both sides of the Atlantic, dealing with all aspects of system design, implementation, and post-production support, solving practical problems in a wide variety of environments. He now works as a Principal Oracle Support Engineer developing time- critical solutions for tier-one customers with high-profile performance problems. Though born in the UK, Stelios now lives in New Hampshire with his wife, two children, and two dogs. Once this book is complete, he looks forward to devoting more time to his hobbies.

About the Technical Reviewer

Mark Bobak is a Senior Oracle DBA at ProQuest Company in Ann Arbor, MI. He's been working in IT for over 25 years. For the past 13 years, he has worked as an Oracle DBA. He is an Oracle ACE and a member of the OakTable Network. He is also active in his local Oracle User Group (SEMOP) and attends and presents at conferences at the local, state, national, and international levels.

Acknowledgments

Carlos Sierra is the author of SQLTXPLAIN. His enthusiasm, patience, and willingness to listen allowed me to enter the world of Oracle SQL tuning with SQLT. Without his help and inspiration, this book would not have been possible. I hope that the text meets his expectations and that I haven't made any errors in my description of his baby. If I have I apologize. The error was all mine.

Thanks are also due to my first family within Oracle (The Oracle Performance Group), Chris Crocker, and Cathy Scully (who had the good sense to hire a Brit) and Mike Matagranno, whose persistence in getting me to move to Oracle was truly astounding. I hope none of you have regretted that decision. Special mention must go to Mauro Pagano, whose special knowledge and endless enthusiasm convinced me that I had something useful to say about SQLT.

Many thanks also to Mike Gervais, my first manager, whose kindness helped so much during my first months at Oracle. Alan Bashy, Nick Meola, and Peter Gadah are fellow members of the performance team who also deserve special mention. They are truly awesome and showed me that no matter how much you think you know about tuning, there is always more to learn. I have been very lucky to come across so many knowledgeable and enthusiastic colleagues and managers. There isn't room here to thank them all, and for that I am sorry.

Thanks are also due to Professor Garry Hunt who always believed in my potential and taught me that anything is possible if you put your mind to it.

From Apress, Jonathan Gennick made the final decision to publish this book, showing faith when others were doubtful. Thanks also to Anamika Panchoo, who put up with my missed deadlines: I'm sorry, I did my best. Last but by no means least, Chris Nelson's herculean efforts have made immeasurable improvements to my poor illiterate scratchings (don't change that) and allowed me to produce a book I hope will be useful for years to come. Chris shouted, pushed, and otherwise kept me going by telling me the finish line was just around the corner. For your patience and hard work I sincerely thank you.

Foreword

You are about to read this book on the subject of SQL Tuning using SQLTXPLAIN (also referred to as SQLT for short). Chances are you know a bit about what this SQLT tool offers but not much about the story behind it. This foreword provides some background on how this tool became what it is today. I hope you enjoy this behind-the-scenes look at the motivations and events that slowly came together over the years as I've worked on the tool you're about to learn.

In the late 1990s I was a "road warrior" like many consultants back then. I was an Oracle "field support engineer" with pretty good expertise in manufacturing and SQL tuning, and I had chosen to be part of the Short-term Assignments Team. (We had two options back then, short or long-term assignments!). I used to travel a lot from site to site for these assignments.

Manufacturing and SQL Tuning was a good combination for someone like me, willing to go on site and handle some of the big fires affecting large manufacturing corporations using Oracle ERP. Life was intense, and it was good! After several week-long assignments I started noticing some patterns: when trying to diagnose a SQL statement performing poorly, there were many areas to look at, and trace/tkprof would not give me everything I needed to effectively diagnose SQL tuning issues promptly! Through my assignments, I developed a set of flexible scripts. That is how the legendary coe_xplain.sql came to life.

Karl Daday, my manager at the time, supported my endeavors and actually encouraged me to build my own set of tools so I could deliver results in a one-week time frame. Thus, I used coe_xplain.sql on pretty much every performance assignment and always with good results. Over time I was glad I developed this little tool, since most of my enhancement requests for tkprof are still pending.

One day, while I has happily using my coe_xplain.sql personal script, as well as some other scripts I had developed, a talented DBA in one of the corporations I was working with at the time asked me if he could keep my set of tools for later use. My concern over leaving behind my toys was that I knew from experience that anything you code will follow you your entire life, which is both a reward and a curse. If your code has only a few bugs, you do fine. Otherwise, you feel haunted and hunted for decades!

Nevertheless, I decided to share my coe_xplain.sql and other small tools, with the understanding that their users would take them "as is." A year later, the same DBA asked me if I would publish my coe_xplain.sql script in Metalink (now known as MyOracle Support), so he could keep getting new versions if I decided to keep it updated. This was a turning point for me. Making a tool available in Metalink back in 2001 meant only one thing to me: I was willing to compromise to keep this tool free of bugs as much as possible, and I would have to react to enhancement requests even if that meant declining them all. I knew back then (as I know today) that I always have a hard time saying "no" when I actually can and want to say "yes."

So after coe_xplain.sql was published in Metalink, I quickly started getting some questions like: "Can you add this little tiny functionality to your otherwise nice script?"

Late in 2002 the coe_xplain.sql script had become a large script, and I decided it was time to actually upgrade it to PL/SQL. That would mean rewriting the entire functionality but using one PL/SQL package instead of several SQL statements embedded into one large script. In those days I was still part of a team named the "Center of Expertise." That is why coe_xplain.sql had that prefix "coe_". Today there are many teams within Oracle sharing the same "CoE" name, so I feel its meaning is somewhat diluted. (Something similar happened to the "BDE" team, which means "Bug Diagnostics and Escalations." That's the reason why some of my scripts had and still have the prefix "bde_".)

I decided it was time to upgrade coe_xplain.sql to something more robust. I no longer wanted to name my scripts after the name of the team I was working for. So in 2002, on the second major version, this coe_xplain.sql tool came to be SQLTXPLAIN, and I published it on Metalink (MyOracle Support) under note 215187.1, which still is its location today. The name SQLTXPLAIN is loosely derived from "SQL Tuning and Explain Plan". I had searched the Internet and had not found any references to this SQLTXPLAIN name. I was trying to avoid collisions with other tool names, products, or companies, and as of today I have succeeded, at least in regard to this naming!

SQLTXPLAIN was rapidly adopted by many Oracle ERP teams within Oracle Support and gradually through Oracle Development. Most of the SQLT enhancement requests I used to get in those days were E-Business Suite (EBS) specific, so SQLTXPLAIN became mostly a tool to diagnose SQL statements performing poorly within EBS. From 2002 all the way to 2006, this tool was installed inside the application schema, what was APPS for EBS. So SQLTXPLAIN used to have strong EBS dependencies. It was created inside an application schema, creating and dropping objects there. But don't panic! It no longer works like this.

Those years between 2002 and 2006 were difficult for me at Oracle. I was extremely busy developing internal applications. I was also getting my master's degree in computer science, so I had a double challenge on my hands. And that's not to mention that my wife and I were raising four teenagers! Thus, SQLTXPLAIN was practically frozen until 2007.

I didn't mention it before, so I will now: SQLTXPLAIN was never an official project with Oracle. SQLTXPLAIN up until 2007 was my weekend pet project (a big one!). I spent pretty much all my personal free time developing and maintaining this tool. It was my baby and still is today. I am very lucky that my wife Kelly Santana, who also works for Oracle, has been so very supportive during all these busy years. Without her patience and understanding I would have had to abandon SQLTXPLAIN and taken it off Metalink years ago!

Late in 2006 I moved to the Server Technologies Center of Expertise (ST CoE) within Oracle. I had been at the Applications CoE before, and now I was getting into the ST CoE. That was "the other side of the house," "the dark side" as some call it, or simply "the other side." I took with me all my EBS expertise and all my tools. Steve Franchi, one of the best managers I have had at Oracle, gave me the green light to keep maintaining my tools, so I was going to be able to work on SQLTXPLAIN during my regular working hours. That was a huge incentive for me, so I was very happy at my new home within Oracle. Until then, most of my scripts were actually worked only at night and on the weekend. That's the problem when your hobby and your job blend so smoothly into one: and on top of that, you have the privilege of working from home. So today I would say I take care of SQLTXPLAIN pretty much around the clock, every day of the week. But of course I still sleep and eat!

Once I joined the ST CoE, I was on a mission of my own: I wanted the Server Technologies Support group to get to know SQLTXPLAIN and actually adopt it as much as EBS, so I requested a meeting with one of the legends and leaders at the performance support team. This is how I got to meet Abel Macias. When I asked Abel about using SQLTXPLAIN for ST performance issues, his answer was brusque. I don't recall his exact words, but the meaning was something like this: "We won't use it, because it sucks big time."

Abel is known for his sometimes no-nonsense (but correct) answers. He is a very humble human being, and is way too direct in his observations, which can be intimidating. After my initial shock, I asked him why he thought that way. This opened the door to a short but significant list of deficiencies SQLTXPLAIN had. I walked away demoralized but with a new challenge in front of me: I would take all that constructive criticism and turn it around into big enhancements to SQLTXPLAIN. After several weeks of hard work (and way too many cups of coffee), I met Abel again and showed to him all those "adjustments" to SQLTXPLAIN. Then and only then did he like it, and Abel and I became friends! Today, we ride our bikes together for miles. And even if we don't want to, we frequently have short but meaningful conversations on SQL Tuning and SQLTXPLAIN while we enjoy our trail rides, or while having a beer . . . or two.

With Abel's input, SQLT became application independent, installed in its own schema, and RAC aware. Since the name SQLTXPLAIN was so unique, it became the name of the application schema into which all the tool objects would be installed. Everything looked fine, but there was something bothering me, and it was the fact that SQLTXPLAIN had been heavily modified and was requiring more and more maintenance. It was time for a third full rewrite. Since the code was already big, I knew I would need a lot of dedicated time with no interruptions at all. I decided to take advantage of all my accumulated vacation time, together with a few holidays; and during December 2009 and January 2010 I pretty much lived in my man-cave developing the third major version of SQLT.

I finished with all the details and testing on April 2010. SQLT now had several packages, and it was getting close to 100,000 lines of code, all developed with two fingers (I still type with two fingers). Some people think SQLT is developed and maintained by a large team. Sorry to disappoint you if you think this way: SQLT was conceived as a tool for someone from support to make his/her life easier and is now somehow shared and used by many others working on the complex task of SQL Tuning. Still the spirit of this tool remains unchanged: a tool to help to diagnose a SQL statement that performs poorly (or which generates wrong results).

SQLTXPLAIN, on its third major version from April 2010, was rapidly adopted by the Server Technologies performance team within support. Since I joined the ST CoE I also have had the pleasure to develop two internal one-week SQL Tuning workshops. After delivering them dozens of times to more than 700 Oracle professionals around the world, I have collected and incorporated hundreds of enhancements to SQLT that were communicated verbally during class.

From all the good and smart people providing me with ideas on how to enhance SQLT, I want to mention two by name, who in addition to Abel have made exceptional contributions to the feature set of the tool: Mauro Pagano and Mark Jefferys. Mauro, with his constant insight, inspired SQLTXPLAIN's XPLORE method, which uses brute force analysis to narrow possible culprits of a regression after an upgrade. Mark, with his strong math and science background, showed me several mistakes I had made, especially around statistics and test case creation. With all sincerity, I think SQLT is what it is today thanks to all the constructive feedback I constantly get from very smart people. Sometimes the feedback is brutal, but for the most part it is just asking me to incorporate a thing here or there, or to fix a small bug. As enhancement examples I would mention some big ones: adding Siebel and PeopleSoft awareness in SQLT, and all the health-checks it performs.

I consider myself a very lucky guy to have had the opportunity to work for such a great company as Oracle since 1996; I have also had some great managers who have been very supportive in taking what I believe is the right approach to diagnose difficult SQL Tuning issues. I do not regret a single minute of my personal time that I have dedicated to this SQLTXPLAIN tool. At the beginning it was from me and to me. Now it is from me and many other bright minds to basically anybody who wants to get his/her hands into some serious SQL Tuning in an Oracle database.

When Stelios Charlambides asked my opinion about him writing a book in which SQLXPLAIN was a central part, I felt honored and happy. Many people have asked me: "When can we have a book on SQLTXPLAIN?" If you are one of those people, please enjoy what Stelios has developed during his own personal time, and thank him for this book! I truly hope you get to enjoy SQLTXPLAIN and even SQL Tuning.

I always say that SQL Tuning is like sushi, you either love it or you hate it! I hope that as you get to learn more, you'll fall in love with it as I did back in the 1990s. Also keep in mind that with SQL Tuning you never finish learning. Good foundations, curiosity, perseverance, and experience—these are just some of the ingredients to one day feeling comfortable doing SQL Tuning. Cheers!

—Carlos Sierra, author of SQLTXPLAIN