Serializable Snapshot Isolation in Shared-Nothing, Distributed Database Management Systems

Yang Lu

Brown University

yanglu@cs.brown.edu

ABSTRACT

NoSQL data storage systems provide high scalability and availability in exchange for limited transactional guarantees. In many cases, however, an application cannot give up transactional support but still needs the scalability provided by such systems. One approach for overcoming this limitation is to implement Snapshot Isolation (SI) on top of these systems. SI prevents most non-serializable executions and its optimistic concurrency control never delays read-only transactions. Nevertheless SI does not guarantee the serializability that many applications require. For example, the "write anomaly" is a well-known problem permitted by SI that violates data consistency [4]. This problem occurs when two or more concurrent transactions update a data item that the other reads. To resolve this problem, we present a new commit algorithm to ensure serializable SI in a large-scale distributed system. The algorithm takes a pessimistic approach to detect and avoid non-serializable execution schedules. We also include a performance study that demonstrates both the correctness and scalability of our new algorithm.

1. INTRODUCTION

The fast growing size of data has necessitated the need for large-scale and highly available data store services. It has been widely recognized that NoSQL systems, such as HBase [8], are scalable in such situations. These systems usually achieve scalability through horizontally partitioning data across a cluster of shared-nothing nodes. However, unlike traditional relational databases managements systems (RDBMS), NoSQL systems provide limited support for transactions, which is critical to large-scale, enterprise-level business applications.

Yahoo's *Reliable and efficient Transaction Status Oracle* (ReTSO) [1] middleware aims at providing transactional support on HBase. It features a centralized lock-free concurrency control algorithm that implements snapshot isolation (SI). Although ReTSO lays the foundation for supporting distributed transactions in HBase, it suffers from the "write anomaly" due to the nature of snapshot isolation and its commit algorithm. For example, executions under SI can corrupt data when programs interleave, even though each program individually preserves the databases' integrity constraints. Many large-scale applications require stronger transactional guarantees, namely serializability [6]. Serializability ensures that every concurrent execution of transactions be equivalent to running the transactions one after another in some order.

In this report, we describe a serializable SI transaction commit algorithm that guarantees serializability in large-scale, distributed DBMSs. The key idea of our algorithm is that by keeping additional information at a central server, the system is able to detect at run time distinctive conflict patterns that occur in every non-serializable execution (write anomaly) under SI. Our detection approach is conservative and prevents every non-serializable execution at the cost of few unnecessary aborts. Experiments show that the algorithm achieves serializability at the cost of 23% of transactions per second committed by the original system in normal cases.

Various techniques [3,4,6,11,12,13] have been developed on traditional RDBMSs to ensure serializability in SI. Our algorithm is inspired and conceptually similar to the algorithm in [3]. Implementing such an algorithm in a distributed system like HBase poses several unique challenges. For example, in

traditional RDBMSs, we can track transaction dependencies from a centralized lock table, while the same information is not available in the current ReTSO and HBase implementation.

The remainder of this report is organized as follows. Section 2 briefly explains the theory of snapshot isolation. Section 3 introduces the ReTSO middleware for HBase and discusses the write anomaly that arises in its commit algorithm. Section 4 describes our new commit algorithm that is designed to fix the problem. This new commit algorithm is evaluated in Section 5.

2. SNAPSHOT ISOLATION

The ideal execution schedule of transactions in a distributed database is when all the interleaved concurrent executions of transactions are equivalent [4] to serial executions. Such a schedule is said to be serializable. One common approach to ensuring a serializable schedule in a distributed system is to use two-phase-commit (2PC) [14]. But a previous study [16] has shown that this approach does not scale well because one participant of a distributed transaction may block while waiting for other participants of the same transaction.

Snapshot Isolation (SI) is an isolation level that does not ensure serializability in database management systems (DBMS). However, it is attractive to implement it on distributed databases, since it prevents most of the common concurrency problems [4] and increases the number of concurrent of transactions by never having a read operation block any updates. Besides these, HBase keeps multiple versions of the same data item, which is necessary to implement SI.

In SI, a transaction T_i receives a start timestamp ts_i when it starts and a commit timestamp tc_i when it commits. Whenever T_i reads data item X, it reads the version that is created by the last committed transaction of all the transactions that committed before ts_i. Whenever T_i updates the data item X, it creates a new version. SI also enforces a restriction called **First-Committer-Wins** (**FCW**) rule: if transaction T_2's commit timestamp tc_2 is in transaction T_1's transaction life [ts_1, tc_1], T_1 can successfully commits only if T_2 did not write data that T_1 also wrote, otherwise, T_1 will abort.

The problem of making SI serializable has been extensively studied in [3,6,11,12,13]. The key idea of solving this problem is to avoid the write anomaly problem. The write anomaly is a concurrent problem permitted by SI that can violate data integrity and consistency. It happens when two or more concurrent transactions when one transaction changes a value that the other transaction reads. FCW allows this, since different items are changed in each transaction. A detailed example of the write anomaly is discussed in Section 3.4. The problem of how to identify the transactions that causes the write anomaly is addressed in [6]. This work uses **Dependency Serialization Graph (DSG)** to identify transactions at runtime. A DSG contains vertices representing transactions and three types of dependency edge defined below:

RW-dependency edge (Vulnerable edge): There is a RW-dependency from T_1 to T_2, if T_1 reads a version of item X and T_2 produces a new immediate successor version X.

WW-dependency edge: There is a WW-dependency between T_1 and T_2, if T_1 produces a version of data item X and T_2 produces a new immediate successor version X. T_1 and T_2 can't execute concurrently.

WR-dependency edge: There is a WR-dependency from T_1 to T_2, if T_1 produces a version of data item X and commits, then later T_2 reads X. T_1 and T_2 cannot execute concurrently since T_2 will not be able to see the version T_2 produced unless T_1 commits before T_2 starts.

Figure 1 shows examples of these three dependency edges described above.

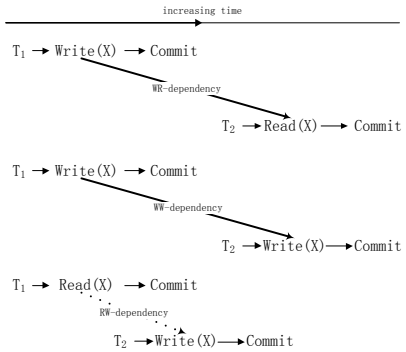

Figure 1 WR/WW/RW-dependency

The write anomaly happens if the DSG of the program contains a cycle and there are two vulnerable edges in a row as part of the

cycle. Figure 2 shows an example. The main work of [6] demonstrates that if a DSG is free of such cycles, then every execution of the programs is serializable. In Section 4, we discuss in detail how to detect and eliminate the write anomaly

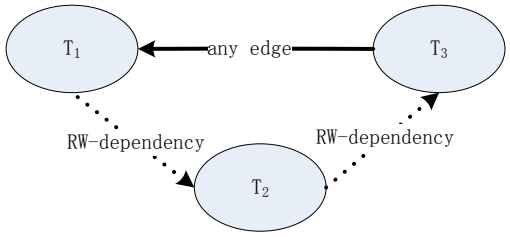

Figure 2 DSG of a non-serializable execution schedule

3. ARCHITECTURE OVERVIEW

This section introduces HBase and ReTSO architecture and describes the lock-free commit algorithm ReTSO uses for concurrency control and its flaws.

3.1. HBase Overview

HBase [8] is an open-source implementation of Google's BigTable [2]. It is a distributed multi-dimensional map that maps a row key, column key and a timestamp to an uninterpreted array of bytes:

{row:string, column:string; time:int64}-> bytes

In HBase, applications store data into tables composed of rows and columns. An HBase table contains multiple versions of the same data indexed by timestamps. These timestamps can be automatically generated by HBase or be explicitly assigned by client applications.

HBase employs a master-slave topology. Tables are horizontally partitioned into disjoint **regions** stored on slave machines called **RegionServer**. HBase provides single-row-level exclusive locks, but does not support multi-row atomicity.

3.2. ReTSO Architecture

ReTSO [9] is an open-source middleware project started at Yahoo! that uses snapshot isolation and to add lock-free transactional support on top of HBase.

ReTSO has two main components: **Transaction Client Library** (TCL) and **Transaction Status Oracle** (TSO). Applications use the TCL to request start timestamps from the TSO server, optimistically writes to an HBase RegionServer, and finally send commit requests to TSO. The TSO server is a single server which monitors the modified rows by transactions and uses that to detect write-write conflict. Additionally, The TSO server uses a distributed logging service called BookKeeper [10] to keep a write-ahead log and to recover the data in memory in case of failure. Figure 3 shows the architecture of ReTSO.

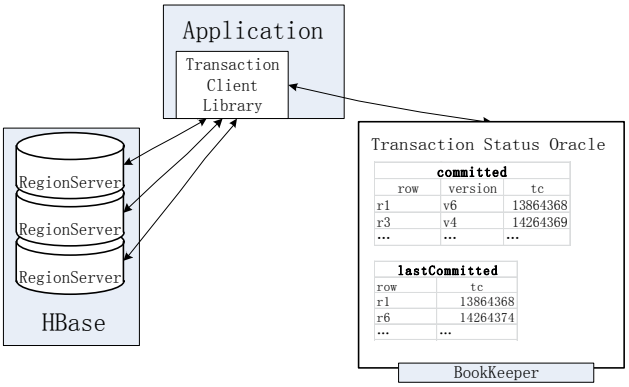

Figure 3 ReTSO architecture overview

To implements SI, the TSO server keeps two hash tables, ***committed*** and ***lastCommited***, in memory. The ***lastCommited*** table contains the most recent commit timestamp of modified rows. The ***committed*** table contains the commit timestamps of completed transactions that created a new version of a particular row.

3.3. Snapshot Isolation in ReTSO

A typical transaction with read and write operations in ReTSO processes as described below:

Transaction Start. An application uses the TCL to request a start timestamp from the TSO server when it initiates a transaction. The TSO server assigns a unique timestamp to each transaction. This start timestamp is also used as transaction ID.

Write. A write is performed optimistically by simply writing the new data with a version equal to the transaction starting timestamp to HBase RegionServer. Each transaction keeps references to all the rows it modified in its own in-memory object at the client side. Whenever the transaction aborts, it cleans up the

new version of the row that it updated.

Read. In order to read a row from the database, the transaction must obtain a portion of all the versions of that row that is committed before its own start timestamp. Noted that a transaction may read a version of a row created by a failed write operation, it must verify the version by querying the TSO server. A version is valid if the transaction that created it is committed. A read operation fetches 10 (by default) latest versions before its own start timestamp and verifies each version in descending order starting from the most recent one until the TSO server acknowledges that the transaction that created the version has committed. If none are acknowledged, the TCL fetches more versions from HBase. **Algorithm 1** [1] shows how the TSO server answers verification queries.

Algorithm 1 isCommitted (row *r*, timestamp *version*, timetamp *ts*) -> {true, false}

```
1:  if committed(r, version) == null
2:      return false;
3:  else
4:      return committed(r, version) < ts
```

Transaction Commit. When a client commits a transaction, the TCL sends a commit request along with all the row identifiers the transaction modified to the TSO server. The TSO server checks if each row modified follows FCW rule by looking up the ***lastCommited*** table. This check guarantees that there are no concurrent transactions updating the same data item. If the check passes, it will update the ***commited*** table and send a commit acknowledgement back to the client. Otherwise, it sends a failure response to the client. **Algorithm 2** [1] shows how the TSO server processes commit request.

Algorithm 2 CommitRequest (modified row set *R*, timestamp *ts*) -> {commit, abort}

```
1:  for each row r belongs to R
2:      if lastCommited(r) > ts
3:          return abort
4:  end for
5:  for each row r belongs to R
6:      committed(r, ts) ← tc //assign a new commit timestamp
7:  end for
8:  return commit
```

Cleanup. After a transaction aborts, the TCL deletes all the versions the transaction created for all of the rows that it modified.

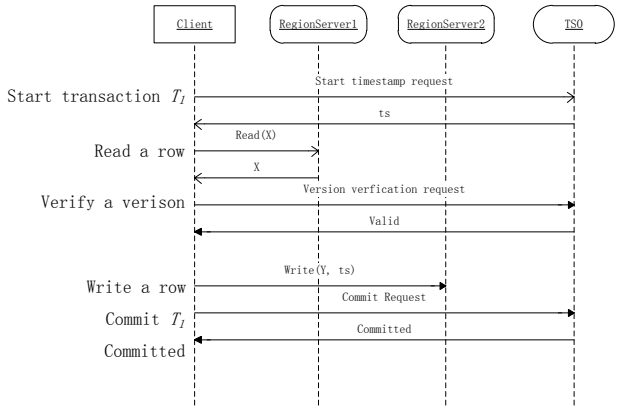

Figure 4 Sequence diagram of a successful commit. Transacion T1 reads X and updates Y.

3.4. Write Anomaly in ReTSO

The current implementation of ReTSO does not take the write anomaly into consideration. Data integrity can be corrupted if it occurs. For example, suppose we have two values *X* and *Y* that represent checking account and savings account of a certain customer, with an invariant that $X+Y>0$. The bank's business logic may permit either account to be overdrawn, as long as the sum of the account balances remains positive. Assume that initially X_0=50 and Y_0=50. Transaction T_1 with a start timestamp $ts_1=1$ reads X_0 and Y_0, subtracts 90 from *X* and creates a new version $X_1=-40$. Transaction T_2 with a start timestamp $ts_2=2$ concurrently reads X_0 and Y_0, subtracts 80 from *Y* and creates a new version $Y_2=-30$. T_1 and T_2 respectively send their commit request to the TSO server. The TSO server accept both requests because T_1 and T_2 update different data items *X* and *Y* respectively and no write-write conflict will be detected in ***lastCommited*** table. In this case each update transaction is safe by itself, but when both occur, the database will violates the invariant $X+Y>0$. This problem cannot be detected using information available in the TSO server. Hence a new mechanism is needed to avoid the write anomaly.

4. MITIGATING WRITE ANOMALIES

We now describe a new commit algorithm that prevents the write anomaly and guarantees serializable SI in a distributed DBMS. Our new algorithm maintains additional transaction dependency information at the TSO server and uses it to detect the distinctive pattern of non-serializable executions. This section discusses in detail how the write anomaly is detected and various design issues in implementing the new algorithm.

4.1. Write Anomaly Detection

To avoid complicating the original architecture, detections will be performed by the TSO server. The design should follow three goals: (1) minimizing the process overhead; (2) ensuring correctness; (3) maintaining the high scalability of the original system. These three goals affect the various design choices described below.

Conservative Detection vs Precise Detection. Many approaches have been proposed to avoid the write anomaly in RDBMSs. These approaches can be generally categorized into two groups. The work in [11][13] keeps a complete DSG graph in memory while the system is running, which leads to aborting only the transactions that causes write anomalies. The other techniques [3][12] tend to have less overhead and only need to keep small amount of information on a central server, however, they are often pessimistic and are afflicted with unnecessary aborts. To achieve scalability and avoid high processing overhead, we choose the conservative detection algorithm. Our algorithm detects a potentially non-serializable execution whenever it finds two consecutive RW-dependency edges in the DSG, where each of the edges involves two concurrent transactions. Whenever such a situation is detected, one of these transactions will be aborted. To support this algorithm, the TSO server needs to maintain the **inConflict** and **outConflict** references for each transaction *T*, tracking the transactions that have RW-dependencies with *T*.

RW-Dependency Detection. The key to detecting the write anomaly lies in detecting the RW-dependencies between transactions. There are two situations when two transactions could have an RW-dependency. One situation arises when a transaction T_1 reads a version of an item *X*, and the version it reads is not the most recent version of *X*. In this case the writer, transaction T_2, of any more recent version of *X* was active after T_1 started, and so there is an RW-dependency from T_1 to T_2. This allows detecting RW-dependency edges for which the read operation is interleaved after a write. To detect edges where a read is performed before a new version is created by a concurrent transaction, we need to somehow keep track of which row is read by which transaction. When a commit request for transaction T_1 comes, using this information, the TSO server checks whether each of its updates conflicts an uncommitted read transaction T_2. In RDBMS, transaction dependency information can be retrieved in lock table. Lacking of such data structure in HBase, we build a similar in-memory hash table.

Memory Limitation. For highly concurrent applications with many clients, it is difficult for a single TSO sever to keep all of the information that is needed in memory. Thus, a timestamp low-bound is set to avoid the TSO server's memory from overflowing. All of the in-memory lookup tables only monitor transactions that start after than the bound. Any uncommitted transaction with a start timestamp earlier than the bound is aborted. An analysis of the system's memory usage is discussed in 5.4.

Victim Selection. When two transactions form an RW-dependency cycle, either transaction could be aborted without loss of correctness in order to break the cycle and ensure serializability. Our new algorithm chooses to abort the transaction that has not been committed to reduce the overhead of cascading aborts. When both transactions are not committed, aborting the younger of the two transactions is preferred, since it may increase the proportion of complex transactions running to completion.

4.2. Implementation Details

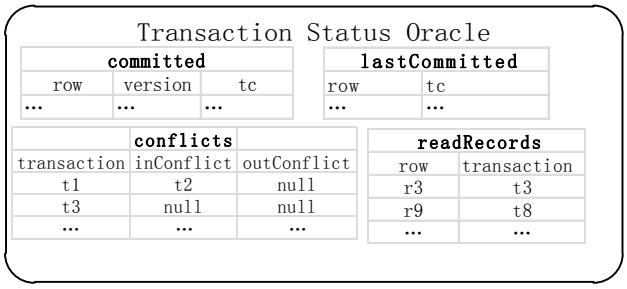

Figure 5 In-memory Tables of the TSO server

Our algorithm needs the TSO to maintain two additional information tables in its local memory. The ***conflict*** table tracks RW-dependencies between transactions. The ***readRecords*** table tracks rows read by the uncommitted transactions and the transaction that reads it.

The following shows the modification we made to the original Algorithm 1 and Algorithm 2 described in Section 3.3. The pseudo-code of our new algorithm is listed in appendix.

Transaction Start. When a client requests to initiate a new transaction, the TSO server assigns the transaction a start timestamp and inserts a new entry into the **conflict** table.

Read. Transactions perform modified read operations by reading versions both before and after its start timestamp. When a transaction reads a version that is created after its own timestamp, it forms an RW-dependency edge to the transaction that creates the newer version. For versions that are earlier than the transaction's start timestamp, the client verifies each version in descending order starting with the nearest one until one of them is valid. The TSO server keeps every successful read in the **readRecords** table until this transaction is committed.

Write. Write operations perform an optimistic write as described in the original implementation.

Transaction Commit. When a client commits a transaction, it sends a commit request to the TSO server. The TSO server handles these requests using modified-CommitRequest method. Other than checking for ww-conflicts in **Algorithm 2**, it checks that for each row that transaction T_1 updated whether it was read by an uncommitted transaction T_2 whose start timestamp is earlier than T_1's commit timestamp. If such a row exists, a RW-dependency from T_2 to T_1 is marked.

When a RW-dependency edge is detected between a reading transaction and a writing transaction, the **conflicts** table is populated with the proper tuple and the victim selection policy discussed in Section 4.1 is applied.

5. PERFORMANCE EVALUATION

In this section, we present the correctness and cost evaluations of our new algorithm. For the correctness evaluation, the goal is to validate that the new transaction commit algorithm guarantees serializable transactions schedules correctly. For the cost evaluation, we present a comparative evaluation between the new commit algorithm and the original one. Lastly, we explore the memory usage in the TSO server.

Our evaluation is performed on a dedicated HBase cluster of six nodes, each running on a machine with Linux 2.6.32, 6 GB of RAM, and a 4-core 3.0GHz AMD Phenom II CPU. The TSO server is deployed on another dedicated machine with the same hardware configuration. The benchmark driver is written in Java and runs on eight separate client machines, each equipped with Linux 2.6.32, 4 GB of RAM and a dual-core 2.0GHz AMD Athlon CPU. We emulate a varying number of concurrent clients using multithreading. We denote the number of concurrent client threads as multiprogramming level (MPL),

5.1. SmallBank Benchmark Overview

The SmallBank benchmark is a workload that is specifically designed to illustrate the write anomaly permitted by SI [7]. SmallBank is a small banking database consisting of three tables: ***Account*** *(Name, CustomerID)*, ***Saving*** *(CustomerID, Balance)*, ***Checking*** *(CustomerID, Balance)*. Each tuple in ***Account*** table represents a customer. The primary key of ***Account*** is *Name*. *CustomerID* is a primary key for both ***Saving*** and ***Checking*** tables.

Account		Saving		Checking	
PK	Name	PK	CustomerID	PK	CusomerID
	CustomerID		Balance		Balance

Figure 6 Tables of the SmallBank Benchmark

The SmallBank benchmark executes five different transaction programs.

Balance, or *Bal(N)*, looks up Account to get the *CustomerID* value for *N*, and then returns the sum of savings and checking balances for that *CustomerID*.

DepositChecking, or *DC(N, V)*, looks up the Account table to get *CustomerID* corresponding to the name *N* and increase the checking balance by *V* for that *CustomerID*.

TransactSaving, or *TS(N, V)*, looks up the Account table to get

CustomerID corresponding to the name *N* and increases the savings balance by *V* for that customer.

Amalgamate, or *Amg(N1, N2)*, reads the balances for both accounts of customer *N1*, then sets both to zero, and finally increases the checking balance for *N2* by the sum of *N1*'s previous balances.

WriteCheck, or *WC(N, V)*, looks up Account to get the *CustomerID* value for *N*, evaluates the sum of saving and checking balances for that *CustomerID*. If the sum is less than *V*, it decreases the checking balance by *V+1* (reflecting a penalty of 1 for overdrawing), otherwise it decreases the checking balance by *V*.

The client threads execute transactions one at a time. The transaction type is chosen uniformly at random from the five transactions. The database is populated with 18,000 randomly generated customers and their checking and saving accounts. The workload is skewed so that some customers are accessed more often than others. We define a certain number of customers as "hotspot". 90% of all transactions deal with a customer who is chosen uniformly from 1000 customers. The other 10% of transactions access customers from outside the hotspot. In the normal contention experiments the hotspot has 1,000 customers, but in the high contention experiments, the hotspot is made to have 10 customers only. Each experiment is repeated three times and the average of results is calculated. Before each experiment, we start the HBase and ReTSO servers and execute a 30-second warm-up, allowing the clients to reach full MPL. At the end of each experiment, we bring down the HBase and ReTSO server after a 30-second cold down.

5.2. Correctness

The SmallBank benchmark requires that the total balance of the two account of the same customer should be non-negative. Thus, if a balance inquiry transaction reads a negative total balance, it indicates that a write anomaly has occurred. Additionally, at the end of each benchmark run, the total balance of each joint account is checked to ensure correctness.

We conduct the experiments using 16 client threads. Each experiment runs the SmallBank benchmark for 30 minutes. Figure 9 shows that out of the total number of aborts, approximately 11% are due to the write anomaly problem. The accumulated number of invalid balance queries over time and the total number of corrupted customer accounts is given in Figure 7 and Figure 8. The results show that applying our new commit algorithm avoids every non-serializable execution permitted in ReTSO.

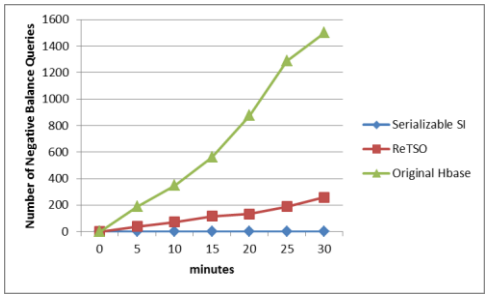

Figure 7 Accumulated number of negative balance queries over time.

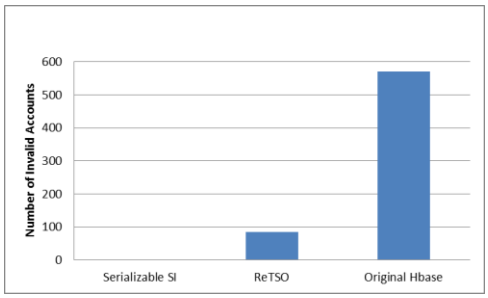

Figure 8 Total number of invalid accounts.

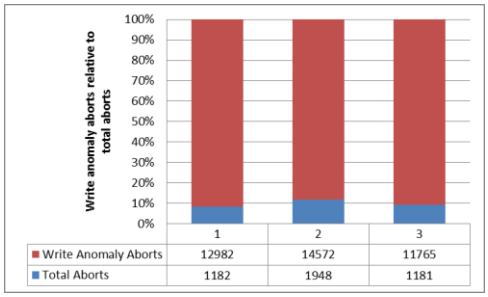

Figure 9 Aborts distribution in three experiments

5.3. Cost of Serializability

Another important aspect to evaluate is the overhead of supporting serializability. We run the SmallBank benchmark with MPL from 8 to 256 under both normal contention and high contention configurations. In the normal cases, the cost of

ensuring serializability shown in Figure 10 is approximately 23% of the original committed transactions per second (TPS). In the high contention experiments, the size of the hotspot region is changed from 1000 customers to only 10 customers. The increased contention from the small hotspot reduces the overall throughput. TPS shows a significant drop in Figure 11, due to an increasing number of conflicts between transactions.

The results in Figure 10 and Figure 11 show a linear growth in throughput over number of client treads when MPL is below 50. When MPL is over 50, our client machine is saturated because the overhead of treads switching on its two CPU cores.

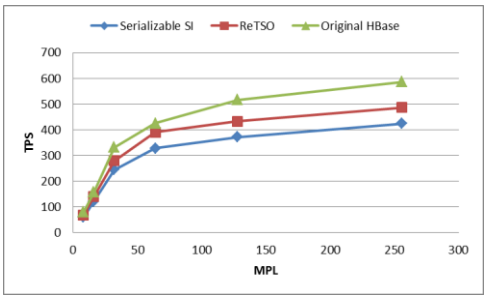

Figure 10 Throughput over MPL

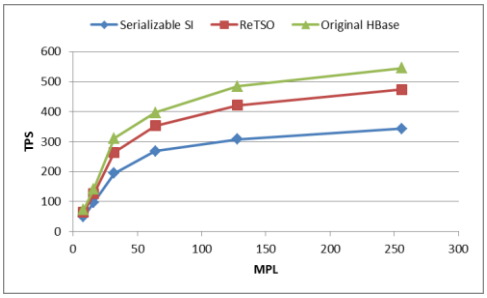

Figure 11 Throughput over MPL under high contention

5.4. Memory Usage

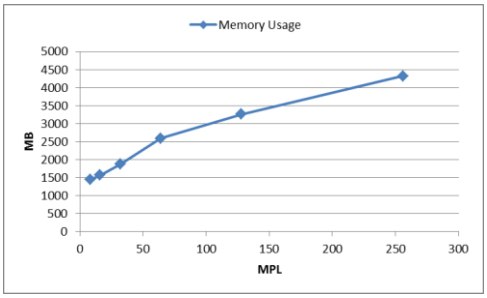

Figure 12 Memory usage in the TSO server

Memory usage in the TSO server is evaluated respectively with MPL from 8 to 256. Figure 12 shows a steady growth of memory use over the number of client threads. This is mainly due to the growing size of the four in-memory hash tables used to maintain the additional information needed by our algorithm. The TSO server running on server with 6GB RAM is predicted to exhaust its memory on 500 client processes if we assume a linear growth of memory use over the number of clients according to Figure 12

Figure 13 shows the relative size of the four hash tables TSO has in memory. A rapid growth in percentage is shown for the ***committed*** table due to the growing number of committed transactions the TSO server has to keep track of. This table, used for verifying read operations and guaranteeing correctness, is the main usage of the TSO server's memory.

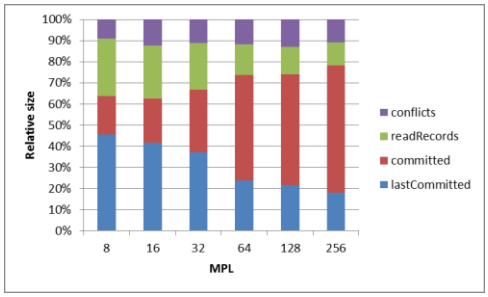

Figure 13 Relative Table Size

6. FUTURE WORK

In the future, we hope to improve the precision of our write anomaly detection algorithm by evaluating and reducing the rate of false positives. We have not yet carried out a thorough analysis of the impact of unnecessary aborts, but we expect that the precise detection would require a prohibitive amount of additional memory and CPU time. Additionally, we are extending our approach to other NoSQL large-scale data services.

7. CONCLUSION

This report describes a new transaction commit protocol that ensures serializable SI in distributed database management systems. The new mechanism makes use of additional information kept at a centralized coordinator to detect and prevent non-serializable executions. We demonstrate the correctness and scalability of our new algorithm by experiments. The evaluation shows that with the cost of only less than 23% throughput loss, we achieve serializability in a distributed NoSQL DBMS.

REFERENCES

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

[2] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. B*igtable: A Distributed Storage System for Structured Data.* In OSDI, pages 205–218, 2006.

[3] M.J. Cahill, U. Roehm, A.D. Fekete. *Serializable Isolation for Snapshot Databases.* SIGMOD 2008: 729-738

[4] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, , and P. O'Neil. 1995. *A critique of ANSI SQL isolation levels.* In SIGMOD '95: Proceedings of the 1995 ACM SIGMOD international conference on Management of data, pages 1–10. ACM Press, June.

[5] Kapali P. Eswaran, Jim Gray, Raymond A. Lorie, and Irving L. Traiger. 1976. *The notions of consistency and predicate locks in a database system.* Communications of the ACM, 19(11):624–633. Philip A. Bernstein and Nathan Goodman. 1981. Concurrency control in distributed database systems. ACM Comput. Surv., 13(2):185–221.

[6] Alan Fekete. 1999. *Serializability and snapshot isolation.* In Proceedings of Australian Database Conference, pages 201 ¨C210. Australian Computer Society, January.

[7] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. 2005. *Making snapshot isolation serializable.* ACM Transactions on Database Systems (TODS), 30(2):492–528.

[8] Cloudera http://hbase.apache.org

[9] Yahoo! https://github.com/dgomezferro/omid/wiki

[10] Cloudera http://zookeeper.apache.org/bookkeeper/

[11] S. Revilak, P. O'Neil, and E. O'Neil. *Precisely Serializable Snapshot Isolation (PSSI).* In ICDE'11, pages 482 –493, april 2011.

[12] S. Jorwekar, A. Fekete, K. Ramamritham, and S. Sudarshan. *Automating the detection of snapshot isolation anomalies.* In VLDB, pages 1263–1274, 2007.

[13] M. Bornea, O. Hodson, S. Elnikety, and A. Fekete. *One-copy serializability with snapshot isolation under the hood.* In ICDE'11, pages 625 –636, april 2011.

[14] M. L. Liu, D. Agrawal, and A. El Abbadi. *The performance of two phase commit protocols in the presence of site failures. Distrib. Parallel Databases*, 6:157–182, April 1998.

[15] Mohammad Alomari, Michael J. Cahill, Alan Fekete, and Uwe Röhm. 2008a. *The cost of serializability on platforms that use snapshot isolation.* In ICDE '08: Proceedings of the 24th International Conference on Data Engineering, pages 576–585, Cancún, México, April 7-12. IEEE.

[16] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. *Ceph: A scalable, high-performance distributed file system.* In Proc. OSDI, 2006.

APPENDIX

Serializable SI Commit Algorithms

The TSO server processes transaction initiation requests from clients

	modified-start(transaction *T*)
1:	*ts* ←new timestamp
2:	insert an empty entry for *T* in the *conflict* table
3:	send start time stamp *ts* to client

Read operation performed by transaction T_1

	modified-read(row *r*, version set *V*, timestamp ts_1)
1:	**for** each version v belongs to *V*
2:	Let ts_2 be start timestamp of transaction T_2 that created v.
3:	**if** $ts_2 > ts_1$
4:	markConflict(T_2,T_1)
5:	**else if** modified-isCommitted(r,v,ts_1)
6:	**return** v;
7:	**end for**

```
8:  fetch more versions
9:  go to line 1
```

The TSO server verifies the version that clients read.

modified-isCommitted (row *r*, timestamp *version*, timetamp *ts*) -> {true, false}

```
1:  if committed(r, version) == null
2:     return false;
3:  else
4:     if committed(r, version) < ts
5:           insert (r, version) into the readRecords table
6:           return true
7:     else
8:           return false
```

The TSO server processes commit requests from clients for transaction T_1.

modified-CommitRequest (modified row set *R*, timestamp ts_1) -> {commit, abort}

```
1:  for each row r belongs to R
2:     if lastCommited(r) > ts1
3:        return abort
4:  end for
5:
6:  for each row r belongs to R
7:     committed(r, ts1) ← tc1 //assign a new commit timestamp
8:  end for
9:  for each row r belongs to R
10:    if readRecords(r)!=null
11:       let T2 be the uncommitted transaction that read r
12:       if ts2<tc1
13:          markConflict(T2,T1)
14: end for
15: return commit
```

When a RW-dependency is identified, the TSO server records it in the *conflict* table and performs proper aborts.

markConflict(Transaction *Reader*, Transaction *Writer*)

```
1:  if Reader has committed
          AND conflicts(Reader).inConflict!=null
2     abort(Writer)
3     return
4:  if Writer has committed
          AND conflicts(Writer).outConflicts!=null
5:    abort(Reader)
6:    return
7:  if conflicts(Writer).outConflict!=null
          OR conflicts(Reader).inConflicts!=null
8:    abort the younger transaction of Reader and Writer
9:    return
10: conflicts(Reader).outConflict←true
11: conflicts(Writer).inConflict←true;
```