Consistent Synchronization Schemes for Workload Replay

Konstantinos Morfonios, Romain Colle*, Leonidas Galanis, Supiti Buranawatanachoke*,
Benoît Dageville, Karl Dias, Yujun Wang
Oracle USA
400 Oracle Parkway
Redwood City, CA 94065
{firstname.lastname}@oracle.com

ABSTRACT

Oracle Database Replay has been recently introduced in Oracle 11g as a novel tool to test relational database systems [9]. It involves recording the workload running on the database server in a production system, and subsequently replaying it on the database server in a test system. A key feature of workload replay that enables realistic reproduction of a real workload is synchronization. It is a mechanism that enforces specific ordering on the replayed requests that comprise the workload. It affects the level of request concurrency and the consistency of the replay results when compared to the captured workload. In this paper, we define the class of consistent replay synchronization schemes and study, for the first time, the spectrum they cover and the tradeoffs they present. We place the only scheme proposed so far [9], the one implemented in Oracle 11g Release 1, within the aforementioned spectrum and show that it is coarse-grained and more restrictive than necessary, often enforcing dependencies between calls that are independent. By enforcing needless waits, it decreases the level of possible concurrency and degrades performance. To overcome these drawbacks, we identify the best scheme within the spectrum; it is finer-grained than its counterparts and strikes the right balance across different tradeoffs: it enforces a partial ordering on the replayed calls that minimizes the number of required waits and maximizes the level of concurrency, without compromising consistency of the replay results. We have implemented the new scheme in Oracle 11g Release 2. Our experiments indicate that it produces better quality replays than the pre-existing one for major classes of workload.

1. INTRODUCTION

An old Chinese proverb says that "a good customer should not change her shop, nor a good shop change its customers", indicating that even a small modification in something that works well is typically risky. The common sense expressed by this saying is reflected in the steadfast reluctance, when it comes to changes in large-scale information systems.

Although safe, such a conservative and cautious approach prevents progress. Changes are required to keep information systems up-to-date and competitive; hence, extensive testing is needed to reduce the related risks. Typically, changes are first implemented on a test system and verified by regression and stress tests. Finally, upon successful termination of all the tests, the changes are implemented on the production system as well. The quality of testing determines the severity and the number of problems that appear after the changes are applied to the production system. Tests that exercise a large portion of the production code and produce realistic loads on the test systems have a higher success rate in uncovering problems before it is too late.

Unfortunately, generating proper synthetic workloads for testing large-scale applications is a very hard problem [10]. Trying to mimic production system scenarios with manually written or tool generated scripts, or with simulated models has been shown inadequate, time consuming, and essentially ineffective [9]. The ideal would be to subject a test system to the actual production workload. Replaying a real workload on a database system enables such type of testing for large-scale database applications. We will use the terms database replay or workload replay to describe this type of testing.

Database replay has been originally introduced in Oracle 11g [9]. Figure 1 sketches how it works. A real workload running on the database server in a production system, appearing on the left side of the figure, is recorded with minimal overhead, preferably during peak period. The recorded workload (also called captured workload, hereafter) is subsequently replayed on the database server in a test system, appearing on the right side of the figure. Replaying such a real, typically heavy and highly concurrent workload has great potential for testing purposes, including regression and stress testing, debugging, and capacity planning. A captured workload is stored in a number of capture files. One capture file corresponds to a single database connection that was active during capturing on the production system. It stores the list of calls that were issued on the database server through the corresponding connection. Replaying the workload involves starting an adequate number of client processes that read the appropriate capture files and issue the recorded calls on the test database server through the corresponding connections. The goal is to recreate the same number of users during replay as there were during capture.

Due to the existence of multiple users issuing calls concurrently on the database server, a key feature of database

*Work done while at Oracle USA.

Proceedings of the VLDB Endowment, Vol. 4, No. 12

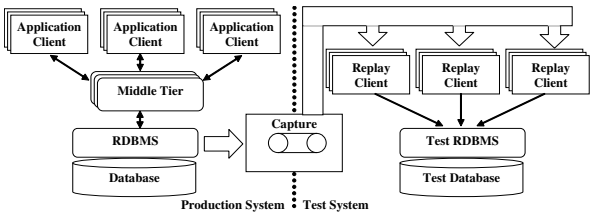


Figure 1: Capture and replay for database testing.

replay that enables realistic reproduction of a real workload is synchronization, i.e., a mechanism that enforces specific ordering on the replayed calls that comprise the workload. The employed synchronization scheme strongly affects the following important properties of replay.

Request concurrency: Enforcing ordering on specific calls coming from different replay clients inevitably introduces waits. Therefore, it affects concurrency.

Result consistency: The order in which data is committed and queried during replay determines the data returned or updated in the replay database. Therefore, the replayed requests may diverge in terms of work done from their capture-time counterparts.

All widely used database management systems (DBMSs) implement some concurrency control mechanism to guarantee database integrity [6, 13, 17]. Most of these mechanisms, including two-phase locking (2PL) and its variations, have been built based on the theory of serializability [6]. Such mechanisms ensure that the schedule they produce is serializable, i.e., equivalent with a serial schedule[1] that consists of the same transactions. Two schedules are equivalent if their individual calls read and write the same values. Note that the equivalent schedule can be any serial schedule.

Employing no replay synchronization scheme and simply relying on the concurrency control mechanism for synchronizing calls during replay only based on their timing at the client side allows for high concurrency, similar to the one during capture, but leads to inconsistent results, i.e., undetermined outcome when calls compete for resources. In other words, calls diverge and the replayed workload does not look like the captured one anymore, since the replay schedule is equivalent with an arbitrary serial schedule, and not necessarily with the schedule that was generated by the concurrency control mechanism during capture.

At the other end of the spectrum, if the scheme employed for replay synchronization imposes a total ordering on the captured calls, say one that strictly follows the execution order of these calls during capture, the data returned or affected during replay will be consistent, i.e., the same as during capture, but the concurrency of the workload will be very low compared to that of the real captured workload, since the replayed workload will be essentially serialized.

An ideal synchronization scheme must lie somewhere in the spectrum between these two extremes. It should enforce some partial ordering on the calls to guarantee consistency, while also allowing good potential for concurrency. Note that a replay synchronization scheme does not perform traditional concurrency control, in the sense that its goal is replay consistency, not database integrity. So, in the rest of this paper we will take for granted that the DBMS implements some additional mechanism for concurrency control [6, 13, 17]. Moreover, we will make the assumption that this mechanism provides snapshot isolation [6], a property supported in almost all widely used DBMSs.

In previous work [9], a practical synchronization scheme has been presented that enforces partial ordering on the replayed requests, thus allowing for reasonable levels of concurrency with no data divergence. In this paper, our goal is to formalize the characteristics of the existing replay synchronization scheme, discover new schemes, and nominate the best. Below, we summarize our main contribution.

Spectrum of consistent schemes: We study, for the first time, the spectrum of consistent replay synchronization schemes. Result consistency between capture and replay is very important, since it allows us to verify that the replayed workload did the same work as the captured workload.

Position of existing scheme within the spectrum: We find the position of the only scheme proposed so far for replay synchronization [9] within the aforementioned spectrum and show that it is coarse-grained and more restrictive than necessary, enforcing dependencies between calls that are independent. Unnecessary dependencies may cause degradation of replay concurrency due to increased waits.

Definition of an advanced scheme: The current replay synchronization scheme, while adequate in several practical applications, leaves room for improvement. In this paper, we present a finer-grained scheme that strikes the right balance across different tradeoffs; it generates a schedule that enforces a partial ordering, which minimizes the required number of waits among the calls coming from different clients and maximizes the level of concurrency, without compromising replay consistency. Our advanced scheme produces the least restrictive schedule that is still equivalent with the schedule produced by the concurrency control mechanism during capture. As we explain below, it achieves this goal by using some similar notions (e.g., the notion of collision dependency) and constructs (e.g., the dependency graph) also defined in the theory of conflict serializability [6]. However, it uses them for a different purpose: to ensure replay consistency rather than transaction serializability.

Efficient implementation: We discuss some ideas for efficiently implementing consistent synchronization schemes in database replay. Our methods apply to all schemes of the identified spectrum, not only to the advanced one.

Experimental evaluation: We have implemented the new scheme in Oracle 11g Release 2. We evaluate it experimentally against the previous one and present the results.

The remainder of this paper is organized as follows: In Section 2, we study the spectrum of consistent synchronization schemes. In Section 3, we describe our efficient implementation of an advanced scheme. In Section 4, we present the results of our experimental evaluation. In Section 5, we review related work. Finally, we conclude in Section 6.

2. CONSISTENT SYNCHRONIZATION

Testing any change routinely involves running some form of application workload on a test system and verifying correctness and performance. In database replay, a real workload is used; so, verification can be achieved by comparing two sets of results: the results of running the captured workload on the test system and the original results recorded while running the same workload on the production system. To make such a comparison meaningful, database replay

[1] In a serial schedule, transactions do not overlap in time.

must guarantee some well-defined behavior of the replayed workload; otherwise verification based on the fact that a real workload was used is impossible. We use the term replay consistency to describe this well-defined behavior. Qualitatively, replay consistency means that each replayed request to the database must perform the same work and operate on the same data as it did during capture. Only then is it possible to compare performance of capture and replay.

Inspired by the significance of consistency in the replay results, in this section, we study the types of inconsistency that can happen during replay and define the class of consistent replay synchronization schemes. Then, we identify the spectrum that these schemes cover. Finally, based on the properties of the points of this spectrum, we introduce an advanced synchronization scheme for workload replay.

2.1 Types of Inconsistency

Let us first focus on the sources of replay inconsistency. They have been also discussed elsewhere [9]; here, we categorize them in a way that will help us develop our new methodology. To begin with, we have to assume that the initial database state at the beginning of capture is identical to the initial state at the beginning of replay, since starting from an arbitrary state can practically never end up with consistent results. The state only includes user data and there are no assumptions on the layout, index, schema, or any other physical database characteristic. Fulfilling this requirement is rather straightforward by using standard features of DBMSs, e.g., export/import or backup/restore.

Having made the above assumption, we can identify two types of replay inconsistency, based on the source of divergence in the associated results: *systematic* and *random*. We define them below and present some examples after that.

Systematic inconsistency: This type of inconsistency appears in a reproducible fashion, when the relative ordering between two collision-dependent calls changes. Assuming that the DBMS guarantees snapshot isolation [6], two calls are collision-dependent if (a) at least one of them modifies the state of the database by committing some changes, and (b) both calls access at least one database object in common. For the sake of simplicity, unless otherwise stated, we can consider that a database object is merely a database table, since our focus is on relational databases. In the case of more complex objects, e.g. views, we assume that they can always be analyzed to the base tables upon which they are defined. Finally, in the case of finer-grained types of database objects (e.g., blocks, and rows) some more attention is required. We will elaborate on them in Section 2.3.5.

Random inconsistency: This type of inconsistency appears in an arbitrary, coincidental fashion, when a call depends on some value or condition related to the runtime state. Typically, calls that cause random inconsistency are calls that use the system date and time, or calls that generate and use random numbers.

Example 1 (systematic inconsistency): Here, we describe an extreme, imaginary situation, only for the sake of illustration. Assume a very large company employing 1,000,000 US employees that has been hit by major financial crisis. Imagine that the treasurer of the company executes query $\mathcal{Q}_1$ shown below to find out how many US employees work for the company and what their average salary is. Assume that the DBMS responds after some considerable amount of time, say 1 sec, since it evaluates two aggregate functions (*count* and *avg*) over one million rows, and returns the following result: ⟨1,000,000, $60,000⟩. Based on this result, the CEO of the company decides to outsource all the employees, but herself, to India. To do so, she executes query $\mathcal{Q}_2$.

```
Q1: select count(*), avg(salary) from emp where country='USA';
Q2: update emp set country='India'
      where country='USA' and job_title<>'CEO';
```

For the sake of illustration, let us assume that $\mathcal{Q}_2$ commits its changes automatically. In this case, we can easily see that $\mathcal{Q}_1$ and $\mathcal{Q}_2$ are collision-dependent, since (a) $\mathcal{Q}_2$ modifies the state of the database by changing table *emp*, and (b) both of them access the same database object (table *emp*).

Furthermore, imagine that the database administrator is planning a change on the company's system, say an upgrade of the DBMS version, but first, he wants to test its effects on the workload consisting of $\mathcal{Q}_1$ and $\mathcal{Q}_2$. To do so, he replays the workload on a testing database. There are two possible orderings for executing $\mathcal{Q}_1$ and $\mathcal{Q}_2$ during replay: $\mathcal{Q}_1 \rightarrow \mathcal{Q}_2$ and $\mathcal{Q}_2 \rightarrow \mathcal{Q}_1$ (symbol "$\rightarrow$" denotes "precedes"). Recall that the first ordering ($\mathcal{Q}_1 \rightarrow \mathcal{Q}_2$) was used on the production system during capture. Let us assume that the synchronization scheme chooses the second one ($\mathcal{Q}_2 \rightarrow \mathcal{Q}_1$) during replay.

After the execution of $\mathcal{Q}_2$, table *emp* stores only one row that corresponds to a US employee, the CEO. Executing $\mathcal{Q}_1$ after $\mathcal{Q}_2$ returns ⟨1, $600,000⟩, assuming that $600,000 is the CEO's salary. This result differs from $\mathcal{Q}_1$'s previous result, the one during capture. Furthermore, computation of $\mathcal{Q}_1$'s result has become trivial during replay, since the aggregate functions operate on a single row, not on one million rows as during capture. So during replay, taking into account some proper indexing on *emp*, the response time of $\mathcal{Q}_1$ must be orders of magnitude faster, say 0.01 sec. Clearly, the speed-up cannot be attributed to the DBMS upgrade.

To conclude, the use of different ordering for two collision-dependent calls has generated systematic inconsistency between capture and replay. As expected, the results are different. The work is also different and therefore the replay performance cannot be used as an indicator of what effect the tested change will have on the production system. ■

Example 2 (random inconsistency): Imagine that in 2010 the financial state of the company of example 1 recovered; so, 100,000 new employees were hired in October, but no more hires were made in 2010. That October, the CEO executed $\mathcal{Q}_3$ shown below to see the number of new hires during that month. In $\mathcal{Q}_3$, function *to_char* converts a date into a string with some given format, *h_date* is a column of table *emp* that stores an employee's hiring date, and *sysdate* is a function that returns the system date. Assume that the result of $\mathcal{Q}_3$ was 100,000 and its response time 0.5 sec.

```
Q3: select count(*) from emp
      where to_char(h_date,'mmyyyy')=to_char(sysdate,'mmyyyy');
```

The next month, replaying a workload containing $\mathcal{Q}_3$ generated a different result for $\mathcal{Q}_3$ (0, since no new hires were made after October). Moreover, during this replay, $\mathcal{Q}_3$'s response time was much faster (say 0.01 sec), since the aggregate function *count* operated on an empty group of rows. In this case, divergence has happened due to random inconsistency, since the result of $\mathcal{Q}_3$ depends on the state of the runtime (in this example, the date returned by *sysdate*). ■

2.2 Consistent Synchronization Schemes

Having studied the sources of replay inconsistency in the previous subsection, in this subsection, we define the class of consistent synchronization schemes for workload replay.

Let us first focus on systematic inconsistency. Clearly, its occurrence during replay depends heavily on the synchronization scheme employed, since the source of systematic inconsistency is the existence of changes in the ordering of collision-dependent calls. Hence, a minimum requirement for a consistent replay synchronization scheme is ensuring that collision-dependent calls are always replayed in the same order as during capture, in order to prevent systematic inconsistency. Note that this order is the one dictated by the schedule of the concurrency control mechanism that was running on the production system during capture.

Let us now focus on random inconsistency. It is easy to observe that this kind of inconsistency is inherent in the type of some captured calls, the ones that depend on the state of the runtime, as explained above. Since it is attributed to the nature of these calls and not to their ordering, synchronization cannot address the corresponding problem. Some additional machinery, which is orthogonal to the synchronization scheme employed, is required for this purpose. In summary, tackling random inconsistency is a matter of correlating and reproducing random results. Existing techniques [9] avoid it by capturing the real values that depend on the runtime state of the production system (e.g., the result of *sysdate* for call $\mathcal{Q}_3$) and subsequently using these recorded values to substitute the values that would be generated based on the runtime state of the test system during replay. In this paper, we will take for granted that random inconsistency does not exist, since our focus is on synchronization schemes and techniques orthogonal to them can take good care of it.

Based on the observations above, we can easily deduce that by avoiding systematic and random inconsistency workload replay adheres to a schedule that is equivalent with the schedule that was executed during capture, since such a replay preserves the same ordering of collision-dependent calls and eliminates all possible sources of divergence.

Having realized how consistent synchronization schemes should behave, we provide below their formal definition. We begin with some notation that we will use throughout the paper. Let $\mathcal{S}$ denote the set of all calls in a given workload, and let $a, b \in \mathcal{S}$ be two of these calls. Moreover, assume that the following predicates exist: $precedes(a, b)$ returns *true* if a was executed before b during capture, or *false* otherwise; $commit(a)$ returns *true* if a modifies the database state by committing some changes, or *false* otherwise; $access\_common\_object(a, b)$ returns *true* if a and b access a database object in common, or *false* otherwise; $must\_wait(a, b)$ returns *true* if a must wait for b to finish before a can be replayed, or *false* otherwise. Finally, let us define the following sets:

- **Set of precedence-dependent pairs:** $\mathcal{S}_{pre} = \{(a, b) \in \mathcal{S}^2 : precedes(a, b)\}$
- **Set of commit-dependent pairs:** $\mathcal{S}_{com} = \{(a, b) \in \mathcal{S}_{pre} : commit(a) \vee commit(b)\}$
- **Set of object-dependent pairs:** $\mathcal{S}_{obj} = \{(a, b) \in \mathcal{S}_{pre} : access\_common\_object(a, b)\}$
- **Set of collision-dependent pairs:** $\mathcal{S}_{col} = \mathcal{S}_{com} \cap \mathcal{S}_{obj}$

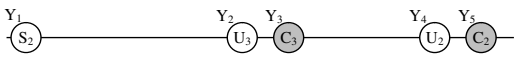


Figure 2: Example of a captured session.

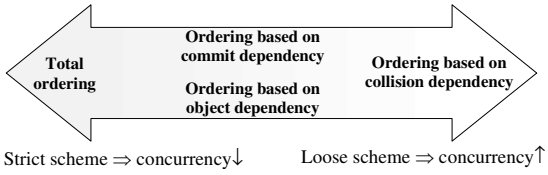


Figure 3: Spectrum of consistent schemes.

Clearly, the following relationships hold for these sets: $\mathcal{S}_{pre} \supseteq \mathcal{S}_{com} \supseteq \mathcal{S}_{col}$, and $\mathcal{S}_{pre} \supseteq \mathcal{S}_{obj} \supseteq \mathcal{S}_{col}$.

To explain our notation, let us give some examples. Assume some captured session Y consisting of five calls: Y_1-Y_5 (Figure 2). The horizontal axis represents time; so, Y_1 has been executed before Y_2 during capture, Y_2 before Y_3, and so on. Furthermore, assume three types of calls: selects, updates, and commits, denoted by letter S, U, and C, respectively. To better distinguish commit calls from non-commit calls, we use grey color for the former (e.g., Y_3) and white color for the latter (e.g., Y_1). Each call operates on a set of objects, denoted by the subscript that follows letter S, U, or C. For instance, Y_1 selects some results from object 2, Y_2 updates object 3, whereas Y_3 commits changes on object 3.

In our example, $precedes(Y_2, Y_5)$, $commit(Y_3)$, and $access\_common\_object(Y_2, Y_3)$ return *true*, but $precedes(Y_4, Y_1)$, $commit(Y_1)$, and $access\_common\_object(Y_1, Y_3)$ *false*. Finally, the elements of various sets follow: $\mathcal{S}$={Y_1, Y_2, Y_3, Y_4, Y_5}, $\mathcal{S}_{pre}$={(Y_1, Y_2), (Y_1, Y_3), (Y_1, Y_4), (Y_1, Y_5), (Y_2, Y_3), (Y_2, Y_4), (Y_2, Y_5), (Y_3, Y_4), (Y_3, Y_5), (Y_4, Y_5)}, $\mathcal{S}_{com}$={(Y_1, Y_3), (Y_1, Y_5), (Y_2, Y_3), (Y_2, Y_5), (Y_3, Y_4), (Y_3, Y_5), (Y_4, Y_5)}, $\mathcal{S}_{obj}$={(Y_1, Y_4), (Y_1, Y_5), (Y_2, Y_3), (Y_4, Y_5)}, and $\mathcal{S}_{col}$={(Y_1, Y_5), (Y_2, Y_3), (Y_4, Y_5)}.

Using our notation, we can formally define a consistent replay synchronization scheme as follows.

Definition (consistency rule): A replay synchronization scheme is consistent, if it satisfies the *consistency rule*: $\forall (a, b) \in \mathcal{S}^2((a, b) \in \mathcal{S}_{col} \Rightarrow must\_wait(b, a))$.

2.3 Spectrum of Consistent Schemes

Conceptually, the main job of a consistent synchronization scheme for workload replay is the following one: given a pair of calls $(a, b) \in \mathcal{S}^2$, it has to decide if one of them must wait for the other one to finish in order to guarantee consistency. There are three possible mutually excluding decisions. *Decision 1:* $must\_wait(a, b)$, *Decision 2:* $must\_wait(b, a)$, and *Decision 3:* $(\neg must\_wait(a, b)) \wedge (\neg must\_wait(b, a))$.

As we show below, depending on how *strict* or *loose* its strategy is for making the aforementioned decision, we can place a synchronization scheme in the corresponding position within the spectrum of consistent schemes. Figure 3 visualizes this spectrum. As we move from left to right within it, we move from stricter schemes to looser ones; hence, the level of replay concurrency increases in this direction. The strictest possible scheme (on the left) enforces a total ordering on all calls that comprise the workload. Looser schemes enforce some partial ordering based on commit, object, or collision dependency. In practice, partial ordering means

that two or more calls can be issued at the same time without any concerns that they will affect each other in terms of the type of work they will do. They will probably contend for DBMS resources, which is one of the aspects we need to test. Below, we discuss the different points of the spectrum.

In order to better explain the differences among the different points of the spectrum, let us extend our running example of Figure 2 and assume that we have captured three concurrent sessions running on our production system, namely X, Y, and Z (Figure 4(a)). Then, Figures 4(b)-4(e) show examples of replaying these sessions using different synchronization schemes. We give further details below.

2.3.1 Total Ordering

Enforcing specific total ordering, the exact same ordering that happened during capture, among all calls that comprise a captured workload is the most straightforward way to avoid systematic inconsistency. By preserving the original ordering of all calls during replay, a synchronization scheme based on total ordering guarantees that it also preserves the ordering of collision-dependent calls; hence, it is consistent.

Formally, a scheme of this category enforces the *total ordering rule*: $\forall(a,b) \in \mathcal{S}^2((a,b) \in \mathcal{S}_{pre} \Rightarrow must\_wait(b,a))$.

Since $\mathcal{S}_{pre} \supseteq \mathcal{S}_{col}$, if $(a,b) \in \mathcal{S}_{col}$, then $(a,b) \in \mathcal{S}_{pre}$, as well. Hence, by enforcing the total ordering rule, a synchronization scheme of this class indirectly enforces the consistency rule, too. Therefore, it is consistent.

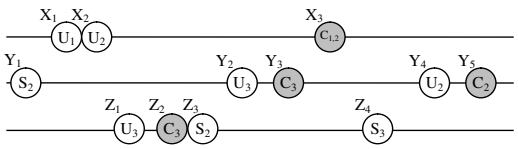
In our running example, if the captured workload is the one in Figure 4(a), a synchronization scheme based on total ordering could have generated a replay workload like the one in Figure 4(b). If we compare the two workloads, we can observe that although some calls have been shifted in time during replay, their ordering still remains the same.

Although consistent, a synchronization scheme based on total ordering is too restrictive for replay concurrency. Actually, due to enforcing a specific ordering between every single pair of calls, it is the strictest scheme possible. This property explains why it appears in the first position (to the left) within the spectrum of Figure 3. For instance, in the example of Figure 4(b), call X_2 has delayed to execute during replay for some reason. Unfortunately, a synchronization scheme based on total ordering has to propagate this delay to all subsequent calls in order to preserve the specified ordering. So, even calls Z_1, Y_2, and all their subsequent calls, most of which are collision-independent with X_2, have to wait until X_2 finishes. This synchronization scheme does not allow for realistic concurrency even though it guarantees consistency. It is presented only for illustration purposes.

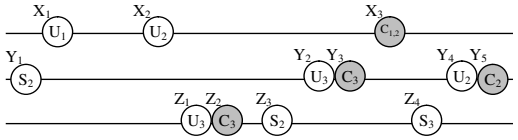
2.3.2 Ordering Based on Commit Dependency

Having described the strictest extreme of the spectrum, a scheme based on total ordering, let us proceed towards looser schemes. A promising candidate scheme that enforces some partial ordering could be based on commit dependency. This scheme is available in Oracle 11g and has been presented elsewhere [9]. Exploiting the property that any pair of non-commit calls is never collision-dependent, a scheme based on commit dependency enforces no specific ordering between non-commit calls. On the other hand, it enforces specific ordering between two calls, if at least one of them modifies the state of the database by committing some changes.

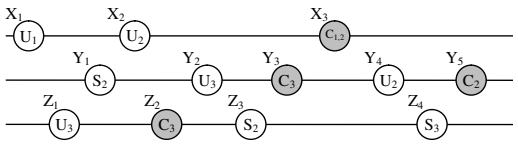
Formally, schemes of this class enforce the *commit dependency rule*: $\forall(a,b) \in \mathcal{S}^2((a,b) \in \mathcal{S}_{com} \Rightarrow must\_wait(b,a))$.



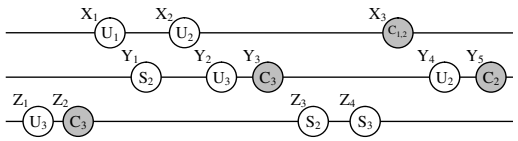
(a) Three captured sessions.



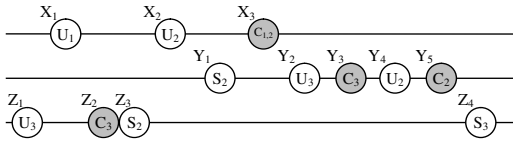
(b) Replay based on total ordering.



(c) Replay based on commit dependency.



(d) Replay based on object dependency.



(e) Replay based on collision dependency.

Figure 4: Examples of replaying three captured sessions using different synchronization schemes.

Since $\mathcal{S}_{com} \supseteq \mathcal{S}_{col}$, if $(a,b) \in \mathcal{S}_{col}$, then $(a,b) \in \mathcal{S}_{com}$, as well. Hence, by enforcing the commit dependency rule, a synchronization scheme of this category indirectly enforces the consistency rule, too. Therefore, it belongs to the spectrum of consistent schemes (Figure 3).

In our running example, if the captured workload is the one in Figure 4(a), a synchronization scheme that enforces ordering based on commit dependency could have generated a replay workload like the one in Figure 4(c). In this example, we can observe that there is no specific ordering among non-commit calls (white circles) between two consecutive commit calls (grey circles). This property allows more freedom and therefore achieves a better level of concurrency during the period between commit calls. For instance, in Figure 4(c), call X_2 has once more delayed to execute during replay. The same had happened in Figure 4(b). The difference is that this time, Z_1 does not have to wait for it, since both of them are non-commit calls. So, Z_1 and X_2 have been reordered during replay. Since they are collision-independent, such a reordering does not harm consistency. Note that similar reordering appears among several non-commit calls that lie between two commit calls. In our example of Figure 4(c), this includes calls Y_2 and Z_3, as well as Y_4 and Z_4.

To the best of our knowledge, a synchronization scheme that enforces ordering based on commit dependency is the only replay synchronization scheme that has been proposed so far in the existing literature [9]. It may be looser than one using total ordering, which explains why we have placed it at the second position within the spectrum of Figure 3, but it is still stricter than necessary, since it does not take into consideration the data a pair of calls accesses when it decides for their ordering. So, it enforces waits between calls that access completely disjoint sets of data, if at least one of these calls modifies the database state, although such calls are collision-independent. For instance, in the example of Figure 4(c), the synchronization scheme would enforce waits between the following pairs of calls, simply because at least one of them commits some changes: *must_wait*(Z_2, X_2), *must_wait*(Z_2, Y_1), *must_wait*(Z_4, X_3). In this example, the aforementioned calls are collision-independent. By enforcing unnecessary waits, a scheme of this category still may degrade concurrency and replay performance.

2.3.3 Ordering Based on Object Dependency

Following some logic similar to the one of schemes that are based on commit dependency, the schemes that are based on object dependency form another category of schemes that enforce some partial ordering among the calls that comprise a captured workload. The property they exploit is that any pair of calls whose members access disjoint sets of objects is collision-independent. Therefore, schemes based on object dependency never enforce a specific ordering between such pairs of calls. On the other hand, they enforce specific ordering between two calls, if these calls access at least one database object in common.

Formally, schemes of this class enforce the *object dependency rule*: $\forall(a,b) \in \mathcal{S}^2((a,b) \in \mathcal{S}_{obj} \Rightarrow must\_wait(b,a))$.

Since $\mathcal{S}_{obj} \supseteq \mathcal{S}_{col}$, if $(a,b) \in \mathcal{S}_{col}$, then $(a,b) \in \mathcal{S}_{obj}$, as well. Hence, by enforcing the object dependency rule, a synchronization scheme of this class indirectly enforces the consistency rule, too. Therefore, it is consistent.

In our running example, if the captured workload is the one in Figure 4(a), then a synchronization scheme that enforces ordering based on object dependency could have generated the replay workload in Figure 4(d). In this example, we can observe that there is no specific ordering between calls that access disjoint sets of database objects. This property allows some more freedom and therefore achieves a better level of concurrency compared to the one of total ordering, since it avoids some unnecessary delays. For instance, during replay (Figure 4(d)), although calls Y_1 and X_1 have delayed to start for some reason, this delay has not affected calls Z_1 and Z_2, which have now become the first calls to execute. The reordering of Z_1 and Z_2 with X_1 and Y_1 is possible during replay, since the former set of calls accesses object 3, whereas the latter set accesses objects 1 and 2, respectively. So, a scheme based on object dependency would never enforce waits between them. In our example, similar reordering has also been possible for other pairs of collision-independent calls, e.g., Z_4 and X_3, as well as Z_3 and Y_3.

A synchronization scheme that uses ordering based on object dependency may be looser than one using total ordering, which explains why we place it on the right of it within our spectrum (Figure 3), but it is still stricter than necessary, since it does not distinguish between commit and non-commit calls when it decides for their ordering. So, it enforces waits even between non-commit calls, if they access objects in common, although they are collision-independent; thus, it creates some unnecessary waits (read-only dependencies). For instance, in Figure 4(d), the synchronization scheme enforces waits between the following pairs of calls, just because they access objects in common: *must_wait*(X_2, Y_1), *must_wait*(Z_3, X_2). The aforementioned calls are non-commit and hence collision-independent. By enforcing unnecessary waits, a scheme of this category still degrades concurrency and decreases replay performance.

When it comes to comparing which replay synchronization scheme is stricter between one that is based on commit dependency and one that is based on object dependency, the answer depends on the properties of the given workload. In more detail, which one is stricter depends on the relationship between the sizes of the corresponding sets, i.e., $|\mathcal{S}_{com}|$ and $|\mathcal{S}_{obj}|$. The more elements a set contains, the more waits the corresponding synchronization scheme must enforce, hence the stricter it is. So, for example, if for a given workload $|\mathcal{S}_{com}| > |\mathcal{S}_{obj}|$, then a scheme based on commit dependency is stricter for this workload. Based on the observation that in the general case we cannot know which scheme between these two is stricter, in Figure 3, we have placed schemes that enforce ordering based on commit dependency at the same coordinates on the horizontal axis with schemes that enforce ordering based on object dependency.

2.3.4 Ordering Based on Collision Dependency

All the synchronization schemes we have presented so far are stricter than necessary, since they enforce waits between pairs of collision-independent calls. The schemes that are based on collision dependency overcome this drawback by directly enforcing the consistency rule (Section 2.2).

In our running example, if the captured workload is the one in Figure 4(a), then a synchronization scheme that enforces ordering based on collision dependency could have generated the replay workload in Figure 4(e). In this example, we can observe that call Y_1 (the first call during capture) has delayed during replay for some reason. However, a very large number of other calls have been executed before it without having to wait for it: X_1, X_2, Z_1, Z_2, and Z_3. If the ordering was based on commit dependency, Z_2 would have to wait. If the ordering was based on object dependency, X_2 and Z_3 would have to wait. However, a scheme of this class does not suffer from these problems, since it enforces the minimum required number of waits, i.e., only between calls that are collision-dependent.

The previous example indicates the advantages of a loose scheme. Actually, as we have shown above, a scheme that orders calls based on collision dependency is the loosest possible (this explains why we have placed in the rightmost position within the spectrum of Figure 3). Hence, it is more promising than the one proposed in the existing literature (Section 2.3.2), and therefore the scheme of our choice. We have implemented the new scheme in Oracle 11g Release 2 as an alternative to the pre-existing one.

2.3.5 Investigation of Using Finer-Grained Objects

In the previous subsection, we presented schemes that enforce ordering based on collision dependency and argued that they are the loosest possible within the spectrum of consistent schemes. Furthermore, in Section 2.1, we mentioned that for the sake of simplicity, we consider that a

database object is merely a table. So the question that arises is whether we could devise even looser schemes, if we used objects at a finer level of granularity, e.g., blocks or rows.

Intuitively, this alternative sounds reasonable. Considering objects at a finer level of granularity should reduce the number of dependencies and could therefore produce a scheme that enforces a smaller number of waits between calls, resulting into even higher levels of concurrency during replay. Unfortunately, this intuitively reasonable approach does not work in practice, since it may generate systematic inconsistency. We show that with a counterexample.

Counterexample (use of blocks or rows): Assume the same company we described in examples 1 and 2 (Section 2.1). Let its database contain a table *emp* storing information about 1,000,000 employees, which correspond to 1,000,000 rows R_1-$R_{1,000,000}$. Furthermore, consider that, on the production system, table *emp* consists of 20,000 blocks B_1-$B_{20,000}$. Let each block fit 50 rows; hence, all of them are full. Moreover, imagine that the company acquires another company, which also has 1,000,000 employees. Finally, assume two queries $\mathcal{Q}_4$ and $\mathcal{Q}_5$, expressed in SQL as follows.

$\mathcal{Q}_4$: *select avg(salary) from emp;*
$\mathcal{Q}_5$: *insert into emp (select * from acquired_emp);*

$\mathcal{Q}_4$ prints out the average salary of all the employees, while $\mathcal{Q}_5$ inserts the employees of the acquired company to the table *emp* of the first company. For simplicity, we assume that $\mathcal{Q}_5$ automatically commits its changes. Executing $\mathcal{Q}_4$ before $\mathcal{Q}_5$ aggregates rows R_1-$R_{1,000,000}$ from blocks B_1-$B_{20,000}$.

Additionally, $\mathcal{Q}_5$ inserts 1,000,000 new rows in *emp*, say $R_{1,000,001}$-$R_{2,000,000}$. Since all blocks B_1-$B_{20,000}$ are full, the new rows go into newly allocated blocks, say $B_{20,001}$-$B_{40,000}$.

If we consider objects at the block level, then $\mathcal{Q}_4$ and $\mathcal{Q}_5$ are collision-independent, since they access different objects (the former accesses blocks B_1-$B_{20,000}$, whereas the latter $B_{20,001}$-$B_{40,000}$). Likewise, if we consider objects at the row level, then $\mathcal{Q}_4$ and $\mathcal{Q}_5$ are once again collision-independent, since they access different objects (the former accesses rows R_1-$R_{1,000,000}$, whereas the latter $R_{1,000,001}$-$R_{2,000,000}$). In both cases, a scheme that orders calls based on collision dependency would not enforce a wait between these calls during replay. This would cause systematic inconsistency, since in case of reordering $\mathcal{Q}_4$ and $\mathcal{Q}_5$ during replay, $\mathcal{Q}_4$ would aggregate 2,000,000 rows (R_1-$R_{2,000,000}$) instead of 1,000,000 that it had aggregated during capture. This difference would affect both the result and the performance of $\mathcal{Q}_4$. ■

Apparently, a scheme based on block-level or row-level dependency cannot guarantee consistency in the general case. Solutions identifying overlaps between queries that dynamically determine their read set in an attempt to solve the problem might be possible in theory. However, we strongly believe that they would not be practical in a real system, due to their expectedly high complexity. Therefore, we do not investigate them any further.

3. EFFICIENT IMPLEMENTATION

The architecture of the first release of Oracle Database Replay has been described in great detail in previous work [9]. We summarized it briefly in Section 1 and Figure 1. In this section, we extend the description found in the original publication by explaining the extensions we implemented in the second release of the tool to make it support the advanced synchronization scheme proposed in this paper.

3.1 Discovery of Dependencies

Oracle Database Replay works as follows: A given number of processes, called replay clients (Figure 1), scan through the calls recorded in the capture files and send appropriately concurrent requests to the testing database server through an adequate number of connections asking for execution of these calls. By doing so, the replay clients simulate the application and middle-tier layers that were running on the production system during capture.

To ensure consistency in the replay results, before executing a call a, the testing database server first identifies all the other calls on which a depends according to the synchronization scheme employed, and enforces a wait on a until all of them have finished. Searching for such dependencies on the fly can be computationally very expensive, due to the typically large number of calls comprising a workload. Hence, since time is critical during replay, an efficient implementation of replay synchronization requires execution of a preprocessing algorithm in an off-line fashion that builds in advance the graph $\mathcal{G}$ of dependencies between calls. $\mathcal{G}$ is a directed acyclic graph (DAG) that represents the partial ordering among the calls that comprise the workload. These calls are the vertices of $\mathcal{G}$. A directed edge starts from a vertex a and ends at a vertex b, if $must\_wait(b, a)$ is true, according to the synchronization scheme employed. The use of a dependency graph for our problem resembles the use of a similar graph in theory of conflict serializability [6]. However, unlike the latter, our graph is acyclic by design. Moreover, we are using it to attack a different problem: replay consistency, rather than transaction serializability.

Algorithm *BuildDependencyGraph* presents in pseudocode our efficient preprocessing algorithm that computes the dependency graph for workload replay. Its input is the set of calls $\mathcal{S}$ that comprise the captured workload. Its output is the dependency graph $\mathcal{G}$. Initially, the algorithm sets $\mathcal{G}$'s vertices to $\mathcal{S}$ (line 1) and initializes $\mathcal{G}$'s edges to the empty set (line 2). Then, it enumerates all the candidate pairs of calls in $\mathcal{S}$ that are potentially dependent in order to identify the edges of $\mathcal{G}$. To do so, it visits every call a in $\mathcal{S}$, ordered by the identifier cf_a of the capture file that stores a and the start-execution time t_a at which the execution of a had started during capture (line 3). Then, it searches for every other call b on which a might depend.

A straightforward implementation of the aforementioned search of b would exhaustively enumerate all possible calls of the workload. Since the number of these calls is typically very large, the exhaustive approach is impractical. To overcome this drawback, below, we present some properties based on which our algorithm can prune its search space by predicting in advance that the synchronization scheme does not need to work on particular pairs of calls.

Property 1 *(Avoid search within the same capture file):* All the calls that are stored in a particular capture file are the ones that were sent to the database server through the same connection on the production system. Anything that came into the database server through the same connection should just be replayed in the same way. Such behavior mimics in a more realistic fashion the production workload a database sees. Recall that reordering of calls during replay tries to achieve better concurrency across requests coming from different threads of execution and not to change the natural order of calls coming from the same one, since the latter change would modify the behavior of the users during

Algorithm BuildDependencyGraph($\mathcal{S}$:in, $\mathcal{G}$:out)

```
1:  G.V = S;
2:  G.E = ∅;
3:  foreach(a ∈ S, ordered by cf_a and t_a) do
4:      foreach(cf ∈ S.capture_files)
5:          if(cf == cf_a) then
6:              continue;
7:          end if
8:          found = FALSE;
9:          b_min = RecallDependency(cf, a.previous_call);
10:         b_max = FindFirstCallAfterGivenTime(cf, t_a);
11:         b = b_max.previous_call;
12:         while(!found && t_b > t_b_min && t_b < t_b_max) do
13:             decision = ApplyRule(a, b);
14:             switch(decision)
15:                 case Decision 1:
16:                     G.E = G.E ∪ {(b→a)};
17:                     found = TRUE;
18:                     break;
19:                 case Decision 2:
20:                     RaiseError("This case can never happen");
21:                     break;
22:                 case Decision 3:
23:                     b = b.previous_call;
24:                     break;
25:             end switch
26:         end while
27:     end foreach
28: end foreach
29: Reduce(G);
```

replay. Hence, although possible, reordering calls stored in the same capture file is not recommended.

Based on this property, *BuildDependencyGraph* does not search for dependent calls stored in the same capture file; hence, for every call a, it scans through every capture file cf (line 4), but skips cf_a, the file that stores a (lines 5-7).

Property 2 *(Constrain search within a range):* For every file cf other than cf_a, our algorithm scans through the calls stored in it (line 12). As we explain below, instead of performing a time-consuming scan of the entire file, it constrains the search space into a particular range of calls.

Let t_a be the time at which a was executed during capture and $a.previous\_call$ the previous call of a in file cf_a ($a.previous\_call$ can be NULL, if a is the first call in cf_a). Moreover, let b_0 and b_∞ be two artificial calls that our algorithm considers as existing by default in cf, assuming that t_{b_0}=0 and t_{b_∞}=∞. They play the role of the first and last call in cf, respectively. They are not replayed, but only used for easier handling of boundary cases, as we explain below.

Furthermore, let *RecallDependency* (line 9) be a method that returns a call in cf, say b_{min}, for which $a.previous\_call$ must wait. If such a call exists, it is unique (based on Property 3 below) and *BuildDependencyGraph* must have already found it in some previous iteration, since it processes calls ordered by start-execution time (line 3). Otherwise, if $a.previous\_call$ is NULL, or such a call does not exist, *RecallDependency* returns b_0, the artificial call defined above.

Finally, let *FindFirstCallAfterGivenTime* (line 10) be a method that returns the first call in cf, say b_{max}, such that $t_a < t_{b_{max}}$ (b_{max} is the first call in cf executed after a). If such a call does not exist, the method returns b_∞.

Based on the previous definitions, the following precedence holds: b_{min}→$a.previous\_call$→a→b_{max}. Hence, our algorithm (line 12) suffices to consider as candidate calls in cf for which a needs to wait, only calls that belong to the range (b_{min}, b_{max}). Note that the aforementioned range is open on both sides. Therefore, it does not include b_{min} and b_{max}, and is valid even when b_{min}=b_0, or b_{max}=b_∞.

Property 3 *(Search in reverse time order and stop on first match):* For every candidate pair it generates, say (a, b), *BuildDependencyGraph* calls method *ApplyRule* (line 13). Depending on the synchronization scheme it employs, *ApplyRule* combines the evaluation of some proper subset of the predicates *precedes*, *commit*, and *access_common_object*, defined in Section 2.2. It returns a decision that can be Decision 1, 2, or 3 (the three alternatives have been defined in Section 2.3), indicating whether specific ordering is required between the individual calls a and b. Note that *ApplyRule* is the only part of our algorithm that depends on a particular synchronization scheme. Other than that, our algorithm is general enough and applies to all consistent synchronization schemes we described in the previous section.

If *ApplyRule* returns Decision 1, call a must wait for b; hence, the algorithm adds edge b→a in $\mathcal{G}$'s edges (line 16). We argue that after finding b, a call in cf for which a must wait, our algorithm does not need to keep searching for other calls in cf preceding b on which a depends; a will be indirectly executed after all of them, since it will be explicitly executed after b, and all of them will have been executed before b, based on Property 1. Therefore, enforcing an explicit wait on a for any one of them would be redundant. This observation explains Property 3, based on which *BuildDependencyGraph* searches within cf in reverse start-execution time order and stops searching upon finding the first match.

BuildDependencyGraph implements searching in reverse start-execution time order as follows: It initializes b with the call in cf that precedes b_{max} (line 11). This is the call with the maximum start-execution time that belongs to the range defined by Property 2. As long as *ApplyRule* (line 13) returns Decision 3 indicating that a and b are independent, the algorithm iterates backwards, by considering as new value for b the call that precedes the old b in cf (lines 22-24). Upon finding the first match (lines 15-18), the algorithm sets flag *found* to TRUE. This forces the algorithm to exit the loop in line 12, interrupting the search.

So far, we have explained what happens when *ApplyRule* returns Decision 1 or 3. Decision 2 implies that call b must wait for a. In our setting, such a result is impossible, since every call b that our algorithm visits belongs to the range defined by Property 2 and all the calls in this range must have been executed before a during capture, by definition. In the impossible case of Decision 2 (lines 19-21), our algorithm raises an error for the sake of completeness.

Property 4 *(Prune edges in $\mathcal{G}$):* After enumerating all required pairs of calls and constructing graph $\mathcal{G}$, *BuildDependencyGraph* finally calls method *Reduce* (line 29). *Reduce* prunes some redundant edges in $\mathcal{G}$. In more detail, it computes the transitive reduction of $\mathcal{G}$ [5], which is the subgraph of $\mathcal{G}$ that has a minimum number of edges and represents the same partial ordering as $\mathcal{G}$. Since $\mathcal{G}$ is a DAG, its transitive reduction is unique and can be found in linear time [14]. By exploiting the properties of transitivity, this optimization minimizes the number of edges in $\mathcal{G}$ and therefore the number of waits the replay client has to enforce.

3.2 Implementation Details

Let us now provide some more implementation details, focusing mainly on the modifications we had to make in the second release of Oracle Database Replay to make it support a synchronization scheme based on collision dependency. Below, we study every step of execution separately.

Capture: During capture, Oracle Database Replay stores some metadata required for consistently replaying a workload, e.g., SQL text, bind values, timing information, and system change numbers[2] (SCNs). On top of that, in the new release, it also stores the identifiers of the objects a call accesses along with the corresponding type of access (read or write). The additional metadata makes our algorithm able to decide whether two calls collide or not, according to the consistency rule (Section 2.2). Note that we have implemented workload recording using callbacks in the DBMS kernel; hence, all the required metadata is easily available.

For performance reasons, Oracle Database Replay stores the additional metadata we described above only once per cursor, during the *open cursor* call. To avoid unnecessary redundancy, it does not explicitly store it again and again for every cursor execution. Instead, in this case, it only stores the cursor number, which is enough for retrieving the related information whenever necessary in subsequent steps.

Finally, note that the aforementioned object identifiers that the tool stores always refer to base tables, even when the SQL text associated with the corresponding cursor refers to complex views. This property guarantees that Oracle Database Replay does not lose any dependencies.

Preprocessing: During preprocessing, Oracle Database Replay spawns a number of threads, each one of which parses a number of capture files, gathers information stored during capture, and populates proper structures and internal tables with metadata required for the replay step. In the new release, it also populates an additional table: *DEPENDENCIES(file_id, call_ctr, object_id, RW, scn, pc_scn)*. This table stores a row for every object a call accesses. This information is necessary for identifying collisions during synchronization. Column *file_id* holds the identifier of the capture file where a given call is stored, *call_ctr* is a counter that identifies the sequence number of the given call in the capture file, *object_id* holds the identifier of an object that the given call accesses, *RW* stores the corresponding access type, *scn* holds the SCN of the database at the beginning of execution of the given call, and *pc_scn* holds the post-commit SCN, i.e., the new SCN produced after the given call committed its changes. If the given call is non-commit, then *pc_scn* is simply equal to *scn*.

Oracle Database Replay uses the raw data in *DEPENDENCIES* for constructing the dependency graph $\mathcal{G}$ we discussed about in the previous subsection. We have implemented algorithm *BuildDependencyGraph* (Section 3.1) in PL/SQL for this purpose. In the pseudocode that describes the general sketch of the algorithm we have used an abstract notion of time for ordering the calls and deciding precedence. Our purpose has been to keep the pseudocode as general as possible. In our particular implementation though, we have used *call_ctr* as a measure of time for ordering calls within a capture file. Furthermore, we have used *scn* and *pc_scn* for ordering calls across different capture files. Since SCN is a sequential counter that identifies a moment in an ORACLE database, it is ideal for deciding precedence.

In summary, our PL/SQL implementation performs the scanning of calls that are stored in the capture files (*BuildDependencyGraph*, lines 3-4) by selecting appropriate rows from *DEPENDENCIES*. It tests whether two calls collide (line 13), based on the corresponding values in columns *object_id* and *RW*. Whenever it discovers collision between two calls (lines 15-18), it inserts a new row in another table *DEP_GRAPH(file_id, call_ctr, dep_file_id, dep_call_ctr)*. *DEP_GRAPH* implements $\mathcal{G}$; a row in it represents a dependency between two calls. The pair (*file_id, call_ctr*) uniquely identifies the first call, which is stored in position *call_ctr* in the capture file with id *file_id*. Likewise, the pair (*dep_file_id, dep_call_ctr*) uniquely identifies the second one.

Replay: During replay, the database server has access to table *DEP_GRAPH*. Hence, it knows which dependencies it needs to enforce. Therefore, it blocks calls that need to wait on others, until the latter have been executed.

Our experience from implementing different synchronization schemes, one based on commit and one on collision dependency, verified our expectation that implementing the former requires less effort. For this scheme, maintaining in the server a central SCN-clock storing the greatest SCN among the calls that have been already executed during replay has been enough for efficiently keeping track of all dependencies, without having to materialize $\mathcal{G}$. The SCN-clock is similar to simulation clocks in that it is advanced by specific events. In the case of replay these events are commit actions. So, in the scheme that is based on commit-dependency, the server enforces consistency by simply blocking all new calls whose recorded SCN is greater than the current value of the clock, until some other call advances the clock to their SCN, or to a value greater than that. For more details, refer elsewhere [9]. The tradeoff is obvious: we chose to implement a more restrictive but simpler scheme for the first release of Oracle Database Replay, and implement the advanced one in the second release.

4. EXPERIMENTAL EVALUATION

In Section 2, we studied the spectrum of consistent synchronization schemes and argued that a scheme enforcing ordering based on collision dependency is the most fine-grained one; therefore, it is expected to maximize the level of concurrency and the associated replay performance. We have implemented this new scheme in Oracle 11g Release 2. To verify our argument, we have compared it experimentally with its only counterpart that has been proposed in the existing literature [9], i.e., the scheme that enforces ordering based on commit dependency. Recall that this scheme was introduced in Oracle 11g. In this section, we present the results of our experimental evaluation. Note that we have excluded from our evaluation schemes based on total ordering or object dependency. The reason is that these schemes exhibit less practical significance. As we already mentioned in Section 2.3.1, a scheme of the former type does not allow for realistic concurrency and has been presented for illustration purposes only. Likewise, as we explained in Section 2.3.3, a scheme of the latter type creates dependencies among read-only calls. This property is required to avoid deadlocks in DBMSs that acquire locks for reading. The explanation of the deadlocks is beyond the scope of this paper.

Hardware: We have run our experiments on two machines running Linux 2.6.9. Machine A has 4 GB of physical memory, and 2 Intel Xeon CPUs at 3 GHz with 6 MB of cache size each. Machine B is more powerful. It has 32 GB of physical memory, and 8 Intel Xeon CPUs at 2.53 GHz with 8 MB cache size each.

Workload Scenarios: We have experimented with different workloads and various capture/replay scenarios. All

[2]SCN is a sequential counter, identifying precisely a moment in an ORACLE database. SCN is advanced by commits.

results have been consistent. Below, we describe the most indicative ones. We omit the rest due to space limitation.

In more detail, we present results related to workloads generated by three different benchmarks: TPC-C [4], Swingbench [11], and a real application (RA), whose characteristics we explain below. TPC-C is a benchmark widely referenced in the database literature. It portrays a wholesale supplier with a number of geographically distributed sales districts and associated warehouses. It takes the number of warehouses as an input parameter. In our experiments, we have set it to 10, which is a commonly used value. Swingbench is a benchmark designed to stress test a database server. It simulates the workload coming from concurrent customers that place orders. We have configured Swingbench to generate 100 concurrent sessions. Finally, RA is a benchmark that uses workloads generated by an application of a real company we are working with. The agents of this company use this application in order to quote prices for different services, based on formulas that take into account the customer's history and profile. In our experiments, we have used 100 concurrent sessions coming from the aforementioned application. Note that our primary goal while choosing values for the input parameters of all the benchmarks we described above has been the creation of a considerable but still manageable load on machine A (CPU was busy on average by 97% during capture).

We can cluster the workloads generated by the benchmarks we have used into two different categories. RA's workload is *read-dominated*, since only 3.3% of its calls are commit calls. The workloads of TPC-C and Swingbench are *commit-dominated*, since the percentage of the commit calls they contain reaches 62% and 36%, respectively. In the case of TPC-C, a percentage of commit calls is so high because most update calls use automatic commit.

The main scenario we used in the experiments we are presenting here is the following: We first captured 32 min of different combinations of the three workloads we described above, while they were being executed on the database server that was running on machine A. Then, we replayed the captured workload on the database server running on machine B. Our scenario simulates a very common real-world situation. An application is running on a moderate-sized production system (like machine A) and the database administrator is planning to deploy it on a more powerful machine (like machine B). Before implementing the upgrade, the administrator uses database replay to estimate the expected performance gains related to the upgrade.

Additionally, we have used three different modes for replay: *same*, *fast*, and *fastest*. We explain their meaning with an example. Imagine two consecutive calls a and b stored in the same capture file, such that $a \rightarrow b$. Let t_a and t_b denote the time at which a and b were executed during capture, respectively. Moreover, let t'_a be the time at which a was executed during replay and d_a the duration of its execution. Then, for each mode, the time t'_b at which b becomes a candidate for execution during replay is given by the following formulas. *Same mode:* $t'_b = t'_a + \max(t_b - t_a, d_a)$, *fast mode:* $t'_b = t'_a + \max(\frac{t_b - t_a}{2}, d_a)$, and *fastest mode:* $t'_b = t'_a + d_a$.

The first formula implies that replaying on same mode tries to reproduce during replay the same rate of calls that was observed during capture. Likewise, replaying on fast mode tries to double that rate. Finally, replaying on fastest mode tries to replay all calls one after the other at the maximum rate possible, i.e., without leaving any gap between consecutive calls. Note that in the previous description we have stressed out that each different mode *tries* to achieve a given rate. Achieving it in practice depends on the duration of call execution and on any additional waits enforced by the synchronization scheme employed.

Analysis of Results: First, we present a set of experiments on a read-dominated workload, the one generated by RA. We replayed this captured workload on three different modes, using two different synchronization schemes. Note that both of these schemes are consistent; hence, by design, they reproduce the captured workload without any data or result divergence. Since both schemes produce the same results, we compare them by focusing on performance metrics: Figure 5 shows the elapsed time of each replay, Figure 6 the corresponding throughput (i.e., the average number of transactions per second), and Figure 7 the total time that commit calls had to wait until they could acquire a lock on the commit log file. All figures contain three groups of two columns. Each group corresponds to a replay mode: same mode on the left, fast mode in the middle, and fastest mode on the right. A pair of columns corresponds to each mode: the black column represents the synchronization scheme that is based on commit dependency, whereas the white one represents the scheme that is based on collision dependency.

The main conclusion that comes from these graphs is that both schemes produce good quality replays for the read-dominated workload. Recall that the duration of the original capture on machine A was 32 min. Figure 5 indicates that both schemes manage to synchronize replay ideally on same mode, since they achieve the same call rate as during capture and make the replay finish after 32 min, too. On the other two modes, they manage to drive the replay faster by reducing the time between calls, taking better advantage of the additional computational power of machine B. So, the elapsed time decreases (Figure 5) and the throughput increases (Figure 6). On same and fast mode, both schemes behave equally well, but on fastest mode, our new scheme, the one based on collision dependency, outperforms the existing one by nearly 23%. This result indicates that even though the number of commits is relatively small in the particular workload, collision-based synchronization can still gain some benefit by the fact that it enforces fewer waits and manages to increase the level of concurrency, driving the load on machine B to higher levels (average CPU usage reaches 47%, whereas the same number for the scheme based on commit dependency is 40%) and revealing to the administrator the real potential of the machine upgrade.

With respect to commit wait times (Figure 7), they are rather low, since the number of commits is small and there is not much contention on the commit log file. Wait time for the scheme based on commit dependency is smaller, since it issues all commit calls in a serialized fashion. Note that in this case, the issuing of the commits is serialized but the completion is not; so, there is still some contention, albeit smaller. Wait times decrease on fast and fastest modes, since both schemes enforce smaller wait times on these modes (recall the formulas that calculate t'_b for the different modes).

In order to better reveal the potential of our new scheme, we further experimented with a commit-dominated workload. To do so, we mixed the previous workload with the workloads generated by TPC-C and Swingbench. The resulting workload is heavier and commit-dominated. Fig-

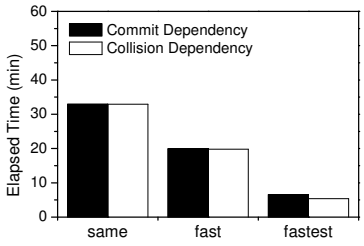


Figure 5: Elapsed time of replay (read-dominated workload).

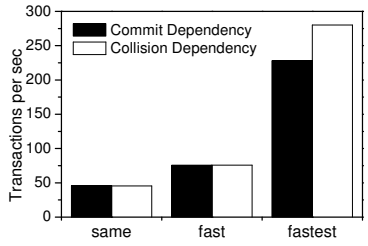


Figure 6: Transactions per second (read-dominated workload).

Figure 7: Commit wait time (read-dominated workload).

Figure 8: Elapsed time of replay (commit-dominated workload).

Figure 9: Transactions per second (commit-dominated workload).

Figure 10: Commit wait time (commit-dominated workload).

ure 8, Figure 9, and Figure 10 show the elapsed time, the throughput, and the commit wait time, respectively, for the different replays of the commit-dominated workload.

The first remark that we can make is that the scheme that is based on commit dependency fails to efficiently drive the replay of the commit-dominated workload. Note that all black columns have approximately the same height within each figure, showing that the corresponding scheme is already saturated, even on same mode. Although individual requests gain some benefit from the increased power of machine B, by holding on to resources for shorter time on average, the scheme is unable to increase the level of concurrency in order to decrease running time of the workload. Note that in Figure 8, the elapsed time of the scheme based on commit dependency is around 38 min on all modes. This time is even higher than capture time, which was 32 min.

Our new scheme does not suffer from the same problem. It manages to replay the workload in approximately 32 min on same mode and also succeeds in accelerating it further on fast and fastest modes, boosting the level of concurrency and increasing the throughput by 40% (Figure 9). By doing so, it generates a faster and better quality replay, since it drives machine B to higher levels of load (average CPU usage reaches 45%, whereas the same number for the scheme based on commit dependency is 35%), revealing the real potential of the upgrade to the administrator, not just the improvement of overall resource consumption.

With respect to commit wait time (Figure 10), we can observe that in this series of experiments it is increased compared to the corresponding time in Figure 7. This is attributed to that the commit-dominated workload includes more commit calls and therefore generates more contention on the commit log file. For the scheme that is based on collision dependency the wait time increases significantly, especially on fastest mode. This indicates that it issues many commit calls concurrently, the ones that are collision-independent. The scheme based on commit dependency cannot do that, by design. Note that commit wait time adds up rapidly as the number of commit calls that stay in the queue increases. This explains the significant height of the white column on fastest mode in Figure 10.

Overall, our results verify that the new scheme behaves equally well with or slightly better than the existing one, when replaying read-dominated workloads. Moreover, it outperforms the existing one by far, when replaying commit-dependent workloads. Being finer-grained, it produces better quality replays, reproducing better the workload on the test machine on same mode, and accelerating it further on fast and fastest modes. Therefore, it leads to more representative and more reliable tests, a property that makes it the scheme of choice within the spectrum of consistent schemes.

5. RELATED WORK

Concurrency control in DBMSs and the associated theory on conflict serializability are rather old and well-studied problems related to our work [6, 13, 17]. Algorithms in this area have used similar notions of dependency between transactions and analogous theoretical constructs, e.g., the conflict dependency graph. Despite the aforementioned similarities, the two problems differ fundamentally. The main difference is that concurrency control mechanisms ensure database integrity, whereas replay synchronization schemes guarantee replay consistency. In summary, the task of a concurrency control mechanism is to generate one legitimate schedule for executing a workload consisting of transactions that compete for resources. A schedule is legitimate if it guarantees the integrity of the database state (e.g., by enforcing serializability). On the other hand, replay synchronization mechanisms do not focus on integrity and therefore they adhere to different constraints. Instead of having to generate just any legitimate schedule, they must generate a schedule that is equivalent with the particular schedule that was actually executed in the production system dur-

ing capture. Another important difference is that concurrency control mechanisms work on the fly, since they do not know the entire workload in advance, whereas replay synchronization schemes can discover all dependencies in an off-line fashion, since they have access to the entire workload in advance. Hence, the former may have to backtrack, if a cycle is detected in the dependency graph they maintain; such a situation is never possible in the latter. The aforementioned differences justify the requirement for different synchronization schemes for workload replay. The study of such schemes and the adaptation of known theories into our new problem has been the main purpose of this paper.

Recording and replaying a real database workload has been studied in the past and some commercial tools already offer some of the features described here. Among them, the tool closer to our work is the first release of Oracle Database Replay [9]. Its implementation in Oracle 11g uses a consistent replay synchronization scheme that orders calls based on commit dependency. As shown in this paper, this scheme is more restrictive than necessary, leaving a lot of room for improvements. In this paper, our goal was to study for the first time the properties and tradeoffs of all consistent replay synchronization schemes, formalize their characteristics, discover new schemes, and nominate the best, which we have implemented in Oracle 11g Release 2.

Other tools of the same category, including Quest Benchmark Factory for Databases [1] and Microsoft SQL Server Profiler [3], rely on SQL trace for recording a database workload. The problem with these approaches is that SQL trace adds considerable overhead on the database server and does not contain transactional information necessary to implement a consistent synchronization scheme. Therefore, their techniques do not apply on our problem.

Furthermore, there is another family of tools, including iReplay [2], which implement capture and replay at the network level. During capture, they use sniffing to record the traffic that enters into the DBMS. During replay, they reproduce the same traffic. A significant disadvantage of such tools is that they are built outside the DBMS; hence, they can only rely on the timing information observed at its entry point. In other words, they only know when a request arrived at the DBMS, but they have no access to information about when this request was actually executed, what dependencies it exhibited, what SCN was assigned to it, or what objects it accessed. So, during replay, they cannot apply any synchronization scheme of the spectrum we discussed in this paper. The best they can do is issuing the same calls in the same order as during capture. Clearly, such a solution does not guarantee that the same order will be preserved during execution in the DBMS. Therefore, these tools do not guarantee replay consistency, which has been our main focus, and they belong outside the scope of this paper.

Finally, several other algorithms generate synthetic data or workloads for testing purposes [7, 8, 12, 15, 16]. None of them studies the problem of synchronizing the replay of the generated workloads though, which is our main focus. Moreover, our approach applies on workloads captured on real systems. Real workloads are expectedly more representative for database testing than synthetic ones.

6. CONCLUSIONS

In this paper, we performed for the first time an extensive study of consistent synchronization schemes for real workload replay. We defined their class, we identified the spectrum they cover, and investigated the properties and tradeoffs of various points within this spectrum. Moreover, we showed that, while adequate in many practical applications, the only consistent replay synchronization scheme proposed so far [9] is coarse-grained and more restrictive than necessary, enforcing dependencies between independent calls. To overcome this drawback, we identified the ideal scheme within the spectrum that uses finer-grained rules for replay synchronization. Furthermore, we described our efficient implementation of the new scheme and showed both theoretically and experimentally that it produces better quality replays than the existing one for major classes of workload.

7. REFERENCES

[1] Benchmark Factory for Databases. *http://www.quest.com*.

[2] iReplay: Database Workload Capture and Replay. *http://www.exact-solutions.com/products/ireplay*.

[3] SQL Server Profiler. *http://msdn.microsoft.com*.

[4] Transaction Processing Performance Council. TPC-C benchmark. *http://www.tpc.org/tpcc/*.

[5] A. V. Aho, M. R. Garey, and J. D. Ullman. The transitive reduction of a directed graph. *SIAM J. Comput.*, 1(2):131–137, 1972.

[6] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[7] C. Binnig, D. Kossmann, E. Lo, and M. T. Özsu. QAGen: generating query-aware test databases. In *SIGMOD Conference*, pages 341–352, 2007.

[8] M. Emmi, R. Majumdar, and K. Sen. Dynamic test input generation for database applications. In *ISSTA*, pages 151–162, 2007.

[9] L. Galanis, S. Buranawatanachoke, R. Colle, B. Dageville, K. Dias, J. Klein, S. Papadomanolakis, L. L. Tan, V. Venkataramani, Y. Wang, and G. Wood. Oracle database replay. In *SIGMOD Conference*, pages 1159–1170, 2008.

[10] G. R. Ganger. Generating representative synthetic workloads: An unsolved problem. In *Int. CMG Conference*, pages 1263–1269, 1995.

[11] D. Giles. Swingbench benchmark. *http://www.dominicgiles.com/swingbench.html*.

[12] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. Quickly generating billion-record synthetic databases. In *SIGMOD Conference*, pages 243–252, 1994.

[13] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, 1981.

[14] T.-H. Ma and J. Spinrad. Cycle-free partial orders and chordal comparability graphs. *Order*, 8(1):49–61, 1991.

[15] M. Poess and J. M. Stephens. Generating thousand benchmark queries in seconds. In *VLDB*, pages 1045–1053, 2004.

[16] D. R. Slutz. Massive stochastic testing of SQL. In *VLDB*, pages 618–622, 1998.

[17] R. E. Stearns, P. M. L. II, and D. J. Rosenkrantz. Concurrency control for database systems. In *FOCS*, pages 19–32, 1976.