Things every update replication customer should know

Rob Goldring
IBM Santa Teresa Laboratory
San Jose, CA

Note: this paper is an abstract of an article which appeared recently in *InfoDB*.

In the mid-1980s, Chris Date's "12 rules" for distributed database systems included *replication*. Replication makes transparent the problems of remote access delays and the management of data redundancy. The commercial market for distributed database features has been slowly building over the years, beginning with simple remote access gateways. Today, replication appears to deliver on the 1980s ideal, with a robust *asynchronous* infrastructure. Current commercial technology though, continues to fall short of that ideal.

"Asynchronous replication" is a pleasant term to describe the operation of a distributed database running without concurrency control. In practice, DBMSs which use locking mechanisms in local operation are connected into replication networks without benefit of a global serialization mechanism, such as a synchronous 2-phase commit protocol. The notion of a *transaction* is thus compromised.

Four properties, *atomicity*, *consistency*, *isolation* and *durability* -- "ACID" for short -- have come to define a transaction system. With asynchronous replication, there is no *isolation* of transactions. Transactions run in parallel without any guarantee that a transaction sees the most current state of the database before making an update. Updates then, are not serialized.

One of the many benefits derived from the ACID properties is a *serial history* of transaction execution, an absolute necessity to satisfy audit requirements in regulated industries. Without a serial history, it is impossible to reliably state who updated a database from state N to state $N+1$. Not all replication systems guarantee a serial history.

Update Conflicts

Update conflicts occur when applications commit competing, potentially incompatible updates to two or more replicas and the existence of these competing updates cannot be detected until propagation occurs. There are, in fact, two general classes of update conflicts:

- **Intra-table update conflicts** are those which are detectable within the scope of a single table.
- **Inter-table update conflicts** are those which are *not* detectable within the scope of a single table, but which are nonetheless conflicting.

Three common, and false, assumptions are routinely made by designers of asynchronous replication systems:

1. All conflicting executions can be detected within the scope of a single global table -- all physical replicas of a single, logical table.
2. Timestamps from uncoordinated local clocks can be relied upon to order events in a distributed system.
3. Constraints can be defined in each local database to protect the global database from permanent inconsistencies resulting from uncoordinated updates.

Peer-to-peer example

For this example, assume that in case of an update conflict, the "most recent update" should persist, as determined by a timestamp accompanying each propagated update. Additionally, there are three sites, *site1*, *site2* and *site3*, each with two replicas:

```
SUPPLIER (SUPPLIER_ID, NAME, ADDRESS,
TOT_RECEIVABLE) PRIMARY KEY (SUPPLIER_ID)

PARTS (PART_ID, SUPPLIER_ID, DESCRIPTION)
PRIMARY KEY (PART_ID)
FOREIGN KEY (SUPPLIER_ID) REFERENCES SUPPLIER
ON DELETE RESTRICT
```

Note that the referential constraints say that a dependent row in PARTS cannot exist if there is no corresponding parent row in SUPPLIER and that a parent row in SUPPLIER cannot be deleted if there exist any dependent rows in PARTS. Figure 1 shows the timetable of events leading to the eventual inconsistency.

These are the events as shown in Figure 1: Beginning at 9:50, all sites are consistent. At 10:00, someone at *site2* inserts a row into the PARTS table. Shortly after, a delete arrives at *site2* from *site1* -- a delete of the parent row in SUPPLIER for the PARTS row just inserted. The delete carries with it a timestamp of 9:58.

By 10:02 the state of the database at *site3* is indeterminate; matching either the state of *site1* or *site2*. The state at any additional sites 4,5,6, etc. is similarly indeterminate.

SIGMOD '95, San Jose, CA USA
© 1995 ACM 0-89791-731-6/95/0005..$3.50

Figure 1 Peer-to-peer example.

time	S1	S2	S3
9:50 (all sites consistent)	SUPPLIER has a row with SUPPLIER_ID=12, PARTS contains no rows with SUPPLIER_ID=12	SUPPLIER has a row with SUPPLIER_ID=12, PARTS contains no rows with SUPPLIER_ID=12	SUPPLIER has a row with SUPPLIER_ID=12, PARTS contains no rows with SUPPLIER_ID=12
9:58	Delete from SUPPLIER where SUPPLIER_ID=12 Ok. (propagate)		
10:00		Insert a row into PARTS with SUPPLIER_ID=12 Ok.	
10:01	Referential constraint violation, as dependent row cannot be inserted when no parent row exists	Referential constraint violation, as parent row cannot be deleted while a dependent row exists.	If the delete is received and processed first, then the same RI violation as at site S1, else the same RI violation as at site S2
10:02	No row in SUPPLIER having SUPPLIER_ID=12, no row in PARTS with SUPPLIER_ID=12.	Not only a row in SUPPLIER with SUPPLIER_ID=12 but also a row in PARTS with SUPPLIER_ID=12.	By chance, either the same situation as at site S1 or the same situation as at site S2

This example illustrates a number of points:

1. The conflict in this example occurs between updates to two related tables but not between two updates to the same global table.

2. Local constraints, or *conflict resolution routines*, could not return the global database to a consistent state. Any attempt to combine the deletion of a parent row and the insertion of a dependent row is *wrong*. A more rigorous multi-site repair strategy is necessary to restore consistency.

3. Timestamps are not useful in resolving this situation. Interestingly, If the delete rule for the SUPPLIER table had instead specified ON DELETE CASCADE and the delete was processed at *site2*, the 10:00 insert at *site2* will be removed from the database as a cascade delete resulting from the 9:58 delete at *site1*, even though the 10:00 insert is surely "more recent" than the 9:58 delete. This is an important point -- the time order of the cascading operation may override the time order you specify in your local conflict constraint.

4. Referential constraints must be defined in the database and not enforced by application programs alone. Application logic cannot detect all potential referential integrity violations when complete detection can occur only *after* propagation occurs. If the database in this example were not enforcing the DELETE RESTRICT constraint, the parent row would have been deleted at *site2*, leaving the row inserted into the PARTS table at *site2* as an orphaned dependent.

The example above illustrates that problems can occur when global serializability is not enforced. In this example, two globally conflicting operations are allowed to be committed locally, with the resulting global inconsistency.

In order to repair the inconsistency, one of the local transactions must be backed out, or *compensated*. Either the delete of the parent row may be durable or the insert of the dependent row may be durable, but not both. Automatic transaction compensation then, is a desirable feature for an asynchronous replication system.

Conclusions

Asynchronous update replication should only be used after carefully assessing the risks. Replication products which do not enforce serializability may not be appropriate for applications requiring transaction integrity.

Bibliography

C.J. Date. What is a Distributed Database System? InfoDB2 Vol. 2, No. 2, Summer 1987 and Vol. 2, No. 3, Fall 1987.

Rob Goldring. Things every update replication customer should know (working title), to appear: InfoDB, April 1995.

Jim Gray and Andreas Reuter. Transaction Processing: concepts and techniques. Morgan Kaufmann, 1993.

Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. Communications of the ACM Vol. 21, No. 7, July 1978.