Scaling Out without Partitioning

Philip A. Bernstein, Microsoft Corporation[1]
Submission to HPTS 2009

We propose a database system architecture that can scale out to a large number of server machines without partitioning the database or transaction load. It is a data sharing architecture. That is, each machine (i.e., server machine) can access the entire database. Thus, each transaction can execute entirely on one machine.

Data sharing architectures have been around for decades. A common approach goes something like this: A global lock manager is used to control which machine can read and/or write which database pages. If a transaction needs to update page P, its machine M sends a message to a global lock manager L to lock P. If P is unlocked, then M gets the lock and reads P from shared storage S. If P is locked, then the lock owner M′ is asked to release the lock. When it does, M obtains the lock and reads P either from storage (if the protocol requires M′ to write it to storage before releasing the lock) or directly from M′ or some other fast memory that has the copy of P that M′ was using.

There are several technical challenges in building a data sharing architecture like this. One is logging. If each machine manages its own log, then after a machine fails, another machine has to run recovery for the failed machine's log, so that any data that was locked by the failed machine can be made available again. Another challenge is making the lock manager very highly available, since the failure of the lock manager prevents all machines from running transactions. Protocols to attain very high availability usually affect performance. A third problem is communication overhead. This design works best when it's relatively rare for pages to move between machines, as described in the previous paragraph. If page P is popular and all machines want to write P frequently, then getting the lock on P and passing P around can become a bottleneck. For this reason, applications are usually designed so that each database partition has affinity to a particular machine, so that most accesses to the partition execute there.

There are solutions to all of these challenges, as evidenced by existence of data sharing products, such as IBM Data Sharing, Oracle RAC, and Oracle Rdb/VMS.

We propose a rather different architecture that may offer better performance and scaleout without partitioning or even soft affinity between database partitions and machines. It does not use a lock manager. Instead, it uses an optimistic concurrency control algorithm, which will be described shortly. The main synchronization point is provided by the shared storage system, which offers an atomic operation to append a log record to a shared log.

[1] This work is joint with Colin Reid, Microsoft Corporation

Our design assumes the system consists of servers that execute transactions and storage units that execute operations on persistent storage. In particular, storage units support an atomic operation to append data to storage and return the location where the data was stored [1]. All servers can access all storage units, which is the essence of a shared storage system, over a high-speed network.

Each transaction T executes in the normal way on a single compute server, reading and writing shared data items. We use a multiversion storage structure, so that T only reads data that was written by committed transactions. When T writes a data item, its update is stored in a cache that is private to T. When it has completed, the after-images of its updates are gathered together into a log record that is appended to the log. The log record also contains a pointer P to an earlier log record that identifies the state of the database that T read. After a server appends a log record L, L is broadcast to all other servers. The shared logging mechanism results in a well-defined total order of all log records. Thus, all servers have a copy of the log consisting of the same sequence of log records.

Ordinarily, the storage unit is managing other storage besides the log, and updates to much of this storage does not need to be broadcast to servers. The storage unit that processes the operation to append L to the log could be a software process or a physical storage controller that is customized to support this operation to append and broadcast.

Unlike today's standard database system architecture, the operation that appends T's log record does not cause T to commit. Instead, there is an optimistic test that is run after the append operation to the log that determines whether T committed.

When a server receives a log record L for a transaction T (after L was appended to the log), it needs to determine whether T committed. To do this, it checks whether there is a log record in between P (the pointer to the log record defining T's readset) and L that contains a transaction's update that "conflicts with" T. If not, then T has committed and T's updates should be incorporated in the database state. Otherwise, T has aborted and its updates should be ignored. Note that the decision to commit or abort T is based only on log records that precede L. Thus, the decision to commit or abort each transaction can be made as soon as the server has received all log records that precede L. Also, since all servers see the same log, for every transaction T they all make the same decision whether to commit or abort T, without any server-to-server communication.

The definition of "conflicts with" varies based on the isolation level being used. For example, if T executed using read-committed isolation and T only read data that was last written by committed transactions as assumed above, then no conflict check is needed. If it executed using snapshot isolation, then there must be no log record in between P and L for a committed transaction that wrote into data items that T wrote (i.e., its writeset). If it executed with serializable isolation, then there must be no committed transaction between P and L that wrote into T's readset or writeset. The latter check requires that L includes the identity of data items that T read (i.e., its readset). If a conflict was detected, then T has aborted (even though it appended L to the log) and (presumably) should be re-executed. It's

just like what happens in a system that uses two-phase locking when a transaction aborts because it was a deadlock victim.

This optimistic mechanism for committing transactions will experience many aborts of transactions that access write-hot data. To predict the transaction throughput can we expect, consider the following facts:

1. When two or more transactions conflict, one of the transactions will surely commit, namely the first one to have written the data on which the transactions conflict. So there is always forward progress.
2. If a transaction T aborts, then almost surely all of the data that T needs is in its server's cache. This includes the data that was written by the other transaction whose update(s) caused T to abort, because such updates are broadcast in the log. So the re-execution of T should not require I/O and hence be very fast. This significantly reduces the chances that another conflicting transaction will have enough time to execute and to cause T to abort again.
3. We believe the system design will enable a log record to be appended in about 100 μs using today's state-of-the-art network technology and flash memory storage units , a long story that is partly discussed in [1]. So in principle, the system can append 10,000 commit records per second (or more, using group commit and other optimizations in the storage units). Even considering that many of these commit records are for transactions that abort, it seems fairly safe to assume that the system will be able to commit thousands of transactions per second that update the same write-hot data. Of course, this will need to be validated by measuring the behavior of our prototype.

If the required transaction rate on write-hot data exceeds the system's ability to process it, then conventional partitioning techniques for transactions and databases can be used on top. For example, one can use a scheduler that assigns all transactions that update a write-hot data item *x* to the same server, and have that server use a lock to ensure it single-threads such transactions to avoid aborts.

References

1. Colin Reid, "Synchronizing Distributed Systems with Shared Appendable Storage," submitted to HTPS 2009.