Lock-free Transactional Support for Distributed Data Stores

Daniel Gómez Ferro Flavio Junqueira Benjamin Reed Maysam Yabandeh [*]
Yahoo! Research
{danielgf,fpj,breed,maysam}@yahoo-inc.com

The data in large-scale data stores is distributed over hundreds or thousands of servers and is updated by hundreds of clients, where node crashes are often frequent. In such environments, supporting transactions is critical to enable the system to cope with partial changes of faulty clients. The behavior of the system when accessed by concurrent transactions is specified by an *isolation level*. Ideally, the data store provides *serializability*, in which the concurrent execution of transactions is equivalent to a serial execution of them. Serializability could be ensured by detecting non-serializable executions at run-time and aborting the corresponding transactions. This, however, leads to a high overhead on the data store and lowers the level of concurrency that the system can offer. Alternatively, a recent proposal [4] uses a centralized server to assign a serial order to transactions prior to their execution. The efficient implementation of this approach based on low-granularity locks, requires a prior knowledge of the rows that will be touched by a transaction, obtaining which becomes complex as well as inefficient in the case of non-trivial transactions [4].

To allow for high concurrency between transactions, commercial data stores [3] often implement isolation levels that are weaker than serializability, such as *snapshot isolation* (SI) [1]. SI guarantees that all reads of a transaction are performed on a snapshot of the database that corresponds to a valid database state with no concurrent transaction. To implement SI, the database maintains multiple versions of the data and the transactions observe different versions of the data depending on their start time.

Implementations of SI have the advantage that writes of a transaction do not block the reads of others. Two concurrent transactions still conflict if they write into the same data item. The conflict must be detected by SI implementation, and at least one of the transactions must abort. To be able to detect the conflicts, the SI implementation must have access to transactional data such as start and commit time of transactions, and the list of transactions that have written into a data item. Due to the large volume of data in distributed data stores, the transactional data cannot fit into a single node and hence should be partitioned into multiple nodes [3]. To maintain the partitioned data, the SI implementations use locks on nodes that store transactional data and run a distributed agreement algorithm such as two-phase commit among them.

The immediate disadvantage of this approach is the need for many additional resources [5]. To avoid this cost, Percolator [3] uses the same data servers to also maintain the transactional data. This, however, resulted into a non-negligible overhead on data servers, which forced the Percolator designers to use heavy batching of messages sent to data servers, inflicting a nontrivial, multi-second delay on transaction processing. More importantly, the locks that are held by the incomplete transactions of a failed client prevent others from making progress.

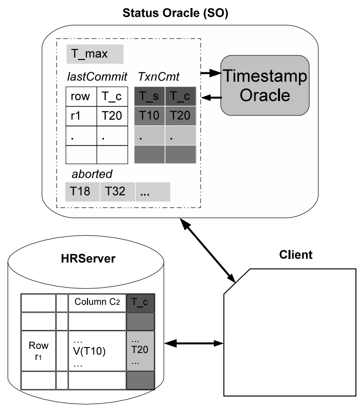
The alternative, lock-free approach could be implemented by using a centralized Transaction Status Oracle (SO) that monitors the commits of all transactions. Note that in contrast to some lock-based approaches in which a centralized lock-server is accessed *prior* to transaction execution [4], SO is accessed *afterwards*. In the centralized approach, each transaction submits the identifiers of modified rows to the SO, where the transaction is committed only if none of its modified rows is committed by a concurrent transaction. The main challenges in design of a SO are the following:

1. To scale to a large number of transactions per second, SO has to service requests from main memory. Plainly, we cannot keep the full history of all transactions in the memory. The partial data could, however, violate the correctness of the SI implementation by missing the detection of a write-write conflict.

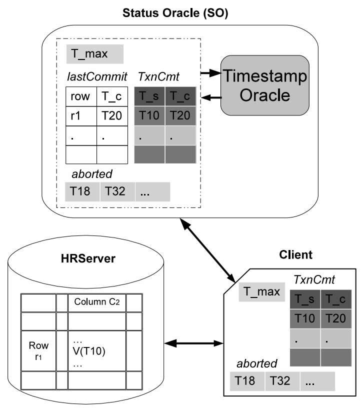
2. The cost of processing TCP/IP headers limits the rate of messages that a single server could receive. This makes SO a bottleneck for large volume of transactional traffic.

3. It is preferable to avoid extra overhead on the data servers. The extra overhead stemmed from maintaining some additional columns and writing the commit timestamp into the data servers, forced the Percolator designer to heavily batch the messages, resulting in a nontrivial

[*] Presenter



(a) Replication of commit data on the data servers.



(b) Replication of commit data on the clients.

Figure 1: Architecture.

multi-second latency in transaction processing.

4. To function on top of any key-value store, we should avoid any modification into the underlying data store.

We have developed CrSO, a client-replicated implementation of SO, that manages the missing data in SO memory. Moreover, CrSO lightly replicates the commit data into the clients where they can locally service a large part of transactional requests. Our approach is based on the following, main observations on our previous implementation of SO [2]: (i) The SO server is not network-bounded; (ii) To process a transaction, the client does not require the data of transactions that commit after its start timestamp is assigned; (iii) The commit timestamp of a transaction is required only during the lifetime of overlapping transactions, and therefore does not have to be persistently stored. Based on these insights, our new approach replicates the commit timestamps of the SO server on the clients, by piggybacking the recent commit data on the timestamp response message. The client maintains a *read-only* copy of the commit data in memory only for a short period and uses them to locally decide which version of data should be used by its transactions.

Figure 1 contrasts our client-based approach with the traditional approach of replicating the commit timestamp on the data servers. In contrast with previous approaches, the commit timestamps are not persistently stored into data servers [2, 3]. CrSO, therefore, brings transaction support to distributed data stores with a negligible overhead on the data servers. Moreover, being client-based, CrSO does not require any change into the data store and could be run on top of any existing distributed data stores. Note that replicated data on clients includes only the data related to transactional management, and the actual data is still stored in the data servers. Moreover, since the client's replica of commit data is read-only, failure of the client affects neither the status oracle nor the other clients.

We have implemented a prototype on top of HBase, which is widely used in industrial applications. The experimental results show that our implementation on a simple dual-core machine can service up to 50K transactions per second (TPS), which is multiple times larger than the maximum achieved traffic in similar data stores. Consequently, we do not expect CrSO to be a bottleneck even for current large distributed storage systems [3]. CrSO is scalable to thousands of clients and since it does not add overhead on data servers, it scales as much as the data store scales (thousands of data servers, in the case of HBase). The overhead on clients is negligible both in terms of memory space and CPU usage.

Acknowledgement

We thank Rui Oliveira and our shepherd, Ion Stoica, for the useful comments. This work has been partially supported by the Cumulo Nimbo project (ICT-257993), funded by the European Community.

References

[1] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ansi sql isolation levels. *SIGMOD Rec.*, 1995.

[2] F. Junqueira, B. Reed, and M. Yabandeh. Lock-free Transactional Support for Large-scale Storage Systems. In *HotDep*, 2011.

[3] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *OSDI*, 2010.

[4] A. Thomson and D. Abadi. The case for determinism in database systems. *VLDB*, 2010.

[5] Z. Wei, G. Pierre, and C.-H. Chi. CloudTPS: Scalable transactions for Web applications in the cloud. *IEEE Transactions on Services Computing*, 2011.