Oracle Real Application Testing: Testing the SQL Performance Impact of an Oracle Database 9i/10g Release 1 to Oracle Database 10g Release 2 upgrade with SQL Performance Analyzer

An Oracle White Paper
April 2008

ORACLE

Testing the SQL Performance Impact of an Oracle 9i/10g Release 1 to Oracle Database 10g Release 2 upgrade with SQL Performance Analyzer

Testing the SQL Performance Impact
of an Oracle Database 9i/10g Release 1 to Oracle Database 10g
Release 2 upgrade with
SQL Performance Analyzer

INTRODUCTION

SQL Performance Analyzer (SPA) has been enhanced to support Oracle 9i/ 10g Release 1 to Oracle Database 10g Release 2 upgrades.
SPA is licensed with the Real Application Testing option.

This document describes changes made to the SQL Performance Analyzer (SPA) feature to support customers upgrading their database systems from Oracle 9i to Oracle Database 10g Release 2. In Oracle Database Release 11.1.0.6, SPA is a general-purpose SQL performance testing feature that adapts equally well to any number of changes. As far as upgrade testing is concerned, we focused on supporting Oracle Database 10g Release 2 to Oracle Database 11g upgrades in the first release of Oracle Database 11g. In the past few months, we have enhanced SPA to support the Oracle Database 9i/10g Release 1[↑] to Oracle Database 10.2 upgrade use case as well. These enhancements can be installed on top of test database systems running Oracle Database 11g and 10.2 releases ([SPAMET]). With these additions to SPA, now you can load Oracle 9i production SQL trace files into an Oracle Database 11g test system, execute the SQLs on an Oracle Database 10g Release 2 test database to build the post-upgrade performance, and then compare the performance data on the Oracle Database 11g test system.

In this white paper we enumerate as specifically as possible a set of best practices for testing the impact of this upgrade on SQL performance. Some sections focus on describing important concepts in SQL performance; others offer our precise recommendations on how to carry out each step of the test. We try not to stray from the intended use case of an Oracle 9i to Oracle Database 10.2 upgrade, but the principles described here apply equally well to other database changes or upgrades. SPA is a tool that can be used just as effectively to test an upgrade from Oracle Database 10.2 to Oracle Database 11g, from one patchset of Oracle Database 10.2 to another, or practically any other database change. With an understanding of the components of a SPA experiment and the necessary foundations of SQL performance testing, you should be ready to devise your own

[↑] Note: The steps used for upgrade testing of Oracle Database 9i and Oracle Database 10g Release 1 are similar, both use SQL trace based mechanism. In the rest of the document, any reference to SQL trace based mechanism to generate a SQL Tuning Set in Oracle Database 9i upgrade testing also refers to Oracle Database 10g Release 1.

test scenarios following in the same spirit. Some experiments might be able to follow the steps in this document with hardly any changes (for example, an Oracle Database 10.1 to Oracle Database 11g upgrade can be tested using exactly the same methodology as the one described here), while others will require slightly different test configurations.

We begin with an overview of SPA functionality and then go through the major steps of testing a system upgrade using SPA as the guide: preparing the systems that we will use for the test, capturing pre-upgrade performance data from Oracle 9i production, executing the SQLs in an Oracle Database 10g test environment, and comparing and analyzing the results. We finish with our recommendations on how to investigate and tune any regressions detected by SPA and, finally, we show that SPA can then be used to test those fixes just as it was used to test the impact of the upgrade itself. This way, the experiment will eventually reach a steady state, and you can upgrade production knowing that you have fixed a large subset of the SQL performance problems before they can negatively impact your production workload.

SQL PERFORMANCE ANALYZER FUNCTIONALITY

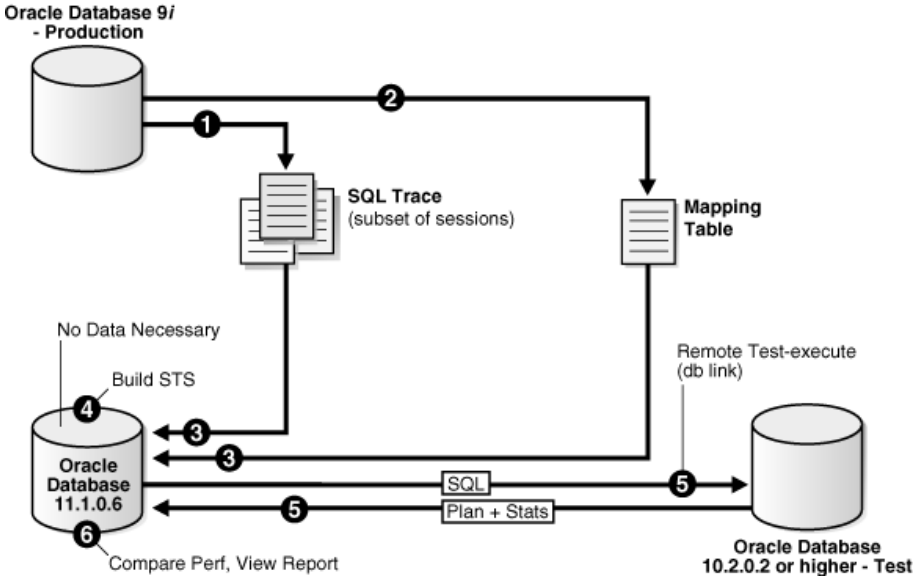
Changes that affect SQL execution plans, such as upgrading a database or adding new indexes, can severely impact SQL performance in unpredictable ways. As a result, DBAs spend an enormous amount of time and effort identifying and fixing SQL statements that have regressed due to the changes. SQL Performance Analyzer (SPA), a key feature of the Real Application Testing option introduced in Oracle Database 11g, can predict and quantify SQL performance improvements and regressions caused by system changes.

SQL Performance Analyzer provides a granular view of the impact of changes on SQL execution plans and execution statistics by running the SQL statements on a test system in isolation before and after a change. It does not consider more complex issues, such as load or concurrency problems, in an effort to provide a manageable report. SPA compares the SQL execution result, before and after the change, and generates a report outlining the net benefit on the workload due to the changes as well as the set of regressed SQL statements. For regressed SQL statements, appropriate execution plan details along with recommendations to remedy them are provided.

For additional information on the functionality and usage model of SPA, see the SPA Oracle OpenWorld white paper ([SPAOOW]).

RECOMMENDED USAGE SCENARIO OVERVIEW

The following diagram shows a high-level overview of the major steps involved in testing a database upgrade from Oracle 9i to Oracle Database 10.2 with SPA:



Our SPA test involves three databases: an Oracle 9i production db running the workload, an Oracle Database 10.2 replica of production in test, and a small Oracle Database 11g test system.

1. Begin by capturing an authentic SQL workload on your Oracle 9i production system in a set of SQL trace files.
2. Next, create an object-mapping table for decoding some of the information in the SQL trace files on the Oracle 9i production system.
3. We then explain how to build a pair of Oracle Database 11g and 10.2 test database systems that will become the principal pieces of our SPA experiment. The SQL trace files and mapping table generated in steps 1 and 2 will be copied to the Oracle Database 11g test system.
4. On the Oracle Database 11g test system, you will construct a SQL Tuning Set (STS) from the SQL trace files and the mapping table. The STS is a database object representing a set of SQL statements with their execution context and performance data; it will serve as the primary input to SPA.
5. From the Oracle Database 11g test system you will use SPA to connect to the Oracle Database 10.2 system through a database link and execute the

SQL workload one statement at a time, collecting all of the performance data in the Oracle Database 11g database.

6. Finally, we show how to use SPA to analyze the two sets of performance data collected, tune any SQLs found to have regressed, and iteratively test the proposed fixes until the system reaches a steady state.

LIMITATIONS

The following technologies are not supported for the Oracle 9i to Oracle Database 10.2 upgrade use case:

- The Shared Server (MTS) architecture is not supported.
- Parallel Query and Parallel DML are not supported. The architectural changes in parallel processing between these releases greatly complicates the comparison of parallel plans between these releases.
- Remote SQL are not supported except for the REMOTE plan operation where rows are fetched from the remote database but processed locally.

PREPARING TO TEST THE UPGRADE

As we have described already, testing an upgrade with SQL Performance Analyzer consists of a number of steps to be performed across different database systems. In this section we describe how to collect the necessary information from the production system running Oracle 9i database and how to set up each system properly to ensure a valid test. With this work done we will be ready to begin the actual SPA experiment itself.

Capturing Necessary Information from Production

Recommended strategy for capturing SQL trace

The baseline performance data SPA uses to judge Oracle 9i SQL performance will come from SQL traces captured on the production system. We believe the production system is a more trustworthy location than any test system for measuring performance, since it contains your real on-line workload. However, if you have a test system running your entire SQL workload, you can do the tracing there instead. This has the advantage that the performance overhead of enabling SQL tracing will not impact your production system.

On the Oracle 9i production database, capture SQL traces with bind values for all of your important SQL statements.

The major requirement for this step is to capture SQL trace with bind values and cover as many of the important SQLs on your system as possible. Be sure to cover any key business transactions and batch jobs. Each SQL need not be covered multiple times – SPA will just consider the first execution seen and ignore the rest. Conversely, any SQL not captured will not be covered by the SPA analysis, so you

should be sure to capture the entire performance-critical workload. Because enabling SQL trace can significantly slow down the SQLs being traced in addition to incurring an expense on I/O system by writing very large trace files, we do not recommend turning it on system-wide on production. Instead, you should enable it a few sessions at a time, cycling through them until covering as many important SQLs as possible.

Even though this strategy prevents any throughput issues, tracing can still impact the response time of the SQLs being traced. It will create additional hard parses for these SQLs when they are first run under tracing and also slow down the SQL execution (chiefly the CPU time) when gathering run-time performance metrics. To give you an idea of how the response time can be impacted, we have conducted an experiment to calculate the overhead of SQL trace for a set of OLTP and DSS queries. We show our full results in Appendix C. To summarize them briefly, we found that the response time impact was less than 20% for the vast majority of SQLs; however, fast-running (< 10ms) SQL statements saw a higher overhead due to the fixed per-execution cost of writing to the trace file. We believe for most systems, if you are following our recommendation of enabling tracing for only a subset of the sessions, this impact on response time should be acceptable.

Enabling SQL trace

Some applications have native support for SQL tracing; so long as your application supports tracing bind values, and it will allow you to cycle through your sessions to capture enough SQLs, it should be easiest to produce an acceptable set of trace files this way. Otherwise, here we outline the preferred ways to enable SQL tracing. Note that the instructions for 9i are different from 10.1, so we show them separately below.

- Release 9i: Add calls to dbms_support.start_trace for some database sessions: this will enable sql tracing in these sessions, as long as they are connected. The dbms_support package needs to be loaded by running the following in sqlplus:

  ```
  @?/rdbms/admin/dbmssupp
  ```

  (see [SUPP] for more information). To enable tracing in the current session, ensure that binds are captured:

  ```
  dbms_support.start_trace(binds => TRUE,
                           waits => FALSE);
  ```

  Call stop_trace when the session should stop SQL tracing:

  ```
  dbms_support.stop_trace;
  ```

  This method works well, but requires altering application code which may not always be possible.

- Release 9i: Invoking the dbms_support.start_ trace_in_session procedure externally to enable sql trace in another session:

```
dbms_support.start_trace_in_session(sid => sid,
                                    serial => ser,
                                    binds => TRUE,
                                    waits => FALSE);
```

  Query v$session to find the sid and serial number of the session you wish to trace:

```
select sid, serial# from v$session where ...;
```

  Use this method when modifying application source is not an option. This will enable tracing for all future SQLs executed by the given session until it disconnects. Call stop_trace_in_session to terminate SQL tracing:

```
dbms_support.stop_trace_in_session(sid => sid,
                                   serial => ser);
```

- Release 10.1: Invoking the dbms_monitor.session_trace_enable procedure to enable sql trace either in the current session or in another session, externally. This is the preferred method to enable SQL trace in release 10.1. Follow the instructions above ("Release 9i: Invoking the dbms_support…") for information on how to find the session id and serial number to use when enabling tracing externally.

```
dbms_monitor.session_trace_enable(session_id => sid,
                                  serial_num => ser,
                                  waits => FALSE,
                                  binds => TRUE);
```

  Call session_trace_disable to end SQL traceing:

```
dbms_monitor.session_trace_disable(session_id => sid,
                                   serial_num => ser);
```

Alternatively, if capturing SQL trace on a test system, you can enable it system-wide with the following command:

- `alter system set events '10046 trace name context forever, level 4'`: this will enable SQL trace for the entire system, for future sessions only, so you can guarantee that all SQLs run will be traced. Disable tracing with "`alter system set events '10046 trace name context off'`".

You can also enable SQL tracing system-wide via the EVENT parameter in your pfile/spfile, which of course requires bouncing the instance:

```
EVENT='10046 trace name context forever, level 4';
```

Again, this is not recommended for production systems because of the volume of tracing generated and because of the performance overhead involved in doing so.

You should set the following parameters to control the behavior of SQL trace:

- timed_statistics = TRUE: crucial for capturing performance timing
- user_dump_dest = '<dir>' : output location for trace files

You can optionally set the following parameter to make it easier to find the trace files once they are generated:

- trace_file_identifier = '<string>' : tag SQL trace files with a string to make them easy to find later

When tracing completes, copy the generated ORA trace files to a safe external location. We will move them to the Oracle Database 11g test system once it is set up.

Create and export a mapping table

The SQL Trace files in Oracle 9i Database refer to objects and users by their numerical identifiers rather than their names. To convert the SQL trace files into a SQL Tuning Set, SPA will need a mapping table to specify which IDs map to which names. We will create this mapping table on the system where the traces were captured and move it to the Oracle Database 11g test system. Create it on the Oracle 9i database system with the following DDL (as a user with SELECT_CATALOG_ROLE):

```
create table mapping_table as
       select  object_id id, owner,
               substr(object_name, 1, 30) name
       from    dba_objects
       where   object_type NOT IN
                 ('CONSUMER GROUP', 'EVALUATION CONTEXT',
                  'FUNCTION', 'INDEXTYPE', 'JAVA CLASS',
                  'JAVA DATA', 'JAVA RESOURCE', 'LIBRARY',
                  'LOB', 'OPERATOR', 'PACKAGE',
                  'PACKAGE BODY', 'PROCEDURE', 'QUEUE',
                  'RESOURCE PLAN', 'SYNONYM', 'TRIGGER',
                  'TYPE', 'TYPE BODY')
     union all
       select  user_id id, username owner, null name
       from  dba_users;
```

After creating the mapping table, export it to a dump file. We will import it into the Oracle Database 11g test system later.

Creating the Oracle Database 10.2 Test System

The Oracle Database 10.2 test system should be built with an environment matching your eventual Oracle Database 10.2 production system.

One of the key components of the SPA experiment is an Oracle Database 10.2 system to measure the post-upgrade performance. SPA will connect to this system from the Oracle Database 11g test system to remotely test-execute the SQLs in the workload. Setting up this system correctly is a critical step in generating reliable Oracle Database 10.2 performance data that can be accurately compared to the Oracle 9i data and used to forecast regressions. To ensure a valid comparison we will need the Oracle Database 10.2 system to resemble the Oracle 9i database system where traces were captured as much as possible.

We recommend the following process for creating this Oracle Database 10.2 test system:

1. Start from a full backup of the Oracle 9i Database baseline system restored in a test environment, using the same configuration as production whenever possible (e.g. init.ora parameters, etc).
2. Upgrade this system to the target Oracle Database 10.2 version (>= 10.2.0.2) for the test and make any environmental changes you intend to make in production in Oracle Database 10.2. The environment of the Oracle Database 10.2 test system should be identical to the environment you intend to use in 10.2 production, i.e. the Oracle 9i Database production environment plus any changes you will make during upgrade.
3. For Oracle Database Releases 10.2.0.2/3/4, install the SPA enhancement on the system ([SPAMET]). Future Oracle Database 10.2 releases will have the enhancement included by default. This system should be licensed with the Real Application Testing option.
4. Grant the EXECUTE privilege on the DBMS_SQLPA package and the ADVISOR privilege to the user whom the database link will connect as.
5. Drop any local PLAN_TABLEs in the schema where the database link will connect. The name PLAN_TABLE should resolve to the global temporary table created by default in 10.2 (SYS.PLAN_TABLE$). It is used by SPA as a temporary storage location for the captured execution plans in the remote db.

Be sure to initialize the following key pieces of system environment properly:

- The Rule-Based Optimizer (RBO) is no longer supported in Oracle Database 10g. If you are using the RBO in Oracle 9i, you will have to migrate to the Cost-Based Optimizer (CBO) as part of the upgrade. See Oracle's Whitepaper on migrating to the CBO for more information ([CBO]).
- DBMS_STATS should be configured in the same way as it will be on production. There have been several critical changes to the default behavior for collecting optimizer statistics in Oracle Database 10g which may have a notable impact on SQL performance. Read the Oracle Whitepaper on Optimizer Changes ([OPT]) and configure DBMS_STATS using the DBMS_STATS.SET_PARAM API:

  ```
  dbms_stats.set_param(pname, pval);
  ```

- Disable the Automatic DBMS_STATS Collection Job and gather statistics manually. This job has been enabled by default in Oracle Database 10g, but it should be disabled on your test system even if you will use it on production. Allowing optimizer statistics to change during the test will make problem diagnosis much more painful. This can be done as follows:

  ```
  dbms_scheduler.disable('GATHER_STATS_JOB');
  ```

  Before gathering fresh optimizer statistics in Oracle Database 10g, you should back up your Oracle 9i optimizer statistics into a table and export it to a dump file for safe keeping. This will allow you to revert back to the old statistics to help diagnose any performance regressions after you upgrade. Use the DBMS_STATS.CREATE_STAT_TABLE/ EXPORT_DATABASE_STATS APIs:

  ```
  dbms_stats.create_stat_table(<user>, 'STATTAB');
  dbms_stats.export_database_stats('STATTAB');
  ```

  Next, gather fresh statistics using the same strategy you will employ on production. Optimizer statistics are gathered during the upgrade for the dictionary only, so you will have to collect them manually for your application data so the SPA test can use the new Oracle Database 10g statistics. For example, to gather statistics for the entire database in a similar way to how the auto job will behave, use the DBMS_STATS.GATHER_DATABASE_STATS API:

  ```
  dbms_stats.gather_database_stats( -
    options => 'GATHER AUTO');
  ```

- Optimizer System Statistics should be collected to take advantage of the optimizer's new CPU costing algorithms. On your test system, you can generate them with the DBMS_STATS.GATHER_SYSTEM_STATS API as follows:

```
dbms_stats.gather_system_stats( -
  gathering_mode => 'NOWORKLOAD');
```

  Refer to the Oracle white paper on optimizer changes ([OPT]) for more information.

- In Oracle Database 10g, PGA memory is managed by default using the PGA_AGGREGATE_TARGET parameter, ignoring values set for the *_AREA_SIZE parameters. If it is not set, it assumes a default value of 20% of your SGA size. If you do not believe this value is sensible for your system, you should set it to your own custom value on the test system just as you will on production.

We recommend building the Oracle Database 10.2 test system from a clone of the Oracle 9i system. This is the best way to ensure that the Oracle Database 10.2 system will be as similar as possible to the Oracle 9i system where the pre-upgrade performance was captured. When SPA remotely executes SQLs on this cloned Oracle Database 10g system, this similarity will give us a high degree of confidence that any differences observed were caused by the upgrade and not an unintended system difference. If the Oracle Database 10g test system is not a full-scale copy of production with identical hardware, it will be necessary to run the experiments on the scaled-down version for both Oracle 9i and 10g databases. See Appendix A for more information.

Creating the Oracle Database 11g SPA System

The SPA experiment itself will be orchestrated from an Oracle Database 11g test system. This can be a small database since it does not need to keep any of the application data. Likewise, the configuration of this system is not critical because none of the performance measurements will be taken here. It does, however, need to have the Real Application Testing option installed locally. The Oracle Database Release 11.1.0.6 database test system will need the SPA enhancement with the new functionality for this use case installed ([SPAMET]). Future Oracle Database 11g releases will have the full SPA functionality installed by default.

To allow SPA to connect to the Oracle Database 10.2 test system, create a public database link from the Oracle Database 11g test system to the Oracle Database

10.2 test system. The link should connect to a user with the ADVISOR privilege and the EXECUTE privilege on the DBMS_SQLPA package to allow for remote test-execution of SQL statements.

Creating a public database link requires the following syntax:

```
CREATE PUBLIC DATABASE LINK mylink CONNECT TO user IDENTIFIED
BY pwd USING 'connect_string'
/
```

All that remains to prepare for the SPA experiment is to load the Oracle 9i SQL trace data into a SQL Tuning Set that SPA can consume directly. Carry out the following steps to create and load the SQL Tuning Set:

- Copy the ORA trace files to the system hosting the Oracle Database 11g database.
- Create a directory object pointing to their location, e.g.

  ```
  CREATE OR REPLACE DIRECTORY MYDIR AS
  '</path/to/traces>';
  ```

- Import the mapping table created on the Oracle 9i production database system.
- Build a SQL Tuning Set with the DBMS_SQLTUNE APIs. You will use the CREATE_SQLSET API to create the STS, and then pass a cursor returned from SELECT_SQL_TRACE to LOAD_SQLSET:

  ```
  declare
    mycur dbms_sqltune.sqlset_cursor;
  begin
    dbms_sqltune.create_sqlset('9i_prod_wkld');
     open mycur for
      select value(p)
      from table(dbms_sqltune.select_sql_trace(
                  directory => 'MYDIR',
                  file_name => '%ora%',
                  mapping_table_name => 'MAPPING_TABLE',
                  select_mode =>
                     dbms_sqltune.SINGLE_EXECUTION)) p;


    dbms_sqltune.load_sqlset(
      sqlset_name => '9i_prod_wkld',
      populate_cursor => mycur,
      commit_rows => 1000);

    close mycur;
  end;
  /
  ```

This will load data from the SQL trace files into the STS. Data from only one execution per SQL will be used. While it would be advantageous for SPA to capture execution frequencies needed to tell frequently-executed SQLs from rarely-executed ones, we cannot get this information from SQL trace since it does not cover the entire cumulative workload. Detailed API examples are given in

Appendix B. Specifying the value for COMMIT_ROWS makes the load API commit after loading every 1000 SQLs. This way you can monitor the progress of the load by querying the DBA_SQLSET/USER_SQLSET views. This can be useful as loading a SQL Tuning Set from a very large trace file can take some time.

RUNNING THE TEST

A SPA experiment is modeled in a database container called a task. Individual sub-experiments are called "trials" within that task.

With the Oracle Database 10.2 and 11g test systems set up, we are ready to run through the principal steps in a SPA experiment: creating the SPA task, converting the statistics in the SQL Tuning Set into a SPA trial, and then remotely executing the SQLs on the Oracle Database 10.2 database to build a second SPA trial. A SPA trial represents a discrete set of performance data generated in the testing workflow -- a set of base measurements for each SQL statement in isolation. The SPA task serves as a container to provide some context around a set of trials and the comparisons you will create among them. This section will guide you through the steps of creating the task and the first two trials, which will contain the necessary performance data to run a comparison in the next step.

SPA has full support in Oracle Enterprise Manager in each release starting with Oracle Database 11g, but the new functionality that supports the Oracle 9i database upgrade currently contains the database enhancements only. For this reason this document refers to the command-line APIs only. We give usage examples for each API here in Appendix B to ease the readability of the document. Enterprise Manager support will be added in a future release.

Creating the SPA Task Container

SPA experiments center around a single database object called a task that manages the state of the current testing being done. We will create a single SPA task and use it for the duration of the upgrade experiment. In each major step we will execute the task to add to the test by creating a new trial. The primary inputs of the task are the SQL Tuning Set with the workload to be tested and a set of parameters dictating how the SPA experiment should be carried out.

Creating a task is a lightweight operation that can be done with the CREATE_ANALYSIS_TASK API in the DBMS_SQLPA package:

```
dbms_sqlpa.create_analysis_task(
  task_name => '9i_10g_spa',
  description => 'Experiment for 9i to 10gR2 upgrade',
  sqlset_name => '9i_prod_wkld');
```

- task_name: name of the task container to create
- description: experiment description

- sqlset_name: name of SQL Tuning Set created from Oracle 9i SQL Trace files (see "Creating the Oracle Database 11g SPA System")

Once the task is created, it will be visible in the USER_/DBA_ADVISOR_TASKS views. We recommend setting the following task parameters which will be used for the testing (see the SET_ANALYSIS_TASK_PARAMETER API):

```
dbms_sqlpa.set_analysis_task_parameter(
  task_name => '9i_10g_spa',
  parameter => [see below],
  value => [see below]);
```

- LOCAL_TIME_LIMIT: the maximum amount of time, in seconds, to allow for executing a single SQL statement in the workload in the remote execution step. This will prevent SPA from getting stuck on one highly regressed SQL for a long period of time. Set it to the longest amount of time any important SQL in your workload should need to execute fully.
- WORKLOAD_IMPACT_THRESHOLD: the minimum threshold, as a percentage, for a SQL to impact the workload before SPA will label it improved or regressed. Set this to zero, since the SQL Tuning Set you will use for SPA does not contain execution frequency information.
- SQL_IMPACT_THRESHOLD: the minimum threshold, as a percentage, for a SQL's per-execution performance to be impacted before SPA will label it improved or regressed (in addition to the workload impact threshold). The default value for this parameter is 1. We recommend increasing it to 5 to compensate for having removed the workload impact threshold and to avoid flooding the report with tiny, irrelevant regressions. Setting these two values will cause SPA to make improvement/regression findings for any SQL whose individual performance increases or decreases by at least five percent, ignoring the rest of the workload.

Create SPA Trial from Oracle 9i Performance Data

The SQL Tuning Set used as an input to the task contains all of the performance data SPA needs for the Oracle 9i production system. For SPA to process this data, we need to model it as a trial execution itself. We build this trial by executing the task in a special mode through a simple API call in the DBMS_SQLPA package. Because this step just copies statistics from the STS into the task and does not involve executing the SQLs, it is also fairly lightweight. Just issue a call to EXECUTE_ANALYSIS_TASK as follows:

```
dbms_sqlpa.execute_analysis_task(
  task_name => '9i_10g_spa',
  execution_name => '9i_trial',
  execution_type => 'CONVERT SQLSET',
  execution_desc => '9i sql trial generated from STS');
```

The execution name passed to the API will be used as a handle for the SPA trial; 'CONVERT SQLSET' is the type of execution for which SPA creates a SQL trial by reading statistics from the STS rather than executing the SQLs anew.

Create SPA Trial from Oracle Database 10.2 Performance Data

To create a SPA trial with your post-upgrade performance data, we will issue another call to EXECUTE_ANALYSIS_TASK on the Oracle Database 11g system, which will remotely execute each SQL in the workload on the Oracle Database 10g system once.

```
dbms_sqlpa.execute_analysis_task(
  task_name => '9i_10g_spa',
  execution_name => '10g_trial',
  execution_type => 'TEST EXECUTE',
  execution_desc => 'remote test-execute trial on 10g db',
  execution_params => dbms_advisor.arglist(
                        'DATABASE_LINK',
                        'MYLINK.SERIES.OF.GLOBAL.SUFFIXES'));
```

Passing task parameters to the EXECUTE_ANALYSIS_TASK API as we do here for the DATABASE_LINK task parameter allows you to specify a value for this execution only. Be sure to pass a global (fully-qualified) name of a public database link for the DATABASE_LINK task parameter.

This step will take significantly more time than the previous ones because it needs to actually execute each SQL on the remote system. You can easily monitor the progress of this step by querying the V$ADVISOR_PROGRESS view on the Oracle Database 11g system as follows:

```
select sofar, totalwork from v$advisor_progress
where task_id = <tid>;
```

The 'sofar' column will indicate the total number of SQLs executed so far, while the 'totalwork' column will show the number of SQLs in the workload.

Generate the Comparison Report

The Nature of SPA Performance Analysis

At this point SPA has made all of the necessary performance measurements for the experiment. In this step, we will use SPA to analyze this data, in which it will step through the workload SQL by SQL, comparing the execution plans and statistics from the two trials. SPA looks for interesting differences between the two, including:

- Performance that has significantly improved or regressed
- Execution Plan changes
- Errors encountered in either trial
- Different number of rows returned due to data differences

Once the analysis completes, SPA can build reports in HTML or Text explaining its findings. First, to perform the analysis, SPA must be given the performance metric that it should use to guide its comparisons. It is only with reference to a particular performance metric that a SQL can be said to improve or regress, since for a given pair of executions, it is possible for one metric to improve and another to regress. For example, if a join changes from a Nested Loops to a Hash Join, you might see an improvement in the number of I/O operations performed but a regression in CPU time. On an I/O-bound system this could be considered an improvement, while on a CPU-bound system this might not be acceptable. SPA needs you to supply the most relevant performance metric to your environment to give it the subjective knowledge needed to judge the change in performance.

Picking the best metric for comparison

Most experts agree that it is important to consider multiple metrics when comparing SQL performance. In the context of a SPA experiment, this can take one of two forms, either performing a single analysis based on a user-supplied formula that combines some metrics (e.g. CPU time + I/O time), or by performing multiple analyses, each one based on separate metrics. We recommend the latter approach because it considers the statistics in isolation and allows you to find the SQLs improving and regressing along either dimension and perform your tuning taking the full knowledge into account. Also, not all performance metrics are equally meaningful, so combining a very reliable statistic with a less reliable one can make it very hard to view the results with a high degree of confidence. A combination along these lines often requires a heuristic-based conversion in units (e.g. from logical I/Os to time based on an estimated time required per I/O) which further corrupts the results.

We believe that good SQL performance analysis focuses on statistics that are both repeatable and comprehensive in what they seek to measure. The first condition, repeatability, is critical in any experiment: if one's measurements change significantly for each collection, it is impossible to say anything definitive about the results that were captured. The second, comprehensiveness, is equally essential: if meaningful aspects of the SQL's performance are not captured in the metrics considered, the analysis is only telling part of the story. With these two goals in mind, we have examined the available performance metrics to find the ones that meet our conditions the best. From the metrics that SPA collects, we will pick a small set that are highly repeatable and, when considered together, will form a complete picture of the SQL's performance.

The table below shows each metric captured by SPA along with its suitability according to the requirements defined above:

Metric	Dimension	Repeatable	Compr.	Comments
PARSE_TIME	CPU Used in Parsing	Yes	No	Should be amortized over multiple execs
ELAPSED_TIME	Time	No	Yes	Very dependent on current environment (e.g. buffer cache state)
CPU_TIME	CPU Used in Executing	Yes	Yes	Very repeatable metric that makes a compr. measurement of a large dimension of SQL perf
USER_IO_TIME	Physical I/O	No	Yes	Comprehensive, but too dependent on buffer cache state
BUFFER_GETS	Logical I/O	Yes	Yes	Repeatable and Compr., but hard to translate to physical I/O
DISK_READS	Physical I/O Reads	No	Yes	Too dependent on buffer cache state
DIRECT_WRITES	Direct Path Writes	No	Yes	Too narrowly focused to be widely useful; dep on buffer cache state
OPTIMIZER_COST	Cost Estimate	Yes	No	Does not seek to measure real-world performance

We recommend concentrating most on the BUFFER_GETS and CPU_TIME SQL performance metrics.

As the table shows, we believe that CPU_TIME is the statistic that can tell the most about a SQL's performance without sacrificing repeatability. Because its comprehensiveness is lacking in the area of I/O, and because the CPU_TIME for the Oracle 9i sql executions is artificially influenced by sql trace, we also recommend analyzing the BUFFER_GETS statistic. While BUFFER_GETS does have important drawbacks in that it is very hard to predict how many logical I/Os will translate to physical ones, where all the cost is incurred, it is our only repeatable I/O statistic so we believe it still merits consideration.

Performing the analysis

We recommend performing two SPA performance analyses, one for CPU_TIME and one for BUFFER_GETS. Creating a SPA analysis is a quick and easy step even for large workloads. We will create two new task trials, one for each statistic:

```
dbms_sqlpa.execute_analysis_task(
  task_name => '9i_10g_spa',
  execution_name => 'compare_9i_102_cpu',
  execution_type => 'COMPARE PERFORMANCE',
  execution_params => dbms_advisor.arglist(
                        'COMPARISON_METRIC', 'CPU_TIME',
                        'EXECUTION_NAME1', '9i_trial',
                        'EXECUTION_NAME2', '10g_trial'),
  execution_desc => 'Compare 9i SQL Trace Performance ' ||
                    'to 10g Test-Execute for CPU_TIME');

dbms_sqlpa.execute_analysis_task(
  task_name => '9i_10g_spa',
  execution_name => 'compare_9i_102_bgets',
  execution_type => 'COMPARE PERFORMANCE',
  execution_params => dbms_advisor.arglist(
                        'COMPARISON_METRIC', 'BUFFER_GETS',
                        'EXECUTION_NAME1', '9i_trial',
                        'EXECUTION_NAME2', '10g_trial'),
  execution_desc => 'Compare 9i SQL Trace Performance ' ||
                    'to 10g Test-Execute for BUFFER_GETS');
```

For each statistic, we recommend generating the following reports in HTML:

- Summary Report
- Detailed report for all sqls with regressed performance

The following report can be generated just once (it will be the same for either trial):

- Detailed report for all sqls with changed plans

Reports are generated with the REPORT_ANALYSIS_TASK API, for example:

```
-- spool reports to files
set heading off long 1000000000 longchunksize 10000 echo off;
set linesize 1000 trimspool on;

-- summary report for CPU_TIME metric
--
spool cpu_summary.html
select xmltype(dbms_sqlpa.report_analysis_task(
                 '9i_10g_spa',              /* task_name */
                 'html',                      /* type */
                 'typical',                  /* level */
                 'summary',                /* section */
                 null,                       /* object_id */
                 100,                         /* top_sql */
                 'compare_9i_102_cpu')  /* execution_name */
               ).getclobval(0,0)
from dual;
spool off
```

Change the `execution_name`, `level`, `section`, and `top_sql` parameters to fetch the reports from other trials and at different levels of detail. The complete API specifics are given in Appendix B. Each report will be a self-contained HTML file that can be opened in any web browser. You can read the summary report to see a broad overview of how the upgrade has impacted your SQL performance. Because we believe CPU_TIME to be the better statistic, we recommend starting with its summary report.

UNDERSTANDING THE RESULTS

There is a lot to look at in the SPA reports you have just built. In this section we go over a few major things to keep in mind when analyzing the results so you can draw the proper conclusions from the report.

Test-Execute Methodology

When SPA performs the remote trial executing SQLs on Oracle Database 10g, it executes each SQL just once in the Oracle Database 10g environment, using just the single set of bind values in the SQL Tuning Set (STS). The SPA analysis compares the statistics observed during this single test-execution to those loaded into the STS from the Oracle 9i sql trace. Since we recommended using the SINGLE_EXECUTION option to the SELECT_SQL_TRACE API in section "Creating the Oracle Database 11g SPA System" above, the SQL Tuning Set will contain information about just a single execution for each SQL. In the context of a SPA comparison, this means that we are comparing a single execution with one set of bind values in Oracle 9i to a single execution with the same binds in Oracle Database 10g. In both cases, the single execution we have measured may not be completely indicative of the average performance for that SQL. For example, some

For each SQL, our SPA experiment has data about only one execution before the upgrade and one execution after the upgrade.

Variance between individual executions can create a margin of error in SPA's conclusions.

SQLs execute much faster with some bind values than others when the different bind values lead to vastly different selectivities. Alternatively, a SQL might run faster in one environment than another if the amount of available memory changes. The larger the per-execution variance in performance for a SQL, the bigger SPA's margin of error will become. For example, if a SQL ran in 0.1 seconds in 9i and 0.2 seconds in Oracle Database 10g, SPA will show a 2X regression, even though the 0.1 second difference is probably well within the margin of error for the measurement. SPA does not compute this margin of error itself because it is very complicated and has significant data dependencies.

You should realize that SQL performance can exhibit significant variance from one execution to the next. Because SPA cannot compute this variance, the workload impacts it reports may be overly optimistic or pessimistic. For this reason we recommend looking at as many changed plans as possible to be sure that any regressions SPA detects will occur for most SQL executions, and that any plan changes it deems insignificant will be so for most executions. By relying on your own knowledge of the system and verifying any suspicions through manual testing, you can focus your tuning efforts on the most important regressions.

Environment Considerations

Differences in environment between Oracle 9i production and Oracle Database 10g test can cause noise in SPA's measurements.

Just as SPA does not make special provisions regarding the margin of error in the measurements it takes, it is also agnostic of the environment in which it executes SQLs. The list of all possible factors that might influence SQL performance is simply too large and varied for SPA to handle them all, so at this point it does not detect environmental changes from one trial to the next. Because SPA will not alert you to any environmental differences, it is all the more critical that the Oracle Database 10g test system is set up with the same environment as your eventual Oracle Database 10g production system – i.e., the Oracle 9i production system's environment plus any changes you intend to make as part of the upgrade. The SPA environment encompasses all performance-relevant SQL parameter values as well as memory configuration and anything else that might impact your SQL performance. For example, if in Oracle 9i you are managing PGA usage with the *_AREA_SIZE init.ora parameters but in Oracle Database 10g you plan to set PGA_AGGREGATE_TARGET to your own custom-tuned value, the Oracle Database 10g test system should have that setting.

Performance Metric Considerations

There are two major reasons why performance metrics captured in Oracle 9i sql trace can differ artificially from the ones observed during Oracle Database 10g test-execution. We discuss them here and explain the biases they introduce into the experiment. Users should keep these factors in mind when looking at the statistics reported by SPA for pre- and post-upgrade performance.

Bias caused by Hardware / Data Differences

In "Creating the Oracle Database 10.2 Test System" above, we recommended building a test system with exactly the same hardware and data as the production system. This is because SPA has no explicit knowledge about the hardware or data for the system, so, when it analyzes the performance it sees, it cannot tell the difference between an artificial regression introduced by a hardware change or a real regression caused by the upgrade. Comparing results gathered on different hardware when you intend to be testing an upgrade will lead to an analysis that is heavily biased against the test system and is almost impossible to understand. If building an exact replica of production is not a possibility, you will need to run both the Oracle 9i capture and Oracle Database 10g remote test-execute on the smaller test system. See our recommendations in Appendix A. The only difference between test and production should be the database version, not the hardware, data, data dictionary, or anything else under our control.

Of course, this rule would not apply were you using SPA to test a hardware change, where you would want to test on a system with the new hardware. SPA can test a hardware change just as effectively as a database upgrade, but the configuration of the test system will differ to support each case. Both experiments would follow the same principle, however: the test system should be a copy of the production system plus exactly the changes you intend to make on production.

Bias caused by SQL Trace

The act of enabling SQL trace on the Oracle 9i system can itself cause a significant impact on the SQL performance we are seeking to measure. When we compare the SQL trace performance to the Oracle Database 10g test-execute performance, we are comparing to a fairly different execution codepath, where SQL trace is not enabled. SQL trace is known to impact performance both by writing tracing information to disk and by introducing extra statistical measurements into the SQL execution. In some cases this can create a very significant bias against the SQL trace data, so consumers of SPA reports should be very aware of this. You should also note that these factors have a bigger impact on some statistics than others.

For this use case, we have recommended examining CPU_TIME and BUFFER_GETS measurements, so we have done some experiments with a set of OLTP and DSS queries to determine how these statistics are impacted by enabling SQL trace. As expected, there was no impact on buffer gets, but the CPU time was affected for most queries. We show our full results in Appendix C. To summarize them briefly, we found that the impact was less than 20% for the vast majority of SQLs; however, fast-running (< 10ms) SQL statements saw a higher overhead due to the fixed per-execution cost of writing to the trace file, and some longer running SQLs experienced a high overhead when many rows were exchanged between intermediate execution plan operations.

SQL tracing itself creates a bias against the Oracle 9i performance data, making any regressions detected by SPA all the more serious.

Thus, the impact of tracing can vary significantly from one SQL to the next, and the profiles for OLTP and DSS workloads look quite different. SPA does not make any adjustment for this impact because it is so unpredictable. This means the CPU comparison is biased against the Oracle 9i data, so any performance regression detected by SPA could potentially be very important, as it was large enough to overcome the bias against SQL trace. For this reason they should all be treated very seriously. SPA might not be able to find the precise CPU impact for all SQLs, but it can quickly point you to the most important regressions. Once you have fixed them you can look to the other plan changes for other potentially regressed SQLs – a SQL for which SPA detects a plan change but no performance change could be a regression that was just not quite large enough to overcome the SQL trace bias. However, regardless of the impact of tracing, SPA will be able to notify you of any plan changes that have occurred, so even if important performance changes are masked by the effect of tracing, plan changes will still be visible and serve as a good source for investigation.

Data changes during SQL tracing

Although we have stressed again and again that the data on your test system should match your production data, it is not always possible for the data seen during SQL tracing to perfectly match the data seen during the remote test-execute trial. If your workload contains DMLs or DDLs, the SQL trace data will be collected while the data is changing, but the SQLs executed during the remote test-execute trial will see an identical set of data. The latter point is guaranteed by the fact that, when we test-execute DMLs, we execute the query part only, so no modifications to the data are made.

The implication here is that, no matter how hard we try, we may not be able to collect execution statistics generated for exactly the same data. Fortunately, SPA can detect data differences when they cause a different number of rows to be returned by the SQL. When this happens, SPA makes a finding that it presents in the detailed reports. You will also see this finding if the production workload does not fetch all the rows for the SQL, since SPA test-execute always consumes the cursor completely. If you see this finding for a SQL, keep in mind that the projection made by SPA may have been affected by the data difference; for example, if a SQL returns 100 rows on Oracle 9i but only 10 on Oracle Database 10.2, it is probably seeing less data, so you should not be surprised if the performance improves. If you see a plan change for such a SQL, you should try executing each plan yourself to verify that no regression is seen when executing on the same data set.

Recommended Comparison Strategy

In the section above entitled "Generate the Comparison Report", we have generated the following five reports, all in html:

- Summary of change impact on CPU_TIME statistic
- Summary of change impact on BUFFER_GETS statistic
- Regressed sqls judging from CPU_TIME statistic
- Regressed sqls judging from BUFFER_GETS statistic
- All SQLs with changed plans

In this section we take you through the recommended strategy for studying these reports, making note of the most important things to look for in each.

Summary Reports

By looking at the summary reports you can quickly see the overall impact of the upgrade and determine the best way to proceed. For example, if the summary report shows that a significant number of SQLs had errors in the Oracle Database 10g test-execution but not in the Oracle 9i sql trace, you would want to investigate why they occurred (perhaps some poor environment setup) rather than beginning to look at the performance data.

Try to tell if the upgrade is helping or hurting your performance overall. If the test was largely a success, get a handle on the number of SQLs that have improved and regressed and see how the change impact is spread across your SQLs. When we tune the regressions in the next step, our actions will be determined by the information you can collect about the various root causes for these issues. On the other hand, if the test seems largely a failure, try to look for signs that the system was not correctly built to match the Oracle 9i system where the traces were collected. If you made a mistake configuring the system, you should correct it and re-run through the last few steps to build a new remote trial and generate new reports before going any further.

Key "not to miss" elements in the summary reports include:

- Overall Impact: the cumulative impact of the change on the workload; in this case, since the SQL Tuning Set has only single-execution statistics, it will be the difference in the sum of the per-execution performance of each SQL in the workload, according to the metric chosen for the comparison.
- Improvement Impact: the workload-level impact for improved SQLs (any SQLs whose performance improved by at least five percent).
- Regression Impact: the workload-level impact for regressed SQLs (any SQLs whose performance regressed by at least five percent). This is the

target that you should seek to shrink to zero, thus making the Overall Impact equal the Improvement Impact.

- Counts of SQLs broken down by plan and performance changes
- Top 100 SQLs ordered by workload-level impact of the upgrade

The following screenshots from a CPU_TIME summary report show an example of how this information is presented:

Report Summary

Projected Workload Change Impact:

Overall Impact	**13.8%**
Improvement Impact	**15.99%**
Regression Impact	**-2.19%**

SQL Statement Count

SQL Category	SQL Count	Plan Change Count
Overall	51	42
Improved	45	39
Regressed	3	3
with Errors	3	0

SQL Statements Sorted by their Absolute Value of Change Impact on the Workload

object_id	sql_id	Impact on Workload	Metric Before	Metric After	Impact on SQL	% Workload Before	% Workload After	Plan Change
111	3sq0m8sruc0jz	1.97%	43783	16262	62.86%	3.14%	10.84%	y
139	bys41b7surq60	1.77%	35010	10358	70.41%	2.51%	6.9%	y
116	54yk0aw1h2b6r	-1.6%	1854	24135	-1201.78%	.13%	16.08%	y

(remaining statements removed)

SQL Statements with Errors Sorted by their object_id (3)

object_id	sql_id	Error Message
112	3tyzbx9vwuu3s	Error in execution '10g_trial': The current operation was interrupted because it timed out.
120	5qqz1p0cut7mx	Type of SQL statement not supported.
151	q4y6nw3tts7cc	Type of SQL statement not supported.

After looking over the summary reports for both CPU_TIME and BUFFER_GETS you should have a good idea of the scope of the upgrade's impact on your SQLs. When the basic test seems successful, the next thing to do is look over each regressed SQL in detail.

Note that the "% Workload Before", "% Workload After", and "Impact on Workload" statistics are computed based on the execution frequency of each SQL statement. Because the execution frequency is not collected from SQL trace data, you should ignore these values and focus on the per-execution statistics ("Impact on SQL" / "Plan Change"). The same is true for performance comparisons done on the detailed reports in the sections that follow.

Regressed SQLs Reports

After getting a broad feeling for the impact of the upgrade and resolving any mistakes made in setting up the environment, you will be ready to begin looking SQL by SQL at the details. The regressed SQLs report is the best place to start for this. It contains the full set of information about those SQLs whose per-execution performance was shown to regress by at least five percent for the chosen comparison metric. It will help you analyze as many SQLs as possible in depth and categorize those regressions so you can make an informed decision about how best to eliminate the regression impact.

The report contains the following information about each SQL:

- Full SQL text and parsing schema
- Full set of execution statistics for Oracle 9i and Oracle Database 10.2 performance (even beyond the comparison metric)
- Execution plans captured both from Oracle 9i sql trace and Oracle Database 10.2 test execution
- Any findings made by SPA during the analysis

The following screenshots show an example for one SQL:

SQL Details:

Object ID	: 116
Schema Name	: DWH_TEST
SQL ID	: 54yk0aw1h2b6r
Execution Frequency	: 1
SQL Text	: SELECT ...

Execution Statistics:

Stat Name	Impact on Workload	Value Before	Value After	Impact on SQL	% Workload Before	% Workload After
elapsed_time	-1.52%	1.867	25.043	-1241.35%	.12%	15.65%
parse_time			.017			1.12%
cpu_time	-1.6%	1.854	24.135	-1201.78%	.13%	16.08%
buffer_gets	-3.52%	53580	2082490	-3786.69%	.09%	20.33%
cost	.3%	105	75	28.57%	1.06%	1.07%
reads	0%	1	0	100%	0%	0%
writes	0%	0	0	0%	0%	0%
rows		344	172			

Findings

1. The performance of this SQL has regressed.
2. The structure of the SQL execution plan has changed.

Execution Plan Before Change:

Plan Hash Value : 1159849121

Id	Operation	Name	Rows	Bytes	Cost	Time
1	HASH GROUP BY		53	4770	105	00:00:02
2	MERGE JOIN		67	6030	104	00:00:02
3	SORT JOIN		2	150	91	00:00:02
4	NESTED LOOPS					
5	NESTED LOOPS		2	150	90	00:00:02
6	NESTED LOOPS		1	41	29	00:00:01
7	MERGE JOIN CARTESIAN		1	22	9	00:00:01
8	TABLE ACCESS BY INDEX ROWID	LU_ELEMENTGROUP_REL	1	11	2	00:00:01
9	INDEX RANGE SCAN	LU_ELEMENTGROUP_REL_IDX1	8		1	00:00:01

Execution Plan After Change:

Plan Id : 115
Plan Hash Value : 3454291636

Id	Operation	Name	Rows	Bytes	Cost	Time
0	SELECT STATEMENT		1	90	75	00:00:01
1	SORT GROUP BY		1	90	75	00:00:01
* 2	TABLE ACCESS BY INDEX ROWID	LU_ELEMENTRANGE_REL	1	15	4	00:00:01
3	NESTED LOOPS		1	90	74	00:00:01
4	NESTED LOOPS		2	150	69	00:00:01
5	NESTED LOOPS		1	41	22	00:00:01
6	MERGE JOIN CARTESIAN		1	22	6	00:00:01
* 7	TABLE ACCESS BY INDEX ROWID	LU_ELEMENTGROUP_REL	1	11	3	00:00:01
* 8	INDEX RANGE SCAN	LU_ELEMENTGROUP_REL_IDX1	8		2	00:00:01

Ordinarily the regressed SQLs report is ordered by the absolute impact of the regression on the workload, multiplying the change in execution performance by the execution frequency to precisely define the relevance of each change. That is not possible in this case because the SQL traces we have collected do not contain meaningful execution frequency information. The SQL traces were not intended to form a comprehensive view of the complete workload: we merely wanted to capture each important SQL at least once. Therefore, the report is ordered by the per-execution change in the comparison metric for each SQL. This means that for this particular use case SPA cannot discern the relative importance of each SQL. You must use your own knowledge of the workload to discern which regressions or plan changes reported are for more important SQLs and which are for less

important ones. Similarly, because the execution frequency is not available, you should ignore values in the "% Workload Before", "% Workload After", and "Impact on Workload" columns as they are computed based on an assumed execution frequency of 1 and instead focus on the "Impact on SQL" column for gauging a change in performance.

Looking over the report, keeping this information and the points from "Performance Metric Considerations" above in mind, first try to eliminate any false positives. SQLs with regressions detected by SPA but for which you do not expect real issues in production do not need further consideration. Next remove any SQLs with insignificant regressions. If a SQL executes only very rarely, or its regression is so small that it will not impact performance in any meaningful way, attempting to tune it is not worth the effort and carries more potential risk than reward.

The remaining SQLs compose all of the meaningful regressions you seek to tune. Separate them into the following separate lists:

a) SQLs with regressed performance for the same execution plan
b) SQLs with regressed performance for a different execution plan

Categorizing your regressions will help you understand their root causes and lead you to possible fixes.

Building these two lists of SQL tuning candidates will help us design a strategy for improving the performance later ("Fixing regressed SQLs"). Within each, as much as you can build sub-groups of similar issues, the more effectively you will be able to fix the problems later. If you notice that a large set of SQLs are suffering from the same underlying issue, you will be able to apply just one fix to solve the whole group's performance problems.

SQLs with Changed Plans Report

Once you have analyzed the known regressions found by SPA, it is a good idea to take a look at as many changed plans as possible, regardless of the impact of the change. While the summary and regressed SQLs reports suggest highly suspicious changes in SQL performance, the changed plans report can point out an additional set of potentially improved and regressed SQLs. Once you have exhausted the set of known issues, it makes sense to do a brief survey of potential issues that the SPA analysis may have missed for any number of the reasons already described here.

After looking at the changes to execution plans, you might find a set for which the change appears problematic. In this case it would be a good idea to do some manual testing to try to seek out conditions (e.g. certain environments, bind values) where the plan change might be harmful. If further testing confirms your suspicion, these SQLs should be added to list b) above. Again, you should take note of any common symptoms that categorize these undesirable plan changes. Even if making an exhaustive list of all SQLs with each change type would be

impractical, these patterns will be very helpful when we seek to eliminate such changes.

FIXING REGRESSED SQLS

After running through the steps of SPA analysis and categorizing the discovered and suspected performance regressions, you are ready to begin tuning the SQLs. Different types of issues will merit different tuning strategies. In this section we go through the major categories of issues you might encounter after this upgrade and recommend approaches for solving each. We recommend a strategy of first categorizing the regressions discovered, then drilling down into each category to discover the *root cause* of the regression, and finally verifying the suspected root cause with further SPA analyses. We will begin the search for the root causes of your plan regressions with the two top-level categories defined in the last step: first those SQLs with regressed performance for the same execution plan, followed by the SQLs with regressed performance and a plan change.

SQLs with regressed performance for the same execution plan

When SQL performance regresses without a plan change, the cause is generally an inconsistency in the execution environment for the two trials. Execution-time behavior changes in the database are generally assured to be improvements in all situations, so we do not expect a large number of regressions in this area. Should there be unintended differences in the two environments, however, performance can regress from one execution to the next. For example, if less memory was available in the test-execute environment or if there was some other workload running on the system in one case but not the other, you could see different performance. If you see a large number of regressions of this type we recommend doing a thorough comparison of the Oracle 9i and Oracle Database 10g database systems to find and correct any environmental mismatches. You should manually reproduce any suspected execution-time regression on both systems before making any changes related to the suspected problem.

In the rare case that execution-time behavior changes are the cause of an important issue, look at a few such SQLs in depth to discover what the issue is and fix the problem by disabling the inappropriate behavior. Contact Oracle Support if you are unsure how to investigate and resolve the issue.

SQLs with regressed performance and an execution plan change

We expect that the bulk of your SQL tuning will pertain to execution plan changes. You should investigate each problem found by SPA to discover its root cause, apply a change to fix it, and then test the effect of that change. Oracle provides a number of different approaches to fixing these problems and it is very important to consider each of them and understand the advantages and disadvantages of each. Problem fixes are generally split between point solutions, which involve a change

Understanding the root cause of a set of regressions is the best way to find an appropriate fix.

made to impact only a single SQL, and more systematic solutions in which many SQLs are impacted by a single fix. Choosing the wrong approach might eventually create more problems than it solves if it does not adapt well to future changes.

The more you can categorize the types of regressions discovered, the better. Large sets of SQLs with similar issues can often be fixed with single changes. Choosing a point solution for a systematic problem forces you to fix the problem once for each SQL suffering from it, resulting in a large number of system changes to solve a single issue. Also, because the SQL trace captured may not contain every SQL in the workload, any systematic issues detected may also impact some unknown SQLs; any point solution used for the problem would then have to be re-implemented after upgrading once these unseen SQLs regress. Making a change that could impact many SQLs to fix just one makes just as little sense – in fixing one SQL you might break one hundred others.

If you find just a few regressed SQLs, we recommend trying a point solution for each. In this section we describe a few possible sources of systematic and point issues and propose some solutions to explore. In the next, "Testing proposed fixes," we explain how to use SPA to be sure the fixes will have their intended effect. Please should refer to Oracle's Whitepaper on Optimizer Changes ([OPT]) for help understanding the root cause of your plan regressions. For additional assistance, contact Oracle Support.

Investigating and solving systematic problems

The most notable changes in Oracle Database 10g that might regress a subset of the SQLs on your system include:

- Optimizer parameter changes: in Oracle Database 10g, the default optimizer_mode has changed to ALL_ROWS because the rule-based optimizer is no longer supported. The optimizer uses dynamic sampling when object statistics are missing, so the default value of optimizer_dynamic_sampling has also changed. It is now done for all unanalyzed tables, with different sample sizes. Both parameters can impact your execution plans. You can test to see whether these are the cause of your regression by reverting to the Oracle 9i default behavior for testing by setting the optimizer_mode and optimizer_dynamic_sampling parameters to CHOOSE and 1, respectively. Note that these cannot be implemented as a permanent fix since the CHOOSE value for optimizer_mode is no longer supported in Oracle Database 10g.

  For help fixing regressions caused by the transition to the cost-based optimizer for some SQLs see Oracle's Whitepaper on migrating to the Cost-Based Optimizer ([CBO]). You can also run the SQL Tuning Advisor on these SQLs, as we describe in the next section ("Fixing problems that impact only a single SQL").

- PGA_AGGREGATE_TARGET: this parameter is now enabled by default in Oracle Database 10g, set to 20% of the SGA size unless overridden explicitly. While it is not strictly an optimizer parameter, it is an important factor in the optimizer's costing model. The amount of memory available when a SQL is executing can have a big impact on the performance of certain operators such as hash joins and sorts, and the optimizer uses this parameter to judge how expensive each should be with the expected amount of available memory. By default the *_AREA_SIZE parameters are ignored in Oracle Database 10g. If you were setting them in Oracle 9i, and you think this might be the cause of some regressions, you can revert back to the old model by setting the WORKAREA_SIZE_POLICY init.ora parameter to MANUAL along with your old *_AREA_SIZE parameter values and see if the regressions disappear. If this is the cause of your issue we recommend manually tuning the PGA_AGGREGATE_TARGET value to come up with one that makes sense for your system or keeping the *_AREA_SIZE/WORKAREA_SIZE_POLICY values.

- DBMS_STATS changes: a number of changes have been made in the default behaviors for how optimizer statistics are gathered, including enabling automatic collection, default histogram collection, and sample size calculation. Any of these could result in plan changes. To test whether your regression was caused by new statistics, you can revert back to your previous statistics and see if the regression is fixed. In the section titled "Creating the Oracle Database 10.2 Test System" we recommended copying the Oracle 9i statistics into a stat table so that, in this step, they could be restored to see if they were the root cause of your regressions. You can do this with the DBMS_STATS.IMPORT_DATABASE_STATS API:

```
dbms_stats.import_database_stats('STATTAB');
```

- CPU costing: by default in Oracle Database 10g the CBO's costing algorithms take the estimated CPU cost into account; the default behavior in Oracle 9i was to cost plans based only on I/O unless you collected system statistics yourself. Unless you gather system statistics with DBMS_STATS, the values used by the CBO might not apply well to your system, leading to important plan changes. If you suspect your issues are due to these changes to optimizer costing, make sure you have collected optimizer system statistics with the DBMS_STATS package. This can be done with the DBMS_STATS.GATHER_SYSTEM_STATS API:

```
dbms_stats.gather_system_stats( -
  gathering_mode => 'NOWORKLOAD');
```

- New optimizer transformations/behaviors: Oracle is always working to improve the optimizer from one release to the next, and we believe that new behaviors introduced will be for the better in general. Still, some SQLs and some workloads may not benefit from a particular change. Common kinds of plan changes may be caused by new optimizer features; you can disable them by setting OPTIMIZER_FEATURES_ENABLE to a previous release and testing with that value, if you do not need to take advantage of any performance improvements in Oracle Database 10g. For help disabling a particular feature, contact Oracle Support.

  To test whether a new optimizer feature might be the cause of your issue, you can set the OPTIMIZER_FEATURES_ENABLE parameter to '9.2.0' and see if the regressions disappear.

If you have trouble determining the cause of a regression covering many SQLs, just make your best guess. Because you are going to test your fixes in the next step ("Testing proposed fixes"), there is no risk in trying something that does not end up solving the problem. The subsequent SPA run will show that you have not fixed the issue, and you can return to try a different fix in the next iteration.

Fixing problems that impact only a single SQL

For SQL statements that appear to be suffering from independent problems, we recommend choosing from the following potential fixes:

- SQL profiles: With the SQL Tuning Advisor in Oracle Database 10g, the system can recommend a new solution, called a SQL profile, when a SQL has a suboptimal plan. We recommend the SQL profile as the first fix to attempt because it is a flexible solution that addresses the root cause of the problem. SQL profiles also adapt well to future situations when data changes might require corresponding changes in the execution plan. Pay close attention to any other recommendations from the advisor – they may indicate a strong alternative that could have a larger benefit.

  You can run SQL Tuning Advisor on one SQL at a time by passing the SQL Text and bind values directly to the command-line API or on multiple SQLs in one step by exporting the SQL Tuning Set to the Oracle Database 10g database and creating a single tuning task for the entire STS. Exporting the STS like this requires a special step because we do not normally support moving an STS from Oracle Database 11g to Oracle Database 10.2. See Appendix B for an example. Note that running the SQL Tuning Advisor requires licensing the tuning pack on the system where it is used.

- Stored Outlines: in the event that SQL Tune could not fix the regression, you can use a stored outline captured from the Oracle 9i production system. You can use the CREATE STORED OUTLINE DDL on the production system to create a stored outline and then export it from the OUTLN schema into the Oracle Database 10g test system, where you can test it with SPA. See the optimizer changes whitepaper ([OPT]) for more information. The stored outline offers stability for the query plan but will not adapt to any data changes in the future or any optimizer enhancements in future releases. Should the plan become more expensive at a later date, the optimizer will continue to pick it without exception until the DBA returns to re-tune the SQL.
- Manual Hinting: as an alternative to creating a stored outline, should the SQL Tuning Advisor fail to produce a SQL profile, consider manually hinting the SQL. For example, you can add an /*+ OPTIMIZER_FEATURES_ENABLE ('9.2.0') */ hint to generate a 9.2 plan. While this approach does require modifying application code, it is more flexible than a stored outline, freezing only a limited number of decisions made by the optimizer rather than permanently forcing it into a single plan. The disadvantage to this approach is that, should it be done incorrectly, further application changes will be required to fix the issue later; however, when done well by someone with knowledge of the data relationships involved, manual hinting can guarantee that certain important optimization decisions are always made correctly.

TESTING PROPOSED FIXES

Testing is not simply a matter of verifying that a proposed change is as good as expected and then implementing it. Often a change will help some SQLs and hurt others, obliging you to make further changes to ensure that no important SQLs regress in production. In this spirit, SPA is meant to be used iteratively. It allows multiple test-execute trials within the same task container and supports comparisons across any pair of trials produced.

After devising a set of proposed fixes to the issues found in the last step, we recommend making those changes on the Oracle Database 10g test system and performing another SPA test-execute trial, followed by a pair of new SPA analysis trials comparing the existing Oracle 9i trial to the new Oracle Database 10.2 trial. Repeat the steps in the sections above and continue finding and testing new fixes until you have eliminated all of the important regressions. Searching for the root cause of a regression can be very challenging and error-prone; testing each possible fix in isolation is the best way to bring sanity to the process.

PREPARING THE PRODUCTION SYSTEM FOR UPGRADE

After completing a sufficient number of SPA iterations, you will converge to a set of fixes for known performance regressions. These should be deployed on the production system at the time the upgrade is performed. Since, with SPA, the

issues and fixes are now known in advance, it is not necessary to wait until the regressions occur and then react.

Most of the fixes recommended here can just be implemented on production as they were on the test system. For the case of SQL profiles, you can avoid the cost of re-running SQL Tuning Advisor on production after the upgrade by simply exporting the SQL profiles created on the test system to the production system via a staging table, using APIs in DBMS_SQLTUNE (see Appendix B). Because SQL profiles express relationships in the underlying data, if your test system does not contain the full production data you should manually test any SQL profiles on production after migrating them.

DEALING WITH UNEXPECTED PROBLEMS AFTER THE UPGRADE

Even the most thorough testing cannot always prevent unexpected issues after the upgrade due to environmental changes, concurrency issues, and test/production differences, among other reasons. The purpose of testing with a scientific tool like SPA is to limit these surprise regressions as much as possible. We recommend licensing the diagnostic pack to take automatic AWR snapshots of performance data after upgrading and comparing them to statspack snapshots taken before the upgrade to understand if (and how) performance has improved or regressed. If there are any regressions, this data will be very helpful in searching for the portion of the workload that accounts for the regression and finding the cause (e.g. plan change).

When SQLs are found to have regressed after upgrading, we recommend running SQL Tuning Advisor. It will search for the root cause of a SQL performance problem and attempt to offer a fix for the problem in a timely fashion. Should it fail to produce a profile, you can attempt to generate a stored outline in a session with OPTIMIZER_FEATURES_ENABLED set to your old release or else manually hint the SQL to revert back to the original plan.

CONCLUSION

Upgrading a production database is a major task for most Oracle customers, requiring many man-hours in preparation and testing. Without the right testing tools and strategies, problems may not be detected until after the upgrade, and the promise of features in the new release can be quickly forgotten from the frustration of just trying to restore the acceptable performance of the previous release. With SQL Performance Analyzer in Oracle Database 11g we have built into the database a reliable, scientific testing tool to help uncover most SQL performance regressions before the upgrade.

To address the needs of our customers currently upgrading from releases Oracle 9i to Oracle Database 10g, we have built support for a new scenario into SPA: comparing SQL performance gathered in SQL trace files on 9i to that from remote

test-executions on Oracle Database 10g. In this document we have explained how you can implement this scenario, how SPA carries out each step in the process, and the things you should keep in mind when viewing SPA results. Armed with this knowledge, many DBAs will, for the first time, be able to accurately and reliably forecast how the upgrade will impact their production system, implementing fixes for problems before they occur in production and avoiding undesirable and expensive downtime.

APPENDIX A: TESTING WITH A SMALLER HARDWARE CONFIGURATION

Throughout this paper, beginning in the section "Creating the Oracle Database 10.2 Test System," we have insisted on the need for performing SPA testing on a Oracle Database 10g test system with the same hardware and data as the Oracle 9i production system. This was essential in the scenario described here because SPA was comparing performance from one of these systems to the other, and if hardware changes were mixed with a database upgrade, it would be impossible to tell which one was the cause of any issues.

Saying this, we do understand that it is not practical for many customers to build a full-scale replica of their production system for testing purposes. Here we offer an alternative scenario for this case to show that SPA can still be used effectively under such constraints. In this case, we recommend building a Oracle 9i test system with a similar environment to production, only at a reduced scale, and running as similar a workload to production as possible with SQL trace enabled. The SPA remote test-execute infrastructure does not exist in Oracle 9i, so use whatever means is at your disposal to execute the SQL workload, whether it be functional tests, a third party tool, or manual scripting. After cycling through the workload, move the SQL trace files produced along with a mapping table to a small Oracle Database 11g test system and load the performance data into a SQL Tuning Set as described in "Creating the Oracle Database 11g SPA System".

Next, upgrade the Oracle 9i test system where the SQL trace was generated to Oracle Database 10.2 and use it for the remote test-execute trial. This way, both the pre- and post-change performance data will be generated on the same system, and you can perform reliable performance comparisons since both sets of statistics were generated in the same environment ("Performance Metric Considerations").

The remainder of this scenario does not differ from the rest of this paper – after completing the two trials in the SPA task, use SPA to analyze the data in terms of CPU and BUFFER_GETS, examine the reports in detail, implement fixes for any regressed SQLs, and test these fixes in addition to the upgrade by creating more SPA trials. This scenario does have the disadvantage that the further the hardware

and data used to test get from production, the harder it will be to accurately forecast what will happen on production, but we believe it is the best option at hand when a full replica of production cannot be built. Even without a full clone of production, testing can still be a very valuable exercise when done right.

APPENDIX B: SPA COMMAND-LINE API EXAMPLES

Loading SQL trace data into a SQL Tuning Set

The following example shows how to read SQL trace files into the database and load them into a SQL Tuning Set to prepare for SPA. We create an STS named '9i_prod_wkld' in the current schema.

Parsing very large trace files can take some time; you can monitor the progress of the load operation by querying the DBA_/USER_SQLSET views. The STATEMENT_COUNT column will indicate the number of SQL statements loaded thus far.

```
-- create a database link to the 10.2 system
CREATE PUBLIC DATABASE LINK mylink CONNECT TO user IDENTIFIED
BY pwd USING 'connect_string'
/

-- create a dir obj pointing to the location of the trace files
create or replace directory mydir as '</path/to/trace/files>';

declare
  mycur dbms_sqltune.sqlset_cursor;
begin
  -- read all ORA trace files in directory 'mydir' into an STS
  dbms_sqltune.create_sqlset('9i_prod_wkld');

  open mycur for
    select value(p)
    from table(dbms_sqltune.select_sql_trace(
      directory => 'MYDIR',
      file_name => '%ora%',
      mapping_table_name => 'MAPPING_TABLE',
      select_mode => dbms_sqltune.SINGLE_EXECUTION)) p;

  dbms_sqltune.load_sqlset(
    sqlset_name => '9i_prod_wkld',
    populate_cursor => mycur,
    commit_rows => 1000);

  close mycur;
end;
/
```

Running SQL Performance Analyzer

After loading the STS we are ready to run SPA. The following example creates a task, performs test-execute on Oracle Database 10g and, finally, generates HTML reports to compare CPU_TIME and BUFFER_GETS changes across trials:

```
declare
  tname varchar2(30);
begin
  tname := dbms_sqlpa.create_analysis_task(
    task_name => '9i_10g_spa',
    sqlset_name => '9i_prod_wkld',
    description => 'Experiment for 9i to 10gR2 upgrade');

  -- no more than 5 minutes to execute each SQL
  dbms_sqlpa.set_analysis_task_parameter(
    task_name => '9i_10g_spa',
    parameter => 'LOCAL_TIME_LIMIT',
    value => 300);

  -- ignore workload impact (only one exec per sql captured)
  dbms_sqlpa.set_analysis_task_parameter(
    task_name => '9i_10g_spa',
    parameter => 'WORKLOAD_IMPACT_THRESHOLD',
    value => 0);

  -- at least 5% impact per execution
  dbms_sqlpa.set_analysis_task_parameter(
    task_name => '9i_10g_spa',
    parameter => 'SQL_IMPACT_THRESHOLD',
    value => 5);

  -- load 9i data into a spa trial
  dbms_sqlpa.execute_analysis_task(
    task_name => '9i_10g_spa',
    execution_name => '9i_trial',
    execution_type => 'CONVERT SQLSET');

  -- collect 10g data through remote test-execute
  dbms_sqlpa.execute_analysis_task(
    task_name => '9i_10g_spa',
    execution_name => '10g_trial',
    execution_type => 'TEST EXECUTE',
    execution_params => dbms_advisor.arglist(
```

```
                        'DATABASE_LINK',
                        'MYLINK.SERIES.OF.GLOBAL.SUFFIXES'));

  -- compare 9i to 10g for CPU
  dbms_sqlpa.execute_analysis_task(
    task_name => '9i_10g_spa',
    execution_name => 'compare_9i_102_cpu',
    execution_type => 'COMPARE PERFORMANCE',
    execution_params => dbms_advisor.arglist(
                          'COMPARISON_METRIC', 'CPU_TIME',
                          'EXECUTION_NAME1', '9i_trial',
                          'EXECUTION_NAME2', '10g_trial'),
    execution_desc => 'Compare 9i SQL Trace Performance ' ||
                      'to 10g Test-Execute for CPU_TIME');


  -- compare 9i to 10g for buffer_gets
  dbms_sqlpa.execute_analysis_task(
    task_name => '9i_10g_spa',
    execution_name => 'compare_9i_102_bgets',
    execution_type => 'COMPARE PERFORMANCE',
    execution_params => dbms_advisor.arglist(
                          'COMPARISON_METRIC', 'BUFFER_GETS',
                          'EXECUTION_NAME1', '9i_trial',
                          'EXECUTION_NAME2', '10g_trial'),
    execution_desc => 'Compare 9i SQL Trace Performance ' ||
                      'to 10g Test-Execute for BUFFER_GETS');
end;
/

-- spool reports to files
set heading off long 1000000000 longchunksize 10000 echo off;
set linesize 1000 trimspool on;

-- summary report for CPU comparison
spool cpu_summary.html
select xmltype(dbms_sqlpa.report_analysis_task(
                '9i_10g_spa',              /* task_name */
                'html',                      /* type */
                'typical',                  /* level */
                'summary',                /* section */
                null,                     /* object_id */
                100,                        /* top_sql */
                'compare_9i_102_cpu')  /* execution_name */
              ).getclobval(0,0)
from dual;
spool off


-- regressed details for CPU comparison
spool cpu_regress.html
select xmltype(dbms_sqlpa.report_analysis_task(
                '9i_10g_spa',              /* task_name */
                'html',                      /* type */
                'regressed',                /* level */
                'all',                    /* section */
                null,                     /* object_id */
                null,                       /* top_sql */
                'compare_9i_102_cpu')  /* execution_name */
              ).getclobval(0,0)
from dual;
```

```
spool off


-- summary report for BUFFER_GETS comparison
spool bg_summary.html
select xmltype(dbms_sqlpa.report_analysis_task(
                 '9i_10g_spa',                /* task_name */
                 'html',                           /* type */
                 'typical',                       /* level */
                 'summary',                     /* section */
                 null,                         /* object_id */
                 100,                            /* top_sql */
                 'compare_9i_102_bgets') /* execution_name */
               ).getclobval(0,0)
from dual;
spool off
-- regressed details for BUFFER_GETS comparison
spool bg_regress.html
select xmltype(dbms_sqlpa.report_analysis_task(
                 '9i_10g_spa',                /* task_name */
                 'html',                           /* type */
                 'regressed',                     /* level */
                 'all',                         /* section */
                 null,                         /* object_id */
                 null,                           /* top_sql */
                 'compare_9i_102_bgets') /* execution_name */
               ).getclobval(0,0)
from dual;
spool off

-- changed plans
spool chgd_plans.html
select xmltype(dbms_sqlpa.report_analysis_task(
                 '9i_10g_spa',                /* task_name */
                 'html',                           /* type */
                 'changed_plans',                /* level */
                 'all',                         /* section */
                 null,                         /* object_id */
                 null,                           /* top_sql */
                 'compare_9i_102_cpu')   /* execution_name */
               ).getclobval(0,0)
from dual;
spool off
```

Running SQL Tuning Advisor

In order to run SQL Tuning Advisor on the Oracle Database 10g system, you can either create a separate tuning task for each SQL or a single one for all regressed SQLs. If you have just a few SQLs, you can create a tuning task for each by pasting the SQL text and bind values into the single statement CREATE_TUNING_TASK API. Here we show examples of the second option, since it is more involved.

Assuming you have more than a handful of SQLs to tune, you will want to use the second option, which requires moving the subset of the SQL Tuning Set (STS) with the regressed SQLs over to that database. This can be accomplished by packing the SQL Tuning Set into an Oracle Database 11g staging table, copying the SQLs into a second staging table that will be compatible with Oracle Database 10g, and moving this second staging table to the 10g system using the data pump client. On the Oracle Database 11g system, first create a table called tuning_candidate_tab with a single column, sql_id, containing the list of SQL_IDs you wish to tune (as built in "Fixing problems that impact only a single SQL" above).

Then run the following to materialize an STS with the subset of the SQLs you intend to tune and pack it into a staging table:

```
declare
  mycur dbms_sqltune.sqlset_cursor;
  basf varchar2(32767);
begin
  -- put the subset of sqls we want to tune into another STS
  basf := 'sql_id in (select sql_id
                      from   tuning_candidate_tab)';

  dbms_sqltune.create_sqlset('sqls_to_tune');

  open mycur for
    select value(p)
    from   table(dbms_sqltune.select_sqlset('9i_prod_wkld',
                                            basf, null,
                                            null, null, null,
                                            1, null,'BASIC')) p;

  dbms_sqltune.load_sqlset('sqls_to_tune', mycur);

  close mycur;
```

```
  -- pack the sts into a staging table
  dbms_sqltune.create_stgtab_sqlset('STGTAB');
  dbms_sqltune.pack_stgtab_sqlset(
    sqlset_name => 'sqls_to_tune',
    staging_table_name => 'STGTAB');
end;
/
```

Then run the following to copy these SQLs into a 10gR2-compatible staging table:

```
exec dbms_sqltune.create_stgtab_sqlset('STGTAB_10G');

insert into stgtab_10g
  (name, owner, description, sql_id, force_matching_signature,
   sql_text, parsing_schema_name, bind_data, bind_list, module,
   action, elapsed_time, cpu_time, buffer_gets, disk_reads,
   direct_writes, rows_processed, fetches, executions,
   end_of_fetch_count, optimizer_cost, optimizer_env, priority,
   command_type, first_load_time, stat_period,
   active_stat_period, other, plan_hash_value, spare2)
  select name, owner, description, sql_id,
         force_matching_signature, sql_text,
         parsing_schema_name, bind_data, bind_list, module,
         action, elapsed_time, cpu_time, buffer_gets,
         disk_reads, direct_writes, rows_processed, fetches,
         executions, end_of_fetch_count, optimizer_cost,
         null, priority, command_type,
         first_load_time, stat_period, active_stat_period,
         other, plan_hash_value, 1
  from stgtab;

commit;
```

Next, move the staging table (STGTAB_10G) to the 10gR2 system using data pump. For example,

```
expdp user/pwd tables=STGTAB_10G version=10.2.0.4
directory=WORK_DIR dumpfile=stgtab10g.dmp
```

<copy file to 10.2 system>

```
impdp user/pwd directory=WORK_DIR dumpfile=stgtab10g.dmp
```

Finally, on the 10g system we can unpack the staging table to rebuild the STS:

```
begin
  dbms_sqltune.unpack_stgtab_sqlset(
    sqlset_name => '%',
    sqlset_owner => '%',
```

```
    staging_table_name => 'STGTAB',
    replace => TRUE);
end;
/
```

With an STS on Oracle Database 10g, running SQL Tuning Advisor and getting a report is simple:

```
declare
  tname varchar2(30);
begin
  tname := dbms_sqltune.create_tuning_task(
    sqlset_name => 'sqls_to_tune',
    task_name => 'tune_regressed_sqls');

  -- 10 mins to tune each SQL
  dbms_sqltune.set_tuning_task_parameter(
    task_name => 'tune_regressed_sqls',
    parameter => 'LOCAL_TIME_LIMIT',
    value => 600);

  dbms_sqltune.execute_tuning_task('tune_regressed_sqls');
end;
/

set heading off long 1000000000 longchunksize 10000 echo off;


-- sql tune report
spool sqltune_report.txt

select dbms_sqltune.report_tuning_task('tune_regressed_sqls')
from dual;

spool off

-- accept a SQL profile
exec dbms_sqltune.accept_sql_profile( -
  task_name => 'tune_regressed_sqls', -
  object_id => <object id from report>);
```

Preparing SQL profiles for export

If you choose to fix some regressions by implementing SQL profiles, you will need to copy those profiles to the production system after upgrading it to Oracle

Database 10.2. To do this you will first pack them into a staging table on the Oracle Database 10.2 test system, and then unpack them on production, as follows:

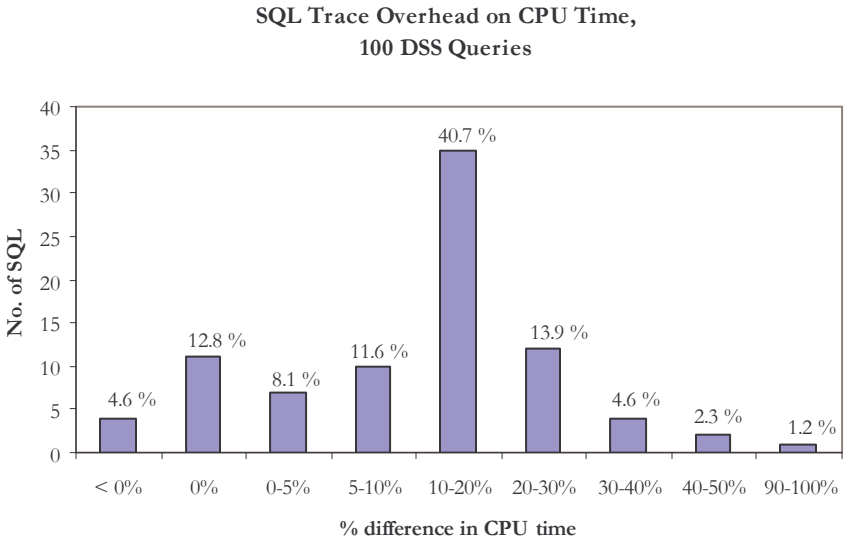
```
begin
  dbms_sqltune.create_stgtab_sqlprof(
    table_name => 'SQLPROF_TAB');

  -- pack all sql profiles, for export
  dbms_sqltune.pack_stgtab_sqlprof(
    profile_name => '%',
    staging_table_name => 'SQLPROF_TAB');
end;
/
```

Then, after upgrading production to Oracle Database 10.2, move the staging table to production and unpack the staging table as follows:

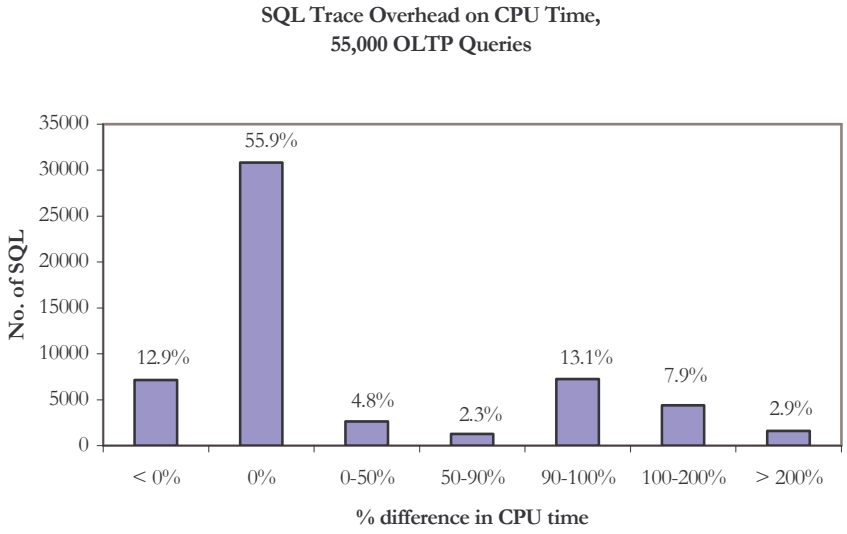
```
begin
  -- pack all sql profiles, for export
  dbms_sqltune.unpack_stgtab_sqlprof(
    replace => TRUE,
    staging_table_name => 'SQLPROF_TAB');
end;
/
```

APPENDIX C: PERFORMANCE IMPACT OF ENABLING SQL TRACE

To understand how SQL trace will impact your production system, and to estimate how much it biases the SPA results against the Oracle 9i data, we have done some experiments with a set of 55,000 queries from OLTP-heavy applications and 100 queries from DSS-heavy ones. We found the impact is almost entirely on the CPU Time statistic:

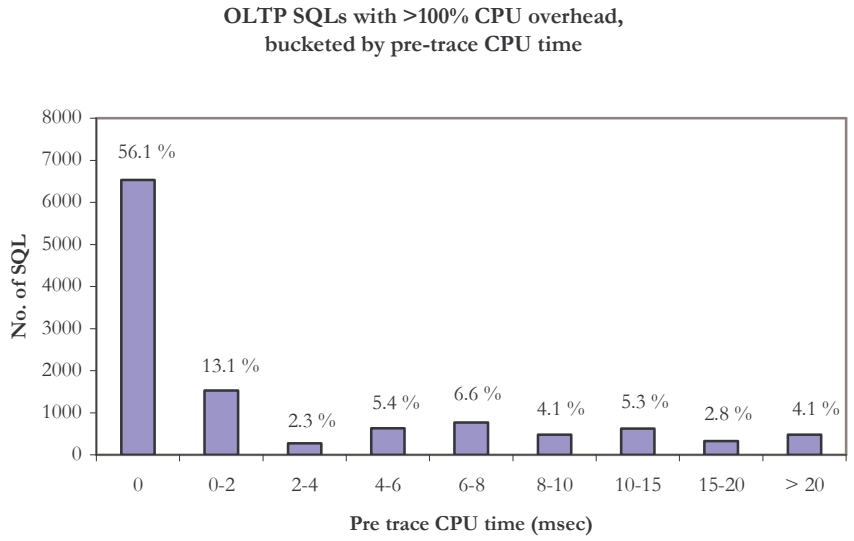


Our DSS results form a more or less bell-shaped curve centered around 10-20% change. These are encouraging results – 77.8% of the SQLs tested saw a SQL trace overhead of less than 20%. This means you can expect SQL trace will not have a drastic impact on too many of your long-running SQLs. The story for OLTP queries is more complicated, however:



For some SQLs, this graph looks even better than the last: 68.8% of SQLs see no discernible impact whatsoever from the sql trace (vs. 17.4% for DSS). Others look much worse, however – 23.9% show regressions over 90% (vs. 1.2% for DSS).

To investigate these highly-regressed SQLs, we looked at how they were distributed in terms of their pre-tracing CPU times to look for a pattern in the data:



These results show that the SQLs with a large overhead almost all had a very small response time: 87.6% of them had response times less than 10 ms. So we can see that most of the SQLs with big regressions have very fast response times. This is because SQL trace has a minimum fixed cost for each execution to the trace files that cannot be avoided. SQLs executing many recursive SQLs (e.g. underneath PL/SQL) saw even bigger overheads because this fixed cost was multiplied many times over for each recursive execution. In contrast, long-running SQLs had their impact amortized over the full execution time so it tended to be less noticeable.

Taken together, these experiments give us a good feeling for the bias SQL trace places on a SQL execution. We see that the overhead varies significantly from one SQL to the next, and it is often noticeable, but it typically does not distort the SQL's performance data to a point that it becomes unrecognizable. When it does have a very large impact, it is usually because the SQL is so short-running that any extra codepath becomes quite noticeable. Also, the patterns for OLTP and DSS workloads are quite different so you should adjust your interpretation according to which pattern your SQLs most closely meet.

REFERENCES

- Real Application Testing Backport for Pre-11g Database Releases: Metalink note #560977.1 [SPAMET]
- Oracle Database Real Application Testing Addendum (Core Documentation): http://download.oracle.com/docs/cd/B28359_01/server.111/e12159/toc.htm
- SQL Performance Analyzer Documentation (Performance Tuning Guide): http://download.oracle.com/docs/cd/B28359_01/server.111/b28274/spia.htm#BABEACFF
- SQL Performance Analyzer (Oracle White Paper) [SPAOOW]: http://www.oracle.com/technology/products/manageability/database/pdf/ow07/spa_white_paper_ow07.pdf
- Upgrading from Oracle 9i Database to Oracle Database 10g: what to expect from the optimizer (Oracle White Paper) [OPT] http://www.oracle.com/technology/products/bi/db/10g/pdf/twp_bidw_optimizer_10gr2_0208.pdf
- The DBMS_SUPPORT Package: Metalink Note #62294.1 [SUPP]
- Migrating to the Cost-Based Optimizer [CBO] http://www.oracle.com/technology/products/bi/db/10g/pdf/twp_general_cbo_migration_10gr2_0405.pdf

- Oracle Upgrade Companion: Metalink Note #466181.1
- Oracle's SQL Performance Analyzer. IEEE Data Engineering Bulletin. March 2008 Vol. 31 No. 1: http://sites.computer.org/debull/A08mar/yagoub.pdf

Oracle Real Application Testing: Testing the SQL Performance Impact of an Oracle 9i to Oracle Database 10g Release 2 upgrade with SQL Performance Analyzer
March 2008
Author: Pete Belknap
Contributing Authors: Benoit Dageville, Prabhaker Gongloor (GP), Shantanu Joshi, Khaled Yagoub, Hailing Yu, Cecilia Grant, Maria Colgan

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200
oracle.com