Oracle Database Replay

Romain Colle, Leonidas Galanis, Supiti Buranawatanachoke,
Stratos Papadomanolakis, Yujun Wang
Oracle USA
400 Oracle Parkway
Redwood City, CA 94065
{firstname.lastname@oracle.com}

ABSTRACT

This demonstration presents Oracle Database Replay, a novel approach to testing changes to the relational database management system component of an information system (software upgrades, hardware changes etc). Database Replay makes it possible to subject a test system to a real production workload, which helps identify all potential problems before implementing the planned changes on the production system. Any interesting workload period of a production database system can be captured with minimal overhead. The captured workload is used to drive a test system while maintaining the concurrency and load characteristics of the real production workload.

The demonstration showcases how important maintaining the concurrency and load characteristics of the real workload is. The current testing solutions do not allow for synchronization based on data dependencies. Without proper synchronization the demonstration workload does not perform the work required and does not exercise the test system appropriately, leading to poor coverage and inadequate load. Thus many issues remain undetected. Database replay with its data based synchronization makes testing realistic and leads to the discovery of potential problems.

1. INTRODUCTION

Any potential change to a production system (such as upgrading the database or modifying configuration) necessitates extensive testing and validation before these changes can be applied. In order to be confident before implementing a change in the production system one needs to expose the test system to a workload that is very similar to the one it would experience in a production environment. With current technology, producing a workload that mimics a production workload is virtually impossible. Therefore, current testing methods often fail to predict problems that frequently plague production system changes. As a consequence in most information system environments, any change in production systems meets great reluctance.

VLDB '09, August 24-28, 2009, Lyon, France.

Oracle Database Replay ([1]) revolutionizes database testing. It allows the recording of the production workload on the production system with minimal performance impact. The captured workload contains all requests made to the RDBMS during the period of capture as well as all concurrency and transactional information. One can then use the captured workload to drive any test system and test any change before implementing it in production. The workload produced by Database Replay accurately reproduces the concurrency and load characteristics of the production workload on the test system. Hence, testing with a real workload ensures that there are no surprises when a change is implemented in the production RDBMS.

This demonstration highlights the importance of using a real workload for testing database changes. A major technological breakthrough introduced by Database Replay is a method of synchronizing the replay of captured requests based on their data dependencies ([1]), which produces a workload on the test system that is virtually identical to the production workload. This ensures that each replayed request operates on the same data as it did when it was captured and therefore performs the same amount of work as it did in the production system. Using a real-world application architecture, we demonstrate that, without data based synchronization, a workload cannot be used to make predictions about potential changes to the production system.

2. DATABASE REPLAY ARCHITECTURE

Database replay allows the recording of a real workload on a production system with minimal performance impact. The captured workload contains all requests made to the RDBMS during the period of capture as well as all concurrency and transactional information. On a test system, Database Replay can then execute the captured workload exactly as it ran on the production system to test any change before implementing it in production. Thus, any change to the RDBMS can be tested using Database Replay. For example, the following changes can be tested before they are applied to a production system: RDBMS software upgrades and patches, indexing and partitioning changes, any configuration changes and any changes of the underlying operating system or hardware. During replay all existing Oracle diagnostic tools can be used to their full extent to diagnose and remedy problems because the RDBMS still accepts and services requests that do not belong to the replayed workload.

2.1. Capturing Production Workloads

Oracle 11g allows any running RDBMS instance to start capturing the incoming workload. The user needs to pick an interesting workload period and find adequate disk space for the

workload before capturing the workload. The workload is stored in a user specified directory in operating system files. The impact on the production system is minimal and the RDBMS continues to behave normally from the viewpoint of the production applications. The overhead of the capture infrastructure to the running system is small (TPC-C throughput degradation up to 4.5%) and is workload dependent. Even in the case that capture errors out or runs out of disc space the production system is not affected.

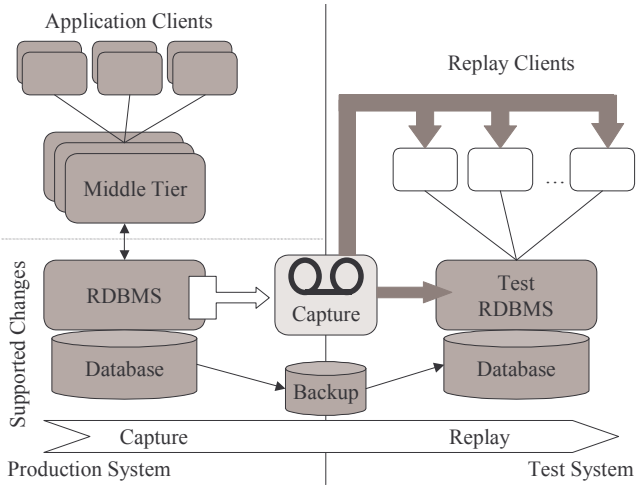

Figure 1 Database Replay Testing

Capturing the workload is made possible by instrumenting the RDBMS kernel with special code (the *capture probes*) that executes during capture and collects all data that is required for replay. These probes produce platform- and protocol-independent data that enables the captured workload to be replayed on any hardware or operating system platform that runs the Oracle Database. Each server process in the RDBMS captures its activity in an OS file stored in the capture directory. Workload from background processes such as maintenance tasks or scheduler jobs is not captured because, at replay time, it will be automatically triggered by the foreground workload that is captured.

The contents of the capture are sufficient to enable the replay on a test system and determine its success afterwards. The workload capture unit (a *database call*) is the minimum amount of information required to replay a request and validate its outcome on the test system. A database call contains data from the following categories:

1. *User data*: This is data that is sent from the client to the RDBMS. Examples include SQL text, host variables, fetch requests, and execution requests.
2. *Server response data*: This is data that is sent from the server back to the user. Result sets are the majority of this data stream. However, results sets are not captured because this would incur prohibitively high overhead during capture. Instead, a synopsis of the work performed in the database is captured: rows affected and error codes. This is enough to determine the outcome of the replayed call.
3. *System Data*: This data is internal to the RDBMS kernel, is not returned to the user, but is required for replay synchronization and runtime data replacement (for more details see [1]).

The capture can be fine-tuned with the definition of workload filters. Users can specify what part of the workload they want to exclude or include in the captured data by setting filters on session attributes, user IDs and other workload specific attributes. After the user is done with the workload capture they can move the captured workload to a test system where it can be replayed.

2.2. Replaying Production Workloads

The replay of the production workload aims to stress the RDBMS on the test system so as to determine whether the test system configuration would be appropriate for use in a production environment. In general, any type of testing consists of 4 distinct phases: 1) Setting up the test system, 2) defining the test workload, 3) running the workload, and 4) analyzing the results. When using Database Replay the time consuming step 2 is unnecessary because the workload is well defined: it was captured on the production system.

At the beginning of the test system setup the database state needs to be restored to a state that is logically equivalent to that at the beginning of the capture. Oracle provides several tools to accomplish this (see [2]). One final step to complete before replay can start is the processing of the workload. This creates the necessary metadata required for replay synchronization and runtime re-mapping and needs to be done only once. Then the processed workload can be replayed as many times as necessary.

A replay is performed by issuing the captured workload to the test RDBMS. To this end we use one or more replay clients running on one or more host systems. The replay client is a special executable that reads the captured workload and submits it to the database. The number of replay clients and hosts required is determined by the maximum concurrency of the captured workload and can be estimated using a utility provided. The replay clients replace the original clients that were present during the capture (Figure 1).

After starting the replay clients the user can start the replay. At that point the server sends a message to all connected clients so that they can start issuing the workload. During replay, the replay clients read the captured workload and convert it to appropriate requests to the database. Each client is assigned a part of the workload by the RDBMS. The aggregate workload generated by all the replay clients accurately mimics the production workload. For example, if during capture 10000 users connected to the RDBMS, during replay the same 10000 users will connect following the same connection and request patterns. Thus, the test RDBMS is subjected to the same load and request rate as the production system during capture. Additionally, the RDBMS makes sure that the replayed requests perform meaningful work, by maintaining the data dependencies seen during capture. For example, if a request updated 10000 rows during capture, during replay the RDBMS makes sure that this request executes after a previous request that inserted these 10000 rows. The result is that using capture and replay testing one can subject a test system to a production workload and perform a highly authentic test.

Each replay client is a multithreaded application that spawns a thread (the *replay thread*) for each captured session file. The replay thread reads the captured workload file and translates the data into the service calls to the RDBMS. The replay thread maintains the timing characteristics of the capture by sleeping appropriately between two consecutive calls. The goal is to maintain the captured request rate unless otherwise specified by the replay options.

Think time preservation during replay is not sufficient to maintain the appropriate data dependencies because it cannot guarantee that the RDBMS will serve the requests in the appropriate order. Our main goal for each replayed request is to make it perform the same work it did during capture, and this can be compromised if the various requests execute out of order and therefore operate on a different snapshot of the database with respect to the data they operated on during capture. Thus, to ensure that each replayed request performs the same work, the RDBMS under replay ensures that it operates on the same data as it did during capture. This is done by enforcing the commit ordering observed during capture by allowing each replayed call to execute only after the appropriate commit has been replayed (for details see section 3.3).

In addition to replay time synchronization, the replay employs runtime re-mapping techniques to facilitate the replay of requests with system specific data that varies from the production to the test system. Replay also uses a method to recreate values of generators of unique number (sequences) appropriately. The details are beyond the scope of this paper (see [1]).

Despite great efforts to ensure that the replayed calls perform the same work during replay as they did during capture, in some situations replayed calls can affect (update/return) different data and run into errors (*Data and Error Divergence*). Database Replay detects data divergence by comparing both the number of rows affected or returned by a SQL execution and the difference in error codes from capture to replay. The divergence data is extensively reported in the replay report, which is used to assess the validity of a given replay.

After the replay is done analysis of the workload can be performed using several reports and utilizing various Oracle utilities. The replay report provides a high-level performance overview and a divergence report that can be used to determine whether the replay performed similar work as the production workload. To further analyze the performance characteristics and diagnose performance problems, existing well-established Oracle features such as the Automatic Database Diagnostic Monitor (ADDM) or the Automatic Workload Repository (AWR) reports can be used. For more details on these tools see [2] and [3].

3. DEMONSTRATION

Our demonstration is based on a real world application that features highly concurrent and synchronized access to the database. We show how a replay of this workload loads the test RDBMS in a realistic manner that is similar to the production workload, whereas other tools that cannot recreate the appropriate data dependencies would simply fail to produce an adequate load.

3.1. Demonstration Application

Our demonstration application is shown in Figure 2. It uses an asynchronous background processing architecture, similar to the one used by SAP (see [5]).

The application logic resides on an application server, while the RDBMS is responsible for application data and for *persisting* requests. The Dispatcher and Worker processes handle incoming client requests in a concurrent fashion.

As shown in Figure 2, when a client sends a request to the dispatcher (Step 1) the latter enqueues it in a database table (2) and notifies the client that its request has been accepted. The dispatcher then picks a free worker process to work on the given request ID (3). The worker process de-queues the request from the table (4) and starts its processing, that involves more accesses to application data in the database (5).

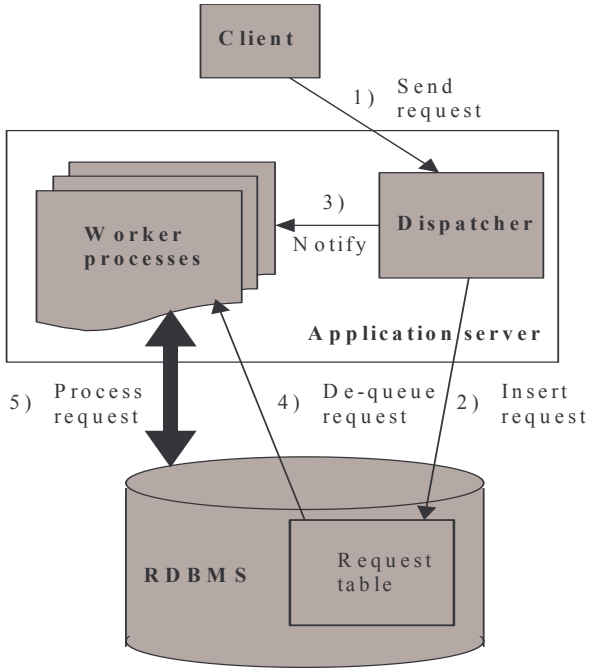

Figure 2 Application architecture

3.2 Testing Strategies

The workload seen by the RDBMS consists of the DML/Queries used to enqueue/de-queue requests and those used in the actual processing. In order to realistically recreate this workload against a test database, it is necessary to somehow emulate the behavior of the worker and dispatcher processes *without actually running the application.*

This is not easy to do. Existing database load testing tools (like the Quest Benchmark Factory [4]) generate synthetic, manually edited workloads. Synthetic workloads might contain some of the requests sent in steps (2), (4) and (5) shown in Figure 2, but fail to capture *all* the interactions with the RDBMS. Most important, they fail to accurately model the *dependencies* between requests.

For example, when testing our demonstration application without synchronization, it could happen that a de-queue request executes on the server *before* its corresponding insert request. This is possible since generic load testing tools do not model request dependencies. In such a scenario, the de-queue request will fail and so will all the subsequent processing on behalf of that request. Such *artificial* errors are not representative of production workloads and thus limit the effectiveness of load testing.

Our demonstration will compare synchronized vs. non-synchronized testing and show how Database Replay takes advantage of the fact that the data dependencies of the production workload are reflected in the database as part of transaction processing and are recorded as part of workload capture. Using this information, Database Replay can recreate the same service patterns seen by the RDBMS in a production setting. More specifically, it makes sure that the part of the database workload that corresponds to each of the client, dispatcher and worker

processes is accurately recreated and performs the same amount of work on the test RDBMS as it did during capture in the production system.

3.3 Database Replay with Synchronization

Database replay allows us to test and assess changes to the database part of our application without running into concurrency issues. Indeed, our novel synchronization infrastructure makes sure that no request will ever be de-queued and deleted before it has been inserted in the table. This way, the worker process' workload in the database always finds the data it is looking for, and the same load as the original one is applied to the test system. The next paragraphs explain how this is achieved.

The captured system records with every database call some system change numbers (SCN) that characterize the database state in which the captured call executed. The SCN is a stamp that defines a committed version of a database at a specific point in time. Oracle assigns every committed transaction a unique SCN. These SCNs are used for ensuring isolation and read consistency within the database server. Each captured call contains the SCN that corresponds to the database state when the call started executing. This SCN is called the *wait-for SCN*. Additionally each commit action contains the *commit SCN* that is the SCN that corresponds to the database state immediately after the commit action and before any subsequent commit action.

The *captured* commit SCN values are used to maintain the *replay clock* during the replay. The replay clock is similar to simulation clocks in that it is advanced by specific events. In the case of replay these events are commit actions. Every commit action sets the clock to the maximum of its commit SCN and the current clock. The replay clock is observed by every replay call (both commit and non-commit actions). During replay at the beginning of each call the server process executing this call checks the value of the replay clock. If the wait-for SCN of the replayed call *is less than or equal* to the replay clock, then the call is allowed to execute. Otherwise the call is blocked waiting to be posted by a clock advance (i.e. a commit action). This enforces during replay the same commit ordering that had been seen during capture.

In the case of our example application a single request is handled as follows. During capture time, when the client request is inserted in the table and committed, we also record the SCN associated with this commit (the commit SCN) as well as the time at which this operation was done. When the request is de-queued and deleted from that table by the worker process, we record with this operation the current SCN in the database (wait-for SCN) as well as the current time. This wait-for SCN value (that we will call **W**) is necessarily greater than or equal to the commit SCN value (that we will call **C**) associated with the "insert request" operation since it happened later.

During replay, our replay client reads the capture files and sends the recorded operations to the database server based on the timestamp recorded with each operation. Inside the database, if it happens that the "de-queue request" operation tries to be executed before the "insert request" operation is done, the synchronization infrastructure will make it wait. Indeed, the replay clock will be strictly less than **C** since the "insert request" operation has not finished executing and committing yet and therefore has not moved the clock to be at least equal to its commit SCN. And because we already established that the "de-queue request" operation's wait-for SCN **W** is greater than or equal to **C**, **W** is necessarily greater than the replay clock. It will thus block at least until the "insert request" operation is done and has committed its work, which is exactly what needs to happen in order to be able to process the request.

Finally, the fact that the replay client issues the calls with the same timing that has been seen during capture makes sure that the "de-queue request" operation will arrive soon enough to the database so that the request does not time out (if it did not time out during capture).

3.3 Testing without synchronization

If a tool, that does not provide a data-aware synchronization feature like the one we just described, is used to "replay" our application workload, it has to rely on timing only to send the calls to the database server. Under a highly concurrent load, it is guaranteed that some "de-queue request" calls will end up being executed before their corresponding "insert request" has finished committing; all the subsequent worker calls for this request will be invalid and the request itself will time out. Therefore, a fair amount of requests will not be processed, resulting in a possibly lighter and different load profile than the one that should have been applied.

In order to highlight these problems, we simply use our database replay tool with the synchronization feature turned off, and compare some performance reports for this period against the ones from the capture period and the synchronized replay period. Whereas the capture and synchronized replay periods look similar, the unsynchronized replay period is fairly different with a much lighter load that is clearly not suitable for drawing conclusions about the production system.

4. REFERENCES

[1] L. Galanis, S. Buranawatanachoke, R. Colle, B. Dageville, K. Dias, J. Klein, S. Papadomanolakis, L. L. Tan, V. Venkataramani, Y. Wang, G. Wood. Oracle Database Replay. SIGMOD 2008

[2] Oracle 11g Documentation. http://www.oracle.com/pls/db111/db111.homepage

[3] K. Dias, M. Ramacher, U. Shaft, V. Venkataramani, G. Wood. Automatic Performance Diagnosis and Tuning in Oracle. CIDR 2005

[4] Quest Benchmark Factory for Databases http://www.quest.com/benchmark-factory/

[5] Architecture of the SAP Web AS http://help.sap.com/saphelp_nw04/helpdata/en/84/54953fc405330ee10000000a114084/content.htm