Concurrent Union
A System for SQL-level Heterogenous Database Replication

Ben Vandiver

May 27, 2005

**System Interface**

The replication system, called ConcurrentUnion (CU), acts as an intermediary between a database application and the database replicas. Sequences of SQL statements, grouped into transactions, are submitted to the CU, which then forwards them on to the database replicas. Database applications interface with the CU by opening a network connection for each transaction. SQL statements are issued and the error-checked responses returned. When the transaction is complete, the application sends either a rollback or a commit signal. Faulty replicas will either be silently fixed in the case of minor errors, or a database error signaled and the transaction aborted in the case of major faults. Any number of transactions may be in progress with the CU at a time.

**System Implementation**

ConcurrentUnion uses a primary-secondary system for transaction ordering and a consensus system for correctness. The primary exists to propose a transaction ordering and the lessen the chance of inter-database deadlock. The result of each SQL statement from each replica is compared with the results of the other replicas in order to decide the correct answer, and to detect faulty replicas. Once the result comparison is performed and the answer deemed correct, then it is returned to the client. When all statements have finished executing on all replicas and the client issues a commit, the transactions are committed.

**Transaction Ordering and Serializability**

In order to ensure that the replicas stay synchronized, each replica must execute the transactions in a serializable manner and each replica must commit the transactions in an equivalent serial order. An offshoot of this issue is that two or more databases picking a different order for transactions can result in an inter-database deadlock. Imagine a situation in which two transactions both attempt to modify the same row in a table. Neither transaction is allowed to proceed until all replicas have executed the update, but if half of the replicas execute one transaction and the other half execute the other transaction, then neither transaction will execute to completion on all replicas. This type of deadlock exists in the CU, thus the database-level deadlock detectors do not detect and recover from it.

The primary imposes an ordering on transactions because an SQL statement must be executed to completion on the primary before it is issued to the secondaries. Any two statements that conflict will result in one of them acquiring the relevant locks on the primary while the other blocks waiting for the other statement to commit. The primary only commits a transaction and releases the associated locks when the transaction has successfully committed on all of the secondaries. Thus, transactions which contain conflicting statements will be ordered by the primary. Transactions that don’t contain any conflicting statements will be capable of
executing simultaneously on the primary and the secondaries. If transactions do not conflict, any ordering is an equivalent serial ordering.

This primary ordering scheme also mostly eliminates the deadlock problem mentioned earlier. If the primary uses a locking scheme at least as coarse as the most coarse locking scheme used by a secondary, then no deadlocks will occur. The proof is by contradiction: if two secondaries entered a deadlock, then they must have executed conflicting statements in different orders. To execute on the secondaries, both statements must have successfully executed to completion on the primary, thus they would have acquired all relevant locks on the primary. Since the statements conflict, they share at least one lock in common, so two transactions must both have acquired the same lock: contradiction. If the the primary uses a finer-grained locking mechanism than the secondaries, deadlocks can result where two distinct locks on the primary map to a single one on the secondaries.

The primary can not necessarily be trusted to correctly serialize transactions. A bug in the primary’s locking mechanism can result in conflicting transactions issuing concurrently on the secondaries. The secondaries could select differing execution orders for the transaction, which would result in a CU level deadlock. However, a transaction requires all replicas to have completed all statements before it can be committed. Thus, secondaries with correctly functioning locking mechanisms will all choose the same order for transactions that conflict. Faulty locking mechanisms can cause an individual replica to commit transactions out of order, but the replica set as a whole cannot be corrupted. Unfortunately, the CU cannot tell the difference between accidental deadlock and failures of the locking mechanism on the primary, thus this type of error will be hard to catch.

Fault Tolerance

Data corruption and hardware, network, and database failures can cause a replica to become faulty. In the case of a secondary replica failure, the replica can be removed from the active set, recovered, and then returned to the active set. A replica that crashes will hopefully eventually come back up, with its database having run its database recovery procedure. This procedure should result in all committed transactions being committed and all uncommitted transactions being rolled back. Since the CU records which transactions have successfully committed on each replica, the CU can determine which transactions must be replayed to a crashed replica to bring it up to date. Should these operations return incorrect answers, the replica can have its data repaired instead.

Failures of the primary require that a new primary be selected from amongst the secondaries. Since the primary does not store any specific state, the only operation necessary is to restart all the transactions on the new primary which had not yet been issued to the secondaries. Since no result can be returned to the client without all the secondaries executing a statement, the new primary is aware of all results that the client has seen. The ordering decided by the old primary has been passed on to the new primary due to normal operation, and the new primary picks up where the old primary left off: issuing statements to the secondaries as it finishes executing them.

Failure of the CU itself results in the rollback of all transactions in progress, as all database connections are lost. All clients will receive errors to this effect. When the CU restarts, the clients can reissue the failed transactions and resume operation.

Replica Repair

When a replica consistently returns incorrect results for a transaction statement, either the RDBMS software has a persistant bug for this statment, or the data in the database has become corrupted. In the former case, transaction should be executed on a different RDBMS and the results copied back. In the latter case, the transaction should be rolled back, the data repaired, and the transaction re-executed and checked for
correctness. Should the wrong result be returned after the attempted repair, the situation is most likely the former case. All the cases below assume that the faulty replica is not the primary. Should the primary require repair, the primary will be retired to secondary status and new primary promoted.

**Immediate Online Repair**

The table repair operation can be performed immediately, blocking the completion of the transaction in progress until it completes. There is an efficiency problem and a correctness problem. The efficiency problem is that the rows whose existence/nonexistence/differing state are the source of the incorrect answer are very hard to isolate in the general case. For simple queries, a more accurate transfer can be constructed, but for more complicated ones, all the tables involved must be transferred.

In order to produce a correct table transfer, equivalent serial transaction order is not good enough, because multiple transactions can update the same table without conflicting. Thus, the requirement is that the table image must occur in exactly the same place as the table delivery in the overall transaction ordering. Depending on the database locks to accomplish this will result in deadlocks at best. Therefore, all replicas, including the primary, are quiesed before the image is taken. Transactions are prevented from issuing on the primary and the transactions in progress are given some amount of time to complete. Transactions that don’t complete in the allotted time are rolled back. The primary must be included because if the primary is malfunctioning in ordering the transactions, the re-ordering of the erroneous transaction could be an issue.

1. Data corruption discovered in replica A, which is not the primary.
2. Prevent transaction issue to the primary.
3. Wait some amount of time, then rollback any transactions which haven’t entered the commit phase.
4. In a new transaction on A, delete contents of tables involved in the faulty transaction on replica A.
5. In a new transaction on some other replica B, issue selects on erroneous tables on replica B, then commit the transaction on B.
6. Re-enable execution of transactions and transaction issue to the primary.
7. Issue inserts to insert contents of select from B into A, then commit the transaction on A.

An alternative to draining the whole system is to drain one replica. Execute on the faulty replica and the repair replica until they have committed exactly the same transactions. Then rollback all other transactions in progress on both replicas, and snap the image.

**Offline Repair**

As above, acquire a consistent image for the faulty replica. Then remove the replica from the list of replicas to verify answers against. Log all transactions and their results executed by the replica set in the meantime. Once the table repair is complete, execute the logged transactions in serial order, verifying the result obtained against the logged result. The replica is likely to catch up to the replica set because the replica set must first execute on the primary and then the secondaries. Once the list is emptied, prevent new transactions from issuing on the primary and wait for the secondaries to drain. Execute these transactions on the replica. Finally, add the replica to the replica set and restart transaction issue on the primary.

An alternative method is to log the exact order that statements are executed on the primary and replay them on the recovering replica. When this statement list is exhausted, lock the primary from executing any more statements, finish off any remaining statements, and then have the recovering replica rejoin the replica set. However, if the primary ever rolls back, the statements from that transaction must be removed from
the list. If the recovering replica has already executed them, the recovering replica must be rolled back to its last checkpoint and the modified list reexecuted.

One final offline repair possibility is to allow transactions to continue to execute on the recovering replica, subject to a block list. This list contains the tables which cannot be interacted with. The list starts with only the tables being repaired, but as other transactions pile up which are waiting on the block list, whose subsequent statements update other tables, these other tables are added to the block list. Using this method, selects on unrelated tables could continue to run on the replica while it was recovering. However, this scheme introduces a lot of extra complexity, where the amount of gain is not clearly worth it.

**Some questions:**

1. What is the minimal disruption possible to achieve a consistent table snapshot to copy during a table repair operation?

2. How is the best way to deal with the “last moment” between when a recovering replica has finished its backlog and when it rejoins the active set?

3. Is the primary switch over really as simple as pick a secondary and restart transactions that haven’t made it to the secondaries yet?

4. The commit-ordering system present in the previous incarnation has been written out in this revision. Is it true that any differences in ordering that matter will result in a deadlock?