Lock-free Transactional Support for Large-scale Storage Systems

Flavio Junqueira
Yahoo! Research
fpj@yahoo-inc.com

Benjamin Reed
Yahoo! Research
breed@yahoo-inc.com

Maysam Yabandeh
Yahoo! Research
maysam@yahoo-inc.com

Abstract

In this paper, we introduce ReTSO, a reliable and efficient design for transactional support in large-scale storage systems. ReTSO uses a centralized scheme and implements *snapshot isolation*, a property that guarantees that read operations of a transaction read a consistent snapshot of the data stored. The centralized scheme of ReTSO enables a lock-free commit algorithm that prevents unreleased locks of a failed transaction from blocking others. We analyze the bottlenecks in a single-server implementation of transactional logic and propose solutions for each. The experimental results show that our implementation can service up to 72K transaction per second (TPS), which is an order of magnitude larger than the maximum achieved traffic in similar data storage systems. Consequently, we do not expect ReTSO to be a bottleneck even for current large distributed storage systems.

1 Introduction

The data in large-scale storage systems 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. Commercial data storage systems [3, 5] often implement Snapshot Isolation (SI) [1], since it allows for high concurrency between transactions. 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 element, say a database row. The conflict must be detected by SI implementation, and at least one of the transactions must abort. There are two general approaches for detecting the conflict: (i) using a centralized Transaction Status Oracle (TSO) that monitors the commits of all transactions, and (ii) locking the modified rows in a distributed way to prevent the concurrent transactions from modifying them. In lock-based approaches, the locks that the incomplete transactions of a failed client hold might prevent other transactions from making progress during the recovery period.

In the centralized approach, each transaction submits the identifiers of modified rows to the TSO, where the transaction is committed only if none of its modified rows is committed by a concurrent transaction. The centralized approach has the advantage of being lock-free, thus not blocking active transactions due to incomplete transactions. It is, however, challenging to efficiently design a TSO that is not a bottleneck for system scalability and at the same time guarantee the reliability of its data in presence of node failures. Large distributed storage systems [5], therefore, implement a lock-based transactional commit algorithm, missing the benefits of lock-free approaches.

In this paper, we present ReTSO, a reliable and efficient implementation of TSO. To recover from failures, each change into memory must first be persisted into a write-ahead log (WAL): the memory state can always be fully reconstructed by reading from the WAL [4]. We make use of BookKeeper, a system to efficiently and reliably perform write-ahead logging.[1] Since servicing commit requests must be performed atomically, the long service time has an inverse impact on the throughput of TSO. Therefore, all the high-latency operations, including writing into the WAL, could make TSO a bottleneck for system scalability. We explain in Section 3 how ReTSO addresses this challenge by using asynchronous writes into the WAL and postponing the response to the user till completion of the write. Furthermore, ReTSO benefits from a lazy, auto-garbage collector hash map to reduce the number of memory access per request to a minimum. The ex-

[1] http://zookeeper.apache.org/bookkeeper

978-1-4577-0375-1/11/$26.00 ©2011 IEEE

perimental results show that our implementation can service more than 72K TPS, which is an order of magnitude larger than the maximum achieved traffic in similar data storage systems [5].

Roadmap The remainder of this paper is organized as follows. Section 2 explains SI and offers an abstract design of TSO. Section 3 explains the ReTSO design, which is evaluated in Section 4. We finish the paper with some concluding remarks in Section 5.

2 Snapshot Isolation

A transaction is an atomic unit of execution and may contain multiple read and write operations to a given database. To implement snapshot isolation, each transaction receives two timestamps: one before reading and one before committing the modified data. In both lock-based and lock-free approaches, timestamps are assigned by a centralized server, the Timestamp Oracle (TO), and hence provide a commit order between transactions. Transaction txn_i with assigned start timestamp $T_s(txn_i)$ and commit timestamp $T_c(txn_i)$ reads the latest version of data with commit timestamp $\delta < T_s(txn_i)$. In other words, the transaction observes all the modifications of transactions that have committed before txn_i starts.

If txn_i does not have any write-write conflict with another concurrent transaction, it commits its modifications by writing new versions of data with a timestamp equal to commit timestamp of the transaction. Two transactions txn_i and txn_j conflict if the following holds:

1. Both write into row r;
2. $T_s(txn_i) < T_c(txn_j)$ and $T_s(txn_j) < T_c(txn_i)$.

Here, we use the row-level granularity to detect the write-write conflicts. It is possible to consider finer degrees of granularity, but investigating it further is out of the scope of this work. The lock-based distributed implementation of SI prevents write-write conflicts by simply locking the modified rows: if a transaction tries to write into a locked row, then it receives a notification of the write-write conflict and aborts.

2.1 Transaction Status Oracle

In the centralized implementation of SI, a single server, *i.e.*, the TSO, receives the commit requests accompanied by the set of the identifiers of modified rows, R. Since TSO has observed the modified rows by the previous commit requests, it has enough information to check condition 2 for each modified row. Algorithm 1 describes the procedure to process a commit request for a transaction txn_i. In the algorithm, *committed* is the in-memory state of TSO containing the commit timestamp of the modified rows, and $lastCommit(r)$ retrieves the last commit timestamp of row r.

Algorithm 1 Commit request: {commit, abort}

```
1: for each row r ∈ R do
2:     if lastCommit(r) > T_s(txn_i) then
3:         return abort;
4:     end if
5: end for
                                          ▷ Commit txn_i
6: for each row r ∈ R do
7:     committed(r, T_s(txn_i)) ← T_c(txn_i);
8: end for
9: return commit;
```

After commit or abort, the client that drives the transaction txn_f is responsible for cleaning up the temporary database updates performed by the transaction. The failure of the client, hence, leaves the future transactions that are reading from the uncleaned rows, uncertain about the commit status of the written data. For each such row r, the reading transaction txn_r needs to know if $T_c(txn_f) < T_s(txn_r)$. This query can be answered by TSO, which has the status of all committed transaction. (TSO can also benefit from the row Id for efficient implementation.) Algorithm 2 shows the TSO procedure to process queries.

Algorithm 2 isCommitted(row r, timestamp $T_s(txn_f)$, $T_s(txn_r)$) : {true, false}

```
1: if committed(r, T_s(txn_f)) = null then
2:     return false;
3: end if
4: return committed(r, T_s(txn_f)) < T_s(txn_r);
```

2.2 TSO in Action

Here we explain an implementation of SI using TSO on top of HBase [2], a clone of Bigtable [2] that is widely used in production applications. HBase already provides a scalable key-value store, which supports multiple versions of data. It splits groups of consecutive rows of a table into multiple regions, and each region is maintained by a single HRegionServer (HRServer hereafter). A transaction client (client for short) has to read/write cell data from/to multiple regions in different HRServers when executing a transaction. To read and write versions of cells, clients submit get/put requests to HRServers. The versions of

[2] http://hbase.apache.org

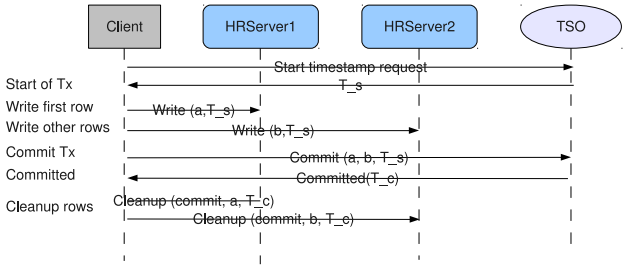

Figure 1: Sequence diagram of a successful commit. The transaction writes two values "a" and "b" into two HRServers.

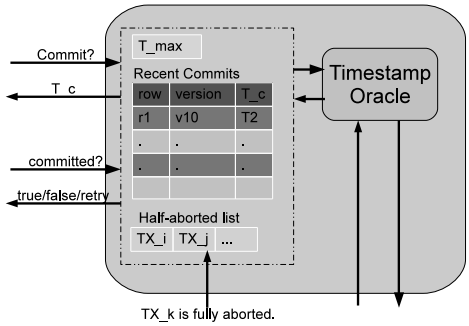

Figure 2: ReTSO: The proposed design of transaction status oracle.

cells in a table row are determined by timestamps. To generate timestamps, the TSO uses a Timestamp Oracle, which guarantees that they are unique and assigned in increasing order. Finally, HRServers must be augmented to also maintain an in-memory `PendingWrite` column.

Figure 1 shows the steps of a successful commit, and the following list details the steps of transactions:

Single-row write. A write operation is performed by simply writing the new data with a version equal to the transaction starting timestamp, $T_s(txn_w)$. Each write will also write into the in-memory `PendingWrite` column.

Transaction commit. After a client has written its values on the rows, it tries to commit them by submitting to TSO a commit request, which consists of a starting timestamp $T_s(txn_w)$ as well as the list of all the modified rows, R. If TSO commits, it returns the commit timestamp, $T_c(txn_w)$, to the client. The whole transaction is then committed and the client starts performing single-row cleanups on the modified rows, with the received timestamp $T_c(txn_w)$. Otherwise, the transaction is aborted and the client must clean up the modified rows.

Single-row cleanup. After a transaction commits/aborts, it should clean up all the modified rows. To clean up each row after a commit, the client updates the timestamp of its written version from $T_s(txn_w)$ to $T_c(txn_w)$ and deletes the corresponding value in `PendingWrite` column. To clean up each row after an abort, the transaction deletes its written version as well as its value under the `PendingWrite` column.

Read. Each read in transaction txn_r must observe the last committed data before $T_s(txn_r)$. To do so, starting with the latest version (assuming that the versions are sorted by timestamp in ascending order), it looks for the first value with commit timestamp δ, where $\delta < T_s(txn_r)$. If the `PendingWrite` column for the version is set by transaction txn_w and $T_s(txn_w) < T_s(txn_r)$, then it is not clear if the last version is committed before $T_s(txn_r)$ or not. (Recall that the commit time is set by TSO and its update on the rows might not be present because of a faulty client.) To verify, the transaction inquires the transaction txn_w commit time of TSO.

3 ReTSO Design

In this section, we analyze the bottlenecks in a centralized implementation of ReTSO and explain how ReTSO deals with each of them. As depicted in Figure 2, the timestamps are obtained from a timestamp oracle integrated into ReTSO. There are three types of requests ReTSO serves: (i) start timestamp requests, (ii) commit requests, and (iii) queries about commit timestamps. A commit request from a transaction txn_i contains: start timestamp $T_s(txn_i)$, and R, the set of modified rows by the transaction. To scale to a large number of transactions per second, ReTSO has to service requests from memory. There are three main challenges that limit the scalability of the ReTSO server: (i) the amount of memory, (ii) number of required memory access for servicing each transaction, and (iii) fault-tolerant replication of the write-ahead log (WAL).

3.1 Memory Limited Capacity

ReTSO first obtains a commit timestamp, $T_c(txn_i)$, from the integrated timestamp oracle. To detect conflicts, ReTSO checks if the last commit timestamp of each row $r \in R$ is less than $T_s(txn_i)$. If the result is positive, then it commits, and otherwise aborts. To be able to perform this comparison, ReTSO requires the commit timestamp of all the rows in the database, which does not fit in memory. To

address this issue, as illustrated in Figure 2, ReTSO keeps only the state of the last NR committed rows that fit into the main memory, but it also maintains $T_{\max}$, the maximum timestamp of all the removed entries from memory. Algorithm 3 shows the ReTSO procedure to process commit requests.

Algorithm 3 Commit request: {commit, abort}

```
1:  for each row r ∈ R do
2:      if lastCommit(r) ≠ null then
3:          if lastCommit(r) > T_s(txn_i) then
4:              return abort;
5:          end if
6:      else
7:          if T_max > T_s(txn_i) then
8:              return abort;
9:          end if
10:     end if
11: end for
                                          ▷ Commit txn_i
12: for each row r ∈ R do
13:     committed(r, T_s(txn_i)) ← T_c(txn_i)
14: end for
15: return commit
```

Line 8 pessimistically aborts the transaction, which means that some transaction could unnecessarily abort. It is not a problem if $T_{\max} - T_s(txn_i) \gg MaxCommitTime$. Assuming 8 bytes for uniques ids, we estimate the required space to keep a row data, including row id, start timestamp, and commit timestamp, at 32 bytes. Assuming 1 GB of memory, we can fit data of 32M rows in memory. If each transaction modifies 8 rows on average, then the rows for the last 4M transactions are in memory. Assuming a maximum workload of 80K TPS, the row data for the last 50 seconds are in memory, which is far more than the average commit time, *i.e.*, around a second.

The third role of ReTSO is to service queries about the status of transactions. In particular when transaction txn_r reads a version of row r that is written by transaction txn_f but it is not sure if the version is committed before $T_s(txn_r)$ or not, it has to check the commit time of transaction txn_f that has written the version. If txn_f is recently committed, the commit times of its rows (including r) are already in the ReTSO memory: if the tuple $(r, T_s(txn_f))$ is in memory, ReTSO verifies whether its commit time, $T_c(txn_f)$, is less than the start time $T_s(txn_r)$. Otherwise txn_f is either incomplete, aborted, or committed long time ago. To distinguish between these three cases, we need to maintain the list of aborted transactions. This list, however, also has to be garbage collected periodically not to fill up the memory space. To do so, each transaction sends a *cleaned-up* request to ReTSO after it has cleaned up its aborted rows. ReTSO then removes the transaction id from the *half-aborted* list.

Algorithm 4 isCommitted(row r, timestamp $T_s(txn_f)$, $T_s(txn_r)$) : {true, false, retry}

```
1: if committed(r, T_s(txn_f)) ≠ null then
2:     return committed(r, T_s(txn_f)) < T_s(txn_r);
3: end if
4: if half-aborted(T_s(txn_f)) then return false;
5: end if
6: return retry;
```

Algorithm 4 shows the procedure to verify if a transaction txn_f has been committed. If neither the committed nor the aborted list contains information about the transaction, at least one of these three cases holds about the transaction:

1. It has been committed a long time ago and its data is garbage collected;
2. It has been aborted and cleaned up, but transaction txn_r has read the row before the cleanup;
3. It has been neither committed nor aborted.

To distinguish between the first and the last case, we keep track of the transactions that are forced to abort because they did not commit in a timely manner, *i.e.*, before $T_{\max}$ advances their start timestamp. To this aim, each time a new start timestamp is assigned by the timestamp oracle TO, we record it in a list. After commit or abort, the start timestamp of the transaction is removed from the list. Once $T_{\max}$ advances due to garbage collection, we check for any uncommitted transaction txn_i in the list for which $T_{\max} > T_s(txn_i)$, and add it to the half-aborted list.

Having the uncommitted, old transactions reduced to aborted ones, we return **retry** for the transaction whose information is missing in the ReTSO, because we could not distinguish between the first and the second case. In both cases, retrying the transaction clears it up: if version v is still there, then it is committed, otherwise it is aborted.

3.2 Processing Overhead

The proposed algorithm for ReTSO is fairly simple. The only major overhead is the number of random memory accesses, *i.e.*, reading and writing into *committed* state. This is because the items of *committed* accessed by different transactions do not exhibit spatial locality (they are likely not to be in the processor caches such as L2 cache) and consequently has to be read from memory. It is important

to reduce the number of memory accesses to a minimum. To this aim, we use the two following policies. First, we try to make the most of data temporal locality by performing the sorting and garbage collection only on the recently accesses data. Second, we avoid using pointers as much as possible because following the pointer references implies jumping to a new random position in the memory, which is probably not loaded into the processor cache. Hash maps are efficient data structures for this purpose, they reduce the average memory access for each lookup to one. A hash map associates each hash value, which is obtained by applying a hash function on the key, with a bucket, and each bucket contains all items that map to the hash value of the bucket. To maintain the elements in a bucket, we use linked lists.

In our design, we use a hash map of row ids to maintain *committed* state, and each hash map item contains the start and commit timestamp. Although a key is the combination of row id and start timestamp, the hash value is computed only on the row id. In this way, the commit times of a row are present in the same linked list. The *put* operation is implemented in a way to ensure that in the linked list of the items with the same hash id, the first one with row id r has the most recent commit timestamp of row r. This feature can be efficiently implemented because, after a *put* operation, the items of the linked list are already loaded into the processor cache and switching the items is cheaply performed in the processor cache. This guarantee pays back in the *lastCommit* operation, since it could return the first item that matches the row id.

Since the size of hash map is fixed (proportional to the size of memory), periodically garbage-collecting old items of the hash map is necessary. Current garbage collection policies require further random accesses to memory, which exacerbates the average processing time of requests. To further reduce average memory access for each transaction, we designed a lazy, auto-garbage collector hash map. The disposal of an old item in the map is postponed to a time that the item is already loaded into the processor cache, *i.e.*, once there is a hit for a *put* operation. After such a hit, the other items with the same hash are already loaded into the processor cache and we can cheaply garbage-collect them.

3.3 ReTSO Data Reliability

The ReTSO must persist the changes into a WAL before updating the memory state. In this way, if the ReTSO server fails, we could still recreate the memory state from the WAL. The WAL is also ideally replicated across multiple remote storage devices to prevent loss of data after a storage failure. Writing into multiple remote machines could be very expensive and it is important to prevent it from becoming a bottleneck. We use Bookkeeper for this purpose, which could efficiently perform up to 20,000 writes of size 1028 bytes per second into a WAL. Since ReTSO requires frequent writes into the WAL, multiple writes could be batched with no perceptible increase in processing time. With a batching factor of 10, BookKeeper is able to persist data of 200K TPS.

Processing commit requests involve two parts: checking the current memory state for conflicts, and changing it based on the new commit. These two steps must be performed atomically. The write into the WAL before the second step, *i.e.*, updating the memory state, therefore, induces a non-trivial latency in service time. We therefore postpone writes into the WAL until after finishing the second step. The response to the user is, however, sent only after the asynchronous write into the WAL terminates, which is simple to implement due to the asynchronous interface of BookKeeper. In the case the ReTSO crashes and loses its memory state, the recovered memory state from the WAL encompasses all the changes that the clients have observed.

When processing queries associated to read transactions, the state of the ReTSO memory while processing such queries might have values that are still being written to WAL. To guarantee that the state this transaction observes is persisted, each read query must also perform a null write into the WAL to flush the channel between ReTSO and the WAL. ReTSO sends the reply only after receiving an acknowledgement, which implies that all the previous pending writes into the WAL have also been persisted. This step ensures that the state observed by the read query is already persisted into the WAL and is amenable to recovery.

4 Evaluation

Here we evaluate the performance of ReTSO. The experiments aim to answer the following questions: (i) is ReTSO a bottleneck for transactional traffic in large-scale distributed storage systems? (ii) what is the overhead of ensuring reliability in ReTSO? We used 10 machines with 2.13 GHz Dual-Core Intel(R) Xeon(R) processor, 2 MB cache, and 4GB memory: 1 machine for the ZooKeeper coordination service, 4 machines for BookKeeper, 1 machine for ReTSO, and 4 machines for clients. Each client runs NT transactions in parallel. Each transaction updates n rows, randomly selected out of 20M rows, where n is a random number between 0 and 20. Since the goal of the experiments is to measure the ReTSO performance,

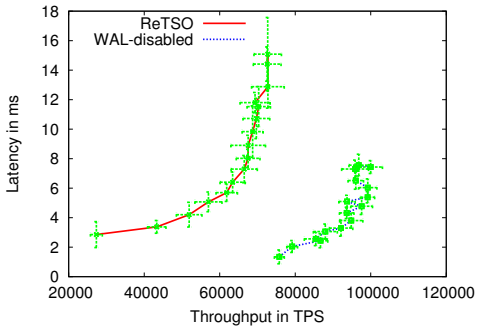

Figure 3: Latency vs. Throughput.

the execution time of transactions is set to zero at clients.

Micro-benchmarks. To measure the highest throughput that ReTSO can handle, we run 3 clients with $NT = 1000$ to keep the buffer of ReTSO full. The average throughput is 81k TPS. To measure the minimum latency, we run the system with one client and $NT = 1$. The average latency is 0.85 ms and the average throughput is 1027.56 TPS. The latency is 0.87 ms more than the latency of simply requesting a commit timestamp from the timestamp oracle (0.17 ms), which is the most that a lock-based approach expects from a centralized server. We also run the same experiment with writes into BookKeeper disabled to measure the overhead of adding persistence. The average latency in this case is 0.171 ms and the average throughput is 3063.08 TPS. Due to the efficient hash map developed, the latency of handling the commit requests is negligible. The major overhead still comes from the reliable writes into multiple remote persistent storage.

Scalability. To assess scalability, we increase the number of clients from 1 to 16 ($NT = 100$) and plot the average latency vs. the average throughput in Figure 3. When increasing the load of the system, the throughput increases up to 72.6k with 16 clients (which is close to the measured limit of 81k). With higher throughput, latency also increases mostly due to the buffering delay at the ReTSO. The system scales up to 72.5k TPS with average latency of 14.4 ms, which is appropriate for transactions in large-scale systems.

WAL-disabled in Figure 3 presents the results of the same experiment with the difference that the writes into BookKeeper are disabled. Adding persistence impacts both throughput and latency. The throughput decreases mostly because of the processing overhead of messages that have to be sent to and received from the remote storage nodes. The lower throughput also exacerbates the latency due to more buffering delays.

5 Concluding Remarks

We have presented a reliable and efficient implementation of lock-free transactions in large-scale distributed storage systems. The traffic that ReTSO can handle is an order of magnitude more than current commercial systems. The highest traffic achieved by TPC-E benchmark is 3,183 TPSE in NEC Express5800 and 11,200 TPS in Percolator [5]. These preliminary results are promising and provide evidence that lock-free transactional support could be brought to large-scale distributed storage systems without hurting the reliability or the scalability of the system.

Acknowledgement

We thank Daniel Gómez Ferro for implementing the mechanism to track uncommitted transactions. This work has been partially supported by the Cumulo Nimbo project (ICT-257993), funded by the European Community.

References

[1] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ansi sql isolation levels. *SIGMOD Rec.*, May 1995.

[2] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26:4:1–4:26, June 2008.

[3] Yi Lin, Kem Bettina, Ricardo Jiménez-Peris, Marta Patiño Martínez, and José Enrique Armendáriz-Iñigo. Snapshot isolation and integrity constraints in replicated databases. *ACM Trans. Database Syst.*, 2009.

[4] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, March 1992.

[5] Daniel Peng and Frank Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, Vancouver, BC, Canada, 2010. USENIX Association.