Inferring a Serialization Order for Distributed Transactions*

Khuzaima Daudjee and Kenneth Salem
School of Computer Science
University of Waterloo
Waterloo, Canada
{kdaudjee, kmsalem}@db.uwaterloo.ca

Abstract

Data partitioning is often used to scale-up a database system. In a centralized database system, the serialization order of commited update transactions can be inferred from the database log. To achieve this in a shared-nothing distributed database, the serialization order of update transactions must be inferred from multiple database logs. We describe a technique to generate a single stream of updates from logs of multiple database systems. This single stream represents a valid serialization order of update transactions at the sites over which the database is partitioned.

1. Introduction

In a centralized database system that guarantees commitment ordering [1], the serialization order of transactions that have committed at the single site can be inferred from the database log. This can be achieved by using the log sequence numbers of transactions' commit records. That is, $seq(T_1) < seq(T_2)$ if and only if T_1 precedes T_2 in the local serialization order. This information can then be extracted from the database log using a standard mechanism such as a log sniffer [5]. Since update transactions are processed at the single site, this makes it relatively easy to determine a serialization order for update transactions.

Data partitioning is often used to improve scalability by distributing a database over a number of sites in a shared-nothing architecture. Each site in this cluster consists of an autonomous database system with a local concurrency controller which is *rigorous* [2] and ensures commitment ordering. Rigorousness [2] is enforced by well-known concurrency control algorithms such as strict two phase locking in which no locks are released until commit.

Consider a distributed database system in which each site's local concurrency control is rigorous and transactions are synchronized using a two-phase (2PC) commit protocol. Our choice of 2PC stems from it being the most widely used protocol for coordinating transactions in distributed database systems [1]. Although serializability is guaranteed at the cluster by the sites' local concurrency controls and the 2PC protocol, the issue is how to determine the resulting serialization order. The contribution of this paper is a technique that determines a serialization order for distributed update transactions that have executed in a partitioned database over multiple sites. Our technique merges log entries of each site into a single stream that represents a valid serialization order for update transactions.

1.1. System Model

The database is partitioned over one or more sites. No restrictions are placed on how the database is partitioned. Transactions execute at the sites using a 2PC protocol [4, 7, 9]. Each transaction executes at one or more sites. One of these sites acts as the coordinator site for the transaction; other sites are participant sites. Different transactions may have different coordinating sites. Each site is autonomous and maintains its own database log and recovery information about its local transactions.

In the 2PC protocol, the coordinator generates a *prepare* message, which is sent to all participant sites involved in the distributed transaction. The participant sites generate and respond with an acknowledgement message, which we shall call *prepare-ack*. After acknowledgement messages are received from all participant sites, the coordinator commits and generates a *coord-commit* message that is sent to all participants. Each participant then either commits, generating a *commit* message, or aborts the transaction. Since the 2PC protocol does not commit a transaction until all read/write operations of the transaction have executed, and each local concurrency control is rigorous, the cluster of sites guarantees 1SR [2].

---

* University of Waterloo School of Computer Science Technical Report CS-2005-34

1 Commitment ordering [10] ensures that transactions can be serialized in the order in which they commit.

Each database log contains update, commit and abort records. Each transaction update log record describes an update executed by the corresponding update transaction. Both update and commit log records of an update transaction contain the transaction's global transaction id (tid). If a commit log record is that of a transaction's coordinator (a coord-commit record), it also contains the list of sites that are participating in the transaction.

Update information can be extracted from the database logs using a standard mechanism such as a log sniffer [5]. The updates from each log are merged into a single stream using the algorithm described in Section 1.3.

1.2. Timestamps

In this section, we describe the use of Lamport timestamps to capture causal relationships between updates at multiple sites. These timestamps, together with the updates committed by transactions, are entered into each site's database log. In the next section, we describe an algorithm that uses these timestamps to merge the updates from the database logs of the sites.

Each site maintains a Lamport clock [6], which is simply a monotonically increasing integer counter. A Lamport timestamp is the value of a site's Lamport clock at some particular time. When a 2PC participant generates a *prepare-ack* message, it increments its clock, and piggybacks the resulting timestamp onto the message. When the coordinator has received all of the *prepare-ack* messages, it sets its Lamport clock to the maximum of its current value and the values of the timestamps attached to the *prepare-ack* messages, plus one. Assuming that the transaction commits, this value becomes the transaction's *coord-commit* timestamp, and it is logged along with the transaction's commit record at the coordinator. The coordinator also piggybacks the *coord-commit* timestamp onto the commit messages it sends to the participants. When a participant receives a commit message, it sets its Lamport clock to the maximum of its current value and the *coord-commit* timestamp, plus one. The resulting timestamp is also logged with the commit record at the participant. A simple example of message exchange using 2PC and the logging of records is shown in Figure 1.

1.3. Algorithm

In this section, we describe our algorithm, which merges database log entries of sites into a single stream that represents a valid global serialization order for update transactions.

The log merging algorithm is executed by a log merger process, which runs at one of the sites. The algorithm reads update and commit records from sites' database logs in a round-robin fashion and writes this information to records in a list called the record list, or r-list[2], at the log merger site.

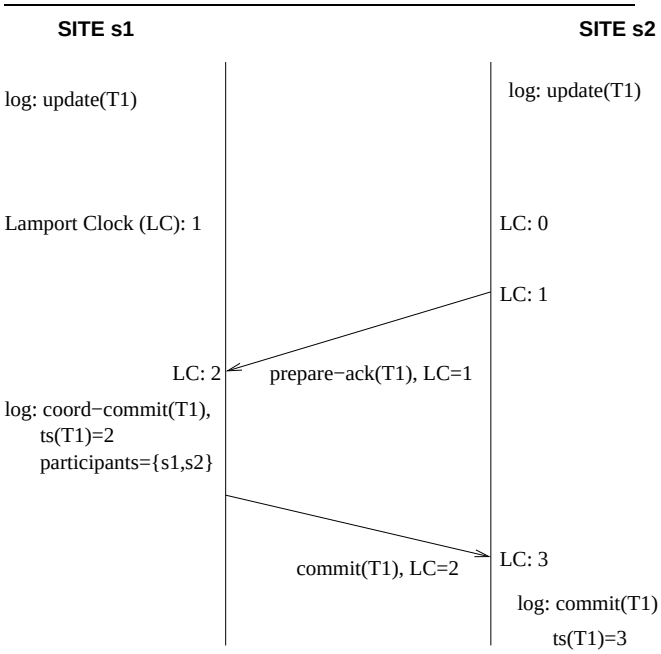

Figure 1. 2PC Message Exchange

There is one r-list record per transaction. Each record in the r-list is uniquely identified by a global transaction id (tid) and stores information from all commit and update records for that transaction from all sites' database logs. Each r-list record holds the minimum transaction timestamp, which is read from the transaction's coord-commit record. The r-list record also holds the site id of each log record read, and the participant list of the coord-commit record. Lastly, the updates of each update record with the same tid are kept in the r-list record.

The log merger visits the database sites in a round-robin fashion. Pseudocode for the actions of the log merger when it visits a site s is shown in Algorithm 1. The log merger can choose any size for the batch of records to read at site s since it does not matter for the correctness of the algorithm (though it may impact the rate at which records are output). A site can be skipped by the log merger if that site does not have any log records available to be read.

Step 14 ensures that a transaction's r-list record will ultimately contain that transaction's coord-commit timestamp, which is always the smallest timestamp from among all commit records of that transaction.

Let $ts(T_i^s)$ be the commit timestamp of the subtransaction of T_i running at site s. Let $ts'(T_i^s)$ be the prepare-ack

2 All fields of a newly created r-list record are initialized to null.

Algorithm 1 Merging of Update Streams

```
1  read batch of log records from head of log site s
2  for each record r in the batch
3       do let t be r-list entry for which t.tid = r.tid
4          if no such t
5             then /* create and initialize */
6                  create t in r-list
7                  t.tid = r.tid
8                  t.ts = ∞
9                  t.sites = ∅
10                 t.participants = ∅
11                 t.updates = ∅
12         if r is a coord-commit record
13            then t.participants = r.participants
14                 t.ts = r.ts
15         if r is a coord-commit record or commit record
16            then t.sites = t.sites ∪ s
17         if r is an update record
18            then t.updates = t.updates ∪ r.updates
19 for each t in r-list in ascending order of t.ts
20       do
21          if t.sites = t.participants /*t is complete */
22             then output t
23          else break /* exit loop */
```

timestamp of T_i's subtransaction at site s.[3] Let $ts(T_i)$ be the timestamp of transaction T_i, which is equal to the commit timestamp of T_i's coordinating subtransaction. From the operation of the two phase commit protocol, we can establish the following properties of timestamps. First, because of timestamps piggybacked on prepare-ack messages, we have:

Property 1 *For all sites s at which T_i runs, $ts'(T_i^s) \leq ts(T_i)$.*

Second, because of the timestamps that are piggybacked on commit messages, we have:

Property 2 *For all sites s at which T_i runs, $ts(T_i) \leq ts(T_i^s)$.*

The transaction serialization graph is consistent with the transaction timestamps. This is established by the following Lemma.

Lemma 1 *If T_1 conflicts with and precedes T_2, then $ts(T_1) < ts(T_2)$.*

Proof: T_1 and T_2 must have conflicting subtransactions at at least one site, since they conflict. Consider any such site, and call it s. There are four cases to consider:

3 For notational convenience, we define $ts'(T_i^s)$ to be equal to $ts(T_i^s)$ when s is the site of T_i's coordinating subtransaction, which does not have its own prepare-ack timestamp.

Case 1: The coordinators of T_1 and T_2 run at site s. Since T_1 conflicts with and precedes T_2, T_1^s must be serialized before T_2^s. Since the local concurrency control at site s ensures commitment ordering, T_1^s commits before T_2^s. Since the timestamping protocol increments timestamps on each commit operation, $ts(T_1^s) < ts(T_2^s)$ and thus $ts(T_1) < ts(T_2)$.

Case 2: T_1's coordinator runs at site s but T_2's does not. Since T_1 conflicts with and precedes T_2, and the local concurrency control at site s is rigorous, $ts(T_1^s) < ts'(T_2^s)$. From Property 1, $ts'(T_2^s) \leq ts(T_2)$. Because $ts(T_1) = ts(T_1^s)$, this gives $ts(T_1) < ts(T_2)$.

Case 3: T_2's coordinator runs at site s but T_1's does not. Since T_1 conflicts with and precedes T_2 and the local concurrency control at site s ensures commitment ordering, $ts(T_1^s) < ts(T_2^s)$. From Property 2, $ts(T_i) \leq ts(T_i^s)$. Since $ts(T_2) = ts(T_2^s)$, we have $ts(T_1) < ts(T_2)$.

Case 4: Neither transaction's coordinator runs at site s. Since T_1 conflicts with and precedes T_2, and the local concurrency control at site s is rigorous, $ts(T_1^s) < ts'(T_2^s)$. Properties 1 and 2 give $ts(T_1) \leq ts(T_1^s) < ts'(T_2^s) \leq ts(T_2)$. □

Theorem 1 *The log merging algorithm outputs transaction (r-list) records in a valid serialization order.*

Proof: It is sufficient to show that for every pair of transactions T_1 and T_2, if T_1 conflicts with and precedes T_2, then T_1 is propagated before T_2. Suppose that this is false, so that there exists at least one such transaction pair for which T_2 is propagated first. Let $\hat{ts}(T_i)$ represent the timestamp of transaction T_i as recorded in the log merger's r-list record for T_i. According to the log merging algorithm, T_2 is not propagated until it is complete. Since T_2 is complete, Property 2 and the fact that the log merger records the minimum commit timestamp for each transaction in its r-list imply that $\hat{ts}(T_2) = ts(T_2)$ when T_2 is propagated. Furthermore, if T_2 is propagated first, the log merging algorithm demands that either T_1 does not exist in the r-list when T_2 is propagated, or it exists in the r-list with $\hat{ts}(T_1) > ts(T_2)$. We will show that these conditions cannot occur.

Since T_1 conflicts with and precedes T_2, there exists at least one site s at which T_1^s conflicts with and precedes T_2^s. Since the local concurrency control at site s is rigorous and ensures commitment ordering, $ts(T_1^s) < ts'(T_2^s) < ts(T_2^s)$. Because of Property 1, $ts'(T_2^s) < ts(T_2)$, and since $ts(T_2) = \hat{ts}(T_2)$, we have $ts(T_1^s) < \hat{ts}(T_2)$. Since T_2 is complete, the log merger must have read T_2's commit record from site s. Since T_1's commit record precedes T_2's at site s, the log merger must have also read T_1's commit record at s, and thus T_1 must appear in the log merger's r-list. Furthermore, since the log merger tracks the minimum timestamp it has observed for each transaction, $\hat{ts}(T_1) \leq ts(T_1^s)$. Thus, we have $\hat{ts}(T_1) \leq ts(T_1^s) < \hat{ts}(T_2)$, which is

the desired contradiction. □

We would like to point out that Algorithm 1 might not output transactions in timestamp order though this order is also a valid serialization order. The output order and the timestamp order may differ when there are concurrent, non-conflicting transactions that can be serialized in either order. In this case, the algorithm outputs transactions as soon as their r-list entries are complete.

2. Related Work

Georgakopoulos and colleagues [3] proposed a technique for generating tickets that represent the serialization order over a cluster of sites by forcing transactions to conflict on a serialization sequence counter. They did not consider inferring this order through the merging of updates.

Liu and colleagues proposed an algorithm that uses Lamport timestamps to merge log entries into a single stream that represents a global order for update transactions [8]. To avoid processing log entries out of order, their approach uses a set of complex rules to determine the log entry with the minimum timestamp that should be processed next. When multiple entries in a log are present, their technique may read only one entry if the next entry with the minimum timestamp belongs to another site's log. As a result, their technique can cause frequent switching between sites. Unlike their approach, our log merging algorithm does not depend on reading entries from the database log in timestamped order and eliminates the complex rules needed for processing the log entries in this order. Since log records may not be read in timestamped order, space is required at the log merging site for storing the r-list. The simplicity of our approach makes it relatively easy to prove its correctness.

3. Conclusion

In this paper, we described a technique that determines the serialization order of update transactions. Using the log merging technique that we presented, a valid global serialization order can be determined for update transactions. The log merging technique works with the widely-used two-phase commit protocol to coordinate distributed transactions over partitioned databases.

References

[1] P. Bernstein and E. Newcomer. *Principles of transaction processing: for the systems professional*. Morgan Kaufmann Publishers Inc., 1997.

[2] Y. Breitbart, D. Georgakopoulos, M. Rusinkiewicz, and A. Silberschatz. On Rigorous Transaction Scheduling. *IEEE Trans. Software Eng.*, 17(9):954–960, 1991.

[3] D. Georgakopoulos, M. Rusinkiewicz, and A. P. Sheth. On serializability of multidatabase transactions through forced local conflicts. In *ICDE*, pages 314–323, 1991.

[4] J. Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, volume 60 of *Lecture Notes in Computer Science*, pages 393–481. Springer, 1978.

[5] IBM. *DB2 Universal Database Replication Guide and Reference*, 2000. version 7.

[6] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *CACM*, 21(7):558–565, 1978.

[7] B. W. Lampson. Distributed systems - architecture and implementation, an advanced course. In *Advanced Course: Distributed Systems*, volume 105 of *Lecture Notes in Computer Science*. Springer, 1981.

[8] C. Liu, B. G. Lindsay, S. Bourbonnais, E. Hamel, T. C. Truong, and J. Stankiewitz. Capturing global transactions from multiple recovery log files in a partitioned database system. In *VLDB*, pages 987–996, 2003.

[9] C. Mohan, B. G. Lindsay, and R. Obermarck. Transaction Management in the R* Distributed Database Management System. *ACM Trans. Database Syst.*, 11(4):378–396, 1986.

[10] Y. Raz. The principle of commitment ordering, or guaranteeing serializability in a heterogeneous environment of multiple autonomous resource managers using atomic commitment. In *VLDB*, pages 292–312, Aug. 1992.