Oracle Database Replay

Leonidas Galanis, Supiti Buranawatanachoke, Romain Colle, Benoît Dageville, Karl Dias, Jonathan Klein, Stratos Papadomanolakis, Leng Leng Tan, Venkateshwaran Venkataramani*, Yujun Wang, Graham Wood

Oracle USA
400 Oracle Parkway
Redwood City, CA 94065
{firstname.lastname@oracle.com}

ABSTRACT

This paper 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 system 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 can be used to drive a test system while maintaining the concurrency and load characteristics of the real production workload. Therefore, the test results using database replay can provide very high assurance in determining the impact of changes to a production system before applying these changes. This paper presents the architecture of Database Replay as well as experimental results that demonstrate its usefulness as testing methodology.

Categories and Subject Descriptors

H.2.m [**Database Management**]: Miscellaneous

General Terms

Management, Measurement, Performance, Verification.

Keywords

Capture, Database, Record, Replay, Testing.

1. INTRODUCTION

Large business-critical applications are complex and experience highly varying load and usage patterns. At the same time they are expected to provide certain service guarantees in terms of response time, throughput, uptime and availability. In an attempt to stay competitive and cope with increasing demand, operators of large-scale information systems strive to keep their systems up-to-date by deploying the latest technology. Constant advances in information technology mandate changes at an ever-increasing rate.

Making any 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, coming even close to *real testing* is virtually impossible. Current testing methods often fail to predict problems that frequently plague production system changes. Therefore in most information system environments, any change in production systems meets great reluctance.

In practically all IT environments the relational database management system (RDBMS) holds a prominent role: it provides safe transactional storage and fast scalable retrieval for virtually all data in an enterprise. Consequently, the ability to safely perform changes in the RDBMS component is of utmost importance. As typical data sizes in an enterprise continuously scale up and as new needs define new requirements for the RDBMS, production systems require frequent changes to their database systems. Some of the possible changes are: 1) A software upgrade of the RDBMS. 2) A hardware or operating system change (for example moving from a single host to a clustered system). 3) A change in the physical organization of the data (adding or removing indices). Extensive testing very often cannot uncover many problems. Thus, changes frequently lead to problems because of new bugs, undesired query plan changes and new resource contention points.

Problems associated with changes to the RDBMS and the resulting reluctance of database administrators to implement these changes are consequences of the shortcomings of current testing procedures. One of the main reasons that current testing fails in many aspects, is that it is not possible to subject a test system to a realistic production workload. Common approaches to testing RDBMS changes are the following:

- Specially crafted test scripts that attempt to mimic production system scenarios: These scripts are either manually written or generated by load generation tools (such as [8]). This approach fails in reproducing the concurrency characteristics of a production workload.
- Real user testing: Real users are asked to use the test system as if it were a production system in the hope to make potential problem surface during testing. This approach is random, time consuming and fails to reproduce the periodically varying load patterns of a production environment.

SIGMOD'08, June 9–12, 2008, Vancouver, BC, Canada.

* Work done while at Oracle USA. Currently veeve@facebook.com.

- Simulation: Defining an adequate simulation model for a complex system like an RDBMS is almost impossible. Thus, this method is only useful for smaller scale systems.

Oracle Database Replay solves the problem of *real 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 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 major contribution of Database Replay is that using the captured workload, it can accurately reproduce the concurrency and load characteristics of the production workload on the test system. Hence, after testing with a real workload, a database administrator can be confident that no surprises are in order when the change is implemented in the production RDBMS.

This paper is organized as follows: Section 2 presents an overview of the testing workflow and methodology of Database Replay. Section 3 details the architecture of the capture infrastructure. Section 4 presents the architecture of the replay infrastructure. Section 5 presents a case study using database replay. Section 6 outlines related work and section 7 concludes the presentation of Database Replay.

2. REAL TESTING

Modern information systems usually contain two distinct environments: *Production systems* and *test systems*. The production systems are the lifeline of the enterprise and are taxed with serving the enterprise's information technology needs. No change should be applied to any live production system without first having been extensively tested on the test systems. The test systems are setup to accurately reflect the production systems' architecture and setup so as to enable realistic testing. This section provides an overview of how Database Replay is used as a testing tool.

2.1 Testing Support

Any change to the RDBMS component of an information infrastructure can be tested using Database Replay. For example the following modifications on the RDBMS can be tested: RDBMS software upgrades and patches, schema changes (new indexing and data partitioning schemes) and configuration changes (from changing initialization parameters up to moving from single host setup to a clustered database). Additionally, the following software or hardware changes below the RDBMS layer can also be tested using Database Replay: Operating system upgrade or operating system change (a move from windows to linux for example), hardware changes (e.g., new cpu, more memory) and storage system changes. Changes that cannot be tested are any changes above the RDBMS such as changes to the middle tier or the application clients.

2.2 Capturing Production Workload

The best possible workload for testing is the actual production workload. Oracle 11g allows any running RDBMS instance to start capturing the incoming workload. The user needs to pick an interesting period of regular business operations, find adequate disk space for the workload and start capturing the workload. The workload is stored in a user specified directory in operating system files. The impact on the production system is minimal (see Section 3) and the RDBMS continues to behave normally from the viewpoint of the applications. Even in the case that capture errors out or runs out of disc space the production system is not affected.

The capture can be fine-tuned with the definitions of workload filters. Users can specify what part of the workload they want to exclude or include in the captured workload by setting filters on session attributes, user IDs and other workload specific attributes. Also the user can either specify for how long the workload should be captured or manually stop the capture. After the user finishes the capture they can move the captured workload to a test system, where real testing can begin.

2.3 Replaying Production Workload

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, 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 step 2 is unnecessary because the workload is well defined: It was captured it in the production system. Hence, test developers do not need to understand the application and spend time writing testing code. This saves a usually very time consuming stage in RDBMS testing.

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. Several tools exist to accomplish this, since backup/restore is one of the most well understood RDBMS technologies, therefore we will not detail this process in this paper. One final step before replay can start is the processing of the workload (see Section 4.2.1). This creates the necessary metadata required for replay and needs to be done only once. Then the processed workload can be replayed as many times as necessary.

The captured workload needs to be issued to the test RDBMS. To this end we use one or more replay clients. The replay client is a special executable that reads the captured workload and submits it to the database (see 4.2.2) and is part of the Oracle 11g RDBMS software. The number of replay clients required depends on 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 (for example the application middle tier).

After starting the replay clients the user can start the replay. The replay is controlled using an API defined by stored procedures. Thus, to start the replay the user needs to connect to the RDBMS and call the appropriate stored procedure. Then the server sends a message to all connected clients so that they 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 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 (see Section 4.2.3). 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 real testing.

While replay is going on the RDBMS can still be accessed normally. This means that all available performance monitoring tools (such as ADDM and ASH [12]) can be used to monitor the RDBMS during replay. Furthermore, additional workload can be executed in parallel with the replay to further load the server. The replay can be stopped at any time or it can run to completion until the captured workload is consumed. After the replay is finished, the RDBMS generates reports that help determine the quality of replay and compare key points of the capture and the replay. Replay can be performed repeatedly using the same captured workload. Obviously, the database state has to be restored appropriately before every replay.

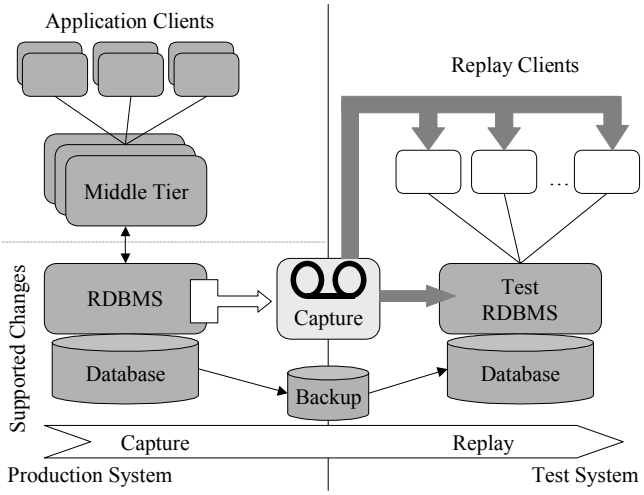

Figure 1 Database Replay Testing

2.4 Use Cases

Change impact testing is one of the most useful test cases for database capture and replay testing. Any conceivable change at or below the RDBMS layer can be confidently tested before it is implemented. One very important use case is *upgrade testing*. One can replay an Oracle 10g production workload against an Oracle 11g test system in order to discover problems before upgrading the production servers. Another testing application is when changing server hardware from a single instance system to a clustered system. The captured workload can be replayed at an increased request rate against a clustered database to figure out the performance gains. The increased request rate is achieved by issuing the captured requests at a rate higher than that observed during capture. The same method can be applied when moving from one operating system to another. Cross OS testing is possible because the captured workload is platform independent. Deterministic debugging of server problems is another good use case of database replay testing. Our experience has shown that the vast majority of issues discovered during testing using database replay are always reproducible.

3. CAPTURE

This section details how the RDBMS workload is captured. The goal of capture is to record *all* necessary activity on the RDBMS that is required to faithfully reproduce the same activity on a test system. Moreover, this needs to be done with minimal overhead to the RDBMS.

3.1 Point of Capture

Capturing the RDBMS workload either requires instrumentation of the software stack of the RDBMS or the use of some external component that captures the network traffic between the RDBMS and its clients. We chose to follow the former approach because it allows the captured data to be protocol independent and makes it possible to record RDBMS internal data that is required for replay (see Section 4.2.4). Furthermore, all *capture probes* are placed below the client-server protocol layer in the RDBMS software stack. This ensures that all captured data is protocol and platform independent.

3.2 Workload Sources

The RDBMS workload can be subdivided into 2 basic categories: 1) External clients requests, 2) internal clients requests. External clients are applications or application server middle tiers. All the activity coming from such clients is captured. Internal clients are considered background processes such as maintenance tasks or scheduler jobs. The workload from these clients is not captured because it is expected to be present during replay. For example, if a client issues a request to schedule a job, this request will be captured, but the execution of the scheduled job will not. Later, during replay, the scheduling of the job will be replayed, which will cause the scheduled job to be executed in the replay system. If we had captured the scheduled job we would have 2 instances of the same job running on the replay system.

3.3 Capturing Agents

The instrumentation of the RDBMS kernel makes up the *capture infrastructure.* It provides capture services to be used by the entities that record the workload, the *capture agents*. More specifically, the capture agents are the oracle server processes that service external client requests. Since, the oracle server processes capture themselves, there are no additional capturing agents. When workload capture is enabled, the server processes use the capture services to record the workload into a plain operating system file, the *capture file*. All processes write into a single directory, the *capture directory*. For performance reasons, the captured workload is not stored in the database, since this would require generation of undo/redo information.

The capture infrastructure is designed to also work with the oracle RDBMS in a cluster configuration. Coordination is only required during starting and stopping. In both cases cross-instance messaging is used to start and stop capture on all instances. One complication in the clustered environment stems from the fact that the capture directory may not be shared. In this case the contents of all directories from all instances have to be consolidated into a single directory for replay.

3.4 Workload Contents

Capturing an RDBMS workload for a period of time yields one file per RDBMS server processes. Each captured process file (*capture file*) contains workload from 1 or more *database sessions*. A database session is the set of interactions of a user between log on and logoff. Multiple sessions can be multiplexed into 1 database server process. Each session consists of consecutive service requests or *database calls*. A SQL query, an update to a table, a commit, and a call to read from a large object are all examples of database calls.

Database calls are divided into two categories. Each captured call is either a *commit action* or a *non-commit action.* A commit action is a call that commits the work that has been done by the captured

session. Examples of commit actions are: 1) a COMMIT, 2) an insert that is issued with auto-commit and 3) a CREATE TABLE statement, which always commits internally. A non-commit action is a call that does not commit any data. Examples of non-commit actions are: 1) a SELECT query, 2) an UPDATE statement and 3) and INSERT statement. The distinction between commit and non-commit actions will become clearer in the context of replay (Section 4.2.3). The basic rationale for the distinction is that commit actions modify the database state and may affect the outcome of subsequent call. Note that DML operations modify database state but the change is only visible to the session that issued the operation because of transactional isolation. So any non-committed operation that changes data does not affect any other session.

Each recorded call contains sufficient information that allows the accurate reproduction of the call during replay. This information can be divided into three categories:

1. *User data*: This is data that is sent from the client to the RDBMS.
2. *Server response data*: This is data that is sent from the server back to the user.
3. *System Data*: This data is internal to the RDBMS kernel, is not returned to the user but is crucial for replay.

The majority of the captured data belongs to the user data category. More specifically, this data contains the full SQL text of user requests, all the bind values, and all non-SQL requests and their arguments (such as calls to manipulate large objects). The recorded user data does not contain system SQL that is executed as part of a user request (for example catalog queries). Similarly, we only record the full text of PLSQL ([12]) scripts, without recording each individual user SQL that is executed as part of the script.

The data that is sent from the server back to the client can potentially be very large, because it mainly contains query results. Recording all the query results of a workload is not feasible since this would incur a high overhead. Nevertheless, we need to capture sufficient data about the outcome of user requests so that we know how much data they processed and if they had any errors. Therefore the captured server response data contains the number of rows affected/returned by the user request and error codes. Additionally, for reasons that will become clear in Section 4.2.4 system specific data that is part of query results is captured. Such data are row identifiers (ROWIDs that allow direct access to rows without the need of an index), large object locators and result set handles.

System data is RDBMS kernel internal data that is required only during replay. Part of the system data is timing information. Additionally, the captured system data contains system change numbers (SCN) that characterize the database state in which the captured call executed and determine whether the call is a commit action or a non-commit action. 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 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 significance of both SCNs will become clear in Section 4.2.3. Finally, values returned by sequence generators (and other system functions that are omitted for simplicity) are part of the captured system data. These values are used during replay as shown in Section 4.2.4.

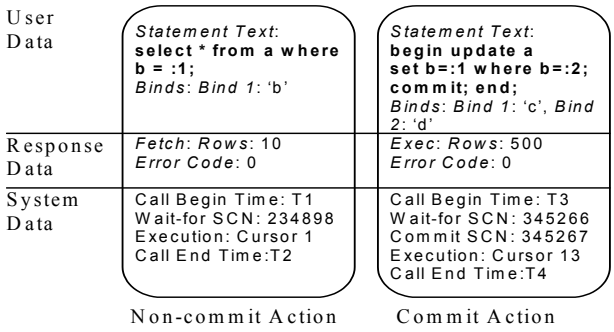

Figure 2 Captured Database Call Examples

Figure 2 depicts two typical captured database calls along with some of the key information captured. The first call is a non-commit action that corresponds to a select query with one bind. As part of this call we capture the call begin and end time, the type of the call (SELECT…), the environment SCN (wait-for SCN) in which the query executed, the full text, the bind data, the internal ID of this results set (cursor number 1) and the number of rows fetched by this call. The number of rows will be compared to that during replay to determine whether this call does the same work during replay. The second call is a commit action that corresponds to a PLSQL script that contains an update statement and a commit. Again, full statement text, binds, call begin and end time are captured. In addition to the wait-for SCN the call contains the commit SCN, which defines the database state immediately after the statement executes.

The capture files are plain binary files that follow a self-describing extensible format. The data in the capture files is platform independent so that one can capture the workload of an oracle database running on a 64bit Linux OS and then replay the workload on a 32bit Windows system. Furthermore, the format automatically allows backward compatibility with pre-existing captures regardless of how many new probes will be added in the future.

3.5 Overhead

As expected the capture infrastructure adds some overhead to the running workload. The goal is to make the capture overhead as small as possible. To this end buffered I/O is used along with code optimizations. Additionally, repeated data (such as SQL text from repeated executions) is captured only once per process. The result is that for most workloads the overhead is reasonable enough so that it is possible to turn on capture on the entire production. Moreover, critical production systems are usually over-provisioned so that the additional overhead of capture is acceptable. For example enabling capture on the TPC-C benchmark [18] only reduces transaction throughput by about 4.5%.

The capture overhead is workload dependent. However, it is not proportional to the work that the RDBMS performs. It is proportional to the data that needs to be captured in order to be able to replay the call. For example, if the workload consists mainly of calls that are very complex queries that each take very long to execute, the overhead is going to be minimal, since the data captured per time unit is small: Only the text of the complex query once per session for each call that executes this query. On the contrary, if the workload is insert-intensive the overhead may be significant. The reason is that each captured insert will contain all the inserted data in addition to the INSERT statement. Figure 3

shows how the overhead varies depending on the type of the workload.

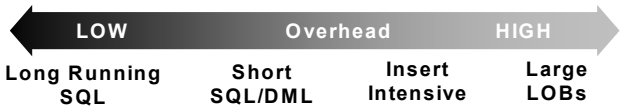

Figure 3 Workload dependent capture overhead

The space requirements of the capture are also workload dependent and so a reliable estimation is not possible. The space required is roughly equal to the data that is sent from the client to the server over the network. Since performance overhead is a big concern, the workload data is not compressed during capture. To estimate the disk space needed one can turn on capture for a brief time when the system processes the workload of interest and then extrapolate for the potential duration of the capture.

3.6 Filters

The capture infrastructure also provides the ability to filter out specific workload based on workload attributes (user, session ID, cluster instance number, session state and others). The defined filters fully specify a part of the workload and are used in two modes: inclusion mode and exclusion mode. If inclusion mode is specified, only the workload specified by the filters is captured. Otherwise (exclusion mode) the workload that is specified by the filters is not captured. Filters are useful in cases where an RDBMS services multiple independent applications. In this case, it is possible to specify appropriate filters that will capture the workload coming from one specific application. Then this workload can be tested in isolation.

4. REPLAY

The high level goal of Replay is to subject a test RDBMS to a realistic workload. This means that replay will make the test system handle a load with near identical concurrency and transactional characteristics as a real workload. The capture infrastructure provides replay with all the necessary data to recreate the same workload on a test system. In this section we will briefly introduce basic replay concepts before we present the Oracle 11g specific design and implementation of database replay.

4.1 Concepts

4.1.1 Replay Testing Success

To determine whether replay can achieve its goal one needs a method to measure success. The goal of this method is to determine whether *the replay of a captured workload is suitable for accurately testing prospective changes*. There are three basic characteristics of a workload that we use to determine replay success: 1) The final database state after the workload, 2) the concurrency and request rate characteristics and 3) the results returned to the clients. A replay is successful if it starts from the same database state as the captured workload, runs on an identical system and exhibits behavior identical to the captured workload with respect to these three workload characteristics. Such a replay achieves *end state consistency*, *result consistency* and *request rate consistency*. Note that there exists technology to create workloads that achieve each of the above characteristics separately. For example one can process the redo log of the database and create a workload that will achieve the same final state as the workload that generated the redo log. This workload, however, will not have any meaningful concurrency and will not execute any queries, only updates. Furthermore, these updates are block level changes. Thus, if the update contained a `WHERE` clause, which could involve substantial processing, it will be lost. Database replay takes on the challenge to create a workload that simultaneously preserves all three characteristics of a real workload in the context of a test system. In reality it is going to be almost impossible to achieve 100% consistency on all 3 characteristics with respect to the captured workload. Nevertheless, a close approximation of each characteristic would still provide high confidence in testing.

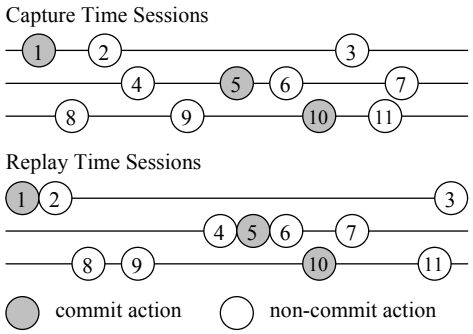

Figure 4 Transactional model example

4.1.2 Transactional Model

To achieve *end state consistency* and *result consistency*, we define the transactional model for replay that determines the replay order of captured calls. This model is based on the classification of the captured database calls into commit actions and non-commit actions. This distinction is based on the fact that a commit action changes the database state and makes the new state visible to all subsequent actions. Since the result of any call depends on the database state, *the replay has to ensure that each call C follows the appropriate commit action so as to satisfy result consistency and ultimately database end state consistency*. In the current version of database replay the appropriate commit action is the most recent commit action in *the entire captured workload* with respect to C. Additionally, non-commit actions do not need to be ordered with respect to other non-commit actions. This allows for concurrency between two subsequent commit actions. Figure 4 shows how a replay would satisfy the transactional model. For example between calls 1 and 5, calls 2, 4, 8 and 9 can be executed in any order during replay but *must* come after 1. Naturally, calls that belong to the same execution thread are issued in the order they were recorded during replay: 9 will always come after 8.

The transactional model currently used for database replay assumes that every commit action affects every subsequent call. This is a coarse model and may consider two captured calls dependent even if they are not. For example in Figure 4, consider calls 7 and 10. If 10 commits updates to table T_1 and 7 selects from table T_2, the model will consider 7 dependent on 10 even though they are not. It is possible to use a more refined transactional model for database replay that only considers true dependencies between captured calls. For the first version of database replay we chose to use the simpler model for several reasons. During capture the required information to define transactional dependencies is small which is beneficial for the capture overhead. Furthermore, during replay enforcing the call order is simple and does not interfere significantly with the replayed workload. Experiments with early prototypes have shown that the simple transactional model used currently is adequate to ensure database end state and result consistency while at the same maintaining the request rate observed during capture.

4.1.3 Time Model

To satisfy *request rate consistency* the transactional model is not sufficient since it only defines the order of replayed calls and not their timing. Timing information is required in order for the replay to accurately recreate the request rate observed during capture. To this end we define two distinct time dimensions:

1. *Connection time*: This is the time between the start of the capture (replay) and the beginning of the first captured call of a captured process (replayed call of a replayed process).
2. *Think time*: This is the time between the end of call and the beginning of the next call in the same execution thread (captured or replayed server process).

To achieve request rate consistency the replay has to maintain the connection time observed during capture for each replayed process and the think time observed during capture between any consecutive pair of calls *within* each replay process provided the call execution time does not change from capture to replay. In reality the call execution time will vary from capture to replay. Section 4.2.2 describes how request rate is maintained in practice.

4.2 Database Replay Architecture

The Oracle 11g database replay feature is an implementation of the transactional model and the time model introduced in the previous section. Using the captured workload on a properly set up test system, database replay achieves high degrees of end state consistency, request rate consistency and result consistency with the respect to the capture. This section explains how this is achieved.

4.2.1 Capture Processing

Capture processing is the operation that in one pass reads all the captured workload and produces metadata required for the replay. This metadata consists of four parts:

1. The data required to order commit actions during replay. (*SCN order table*)
2. A collection of system-generated values used for the replay time emulation of system function. (*SYSID tables*)
3. A collection of the connection descriptors used by the captured processes to connect the database.
4. A summary of the workload used by the workload driver infrastructure. (*Connection queue*)

The SCN order table contains rows that correspond to commit actions in the workload. Each row contains the following columns: The commit SCN (Section 4.2.3), the call ID and the file ID. The table is ordered by commit SCN and contains all commit actions in the order they changed the database state. The reason we cannot use time to order commit actions is that the actual commit may happen anywhere within the database call. Calls can overlap in all possible way, thus only the SCN can determine which call committed its changes to the database first. This table is used during replay to implement the transactional model.

The SYSID tables contain values of system-generated ids that are used in (Section 4.2.4). One of these tables contains per call sequence values. During capture we record the return values of S.NEXTVAL and S.CURRVAL, where S is a sequence generator. These values are used during replay instead of the actual replay time return values of the S.NEXTVAL and S.CURRVAL. Each row in the sequence table contains a reference to the call and the sequence it belongs and a range of values that this call consumed. One or more rows can correspond to a captured call.

The collection of connection descriptors is used to initialize the *connection map* table before replay. The captured connection descriptors are invalid in the replay. So, before replay can start the user must map them to new valid connection descriptors.

The connection queue is a single file that contains an entry for each captured process. The entries are sorted by connection time and contain information used by the workload driver infrastructure as explained in Section 4.2.2.

Capture processing is a one-time operation that transforms the captured workload into the *replay files*. The replay files can be used for replay arbitrary many times.

4.2.2 Workload Driver

The workload driver is the infrastructure that reads the captured workload and issues it to the RDBMS during replay. It is made up of one or more replay client processes (*replay clients*). Each replay client process is a multithreaded application provided as part of the RDBMS distribution. Each replay client thread (*replay thread*) reads *one* captured process file and interprets its contents into appropriate database calls to the RDBMS. The replay client is a regular OCI client ([12]) and as such can connect remotely to the RDBMS. This makes it possible to start the replay clients on multiple hosts. The workload driver infrastructure replaces any clients/middle tiers that were present during capture and is responsible for driving the workload to the server.

Protocol independence in the captured data complicates the choice of replay client protocols. Conceivably the replay client can use multiple protocols during replay based on what was used during capture. However, this would lead to a multi-language code base (for example Java for JDBC client code and C for OCI client code) with significant code duplication that would be difficult to maintain. This technical difficulty combined with the fact the protocol specific code in the RDBMS is not the main focus of testing, led to the adoption of *functional replay*. Functional replay dictates that each replayed call be equivalent to the captured call in terms of the work it performs inside the RDBMS regardless of the protocol used to send the call from the client to the server. Thus, by design, the replay client only uses one protocol to talk to the server: the Oracle Call Interface (OCI). Each captured call is converted to the appropriate sequence of OCI interface calls.

The captured workload can potentially consist of thousands of files and may require the replay of thousands of concurrent sessions connected to the RDBMS. Therefore the number of replay clients that can be used to drive the workload needs to scale appropriately. To achieve scalability the replay clients do not communicate with each other and are instead coordinated by the server. Database replay contains a calibration utility that advises on the number of the replay clients that are needed based on the workload concurrency and the available hardware.

Starting the replay is a two-step process. First the RDBMS is put in a special state (*prepared state*) in which it waits for replay clients to connect. Then the user starts arbitrary many replay clients by pointing them to the prepared RDBMS. After starting a client the user cannot interact with it anymore; the client is fully controlled by the RDBMS server. The user can start the replay by interfacing with the RDBMS through a server API. At replay start, the server determines how many clients have connected, sends each client the replay options (Section 4.2.6) and distributes all the capture files among them in a round-robin fashion. At this point each client know exactly which capture files it will replay. Then it sends the signal to each replay client to start issuing the captured workload.

The workload driver infrastructure is tasked with implementing the replay time model, which ensures request rate consistency. Connection time during replay is determined by the connection queue produced by workload processing. Once replay starts each replay client scans the entire connection queue. For each entry in the queue the replay client checks whether the capture file this entry points to belongs to the client's workload assigned by the RDBMS. If it does the client spawns a replay thread at the appropriate point in time. This thread is completely self-contained and does not communicate with other threads. The replay thread starts scanning the appropriate capture file and connects to the server only after satisfying the timing specified by the connection time in the capture. The connection endpoint to use is determined by the connection map table that the client reads from the server.

The replay thread exclusively maintains the think time prescribed in the time model. Between consecutive calls, the replay thread sleeps for the appropriate amount of time so as satisfy the captured request rate for the captured process it is replaying. Thus, the aggregate request rate requirements are maintained globally as a result of each thread's effort to maintain the request rate locally. The time model think time requirement assumes that call execution time does not change from capture to replay. In reality this will almost never be true. Therefore the replay thread maintains a cumulative time deficit that is subtracted from the think time in order to maintain the captured request rate. Thus, if replay calls are on average slower the think time is shrunk in order to maintain the request rate (Figure 5). However, if calls are on average faster the think time is not expanded, even though this contradicts the request rate consistency. The reason is that it is more appropriate for testing purposes to let the replay run at a higher rate if the test RDBMS can handle the workload. It is possible that the time deficit becomes so high that sleep time needs to be reduced to zero. At this point the replay cannot maintain the captured request rate and starts running slower.

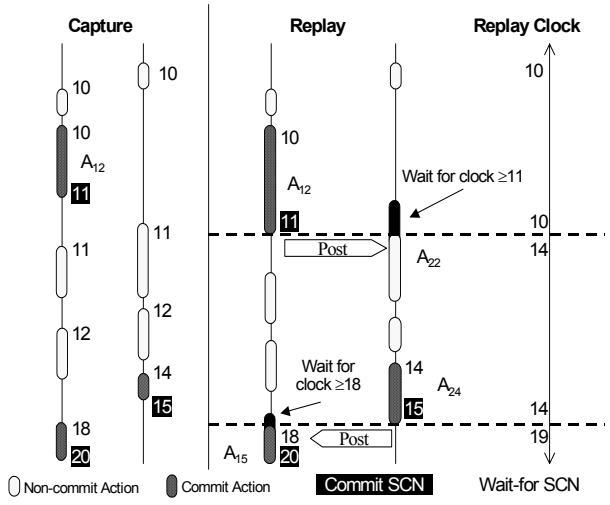

Figure 5 Synchronization example

4.2.3 SCN Based Synchronization

SCN based synchronization is the means with which database replay implements end state consistency and results consistency. This infrastructure resides in the RDBMS server and is used by the replayed server processes. Each captured call contains a wait-for SCN and each commit action additionally contains a commit SCN. Conceptually, in order to achieve result consistency it is sufficient to ensure that every replayed call **C** is executed after the replay of the commit action **A** that is the most recent commit action that was captured before **C**. This is always true in the Oracle RDBMS because of the absence of "dirty reads". Therefore the SCN can provide the desired ordering.

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. This posting happens when a commit action **A** advances the clock to a new value. The values for the clock are calculated using the SCN order table that is produced during capture processing (the appropriate portion of this table is cached in memory during replay). The new value of the clock is calculated by taking the commit SCN of the commit action **B** immediately following **A** in the SCN order table and subtracting one. Then the server process that executed **A** posts all waiting processes that have pending calls with a wait-for SCN less than or equal to the new value of the clock. Note that the system SCN during replay time is not used for replay purposes and so it does not have to match with the captured SCN values.

Figure 5 shows an example of synchronizing 2 sessions during replay. The right side shows the corresponding calls with the recorded SCN values. The SCN values are monotonically increasing. During replay call A_{12} takes longer to complete than it did during capture. This causes call A_{22} to wait since the replay clock is still 10. After A_{12} completes it makes the replay clock 14 and posts A_{22}. Taking the commit SCN of A_{24} and subtracting one produces 14, which is the new clock value. Similarly A_{15} waits to be posted by A_{24}. In reality the replay clock mechanism is more complicated than in this example. For illustration purposes the concept is simplified with respect to the actual implementation.

The replay clock is maintained in global memory in the RDBMS. In case of a clustered RDBMS each instance maintains its own replay clock. These clocks are kept in sync by cross instance messaging. Conceptually each commit action broadcasts the new replay clock to all cluster instances using some optimizations that are outside the scope of this paper. Note that during replay, only sessions coming from replay clients use the SCN ordering infrastructure. Thus the database is still available for serving regular workload even if replay is ongoing.

Gating calls in the database occasionally may lead to replay induced deadlocks. The reason is that the server may block a call **C** that holds resources (locks or latches for example) needed by another commit action **A** to complete. If **C** is (transitively) waiting for the replay clock value that **A** is going to produce, then the replay will hang. To avoid such hangs during the replay a background process periodically checks for such deadlocks by following wait chains and resolves them by posting the process that executes that call that is blocking the commit action from proceeding. This type of deadlock resolution is different than the usual deadlock resolution mechanisms used in Oracle in that the chosen deadlock detection victim is not actually forced to abort a transaction. It is rather allowed to continue without waiting for the replay clock until it has given up the resources (usually locks) it holds (i.e. until the next

commit) so that the clock ticker waiting on these resources can make progress and advance the replay clock.

4.2.4 Replay-time Data Replacement

Implementing the transactional model should in theory be enough to achieve end state consistency and result consistency. However, there are cases where a replayed call will not do the same work it did during capture. This happens when a call contains system dependent data that is invalid in the replay system. Such data in the Oracle RDBMS are row identifiers (*rowids*), large object locators (*lob locators*) and result references (*ref cursors*). If such values are part of the bind values the replay clients re-map them to runtime correct values. Runtime re-mapping is illustrated by the following example (Figure 6). The rowid that is returned to the client as part of the select list is captured (1). Then the same rowid is captured as an in-bind (2). During replay the captured rowid associated with the select statement (3) makes the replay client expect the replay time value of the rowid. At this point the association is made between the captured rowid and the one seen during replay. When the update is replayed (4) the replay time rowid is used. This will make the update succeed and update the employee row.

Replay-time data replacement also takes place in the server. The server intervenes in special server side functions such as sequence generators. When a replayed call uses a sequence value, the replay infrastructure looks up the captured sequence values produced by capture processing (Section 4.2.1). The replayed call is then furnished with the exact same values it saw during capture. If the sequence values were not altered during the replay, a replayed call would most probably use different sequence values than those used during capture and would so diverge. Furthermore, using sequence value replacement ensures the successful replay of requests that contain data that was generated using sequences. For example if the application created a shopping cart id using a sequence generator, this id is going to be the same during replay. Thus, every update using this ID will be successful during replay.

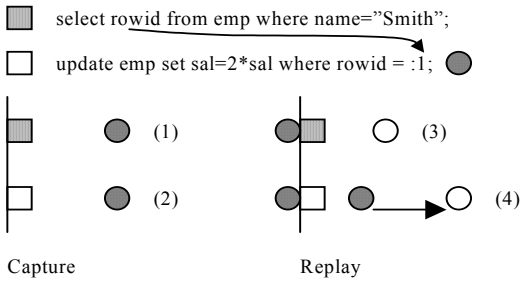

Figure 6 Runtime re-mapping of system dependent values

4.2.5 Replay Divergence

A replayed call is considered to *diverge* if 1) it affects different number of rows during replay than it did during capture or 2) if it sees an error during replay and there was none or one with a different error code during capture. The captured data does not contain any results (Section 3.4), so we cannot know if a call during replay actually has the same results as it did during capture. But for the purposes of replay, row counts and error values are sufficient to indicate whether a replayed call performed the same work as its captured counter part. Moreover, we expect users to have an application level validation script to assess the correctness of replay.

Result consistency is not always possible even with commit ordering and replay-time data replacement. Some corner cases exist that make data divergence likely during replay. In all the workloads we experimented with, these cases appeared in a small percentage of calls and did not invalidate the value of replay as a testing tool. Some example causes of divergence are the following:

1. *Multiple commits within PLSQL scripts*. In this case we consider the entire PLSQL block as a single commit action and don't enforce commit ordering between the statements inside the PLSQL block and the calls from other sessions.
2. *Application logic within PLSQL scripts*. PLSQL scripts contain application logic that might follow different paths during replay. Thus, the replayed call may perform different work. This is especially true in cases where the application logic depends on data that does not reside in the database (i.e. time of day, client host names and others).
3. *In-flight session at the time capture starts*. These sessions may contain calls that operate on data was not committed before capture started. These calls will diverge because the missing part of the in-flight sessions will not be replayed.
4. *Reference to external entities*. When a captured call uses facilities such as database links that point to remote databases it will fail during replay if the remote link does not exist.
5. *Interaction with scheduled jobs*. If there are scheduled jobs executing in the test system that interact with the same user data as some replayed sessions, there maybe divergence because we don't order commits within scheduler jobs.

Replay strives to minimize replay divergence in most practical situations. As a testing tool we assume some divergence due to corner cases can be tolerated.

4.2.6 Replay Options

Strict end state consistency, request rate consistency and result consistency might not be always desirable in all testing scenarios. Therefore replay provides options to tweak the replay behavior suitably for each test case. Commit ordering can be turned on or off. No commit ordering maybe suitable in cases where during replay minimal data divergence is not a requirement. Thus, bypassing the replay synchronization infrastructure may eliminate any synchronization overhead. This mode can also be used for stress testing where we don't want a slow commit action to slow down the calls the would otherwise wait for it.

The remaining three replay options (connect time scale, think time scale and auto-rate) tweak the request rate of the replay. By default connect time scale and think time scale are set to 100%, which is required for request rate consistency. They can be adjusted depending on the situation. For example, when trying to replay a single instance capture on a 4-node RAC system that could handle up to 4 times the request rate compared to a single instance setup, one could set think time scale to 25% thus quadrupling the request rate of the workload issued by the replay clients. Auto-rate is on by default in which case the replay thread will attempt to maintain the captured request rate by reducing the think time appropriately. If auto-rate is turned off the captured think time is not decreased to compensate for longer executing calls.

4.2.7 Replay Reports

After each replay the user is presented with extensive replay specific reports: one after capture and one after each replay. Furthermore, all extensive performance reports (AWR, Compare Period, ASH…) and oracle performance advisors (ADDM [4]) can be used during or after the replay. The goal of this paper is to present how these

reports can generally be used and not to accurately describe them in detail.

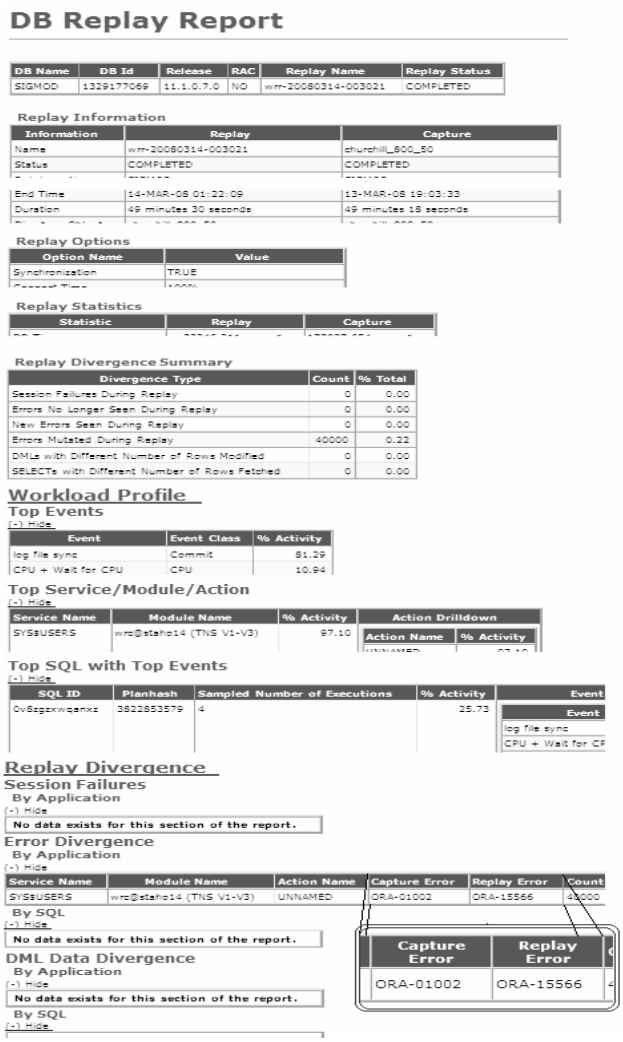

Figure 7 Sample replay report

The capture report contains data on how many calls were captured, what percentage of the total database workload was filtered out, how many errors were seen during capture, how many calls are supported for replay, how much CPU was consumed etc. The goal of the replay report is to enable the user to determine whether the right workload was captured, whether the workload can be replayed. It also provides a concise performance summary of the captured period.

The replay report juxtaposes some key capture data with the corresponding replay data (see Figure 7). This data contains among other items the elapsed time, CPU usage, IO activity etc. An entire section of the replay report is dedicated to replay divergence. The average magnitude of the row count divergence is reported as well as the new errors and changed errors during replay. The goal of this report is to help the user determine whether the replayed workload was a valid test. For example if the data divergence magnitude is too high, then it is highly unlikely that the replayed workload behavior is indicative of how the test system would perform in a production setting. If, however, the replay divergence is really low and the elapsed time difference is so high that it cannot be attributed to the potential replay synchronization overhead, then one can conclude that the test system contains a change relative to the captured system that makes it perform worse. If both the capture report and the replay report indicate that the replay exercises the test system as intended, then further performance specific testing tools can be used to drill down to specific performance areas (such as IO, CPU, latch contention etc).

5. CASE STUDY

This section showcases how Database Replay can be used for real testing. The goal is to experimentally evaluate and confirm that the testing results from a replay can be used to predict how the real workload under test will behave when a change is applied to the production system.

5.1 Methodology

Since using a real production workload from a live system to verify the usefulness of database replay is impossible, we will use two workloads that are as close to reality as possible: 1) An oracle in-house benchmark (IB) that was created from using a real customer application and data and 2) TPC-C, a widely used benchmark that models an OLTP system [18].

The IB workload is modeled after a real oracle customer that maintains a system for creating, storing and retrieving insurance quotes for various assets. The system is used by the insurance agents that are simulated by executing the workflows of the real insurance application used by the customer. The main reason we chose to use IB is because it is widely used internally at oracle for many purposes with very good results. The data for this workload consists of 71 tables and 57 indices on these tables and the database size is approximately 5GB. During the workload simulation 800 users connect to the RDBMS and compute insurance quotes. We run the workload for about 50 min at a time on a "warm" system. The TPC-C workload used 20 warehouses and 5 workload generators. The runtime for TPC-C is approximately 32 min.

The system on which the RDBMS is run is a dual CPU (hyper-threaded 32-bit Intel Xeon at 3.2Ghz) running Oracle Enterprise Linux kernel 2.6.9. The system has 6GB memory but for the IB workload the RDBMS is given 800MB for shared memory and 300MB for data buffer cache. The captured workload is directed to a different disc than the one used for the database. For the replay clients we use a different host with the same characteristics, which provides us with more than enough capacity to run 5 replay clients and 1 replay client for IB and TPC-C respectively.

The change tested in our experiments is advanced compression for tables, which is also a new feature in oracle 11 [12]. In our scenario we want to determine whether Database Replay can be used to predict the impact of compression on the real workload. The experimental scenario is as follows: First we run the real workload (IB or TPC-C) and capture it. Then we replay the captured workload in fully synchronized mode (commit ordering and capture request rate are maintained) and verify that the replay has the expected data divergence. In both cases, there is no divergence, which means that each replayed call performs the same work on the same data during replay as it did during capture. The only instance of divergence is found in the IB benchmark where ORA-1002 errors (fetch out of sequence during) become ORA-15566 errors (unreplayable error) (See Figure 7). This type of divergence is because we cannot yet replay a call so as to make it error out with ORA-1002. After the first replay, we enable advanced compression for all the tables used by each workload and perform another replay. Finally, we run the real workload with and without compression. The oracle automatic

workload repository (AWR) feature [12] is used for the performance comparison of key metrics among the run pairs shown in Figure 8. Each of the runs was executed 2 to 3 times to ensure repeatability.

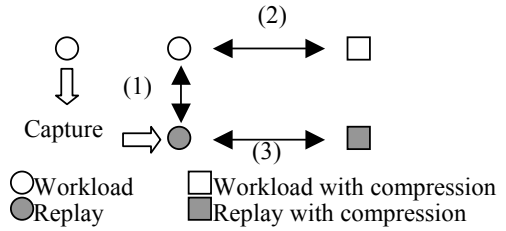

Figure 8 Performance comparisons

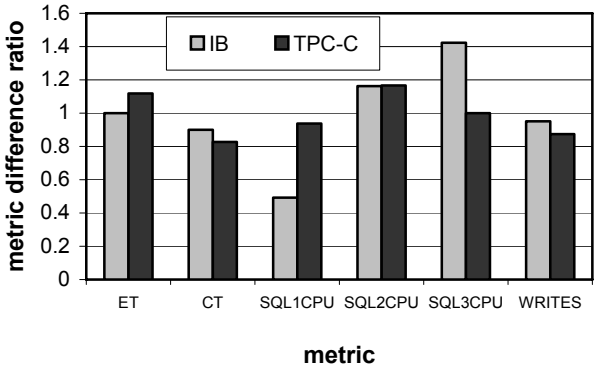

Figure 9 Ratios of metric differences

5.2 Results

5.2.1 Change Impact Evaluation

In Figure 8 the performance comparison between the actual workload and the replayed workload (1) is done to confirm that both runs exercise the RDBMS in a similar way. The performance comparison between the workload with and without compression (2) should lead to the same conclusions as the performance comparison between replay with and without compression (3). Figure 9 shows the ratio of the relative metric difference between (2) and (3). The ratios are computed as follows:

$$\frac{\left(M_{C(3)} - M_{(3)}\right)/M_{(3)}}{\left(M_{C(2)} - M_{(2)}\right)/M_{(2)}}$$

In the equation M_C means the value of a metric when compression is turned on and M is the value when compressions is turned off. The subscripts (3) and (2) refer to the comparison between runs in Figure 8. The ratios shown in Figure 9 are for the following metrics:

1. **ET:** Elapsed time of the workload
2. **CT:** Total CPU time consumed by the workload
3. **SQL1CPU, SQL2CPU, SQL3CPU:** Total CPU time of selected SQL statements (updates and queries) with measurable change when compression is enabled. They are different SQL between the two workloads.
4. **WRITES:** Total writes to the data table spaces.

Ideally all ratios should be +1. Database replay always correctly predicts the direction of the change and provides a good sense of the magnitude of the change. Unfortunately we cannot publish absolute numbers but the ratios are indicative of how the replay points to the direction of change. Also due to space limitations we cannot include the full AWR reports in this section.

5.2.2 Capture Overhead

As mentioned in Section 3.5 the overhead of capture is highly dependent on the workload. Thus the 4.5% throughput degradation of a TPC-C benchmark run is only indicative of how real workloads with similar characteristics will be affected during capture. The other interesting overhead is the space overhead required to store the captured workload. In our experiments the capture size was 2.7 GB and 0.32 GB for IB and TPC-C respectively. These figures are again indicative and depend on the workload. As can be seen the workload write rate is quite different between IB and TPC-C: 54MB/min and 1MB/min.

5.2.3 Transactional Model

In Section 4.1.1 we introduced the concept of replay testing success. In our case study replay testing success was 100% in all aspects: 1) The final database state after the workload, 2) the concurrency and request rate characteristics and 3) the results returned to the clients. Items 1 and 3 were achieved because the benchmarks we used have the following characteristics:

1. Well defined initial state before we start the workload capture
2. None of the captured requests use unsupported or rare features (such as pipes or references to external entities).
3. There is no significant application logic inside PL/SQL scripts that could lead to replay divergence.

Concurrency and request rate consistency was also achieved. In the case of the IB benchmark for example, the elapsed time of replay is almost the same as the elapsed time of capture (Capture: 49.30 min replay 49.41 min) and all replayed requests succeed and perform the same work as they did during capture. Thus the rate of real work of the replay approximates very closely that of capture (Capture: 243.4 transactions per second, Replay 243.2 transactions per second).

In real environments we expect it to be harder to achieve the level of success with respect to the transactional model that we have achieved in this case study. However, using the capture and replay reports as well the existing workload reports provided by the Oracle RDBMS the user should be able to determine whether a given replay is reflecting a production workload accurately enough so as to be used as a guide to determine the impact of changes in the production environment.

6. RELATED WORK

Microsoft SQL Server features capture and an uncoordinated replay functionality that is based on the SQL Trace [14] infrastructure. This infrastructure provides general-purpose event based tracing. SQL Profiler implements the replay functionality in the SQL server by allowing the replay of traces. The replay in SQL Server, however, does not maintain transactional dependencies and session coordination, and does not strive for result consistency of the replayed requests. Also, it is not clear whether SQL Trace can be enabled on the entire workload of a production system with reasonable overhead.

Quest Benchmark factory [6] attempts to provide capture and replay functionality for the Oracle RDBMS. The replay is based on Oracle SQL trace [12], which is a facility for tracing user sessions into text files. The disadvantage of this solution is that SQL trace cannot be enabled for a production system because of very high overhead. It was designed to trace only isolated sessions to diagnose problems. Furthermore, it does not contain transactional information necessary

to implement commit ordering and session coordination, similar to the SQL server approach.

The LoadRunner performance-testing tool [8] can be used to generate load against a test system. It is a very popular load generation tool that is used to test the entire software stack including the application, the middle tier and the RDBMS. Often it is used for RDBMS stress testing. The interaction of users with the application front-end is recorded. The resulting traces are parameterized and used to simulate load for an arbitrary number of users. The disadvantage of this method is that the captured workload contains only a small percentage of the production workflows since usually real users cannot be captured. Additionally, the requests generated are random and may not always do useful work in the RDBMS. Furthermore, it makes RDBMS-only testing cumbersome since it requires the middle tier to be present. A similar approach to LoadRunner is followed by [17] with the emphasis on automating the replay testing for web applications.

VMWare [20] offers with its new virtual machine an experimental feature that allows the recording and replay of the virtual machine activity. This approach captures at the operating system level, well below the RDBMS level and the focus is debugging of programs since it allows for the exact duplication of the operations and the state of the virtual machine thus enabling deterministic debugging. It is not designed to test changes to any software running on the virtual machine.

Deterministic replay of programs has been considered as a method that enables deterministic debugging and testing. The work in [2] describes a method to deterministically replay Java applications. The approach followed is similar to database replay in that they identify synchronization points in java programs and introduce a clock based thread scheduler inside the Java virtual machine. A similar approach is followed by [7]. In parallel systems, deterministic debugging is essential. [11] presents a debugging method for parallel systems based on instant replay.

The work presented in [10] follows a very similar approach to that of Database Replay in the context of file systems. File system traces are used to capture activity at the virtual file system (VFS) level, which is the layer that abstracts file system specific interface. These traces are then used to replay the captured activity at different speeds for stress testing.

Another workload generation approach for testing a database application is described in [5] and uses as input the application code and the database schema to produce user input to the application as well as data for exhaustively testing the application and the database. The generated workload is used to drive the application in an attempt to achieve extensive code coverage. The work in [1] proposes a query aware approach to generating test databases. It emphasizes the importance of test queries producing the desired intermittent and end results when they are executing on a test database. Database replay does not strive for intermediate result consistency but for end result consistency. The impact of intermediate results on the workload is part of testing a change. There is extended literature that proposes database and query generation methods for testing ([3], [9], [19], [13], [15] and [16]). Database replay makes it possible for the first time to use real data and real user workload for RDBMS testing.

Oracle Database Replay is part of the Real Application Testing feature that also includes the SQL Performance Analyzer (SPA) [21]. SPA can capture the SQL of a running RDBMS into a SQL tuning set that is a collection of unique SQL statements along with performance and plan information. The SQL tuning set can be used in a test environment to perform unit testing of individual SQL in isolation. Thus, Database Replay and SPA complement each other to provide a powerful set of tools: Database Replay is geared towards comprehensive throughput testing using real workloads, while SPA is targeted to unit testing that explores every aspect of a SQL statement in detail.

7. CONCLUSION

Change is relentless in today's rapidly evolving IT environments but it doesn't have to be difficult for data center managers and administrators. Database replay helps maintain the undesired effects of change to a minimum by making it possible to subject test systems to production quality workloads. Changes can be evaluated quickly with high confidence, and corrective action can be taken before production systems are negatively impacted. To our knowledge no other database vendor offers tools that come even close to real testing.

ACKNOWLEDGEMENTS

We would like to thank Richard Sarwal for his vision and support.

REFERENCES

[1] C. Binnig, D. Kossman, E. Lo, M.T. Özsu. QAGen: generating query-aware test databases. ACM SIGMOD international conference on Management of data 2007

[2] J.D. Choi, H. Srinivasan. Deterministic Replay of Java Multithreaded Applications. Proceedings of the SIGMETRICS symposium on Parallel and distributed tools 1998.

[3] DTM Data Generator. http://www.sqledit.com/dg/

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

[5] M. Emmi, R. Majumdar, K. Sen. Dynamic test input generation for database applications. International symposium on Software testing and analysis 2007.

[6] C. Fernandez, J. Leslie. Predicting and Preventing Performance Bottlenecks in Oracle 10g. Technical Brief, http://www.quest.com.

[7] A. Georges, M. Christiaens, M. Ronsse, K. De Bosschere. JaRec: a portable record/replay environment for multi-threaded Java applications. Software -Practice & Experience. May 2004.

[8] HP LoadRunner. http://www.hp.com.

[9] IBM DB2 Test Database Generator. http://www-306.ibm.com/software/data/db2imstools/db2tools/db2tdbg/

[10] N. Joukov, T. Wong, E. Zadok. Accurate and efficient replaying of file system traces. USENIX Conference on File and Storage Technologies 2005.

[11] T.J. LeBlanc, J.M. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. IEEE Transactions on Computers 1987.

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

[13] M. Poess, J.M. Stephens. Generating thousand benchmark queries in seconds. VLDB 2004

[14] SQL Server Profiler, RDBMS Documentation, http://msdn.microsoft.com.

[15] D.R. Slutz. Massive Stochastic Testing of SQL. VLDB 1998

[16] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, P.J. Weinberger. Quickly generating billion-record SIGMOD 1994

[17] S. Sprenkle, E. Gibson, S. Sampath, L. Pollock. Automated Replay and Failure Detection for Web Applications. 20th IEEE/ACM international Conference on Automated software engineering ASE '05

[18] TPC-C, http://www.tpc.org.

[19] J. M. Stephens,M. Poess. Mudd: a multi-dimensional data generator. WOSP 2004.

[20] VMWare Workstation 6 Record and Replay. http://www.vmware.com.

[21] K. Yagoub, P. Belknap, B. Dageville, K. Dias, S. Joshi, and H. Yu. Oracle's SQL Performance Analyzer. IEEE Data Engineering Bulletin. March 2008 Vol. 31 No. 1