A look at *Real Application Testing* from a customer's perspective

Jeremiah Wilton
ORA-600 Consulting

As a consultant, I am frequently engaged to help Oracle customers who are experiencing destabilization as a result of a change. It is hard to overestimate the impact to industry in terms of cost and time of IT changes gone wrong. For Oracle sites, instability can result from a wide variety of changes, including version and patchset upgrades. Mostly resulting from optimizer changes, version upgrade instability has resulted in widespread reluctance by Oracle customers to adopt new versions in a timely manner.

Customers have attempted over the years, at significant expense and effort, to develop acceptable testing and benchmarking solutions to help provide peace of mind through changes. The majority of these approaches often did not completely address or identify the impacts of the change, and in the end, many customers came to question the efficacy of the most popular change assurance methods.

During the 11*g* Beta, I combed through the new features guide looking for interesting and promising new features. It was clear to me that 11*g* Real Application Testing would be the feature set that provided the greatest benefit to customers of all new features in 11*g*. The importance of Real Application Testing to customers is so great primarily because it helps assure success through major changes, such as version and patchset upgrades.

Feature overview

Real Application Testing is an umbrella term for two important 11*g* features: Database Replay and SQL Performance Analyzer. These features address change impact in different ways but are largely independent features. For the most part, both features collect production workloads and allow them to be tested in a test database.

DB Replay provides the ability to capture live production workloads into files, and replay that workload faithfully in another database. It is primarily a tool for identifying the potential effects of realistic application concurrency on a changed database. Some examples of potential impacts due to concurrency are latching, locking and resource contention.

SQL Performance Analyzer (SPA) runs SQL statements before and after a change, and compares performance. SPA has the ability to identify both regressed and improved statements, and gage the overall impact of the change on the SQL tested. SPA obtains SQL statements to test from SQL tuning sets (a 10*g* feature), which are collections of SQL statements from a live system. SPA does not run SQL in a way that resembles production workload levels, and also does not run DML. It is a statistical benchmarking tool for the SQL in a database.

Besides version upgrades, Real Application Testing can be used to help ease the impact of large changes like platform migrations, storage reconfigurations, clustering or de-clustering, all activities that have been historically regarded as sources of serious instability. However, these features are also useful for validating small changes such as parameter changes and index addition or removal.

Benefits to Customers

Customers benefit from Real Application Testing by allowing them to make changes to their systems with confidence and agility. Many companies whose databases provide mission-critical services are, with good reason, seriously risk-averse when it comes to change deployment. Change management procedures have become increasingly elaborate and time-consuming over time, and the ability of companies to make changes with ease and agility has suffered.

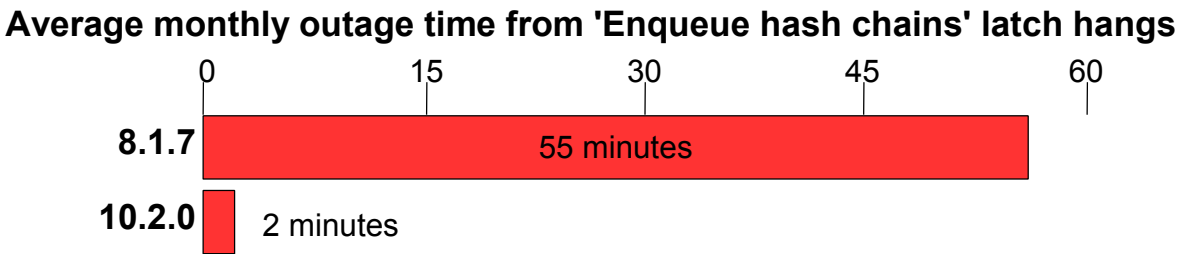
A common misconception in change management holds that the longer and more elaborate the testing cycle, the less the impact of the change will be. The high quality empirical results that are now available from Real

Application Testing features should provide increased stability and availability through changes without an extended and elaborate testing cycle. The confidence in outcome that this feature set provides should offset large portions of the extended change management timeline that were previously devoted to designing and performing elaborate, though often deficient, test plans.

Because of the significant costs and time associated with adopting new versions and features, Oracle customers have often opted to delay version upgrades and feature adoption as long as possible. Such delays do not serve customers well. Often the versions and features that they are avoiding could provide improved service quality, better reliability and scalability, and greater functionality. The cost to the customer of delaying such upgrades and feature adoptions can be significant. One of my own clients provides a dramatic case in point.

One of my clients, an ASP running Oracle to support their application, is very risk averse, and remained on Oracle 8*i* until early 2007. Throughout the lifecycle of 8*i* at this site, the customer had lived with occasional, but persistent whole-instance hanging incidents lasting from 60 to 120 seconds. After upgrading to 10g, they continued to experience the hangs, but were immediately able to view Active Session History (ASH) data from the period of the hang, and back track to the first moments of the incident.

Within the first day my client was on production 10*g*, ASH data revealed many sessions waiting on 'enqueue hash chains' during the hang, with the genesis of the incident coinciding with Oracle detecting a deadlock and raising “ORA-00060: Deadlock detected...” in one session. This led them to discover the fact that a session dumping processtate after receiving ORA-60 can hold the 'Enqueue hash chains' latch for the duration of the dump. They subsequently set event 10027, which changes deadlock detection so that it does not take a processtate dump with the ORA-60 trace.



This anecdote demonstrates the cost of failing to upgrade. Had the customer upgraded to 10.2 earlier, they would have had far fewer outages, and would not have incurred the significant cost in revenue and quality of service that they did under 8*i*. This customer's reluctance to upgrade was above all due to an inability to empirically predict the potential instability that might have arisen from the upgrade.

Whether it improves a site's service quality by allowing faster adoption of new versions and features, or whether it does so by reducing or eliminating the destaboilizing effects of change, Real Application Testing promises to dramatically improve the quality of service that we as customers and third parties can provide. By enabling customers to provide higher quality services, customers will see their investment in Oracle technology as increasingly valuable.

Benefits to Oracle

The benefits of Real Application Testing do not confer solely to the customer. Oracle currently supports customers on database versions going back many releases. Even after a version is desupported, Oracle continues supporting customers who purchase extended support on that version for years. This complicates Oracle Support's ability to effectively serve all customers, diffuses that knowledge needed to support customers, and increases the cost of support. If Real Application Testing results in more customers upgrading to newer versions sooner, then Support and development will have a simpler job and a narrower focus, which can only improve the Quality of Oracle support from a customer perspective.

Oracle Support will hopefully also benefit from customers encountering fewer destabilizing changes that result in high severity service requests. If customers are able to thoroughly and empirically test their changes before

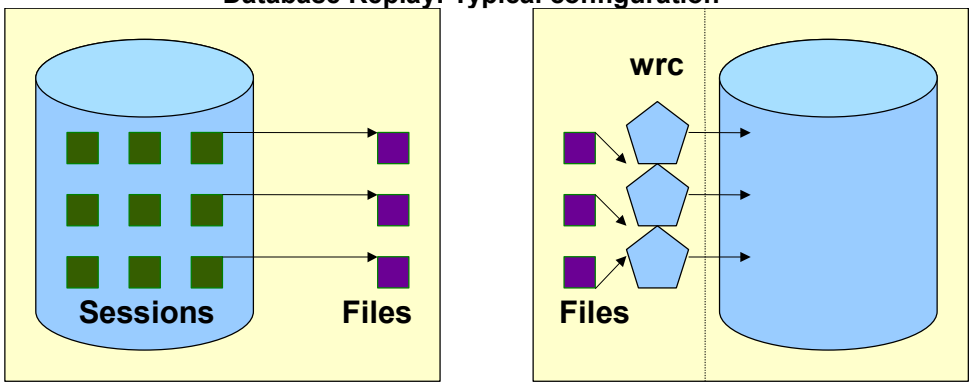
deploying them, then far fewer changes should result in SEV-1 calls to support.

Finally, since Database Replay provides easy access to replay realistic customer workloads against a test database, many more bugs should be discovered in those test databases rather than in production systems. If all of these factors allow customers to adopt new Oracle versions sooner, then more critical and destabilizing bugs should be found in early releases of a new version, compared to previous versions that did not have Real Application Testing.

Feature technical details: Database Replay

Database Replay provides a workload capture feature beginning in 10.2.0.4 and faithful replay facility beginning in 11.1.0.6. The functions of DB Replay are mainly managed by two new PL/SQL packages, dbms_workload_capture and dbms_workload_replay. DB Replay causes running sessions to write their activity into files that can later be cobbled together into a replayable workload. The workload, stored in files, can be easily copied among hosts. A new binary, the Workload Replay Client (wrc) runs the workload from local or remote hosts, and behaves more or less as your application would vis à vis the database.

Database Replay: Typical configuration



Capture workload on production **Replay workload in clone/test**

To demonstrate the basics of DB replay, we can walk through a typical use case, beginning with the workload capture. This use case does not cover all options available for each step of the process. Those hoping to use this feature should thoroughly review the Oracle documentation and become familiar with all available options.

- **First, create a directory for the capture to save files.**

```
SQL> create directory wloadcap as '/opt/oracle/workload';
Directory created.
```

- **(Optionally) Add a filter to limit what is captured. T**hese filters are either INCLUDE or EXCLUDE filters, meaning they allow the inclusion or exclusion of a user (schema), module, action, program, service or instance from the capture. You can specify multiple filters.

```
SQL> exec dbms_workload_capture.add_filter( -
             fname=>'RTUSERFILTER', -
             fattribute=>'USER', -
             fvalue=>'RT')
PL/SQL procedure successfully completed.
```

- **(Optionally) Shutdown/startup restrict.** This provides a consistent start point for all transactions. If you decide not to begin capture with a freshly started database, then capture may record just the ends of transactions that were already started. This creates the possibility that replay will produce errors, complicating the change analysis. Certainly, many sites will not be able to stop and start their database during the peak load periods that they want to capture, so those sites will accept the minor inconvenience of errors resulting from partial transactions being captured at the start of the workload.

- **Start the capture.**
```
SQL> exec dbms_workload_capture.start_capture( -
             name=>'OOW07', -
             dir=>'WLOADCAP', -
             default_action=>'EXCLUDE')
PL/SQL procedure successfully completed.
```

- **Allow work to transpire, then end the capture.** The period of capture should span such time period as you feel is a typical or peak workload for those components you wish to test with replay.
```
SQL> exec dbms_workload_capture.finish_capture
PL/SQL procedure successfully completed.
```

- **Obtain the start SCN of the capture from** `wrr$_captures` or `<dir>/wcr_cr.text`. You need this piece of information so that you will know the exact SCN to which you should recover your target or "replay" database prior to performing replay.

These same procedures may also be performed through Enterprise Manager (Grid Control or Database Control). The EM interface allows the user to specify all options available in the PL/SQL API. Some users prefer to use the PL/SQL API since it allows us to script and repeat out activities predictably. In addition, network restrictions and security rules prevent many customers from using EM.

In EM, the "Set Up Workload Capture" screens provide access to the capture functionality, allowing the user to name the capture, specify a directory, specify filters, and start the capture:

- Once the capture is complete, you must process the capture files before replaying the workload. The replay database must be consistent with respect to the start SCN of the capture. The simplest way to accomplish that is probably to clone database with RMAN:
```
RMAN> duplicate target database to clonedb until scn <scn>...
```

- Moving the workload is as simple as copying the contents of the capture directory to the testing host:
```
$ scp /opt/oracle/workload/* clonedb.host:/opt/oracle/workload/
```

- The next step is to process, initialize and prepare the capture data on replay database.
```
SQL> exec dbms_workload_replay.process_capture( -
             capture_dir=>'WLOADCAP')
SQL> exec dbms_workload_replay.initialize_replay( -
             replay_name=>'OOW07', -
             replay_dir=>'WLOADCAP')
SQL> exec dbms_workload_replay.prepare_replay
```

- On the workload generation host(s), start the Workload Replay Clients. The workload can be generated from the local database host or separate machines. For most scenarios, a more realistic load can be produced using separate hosts. Starting these client processes does not begin the replay. These processes log into the replay database and monitor the status of the replay job, waiting to start replay until the replay is started from the database side.
```
$ wrc userid=system password=foo replaydir=/opt/oracle/workload workdir=/tmp
```

- The next call from the replay database actually starts the replay
```
SQL> exec dbms_workload_replay.start_replay
```

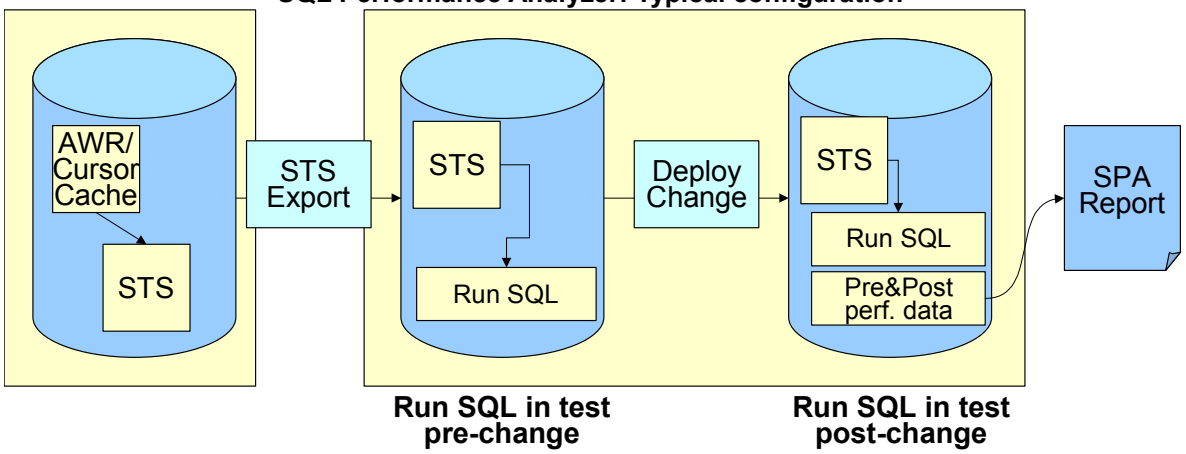
- A series of changes, modifications or parameter settings may be tested one after another if you use Flashback Logging and Flashback Database. After one replay is complete, export any performance statistics you want to analyze for comparison, then simply make more changes, flash back, and start all over again
```
SQL> shutdown
SQL> startup mount
SQL> flashback database to scn <scn>;
SQL> alter database open resetlogs;
```

As with capture, Enterprise Manager provides the ability to run replays using the full set of options avalable via the PL/SQL API. In addition, it provides a very useful monitoring screen, “View Workload Replay,” for tracking the progress and effectiveness of a replay session.

Feature technical details: SQL Performance Analyzer

SQL Performance Analyzer (SPA) leverages an existing 10*g* feature, SQL Tuning Sets, and allows the benchmarking of SQL statements in those sets before and after a change. In most cases, a SQL Tuning Set collected in production is exported to a test system for SPA analysis through a change. SPA tracks and reports increased resource usage, differences in timing and wait events, and execution plan changes after a change.

SQL Performance Analyzer: Typical configuration



You can collect SQL tuning sets on any version of 10*g*, and compare performance with SQL Performance Analyzer starting with version 11.1.0.6. Oracle supports customers using SPA for 10*g* to 11*g* upgrades. Oracle suggests performing the before-change analysis on 11*g* with the optimizer_features_enable parameter set to 10.x, then setting optimizer_features_enable back to 11*g* for the post-change analysis. I will demonstrate just suck a scenario in the following example.

- To create and populate a SQL Tuning Set, first create a logical SQL tuning set object. This is simply a container for the set data.

```
SQL> exec dbms_sqltune.create_sqlset( -
>            sqlset_name=>'OOW07_SET', -
>            description=>'Demo set for OOW 2007')
PL/SQL procedure successfully completed.
```

- Next we populate the tuning set with all SQL executed within a specified time limit. The quality of this tuning set determines effectiveness of analysis. In most cases, a longer timeframe produces a higher qualist set of SQL, in terms of completeness. This example uses 'incremental' STS capture, directly from the live cursor cache.

```
SQL> exec dbms_sqltune.capture_cursor_cache_sqlset( -
>            sqlset_name=>'OOW07_SET', -
>            time_limit=>300, -
>            repeat_interval=>30)
```

- The PL/SQL call will not return when using live 'incremental' capture until time_limit has expired.

```
...
PL/SQL procedure successfully completed.
```

As with DB Replay, SPA functionality and tuning set creation may all be managed through Enterprise Manager Grid Control/DB Control. The same costs and benefits apply here as with DB replay. EM provides a richer and more intuitive interface, but is not scriptable and repeatable in the same way that PL/SQL API calls are.

- Once you have a SQL tuning set in the database where you want to perform before and after-change analysis, you can use DBMS_SQLPA to use the SPA functionality. The first step is to create an analysis task, which once again, is just an empty container for holding the analysis you are about to perform.

```
SQL> variable a varchar2(100)
SQL> exec :a := dbms_sqlpa.create_analysis_task( -
>                  sqlset_name=>'OOW07_SET', -
>                  task_name=>'OOW07_TASK')
PL/SQL procedure successfully completed.
```

- Prior to deploying the change for the purposes of the SPA testing, we collect the pre-change performance of each statement in the SQL tuning set. This is accomplished by *executing* the analysis task we just created. SPA runs each SQL statement once and collects session statistics, waits and execution plans.

```
SQL> exec dbms_sqlpa.execute_analysis_task( -
>            task_name=>'OOW07_TASK', -
>            execution_type=>'TEST EXECUTE', -
>            execution_name=>'OOW07_PRE')
PL/SQL procedure successfully completed.
```

- The next step is to make some kind of change that could impact SQL performance. In this example we are changing optimizer_features_enable from 10.2.0.2 to 11.1.0.6, to simulate a 10g to 11g upgrade.

```
SQL> alter system set optimizer_features_enable = '11.1.0.6' scope=memory;
System altered.
```

- After making the potentially impactful change, we perform an execution of the analysis task to collect the SQL statement performance after the change.

```
SQL> exec dbms_sqlpa.execute_analysis_task( -
>            task_name=>'OOW07_TASK', -
>            execution_type=>'TEST EXECUTE', -
>            execution_name=>'OOW07_POST')
PL/SQL procedure successfully completed.
```

- At this point we have collected two executions of the analysis task. One was generated before the change to optimizer_features_enable, and one was generated after. A final execution performs the comparison of the first two executions.

```
SQL> exec dbms_sqlpa.execute_analysis_task( -
>            task_name=>'OOW07_TASK', -
>            execution_type=>'COMPARE PERFORMANCE', -
>            execution_name=>'OOW07_COMP', -
>            execution_params => dbms_advisor.arglist( -
>               'comparison_metric','buffer_gets'))
PL/SQL procedure successfully completed.
```

- A very useful component of SPA is the analysis test reporting function, which details the improvement or regression overall, and for the specific outlying SQL statements that have either improved or regressed most. The SPA report weights SQL statements appropriately accoring to number of executions. Even though SPA only benchmarked the SQL once, the time and resources consumed by a SQL statement that ran 1000 times during the sample period is multiplied by 1000.

```
SQL> set long 100000 head off longc 100000 lines 130
SQL> variable rep clob;
SQL> exec :rep := dbms_sqlpa.report_analysis_task( -
>                   task_name=>'OOW07_TASK', -
>                   type=>'text', -
>                   level=>'regressed', -
>                   section=>'summary')
PL/SQL procedure successfully completed.
```

```
-- Display the report
SQL> print :rep

Projected Workload Change Impact:
-----------------------------------------
 Overall Impact      :  -2.24%
 Improvement Impact  :  0%
 Regression Impact   :  -2.24%

SQL Statement Count
-----------------------------------------
 SQL Category  SQL Count  Plan Change Count
 Overall             172                  9
 Regressed             1                  1
 Unchanged           162                  8

Projected Workload Performance Distribution
-------------------------------------------------------------------------
|             | Cumulative Perf. |        | Cumulative Perf. |        |
| Bucket      | Before Change    | (%)    | After Change     | (%)    |
-------------------------------------------------------------------------
| < = 1       |            28535 |     0% |            28537 |    0% |
| < = 2       |            38228 |     0% |            38236 |    0% |
| < = 4       |           242070 |     0% |           241986 |    0% |
| < = 8       |            51888 |     0% |            51978 |    0% |
| < = 16      |             2539 |     0% |             2513 |    0% |
| < = 512     |         20566775 |   .24% |         17292752 |   .2% |
| < = 1024    |             6417 |     0% |             6390 |    0% |
| < = 2048    |             1559 |     0% |             1535 |    0% |
| < = 1048576 |       8580458824 | 99.76% |       8772888012 | 99.8% |
-------------------------------------------------------------------------

SQL Statements Sorted by their Absolute Value of Change Impact on the Workload
-------------------------------------------------------------------------------------------------------
|               | Impact on | Metric | Metric  | Impact   | % Workload | % Workload | Plan   |
| sql_id        | Workload  | Before | After   | on SQL   | Before     | After      | Change |
-------------------------------------------------------------------------------------------------------
| f0cxkf0q803f8 |    -2.24% | 948746 | 970023  | -2.24%   |     99.76% |      99.8% | y      |
-------------------------------------------------------------------------------------------------------
```

The above report, based on the tiny 5-minute sample we collected in our SQL tuning set, revealed one of the 172 SQL statements collected has regressed upon switching to 11*g* optimizer logic. This type of information would be extremely useful to someone performing such an upgrade, because it would allow the user to focus on that SQL statement, address the regression under 11*g*, and possibly store an outline to prevent regression upon upgrade.

Life before Real Application Testing

Before the availability of Real Application Testing, customers user a variety of methods to try to quantify the effect of changes before deploying them in production. The methods that customers have used vary greatly, but all have one thing in common. They all were far less accurate and easy to use than DB Replay and SPA. Because customers could not make the fundamental changes to Oracle needed to collect accurate workloads, all of the old methods were imperfect. In the next section we will explore the relative imperfections in the various methods.

Commercial Synthetic Loaders

The most common approach to change validation under load prior to Database Replay was to use a commercial synthetic loader such as LoadRunner. With synthetic loaders, workload is collected at the application layer. Whereas DB Replay collects and replays all external workload that a database is subjected to, synthetic loaders collect only that work that takes place in a particular application layer and replays it through that same application layer.

The strength of synthetic loaders is their generic capabilities. They can provide a testing solution not only for the underlying database, but also for the application service, and all other services upon which that application

depends. For Oracle database changes, however, synthetic loaders are typically too generic a solution. There are basically two major drawbacks to using synthetic loaders when compared with DB Replay for database load testing and change verification:

- To successfully use a synthetic loader that drives an application layer, all services upon which that application depends must be built out to scale in a test environment.
- Because synthetic loaders only capture application-related load, all load to the database not related to the application you are capturing will be missed.

Most applications rely upon more than one service. For instance, an application may connect to two or three database services over the course of normal operation. In addition, an application may require other services for basic operation, such as an SMTP gateway or NFS services. In order to build a working load simulation for the whole application, then all services upon which the application depends must also be built out in the test environment. This increases the complexity, cost and time associated with the build-out needed to accomplish testing.

In the best case, if a single Oracle service serves a single application, and that application relies upon no significant other services, then a synthetic loader may acceptably drive the majority of the database load. However, it will not include a wide variety of ancillary load typical to a database, such as monitoring, ad-hoc SQL queries, regular DSS extracts, any and all applications that connect to the database but are not part of the main application. The high probability that collecting and replaying workload at the application layer leaves many loose ends and increases both the complexity of implementing such a solution and the probability that the solution will be incomplete.

Synthetic Loader case study: 9*i* to 10*g* upgrade

One of my clients, a major wholesale distributor of CDs and DVDs run several large fulfillment centers on home-grown applications. Each fulfillment center has its own Oracle database service. The responsiveness of the Oracle service is of paramount importance to this customer, since any time a call to the Oracle service exceeds one second, the machinery of the production line stops and all work ceases. In such situations, hundreds of workers are idled, and production stops, costing the company significant sums while the Oracle issue is worked out.

Needless to say, the customer was leery of upgrading from Oracle 9*i* to 10*g*, but reassured themselves prior to the first fulfillment center upgrade by deploying and running an elaborately constructed test using LoadRunner to drive their application. They discovered many potential performance problems using this approach. Much to their alarm, despite having addressed all problems exposed in their testing, large numbers of sessions waiting on 'enqueue' caused periodic and extended production outages after the upgrade. It was agter this initial upgrade and subsequent failures that the client retained me to assist with subsequent upgrades.

Despite an eight-week build-out and execution, the test they had constructed had not included jobs run by a third-party scheduler, external to the core application. One of those jobs periodically ran a SQL statement that performed a 'select ... for update...' The SQL was fast and unintrusive on 9*i*, but regressed badly in 10*g* because of new optimizer and statistics collection logic. In this case, the synthetic loader provided a solution that, despite exposing 90% of potential performance problems, was insufficient to prevent costly and destabilizing outages after change deployment.

Comparison of Synthetic Loader vs. DB Replay for case study

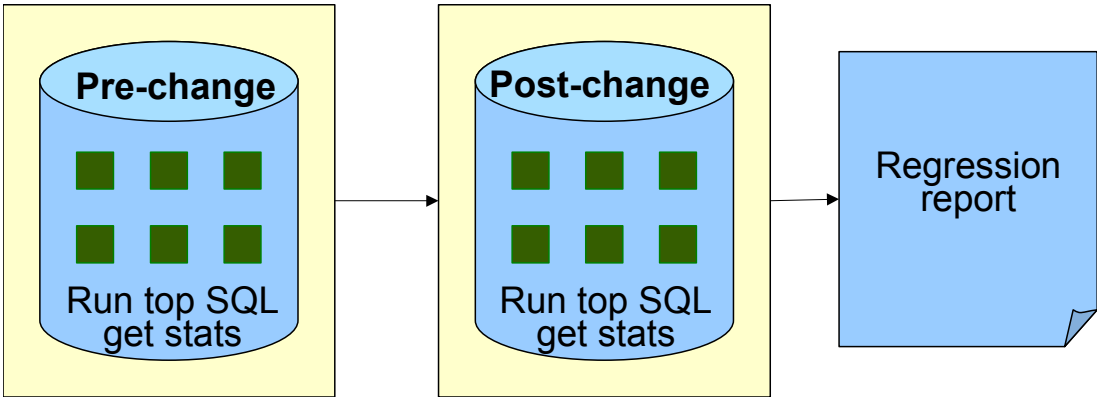
	Load Runner	**DB Replay (hypothetical)**
Capture time	8 weeks	4 hours
Workload completeness	80%	100%
Databases needed	3	1
Accuracy of results	90%	100%

Notably, both DB Replay and SPA would have effectively exposed the issue that ultimately caused the post-upgrade regressions. DB replay would have generated a locking pile-up similar to the one observed in production, and SPA would have detected the regressed SQL statement and reported it.

Pre-SPA SQL benchmarking

Another popular approach before Real Application Testing was to poll v$sql or AWR tables to construct a more or less complete set of SQL running on a database service. That SQL could then be benchmarked in test before and after a change, collecting data from v$sesstat and v$session_wait, as well as execution plans and elapsed time. Those statictics could then be compared to expose regressions and improvements. This approach had two main drawbacks. First, it required significant coding on the part of the customer to build a test harness and coherent statistics recording mechanism. Second, there was no good way to substitute representative bind variables in SQL. So customers who used few bind variables were served better by this approach than those who did.

Overview of Pre-SPA SQL benchmarking



SQL Benchmarking case study: 8*i* to 10*g* upgrade

My client, an ASP providing their in-house built applications to customers, was approaching end-of-life on 8*i*, and needed badly to upgrade to 10*g* to remain supported and to leverage the many advanced features that had been introduced since 8.1.7. The customer's application was responsible for membership, list and content management for a variety of nonprofit organizations and corporations, as well as response and campaign tracking. This database-intensive web application was similarly deployed on multiple sets of vertical systems, so any strategy that they developed for validating upgrade would have to be repeatable.

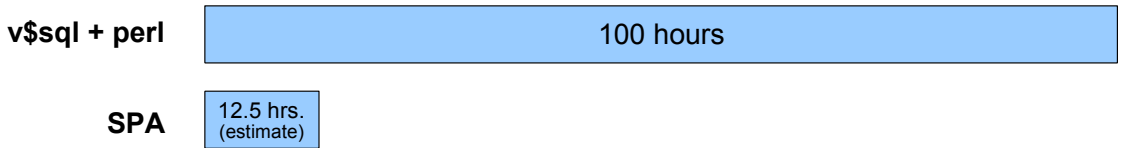
I built a benchmarking tool for the client in Perl using data from v$sql and statspack tables to obtain the SQL. For the first time in my engagement, I was relieved that few of their SQL statements contained bind variables. This fact, which had been the bane of their shared pool for years, suddenly paid off in terms of ease of testing. I found the few statements that contained bind variables, and in consultation with application developers, fudged values for them. My Perl testing harness initiated two connections to the database. One connection ran the SQL statements, while the other recorded v$sesstat and v$session_wait values for the benchmarking session for each statement run. This approach was about as close as someone without direct SGA sampling capabilities could come to simulating SPA functionality in 8*i*.

I ran the benchmark first on their 8i production system, then on their 10g test system, and compared statistics. For several statements that appeared to regress, we invistigated and addressed the causes. The method allowed me to generate an overall estimate of impact from upgrade.

The upgrades were successful, although there were a couple non-critical SQL statements that became heavy resource hogs which the v$SQL/statspack method had missed. These turned out to be monthly jobs that had not been running during any period for which we had obtained collections of SQL. In the end, overall host utilization dropped after upgrade to 10*g*, in part due to the ease of diagnosis of inefficient SQL using Active Session History.

The entire benchmarking project took 100 hours to build and run on all environments, which is far longer than would have been needed if using SPA in a hypothetical analogous situation today. Of course, SPA was and is not available for 8*i*, but for the purposes of assessing the value of SPA as a feature, I have estimated that with SPA I would have spent about 12.5 hours obtaining the same, if not more accurate data. This would have saved the customer many hours in my fees, and all binds would have been accurately collected.

SQL Benchmarking: With and without SPA



Another Pre-11g testing approach: SQL*Trace, parse and run

A oft-cited approach prior to 11*g* for obtaining representative workload was to enable event 10046 at level 4 (SQL+binds) or level 12 (SQL, Binds and waits). A script could then be written to cobble together all the statements from the various trace files into a serial stream. More homegrown code could then be written to run that SQL in a pre and post-change environment.

Despite the frequency with which this method seems to have been cited as a valid approach in the past, I have only ever seen it implemented, and then only partially, at one client site. The main problems with trying to use SQL*Trace data are the prohibitively large performance cost of enabling event 10046 at *any* level in a production instance, and the inability to order dependent operations across multiple sessions when replaying them.

For most environments, even moderately loaded systems cannot sustain the additional wait time and resource cost of enabling event 10046 in every session. Typical fallout includes 100% CPU utilization and unaaceptable service times for database calls by applications.

The lack of ability to parallelize replayed workload generated by event 10046 further condemns this approach to the scrapheap. Extensive engineering went into the capture and replay functions of DB Replay so that the workload could be replayed using the same number of sessions as were involved in the capture. SQL collected with 10046 cannot be reconciled based solely on time index and wait events with other potentially dependent transactions from other sessions during replay. If a transaction on one session regresses and takes three times as long, the 10046-based benchmark has no ability to delay dependent transactions so as to avoid data divergence and resultant errors.

In the one client site where I saw 10046 used as a driver for workload collection and replay, no attempt to achieve realistic wait times or parallelism was made. One of the clients DBAs spent almost all of three months building the collection and replay scripts. They suffered degraded performance during workload collection, and in the end the testing failed to expose the impacts of concurrency on the changes they were trying to validate. Not only was this approach not comparable in the least to DB Replay, but using this method no tool could be. Any 10046-based approach cannot produce a comparable tool to DB replay, because a faithful simulation of production concurrency is not possible to produce before the availability of DB Replay.

Still more suboptimal approaches

I have encountered a wide variety of other approaches to change validation that have been tried or proposed at various times in my work with clients, participation in online forums or through word of mouth. Some of them are:

- **Collect workload with Fine-grained Auditing**
  This method proposes to audit all SQL then cobble the statements into a script for replay. I can't personally estimate the resource cost of this approach, but some significant overhead is inevitable. While partially effective for collecting SQL for individual statement benchmarking, there is no FGA data that would allow for replay using multiple concurrent sessions or with any type of realistic wait times between calls.

- **Implement hooks in the application to log all SQL issued to the database**
  This approach shares many of the same shortcomings as synthetic loaders. Namely, it must be implemented in each and every source of external work or potentially impactful workload will be missed. It also requires some significant work on the part of the customer to code such logging mechanisms. As with most of the pre-11*g* approaches, no cross-session transactional dependency can me coordinated during replay, so the approach is only good for collecting SQL for potential individual-statement benchmarking.

- **SQL*Net snooping tools (OnWire)**
  Some years ago, some clever people built a very clever tool called OnWire that was capable of recording all SQL sent to an Oracle service via SQL*Net. This tool had the ability to accurately obtain timings and could have been leveraged to order statements across multiple replay sessions. But again, dependencies between sessions on transaction commit time for dependent data could not have been reliably reconstructed using the data produced by OnWire. Such tools could have been a good source of SQL for benchmarking individually.

- **Ask developers for a list of queries**
  As preposterous as this sounds on the surface, it is sadly probably the most popular approach to testing at a large number of sites. Developers send a big list of SQL culled from their code, and DBAs benchmark it across a change to try to gage impact. The obvious pitfalls are numerous, but certainly include the remote possibility that the developers might not identify all potentially problematic SQL when asked.

- **Benchmark the top queries from a Statspack or AWR report**
  Eve if a SQL is among the most expensive statements before a change, there is no reason to believe that it will be the source of a problem after a change. Often the worst problems across changes come from SQL that ran very efficiently before a change, but regressed and became problematic only after a change. This approach suffers from failed logic.

- **Compare execution plans before and after a change**
  The worst problem of many with this approach is the fact that there is no way to empirically know if a plan is “bad” or “good” just by looking at it, or even based solely on the types of operations in the plan. Some type of empirical statistics that provide waits, timing or resource usage are needed to provide information on improved or regressed SQL statements.

Beware false successes

Without doubt, customers have used one or many of the above approaches in the past, and successfully performed one or many major changes without encountering a catastrophic failure. Just because you use one of these flawed or incomplete approaches and succeed in deploying a change, it does not mean that the approach was valid or appropriate. Many customers have deployed major changes with no significant testing. This does not mean that untested changes are advisable or responsible. It just means that the customer was lucky. There is no guarantee that the next insufficiently-tested change will succeed.

In many cases, customers tolerate real problems resulting from change as “normal.” A popular excuse for higher resource utilization after upgrading Oracle is that the new release has higher “overhead.” In reality, the higher utilization is likely the result of an avoidable problem such as regressed SQL or a concurrency issue that could be addressed with minimal troubleshooting.

Novel uses for Real Application Testing features

The basic functionality of DB Replay and SPA are fairly straightforward, but the scope as described in the documentation is pretty much limited to change assurance. As I tested these features, a variety of potential other uses presented themselves. Oracle has undoubtedly thought of most of these ideas, but I present them here in the spirit of allowing customers to maximize the value of their investment in Oracle.

- Use Database Flashback with DB replay
  Once a replay is completed, if the test system has flashback logging on, then you can record relevant performance statistics, then flash the database back to the start SCN of the workload. By doing this, you can try a variety of changes one after another, and quantify the impact. For example, you can try several settings for an initialization parameter, and determine which setting provides the best performance under your workload. You can also keep the same test database and workload around for repeated use with any number of unrelated changes. Whenever a DBA wants to quantify the impact of something they are going to change, they can use a canned database and workload that they keep on a test system.

- Perform mid-stream changes under DB Replay workload
  A common source of database service destabilization is the running of maintenance or change tasks

under production load. By performing these activities in the middle of a running DB Replay workload, you can simulate what effects performing that activity under production workload may have. Any global change, or activity that might invalidate SQL or generate waits would be a good candidate for such testing. If you feel afraid to do something in production, it is probably a good candidate for testing under DB replay load.

- Problem simulation under DB Replay workload
  If a particular job or activity is causing performance problems in production, you can try running that same activity under DB Replay workload, and diagnose the causes of the problem in the comfort of a test database rather than in production. You can also use known destabilizing activities to simulate problems under DB replay workload to give DBAs an opportunity to practice diagnostic methods.

- Test case identification
  If a production performance problem occurs predictably, you can use DB Replay capture to capture the database workload during the time of the problem. That problematic workload may then be replayed repeatedly (using Database Flashback) in a test database. This provides a quicker path to root cause identification, because you can run many repeated simulations of the problem as are needed. Without DB replay, you would have to wait hours or another day for the next occurrence of the problem to try to diagnose it.

Active Session History and Real Application Testing

While Real Application Testing may be the star of the show in 11*g*, Active Session History (ASH), introduced in 10*g* is the revolutionary feature that has made empirical comparison of performance across broad timeframes possible. Without ASH, Real Application Testing would have far less useful data for comparison of pre and post-change performance. Before 10*g*, customers were able to tune using sampled wait information over time using third-party direct SGA-attach tools. But these tools were GUI-based, and the SGA-reading agent APIs were unpublished. Thus, customers could only use what data the tools presented in the GUIs, and no scripting or programmatic use of the data was possible.

Some customers used frequent sampling of the contents of v$session_wait to simulate the type of data that ASH now provides. This approach was better than most, but was not as comprehensive as ASH, and had greater database overhead. ASH has revolutionized the ability of DBAs to perform empirical analysis of performance, and has enabled whole other feature sets including many functions of Real Application Testing.

A feature Geared towards stability and high-quality service

Real Application Testing provides significant value to Oracle customers by allowing them to predict the impact of changes and maintain stability through changes. Using this feature set, organizations will be better equipped to implement needed changes with agility, using empirical methods and data, and sound decision-making.

Many organizations rely upon the availability and stability of Oracle services for the core functionality of their businesses. In other words, for many, Oracle is a mission-critical service. For those organizations whose core philosophy holds that outages, instability and poor performance in their mission-critical services is unacceptable, Real Application Testing can provide a valuable tool for adhering to that philosophy.

About the author

Jeremiah Wilton is principal consultant and owner of ORA-600 Consulting. Before becoming independent, he spent seven years at Amazon.com, initially as their first database administrator in 1997. Using expertise he gained during Amazon.com's period of exponential growth, he now specializes in scalability, high availability, stability and complex recoveries. Jeremiah has presented at numerous conferences and user group meetings, and is the author of a variety of technical whitepapers and articles. His publications are available at www.ora-600.net.

