

Oracle® Streams for Near Real Time Asynchronous Replication

Nimar S. Arora

Oracle USA
M/S 4op10, 500 Oracle Pkwy
Redwood Shores, CA
U.S.A.
nimar.arora@oracle.com

Abstract

Replication users typically have two very conflicting needs. They require the online transaction workload to proceed seamlessly, as if there was no replication. At the same time, they need the latency of the replica to be extremely small, or, in other words, near real time. Now, synchronous replication guarantees no latency, but it does so at tremendous cost to the transaction throughput and system availability. Thus, it is not surprising that asynchronous replication is the more popular choice. The challenge is to design an asynchronous replication system that can guarantee a small, fixed latency while at the same time keeping up with the full transaction throughput supported by the database. This paper discusses the design of Oracle® Streams (as configured for replication) to provide near real-time asynchronous replication at throughputs close to the maximum.

1. Introduction

Replication has traditionally been associated with high availability and disaster recovery applications. Recently, however, replication is being used for data warehouse loading and online upgrade or migration of applications. These new uses are placing stringent demands on replication performance.

For data warehouse loading, a customer might want to run a data mining application concurrently with online transaction processing, without affecting the transaction throughput. The solution typically is to perform the data mining on a replica. It is often acceptable in this case that the data mining application is running on a replica with slightly older data than the data in the source database, but strict limits are placed on the maximum latency allowed.

For online upgrade, an application often must be upgraded with essentially no downtime. In this case, the

upgrade is performed on a point-in-time replica, while the older version of the application is running. After the upgrade, the replica database is synchronized with the source database by replaying all the changes made since the point-in-time and mapping them to conform to the new version of the application. When the replica has almost caught up, the old version is taken offline briefly, while the new version completely catches up. Finally, the workload is redirected to the replica. Online migration is similar, whereby the customer is migrating a database to a different operating system or platform with essentially no downtime.

Because synchronous replication requires that the replica must be available while the changes are being made to the source, it does not work for online upgrade. Also, synchronous replication forces each transaction to wait for at least one network round trip before committing. This required round trip not only increases the delay seen by an individual transaction, but because locks are held longer, it affects overall transaction throughput as well. In other words, synchronous replication is unusable for these cases.

Asynchronous replication, on the other hand, has problems of its own. Although potential inconsistencies are a common concern, inconsistencies are less of an issue in practice. Experienced users can easily configure replication to virtually eliminate the possibility of divergence. The more critical problem is for the replica to keep up with the workload at the source.

The maximum transaction throughputs, as reported by popular benchmarks, have been skyrocketing over the years (Figure 1, [4]). Although much of the improvement has been fueled by faster hardware, database algorithms have also been getting better at extracting higher concurrency from multi-processor computer systems. Thus, the challenge for replication is not only to ride the hardware technology curve, but also for the replica to mirror the source's concurrency.

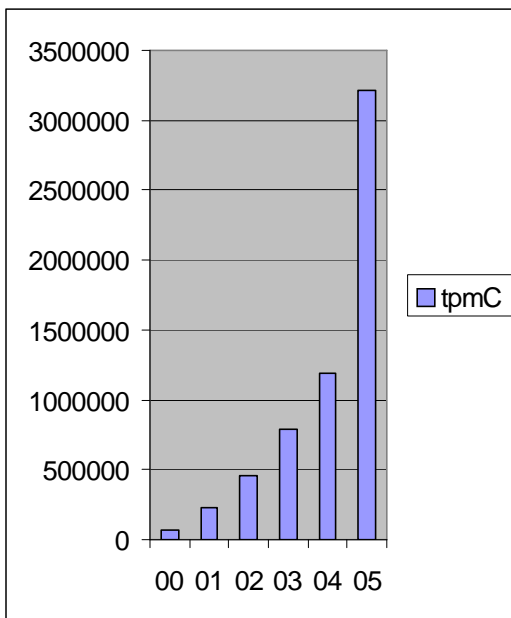


Figure 1. OLTP Performance Trends

This paper discusses how these problems are handled by Oracle® Streams, which can be used for asynchronous replication. Section 2 provides an overview of Streams and some terminology. Section 3 is a brief overview of some of the alternate replication designs provided by Oracle®. Section 4 discusses overall performance characteristics. Finally, section 5 deals with the concurrency issues at the replica.

2. Overview of Oracle® Streams

Oracle® Streams is a unified information sharing infrastructure, which provides generalized modules for the capture, propagation, and consumption of information. A full overview of the many features of Streams and its uses is beyond the scope of this paper; the interested reader is referred to [1]. This document limits this discussion to the replication of changes made to relational databases.

2.1 LCR and CR

In the context of replication, the information representing a change to a relational database is known as a logical change record (LCR). An LCR is a generalized representation of all possible changes to a database, and is independent of the database vendor. A change record (CR), on the other hand, is a term used to denote a database-specific representation of a change.

2.2 Rules and Transformations

The user can specify which LCRs to replicate by using a set of rules with rule conditions that are similar to SQL WHERE clauses. These rules are evaluated against all the changes made to the database to filter out the irrelevant ones. For example, the following rule specifies that only DML changes to table SCOTT.EMP are of interest.

```
:dml.get_object_owner()='SCOTT' and  
:dml.get_object_name()='EMP'
```

Similarly, rules can be specified for non-DML activity, such as ADD CONSTRAINT and DROP USER.

Moreover, rules can have transformations associated with them. A transformation uses a user-specified or built-in stored procedure, and it automatically changes any LCR that satisfies the rule condition of the associated rule. A transformation typically is used to delete sensitive attributes (for example, social security numbers), or to rename a table or a column before replication.

2.3 Queue

A queue is a temporary staging area that stores LCRs as they move between different Streams modules and across databases. The user configures Streams by first creating the queues and then attaching various Streams modules as producers or consumers for the queues. Along with each module, a set of rules or transformations can be specified to filter information flowing into or out of that module.

The queue supports three operations, enqueue, browse and remove. Here, the standard dequeue operation has been broken up into separate browse and remove operations.

2.4 Capture, Propagation, and Apply

The three modules of Streams are capture, propagation, and apply. The behavior of each module is controlled by rules.

Capture reads CRs contained in the redo generated by the database. It converts these CRs into LCRs and enqueues them.

Propagation consumes LCRs from one queue and enqueues them into another queue on the same, or on a different, database.

Apply consumes LCRs from a queue and changes the database as specified in the LCR. Because all database changes are recorded in the redo, apply can be thought of as writing CRs into the redo. In that sense, apply is the inverse of capture.

Figure 2 gives a high-level overview of Streams replication. It shows capture, propagation, and apply modules for replicating changes from one database to another. This configuration is an example of unidirectional replication, but it's not the only configuration that is possible. We could add another set of capture, propagation, and apply modules in the reverse direction to get bi-directional replication, for example. Similarly, by connecting appropriate modules, it's

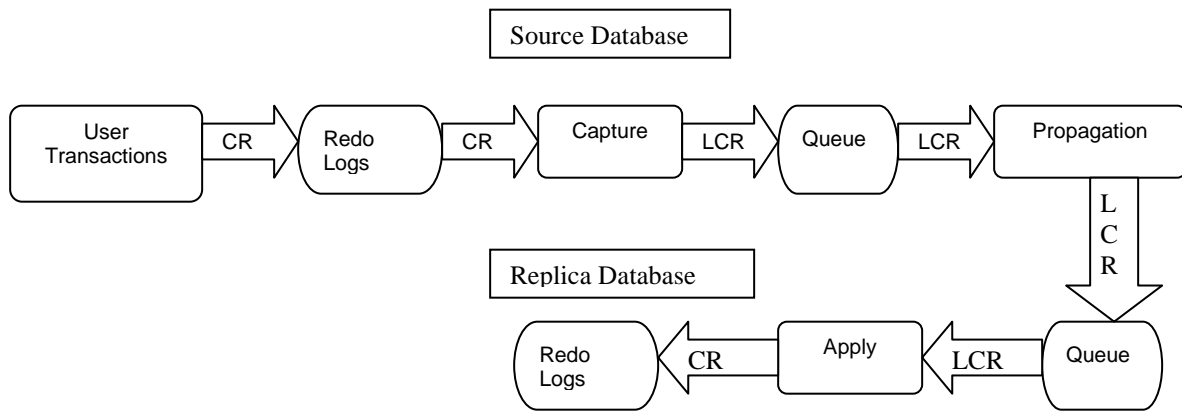


Figure 2. Overview of Streams Replication

possible to configure any conceivable replication topology.

2.5. Supplemental Logging and Replication Based on Primary Keys

The CR from which an LCR is constructed typically has minimal information in it. Usually, it is only possible to extract the modified attributes and the rowid. However, to apply a DML LCR, the primary key of the modified row must be available in the LCR. For parallel apply and inconsistency detection and resolution, other columns might also be needed. Thus, the CR must include extra columns. The addition of these extra columns in the redo log is known as supplemental logging.

2.6. Apply Handlers

In some replication scenarios, it is helpful to specify a user-created stored procedure, called an apply handler, to apply LCRs. For example, when schemas change during online application upgrades, the LCRs from the old version of the application schema cannot be applied as is. An apply handler is needed to convert a change (represented by an LCR) so that it is relevant to the new version of the schema. This conversion often requires querying the current application state. In this case, transformations are not suitable.

2.7. Inconsistencies

One of the drawbacks of asynchronous replication, versus synchronous, is the possibility of inconsistencies. In general, inconsistencies arise when users make conflicting changes on the source and the replica. Inconsistencies lead to two problems: detection and resolution.

For detection, all the columns in the LCR pertaining to “old” data on the source (that is, before the change corresponding to this LCR) are compared with the current values in the corresponding row in the replica during

apply. Also, users can supplementally log additional unmodified columns from the source to strengthen this check. In addition, the database identifies constraint violations, such as unique key, foreign key, and so on.

For resolution, Streams provides built-in conflict handlers for common strategies such as maximum value and overwrite. Alternatively, users can write their own stored procedures for resolving inconsistencies.

In practice, a combination of conflict resolution and conflict avoidance strategies works well for most user requirements.

2.8. Online Instantiation

Customers frequently add new sites to their replication configuration or add new tables to an existing configuration. It is imperative that such reconfigurations do not involve any downtime. The following steps show how Streams supports online instantiation:

- 1 Suspend apply activity, and start capturing changes to the new table and propagating these changes to the new site.
- 2 Briefly hold a shared lock on the source table to ensure that there are no long running transactions.
- 3 Make a point-in-time copy of the table.
- 4 Instantiate the table at the replica.
- 5 Resume apply activity, but ignore transactions which committed before the instantiation point-in-time (from step 3) for this table.

3. Alternate Repliation Configurations

3.1. Propagating CRs vs LCRs

In Figure 2, a modification makes it possible to propagate the CRs directly to the replica. This change has the advantage that the conversion of CRs to LCRs can be shifted to the replica and offloaded from the source.

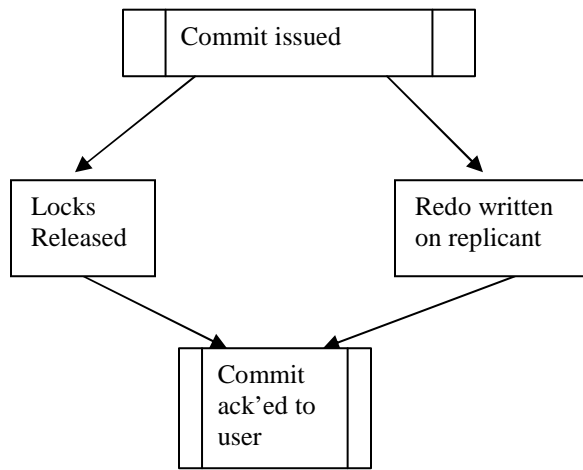


Figure 3. Synchronous Redo Replication

However, for the replica database to be able to decode the CRs, it must be running the same version of Oracle® as the source on the same platform as the source. This restriction disqualifies this approach for platform migration scenarios.

An interesting option that is available when propagating CRs is to obtain a weak form of synchronous replication by synchronously writing out the redo on the replica. This option is similar to synchronous replication in the sense that a user issuing a transaction commit must wait for at least one network round-trip. However, this option is different than synchronous replication in that locks on modified data items can be released before network acknowledgement. Some of the concurrent steps performed during transaction commit are shown in Figure 3.

This form of synchronous *redo* replication works well when the average transaction size is large enough that the network round-trip time is not significant. It also works well when the redo generation for concurrent transaction commits can be batched, so that the transaction commit rate is not limited by the network round-trip latency.

3.2. Applying CRs vs LCRs

If CRs are propagated to the replica, they can be applied directly, as if the replica is being “recovered” from the redo of the source. The obvious advantage is the reduction of the number of steps involved and avoidance of supplemental logging. The disadvantage, however, is that the replica must be an exact copy of the source. Thus, the replica cannot be open for updates.

See [2] for more on these alternate configurations.

4. Replication Performance

4.1. Latency and Throughput Definition

In standard I/O terminology, latency and throughput are defined with respect to bits or bytes, because those are the atomic units at which data is processed. In replication, however, the atomic unit is a change record, and so replication latency is measured in terms of CRs or LCRs. The immediate problem with such a definition is that CRs can vary widely in size, and consequently latency and throughput are not fixed measures. To address this issue, one standard CR is used, and latency and throughput are defined in terms of this fixed CR. Also, variation in latency can be measured with variation in CR size.

If the latency of the capture, propagation, and apply modules is, respectively, $L_{capture}$, $L_{propagation}$, and L_{apply} , then the overall latency for the simple replication configuration of Figure 2 is $L_{capture} + L_{propagation} + L_{apply}$. This method assumes that the throughput of each of the modules is at least as high as the throughput of redo generation. Otherwise, replication could steadily fall behind and the latency would grow unbounded.

4.2. Capture Performance

The capture module has to perform three distinct activities: read the CRs from the redo; convert the CRs into LCRs; and enqueue the LCRs. This document denotes the time taken by each of these activities as L_{read} , $L_{convert}$, and $L_{enqueue}$, respectively.

All the activities of the capture module cannot be performed in a single process (or thread of control). The problem is that the rate of reading redo is about the same as the rate of writing redo (disk I/O rates are typically symmetrical). If the capture process has to perform other activities in addition to reading redo, then it could not keep up with the maximum redo generation rate. Hence, a separate process is used for the reading activity. Although CRs can be read from the redo log buffer, which is much faster, the design of capture must handle situations where capture is so far behind the redo generation that it is forced to read the CRs from disk. For example, in the online upgrade scenario, as described in the introduction, replication might be started after some delay.

It follows that the maximum capture throughput is:

$$\min(1/L_{read}, 1/(L_{convert} + L_{enqueue}))$$

and latency is:

$$L_{read} + L_{convert} + L_{enqueue}$$

Here L_{read} is either $(LCR\ size)/(disk\ bandwidth)$ or $(LCR\ size)/(memory\ bandwidth)$ depending on whether capture is reading from disk or memory. The cost of the conversion and enqueue operations, however, is somewhere in between these two values. In general, the time taken to convert an LCR depends on the LCR size and the number of columns. LCR conversion is a more complex operation than simply copying memory. The

enqueue operation operates on a pointer to the LCR and is independent of the actual size. Therefore, LCR conversion cost primarily determines capture process latency.

It follows that when capture is reading redo from memory, its latency is primarily $L_{convert}$ and its maximum possible throughput is $1/L_{convert}$, which is more than the maximum CR generation rate.

4.3. Queue Performance

Although all queue operations are performed in memory and are independent of the LCR size, some queue performance issues warrant consideration. In some cases, concurrency issues with queue operations can be costly. For example, if a process goes to sleep waiting for a latch, it could be rescheduled as much as 10 milliseconds later (assuming that there are other processes available to be scheduled).

In general, when there are fewer processes concurrently accessing a data structure, there is a smaller likelihood of concurrency related delays. In the simple topology of Figure 2 and in many commonly used topologies, the maximum concurrency is just two (one producer and one consumer per queue). At this level, concurrency is not a significant issue.

4.4. Propagation Performance

Similar to capture, propagation performs three tasks: browse LCRs, transmit LCRs over the network, and remove LCRs. However, because the removal of an LCR is done after the LCR is sent, the cost of removing an LCR does not contribute to the latency. In fact, the removal of the LCR is done in a separate process to avoid any impact on the transmission throughput. The browse activity is extremely lightweight and is done in the same process as the transmission. LCRs are browsed and transmitted continuously to ensure that network latency cannot affect the throughput.

Therefore, the maximum throughput is:

$$\min(1/(L_{browse} + L_{transmit}), 1/L_{remove})$$

and the latency is:

$$L_{browse} + L_{transmit} + \text{network latency}$$

Where $L_{transmit}$ is $(LCR\ size)/(network\ bandwidth)$.

The propagation throughput is dominated by $1/L_{transmit}$ or, in other words, the *network bandwidth*. Also, because network latency is bound by speed of light considerations while network bandwidth is tripling every year [5], it is reasonable to assume that the propagation latency is bound by *network latency*.

The size of an LCR propagated can be comparable to the redo generated for the corresponding change. Thus, for propagation throughput to keep up, network bandwidth must be comparable to disk bandwidth.

4.5. Apply Performance

The apply module also has three broad activities: browse LCRs, execute LCRs (that is, manipulate the database

corresponding to the contents of the LCRs), and remove the LCRs from the queue. As in propagation, the removal activity does not contribute to the latency.

The main problem when applying an LCR is that the act of manipulating the database is slower than the generation of the redo that represents the change. So, if apply were to execute the LCRs serially, it could not keep up with the redo generation rate. For this reason, the apply module executes LCRs in parallel. If $L_{execute}$ is the execution time and L_{write} is the time taken to write the redo for the same LCR, the apply module requires effective apply parallelism of $L_{execute}/L_{write}$ to keep up.

In order to support parallel apply, the apply module must compute dependencies between the transactions so

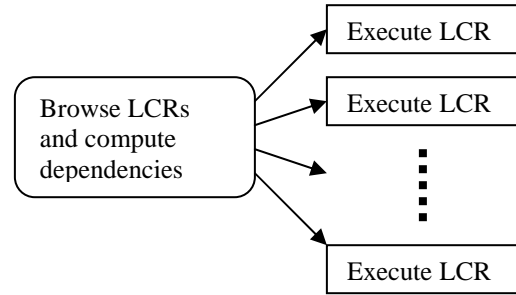


Figure 4. Apply Processes

that it can identify the LCRs that can be executed concurrently. As shown in Figure 4, there is one apply process for browsing and computing dependencies, and one for each active transaction. It is important that LCRs are executed as soon as they are received, even before the corresponding transaction on the source has committed. Otherwise, the latency could grow unbounded. Although the apply process executes an LCR immediately, it still replicates transactionally. For example, if the transaction on the source is rolled back, the apply module rolls back the corresponding replica transaction.

From this description, it follows that the maximum throughput of apply is:

$$\min(1/(L_{browse} + L_{dependency}), p/L_{execute})$$

where p is the effective parallelism. The latency is:

$$L_{browse} + L_{dependency} + L_{execute}$$

The dependency computation, which will be described in detail in the next section, is a relatively simple, in-memory operation, while writing redo is limited by the much slower disk bandwidth, that is:

$$L_{browse} + L_{dependency} \ll L_{write}$$

Combined with the observation that $L_{execute} > L_{write}$, it follows that apply throughput is $p/L_{execute}$, and the latency is dominated by $L_{execute}$.

4.6. Overall Performance

Remember that $L_{convert}$ is the most important latency factor for the capture module, but the conversion time is much smaller than the execution time because conversion is a simple, in-memory operation. Thus, if apply can extract sufficient parallelism to keep up with the redo generation, and if capture is reading redo from memory, the replication latency is essentially $(network\ latency) + L_{execute}$.

5. Parallel Apply

5.1. SCN Ordering

All changes in the Oracle® database are ordered by a number, called the system change number (SCN). Normally, this is a monotonically increasing number, but concurrent changes might be assigned the same value. It is a Lamport Clock as in [3] because a change x which *depends* on a change y has to have a higher SCN than y 's. Here the notion of *depends* says that change x acquires a lock which conflicts something locked by change y .

In addition to changes, the act of committing a transaction also has an SCN associated with it, known as the CSCN (or commit SCN). Because locks on modified data items are released only at the commit of a transaction, a stronger statement is needed for dependent changes: If change x depends on a change y , then x must have a higher SCN than y 's CSCN unless x and y are part of the same transaction.

5.2. Dependency Computation

For each LCR, apply must determine the set of locks that would be acquired during execution. If these locks could conflict with any lock needed by concurrently executing LCRs, then apply must suspend execution. Otherwise, apply might, as a result of a race condition, acquire the locks in the wrong order.

For DML LCRs only, locks taken on individual rows are of interest. In general, DDL LCRs are not executed in parallel. Because Oracle® only takes write locks, it becomes easy to compute dependencies for them. The LCRs are processed in SCN order, and, for each LCR, multiple entries are made into a fixed size dependency hash table, as follows. For each row of a table or index that an LCR execution would involve, apply hashes a unique identifier for that row to locate a bucket of the dependency hash table. If there is an entry in that bucket, then this LCR depends on it. Finally, the contents of the bucket are overwritten with the current LCR so that future LCRs that involve this row depend on the current LCR. Of course, false dependencies are possible due to hash collisions. However, with large enough hash tables, the probability of a false dependency between concurrently executing LCRs is extremely low.

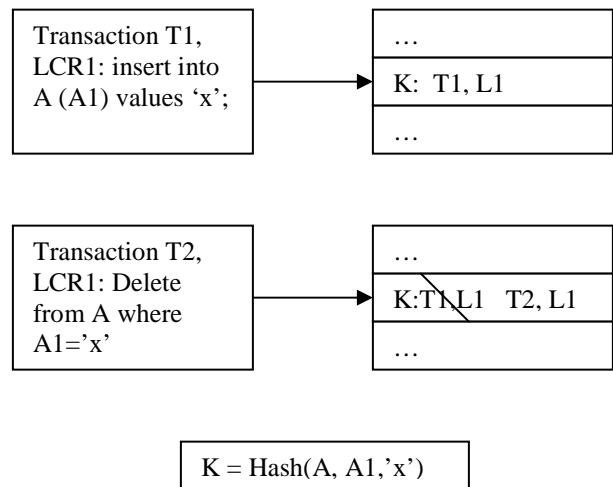


Figure 5. Transaction Dependency Computation

Figure 5 shows an example of dependency computation. The insert LCR of transaction T1 causes an entry to be written in the hash table at the location determined by $\text{hash}(A, A1, 'x')$. Later, when the delete of transaction T2 is processed, the same entry in the hash table is referenced. The delete LCR of T2 must wait on the insert LCR of T1, and the hash entry is updated with the information on the delete LCR of T2.

5.3. In-Order Apply

Another dependency between transactions is the commit SCN order. In general, it is safest to commit transactions in the same order as on the source. Although dependency computation ensures that dependent transactions are always correctly ordered, the CSCN order of transactions is also important. Reordering seemingly independent transactions might transiently violate application constraints that are not expressed as database constraints.

For example, an application might enforce a constraint that at most ten people are in the sales department. Consider a situation where the sales department has exactly ten employees. Now, assume that an employee is moved out of the sales department, and another employee moved into the department in two consecutive transactions. From the dependency computation point of view, these transactions are independent. But if they are not committed in CSCN order, it is possible to reach a state where the sales department has eleven employees.

5.4. Dependency DAG

A dependency directed acyclic graph (DAG) is created with individual LCRs as nodes and edges between any two LCRs that depend on each other. Figure 6 shows an

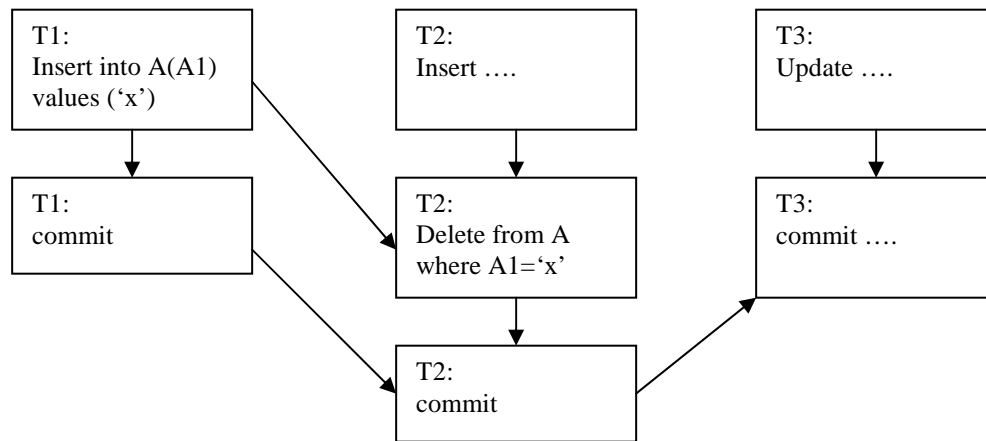


Figure 6. Example of a Dependency DAG

example of such a DAG. Note that the dependencies created by in-order apply requirements are expressed as edges between the commit LCRs of the transactions. Also, the requirement that the changes within a transaction be made in serial order is expressed as edges between consecutive changes of a transaction.

The apply module can therefore be thought of as an online DAG scheduling algorithm.

6. Conclusion

Oracle® Streams achieves near real-time replication by moving changes to the replica faster than the redo generation rate, and by applying them with a high degree of concurrency.

7. Acknowledgements

The author is indebted to Jim Stamos, Bipul Sinha and Mahesh Girkar for numerous discussions that helped shape the arguments presented here. Randy Urbano

provided valuable comments, which helped improve the layout of the presentation, and Marybeth Pierantoni helped with the performance data.

This work wouldn't have been possible without the support of my wife, Geeta, and the encouragement of my managers: Lik Wong and Alan Downing

8. References

1. Oracle® Streams Concepts and Administration 10g Release 2 (10.2). <http://otn.oracle.com>
2. Oracle® Data Guard Concepts and Administration 10g Release 2 (10.2). <http://otn.oracle.com>
3. Leslie Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.
4. TPC-C Benchmark Results. <http://www.tpc.org>
5. G. Gilder, "Fiber Keeps Its Promise: Get ready. Bandwidth will triple each year for the next 25." *Forbes*, 7 April 1997.