Technical Report

Department of Computer Science
and Engineering
University of Minnesota
4-192 Keller Hall
200 Union Street SE
Minneapolis, MN 55455-0159 USA

TR 12-004

Scalable Transaction Management with Serializable Snapshot Isolation on HBase

Vinit Padhye and Anand Tripathi

February 16, 2012

Scalable Transaction Management with Serializable Snapshot Isolation on HBase

Vinit Padhye
University of Minnesota
Minneapolis, MN, USA
padhye@cs.umn.edu

Anand Tripathi
University of Minnesota
Minneapolis, MN, USA
tripathi@cs.umn.edu

ABSTRACT

Key-value based data storage systems such as HBase and Bigtable provide high scalability and availability compared to traditional relational databases. However, unlike relational databases, the existing key-value stores provide only limited transactional functionality, such as single-row transactions. In this paper, we address the problem of building scalable transaction management mechanisms for multi-row ACID transactions on data storage systems such as HBase. To support scalability and availability, the transaction management functions are decoupled from the data storage system. Furthermore, our design does not depend on any central transaction management layer, instead these functions and the related recovery actions are performed in a decentralized and cooperative manner by the application level processes executing the transactions. Our protocol uses snapshot-isolation model as it provides more concurrency. Since the basic snapshot isolation model does not guarantee serializability, our protocol uses a technique based on identifying dependency cycles amongst transactions to avoid serialization anomalies. The protocol for supporting serializability is also performed in a decentralized manner. We demonstrate the scalability and robustness of our approach as well as the correctness of the protocol in ensuring serializable executions of transactions on HBase. We also present and evaluate an alternative approach based on a hybrid model where certain functions, such as conflict detection, are performed by a dedicated service.

1. INTRODUCTION

The emergence of cloud computing platforms has given opportunity for building scalable and highly available services and applications by utilizing the elastic resource pool provided by such platforms. It has been widely recognized that the traditional database systems based on the relational model and SQL do not scale well [9, 10]. The NoSQL databases based on the key-value model such as Bigtable [9] and its open source implementation HBase [4], have been shown to be scalable in large scale applications. These systems achieve scalability through horizontal partitioning of data. However, unlike traditional relational databases the key-value based data storage systems provide only limited transactional functionalities. For example, HBase and Bigtable provide only single-row transactions, whereas systems such as Google Megastore [5] provide transactions over a predefined group of entities. The primary reason for not supporting multi-row transactions in such systems is that it requires distributed synchronization amongs the storage nodes involved in the transaction, thereby limiting the scalability of the system. These two classes of systems, relational and NoSQL based systems, represent two opposite points in scalability versus functionality space. However, many applications such as online shopping stores, online auction services, collaborative editing etc, while requiring high scalability and availability, still need certain transactional guarantees. For example, an online shopping service may require transactional guarantees for performing payment operations.

In this paper, we address the problem of providing multi-row transaction support on systems such as HBase, without compromising scalability properties of such systems. The key aspects of our design are: (1) decoupling of transaction management functions from data storage services, and (2) decentralized transaction management, which does not depend on any central transactional layer. In our design, the transaction management functions are performed by the processes executing the transactions. The general framework of this execution model is shown in Figure 1, where a web service or application is hosted in a cloud environment providing compute and storage services. Clients access the service/application functions over the network. Server processes are created by the front-end servers to perform the requested application-specific functions. These server processes access data stored and managed by the cloud storage services, such as HBase. These processes belong to the trusted domain of the deployed application.

Our approach of decentralized and decoupled transaction management is driven by the goal of eliminating the need of any centralized transaction management component since the scalability and availability of such transaction management component itself becomes a design issue. Towards this goal, our design supports concurrent execution of transaction management functions by the server processes for achieving scalability. Based on this model, we first present and evaluate a fully decentralized approach where all transaction management functions are performed by the server

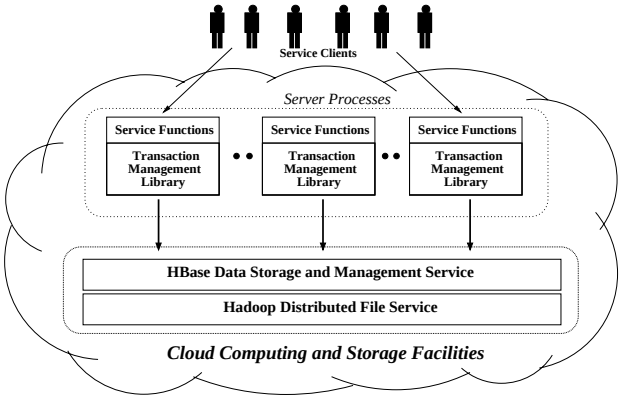


Figure 1: Transaction Management Model

processes through the transaction management library. Then we present an alternative approach based on a hybrid model where certain functions, such as conflict detection, are performed by a dedicated service for better performance. We use the *snapshot isolation (SI)* [6] model of transaction execution. SI is attractive for scalability, as identified in the past [6, 20], since the transactions read from a snapshot, the read operations are never blocked due to write locks, thereby providing more concurrency.

In realizing the transaction management model described above, the following issues need to be addressed. The first issue is related to the correctness and robustness of the decentralized transaction management protocol. In centralized database systems, the commit protocol is executed as an atomic operation, which performs validation and also ensures that all the updates of a transaction are written to the stable storage before committing the transaction. In our approach, such functions are performed by individual server processes in various steps, and the entire sequence of steps is not an atomic action. The transaction management protocol should ensure the transactional consistency when different processes execute the protocol steps concurrently. Furthermore, any of these steps may get interrupted due to process crashes or delayed due to slow execution. To address this problem, the transaction management protocol should support a model of cooperative recovery; any process should be able to complete any partially executed sequence of commit/abort actions on behalf of another process which is suspected to be failed. In such a model, any number of processes may initiate the recovery of a failed transaction, and such concurrent execution of recovery actions by multiple processes should not cause any inconsistencies. The second aspect is related to the snapshot isolation model. The basic snapshot isolation model does not guarantee serializable execution of concurrent transactions as shown in [6, 15]. Specifically, it allows anomalies such as write-skew [6] and read-only anomalies [16]. In the past, researchers have investigated techniques for detecting and preventing such serialization anomalies in SI [15, 29] based on identifying dependency cycles amongst transactions. To ensure serializability, the transaction management protocol should maintain information about dependencies amongst transactions and avoid non-serializable executions by identifying cycles in the dependency graph.

Recently, other researchers have proposed similar techniques for providing multi-row snapshot isolation transactions [28, 31], however, they do not ensure serializability. Moreover, there are several other aspects which motivate further investigation in this area. The technique developed in Google Percolator [28] for its index maintenance system does not address issues related to efficient execution and recovery. For example, in that approach two concurrent and conflicting transactions can potentially abort each other when one could have completed with the abortion of only one. The lazy approach for cleaning the locks enforces the read operations to wait until the locks are released. The work presented in [31] does not adequately address issues related to recovery and robustness when some transaction fails. Moreover, it requires expensive scan operations to identify the latest snapshot and to check for conflicting writes. In regards to the problem of ensuring serializability in snapshot-isolation, prior work in this area has developed various techniques based on static analysis of programs [15] as well as runtime detection of serialization anomalies [7, 8, 29, 18]. However, all these techniques have been developed in the context of traditional RDBMS. In this paper, we develop such techniques for HBase like systems, which pose certain unique challenges. For example, how to maintain read/write set information to track dependencies. In traditional RDBMS such information can be extracted from the lock table. Moreover, in our design the protocol for detecting cycles is also executed concurrently by the transactions.

The major contributions of our work are the following. We present our investigation of a decentralized and decoupled transaction management model on the HBase storage system. We present a transaction execution protocol based on above model which supports snapshot isolation based transaction execution and it also ensures transaction serializability. We evaluate the scalability of this model using the TPC-C benchmark [11] and also using a synthetic benchmark developed primarily to test the correctness of the protocol. We also present and perform comparative evaluation of a hybrid approach of using a dedicated conflict detection service to provide better throughput. We measure the relative cost of supporting serializability in comparison to the traditional snapshot isolation model.

The rest of the paper is organized as follows. In the next section, we discuss the relevant related work. In Section 3 we discuss the snapshot isolation model and techniques for ensuring serializability. In Section 4, we provide the conceptual design of our approach. We provide the detailed discussion of our transaction protocol in Section 5. Section 6 discusses various implementation issues and approaches of performance optimization. Section 7 present the correctness and scalability evaluations of our approach. The conclusions are presented in the last section.

2. RELATED WORK

For transaction management in the distributed databases, various techniques for distributed transactions using *two-phase-commit* [26] and group communication protocols [19, 25] have been proposed. In recent years, researchers have recognized the scalability limitations of relational databases, and to address the scalability and availability requirements, various systems based on key-value data model have been developed [9, 14, 3, 5]. Systems such as SimpleDB [3] and Cassandra [23] provide weaker consistency. Bigtable [9] and

its open-source counterpart HBase [4] provide strong data consistency but provide only single-row transactions. Other systems provide transactions over multiple rows with certain constraints. For example, Megastore [5], and G-store [12] provide transactions over a group of entities. ElasTraS [13] supports multi-row ACID transactions only over a single database partition and provides a restricted *minitransaction* semantics [2] over multiple partitions. The various approaches for transaction processing in cloud storage system are discussed in [21]. CloudTPS [30] provides a design based on a replicated transaction management layer which provides ACID transactions over multiple partitions. In the Deuteronomy system [27, 24], an approach based on decoupling transaction management from data storage using a central transaction component is proposed.

In contrast to these approaches, our design presents a decentralized transaction management protocol to eliminate the need for a central transaction management layer. In the designs based on a central transaction management layer, such as CloudTPS or Deuteronomy, the scalability and availability of this layer itself becomes a design issue. To support scalability of the transaction management layer, CloudTPS partitions the data across transaction managers (referred as LTM in their design). It assumes transactions to be short-lived and access only a small number of well-defined set of items, and therefore a transaction would access only a small number of LTMs. Our design does not make any assumptions about the workload characteristics; a transaction may read and write any arbitrary set of data items. Other researchers have presented techniques based on the decentralized transaction management approach, such as in Google's Percolator system [28] and [31]. However, as discussed earlier further investigations are needed in this area. Our approach is conceptually similar to the approach presented in Percolator system [28]. However, our protocol also ensures serializable execution and addresses issues such as livelocks which can occur in Percolator.

The problem of transaction serializability in snapshot-isolation model is also extensively studied [15, 7, 8, 29, 18]. The work in [15] characterizes the conditions necessary for non-serializable transaction executions in the SI model. Based on this theory, many approaches have been suggested to avoid serialization anomalies in SI. These approaches include static analysis of programs [17] as well as runtime detection of anomalies [7, 8, 29, 18]. Technique presented in [7, 8, 18] tend to be pessimistic and can lead to unnecessary aborts. PSSI approach [29] avoids such problems and aborts only the transactions that lead to serialization anomalies. However, these approaches were developed in the context of traditional relational databases and, except in the case of [18], provided solutions for centralized databases.

3. SERIALIZABLE SI TRANSACTIONS

Snapshot isolation (SI) based transaction execution model is a multi-version based approach utilizing the optimistic concurrency control concepts [22]. When a transaction T_i commits, it is assigned a commit timestamp TS_c^i, which is larger than the previously assigned timestamp values. The commit timestamps of transactions reflect the logical order of their commit points. When a transaction T_i commits, for each data item modified by it, a new version is created with the timestamp value equal to TS_c^i. When a transaction T_i's execution starts, it obtains the timestamp of the most recently committed transaction. This represents the *snapshot timestamp* TS_s^i of the transaction, and a read operation by the transaction returns the most recent committed version up to this snapshot timestamp. Thus a transaction reads only the committed data items and never gets blocked due to any write locks. Two transactions T_i and T_j are concurrent if, and only if, $(TS_s^i < TS_c^j) \wedge (TS_s^j < TS_c^i)$ A transaction T_i commits only if none of the items in its write-set have been modified by any committed transaction T_j such that $TS_s^i < TS_c^j < TS_c^i$.

If two concurrent transactions modify the same data, then only one would be able to commit. It is possible that a data item in the read-set of a transaction is modified by another concurrent transaction, and both are able to commit. An *anti-dependency* [1] between two transactions T_i and T_j is a *read-write (rw) dependency*, denoted by $T_i \overset{rw}{\rightarrow} T_j$, implying that some item in the read-set of T_i is modified by T_j. This is the only kind of dependency that can arise between two concurrent transactions, and it is possible this to be mutual between them, implying a cycle. There are other kinds of dependencies, namely *write-read (wr)* and *write-write (ww)*, that can exist between two non-concurrent transactions. Snapshot isolation based transaction execution can lead to non-serializable executions as shown in [6, 15]. It was shown in [1] that a cycle of dependencies, including at least one anti-dependency edge, among a set of transactions executing under SI represents a non serializable execution. Fekete et al. have shown that a non-serializable execution must always involve a cycle in which there are two consecutive *anti-dependency* edges of the form $T_i \overset{rw}{\rightarrow} T_j \overset{rw}{\rightarrow} T_k$. In such situations, there exists a *pivot* transaction with both incoming and outgoing rw dependencies. In the above example, T_j is the pivot transaction. Several techniques have been developed utilizing this fact to ensure serializable transaction execution. Some of these schemes abort transactions whenever two consecutive rw dependencies are observed, but such abortions may be unnecessary as not all such situations necessarily result in a cycle.

Our approach for ensuring serializable transactions under snapshot isolation requires maintaining the dependency graph for the transactions. In this graph, transactions are represented by nodes and the edges represent the dependencies between them. If during the commit operation of a transaction we detect a cycle, we abort that transaction. This simple approach requires several issues to be further examined. First, we show here that even after a transaction commits, we need to maintain some information about it in the dependency graph until some point in the future. Such committed transactions are referred to as *zombies* in [29]. Also, for efficient execution the dependency graph should be kept as small as possible by frequent pruning to remove any unnecessary transactions that can never lead to any cycle in future.

The read-set and the write-set of a transaction are precisely known only at its commit time, this is because a transaction may determine which items to read or write during its execution, which may take any arbitrary time. At a transaction's commit time, all dependencies with previously committed transactions are precisely known. At the commit time, we may detect any ww, wr, and rw dependencies for this transaction in relation to any of the previously committed transactions. It can have only wr, ww and rw incoming

dependencies with a previously committed non-concurrent transaction. With respect to a committed concurrent transaction, it can have only rw dependencies, possibly in both directions. However, any rw dependencies to or from any of the currently active transactions cannot be precisely known until that transaction commits. We illustrate below these concepts through two examples.

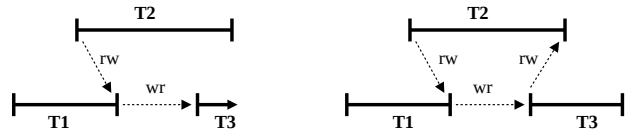
Figure 2(a) shows three transactions, two of which, T1 and T2, are committed. At the time when T2 commits, T3 is active. The *rw-edge* from T2 to T1, and *wr-edge* from T1 to T3 are known at this time. We need to keep this information in the dependency graph because when T3 commits an rw dependency from T3 to T2 is found, as shown in Figure 2(b). At this time a cycle is formed, indicating a non-serializable execution. This leads us to conclude that when a transaction T_i commits, its node and related edges should still be retained in the dependency graph if there are any other transactions active, i.e. concurrent and not yet reached their commit points, at T_i's commit time. Moreover, all transactions reachable from the committing transaction, such as T1 in this example, should also be retained in the graph. Figure 3 shows an example illustrating that sometimes rw, ww, and wr dependencies for a transaction may become known only at its commit time. In this example, when T3 commits, an rw dependency to T2, and ww and wr dependencies from T1 are discovered. This situation results in a cycle.

Based on the above observations, we maintain the dependency graph using the following rules. When a transaction T_i commits its dependency edges with other transactions are inserted as follows:

1. For every active transaction T_j such that $TS_s^j < TS_c^i$, an undirected edge is inserted. When such a transaction T_j commits this undirected edge is removed and appropriate rw dependency edges are inserted.

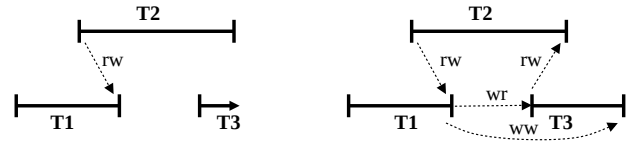
2. For any previously committed transaction T_j an outgoing edge is inserted if there is any dependency $T_i \rightarrow T_j$. Similarly an incoming edge in inserted if there is a dependency $T_j \rightarrow T_i$.

In order to prune the dependency graph, we remove the unnecessary transactions using the following rule: A committed transaction is removed if it is not reachable from any currently active transaction through undirected or directed edges. Such an unreachable transaction can not be part of any future cycle. This is because, since the transaction is unreachable it is not concurrent with any active transaction. Therefore, for such a transaction the only new dependencies that can arise in future are of outgoing ww and wr types. Since such a transaction does not have any incoming edge, it can not become part of any future cycle due to any new dependencies.



(a) T3 is active when T2 commits (b) After T3 commits

Figure 2: Potential dependency with a committed concurrent transaction



(a) T3 is active when T2 commits (b) After T3 commits

Figure 3: Potential dependency with a committed non-concurrent transaction

4. CONCEPTUAL DESIGN

As discussed earlier, our goal is to develop a transaction management model which decouples the transaction management functions from the storage service. In this section, we discuss the various design issues in realizing this model.

4.1 HBase Overview

HBase is a key-value based distributed data storage system based on the Bigtable [9] data model. In HBase, data is stored in the form of HBase tables composed of rows and columns. Columns are grouped into *column-families*. A read or write operation is performed on a row using the row-key and one or more column-keys. HBase also maintains different versions of data by maintaining multiple versions for each cell indexed by a timestamp. We use this multi-version system provided by HBase to support snapshot based data access. In our model, each committed data version is associated with a timestamp, which is the commit timestamp of the transaction that created the version. Update operations on a single row are atomic, i.e. concurrent writes on a single row are serialized. Additionally, HBase provides row-locks to atomically perform multiple writes and reads on a single row. Any update performed is immediately visible to any subsequent reads, i.e. HBase provides strong consistency of data. In our protocol, for operations that require atomicity we use the single-row level atomic transactions such as increments and conditional update APIs provided by HBase.

4.2 Design Overview

Since the transaction management protocol would be performed by the server processes, the following are the important goals in designing this protocol: (1) Support high concurrency and minimize the contention in executing the protocol steps, thereby increasing the scalability of transaction execution; (2) Minimize the protocol overhead and ensure that the time required for commit is bounded; (3) Support cooperative recovery, i.e. any process can perform recovery of a failed or a stalled transaction. These goals would affect the various design choices as described below.

Timestamps Management: In a centralized implementation of the transaction protocol, all the steps in the commit protocol are executed atomically, which include acquiring the timestamp, performing validation, and writing updates to the stable storage. In our model these steps in the commit protocol are executed concurrently by the application processes, and the goal of scalability demands that these steps should not be implemented as a single critical section. Not performing all steps of the commit protocol as one critical section raises a number of issues. There can be situations where a transaction has acquired the commit timestamp but its status is not yet resolved as committed/aborted. There-

fore, we need to maintain two timestamps: GTS (global timestamp) which is the latest commit timestamp assigned to a transaction, and STS (stable timestamp), $STS \leq GTS$, which is the largest timestamp such that all transactions with commit timestamp up to this value are either committed or aborted. When a new transaction is started, it uses the current STS value as its snapshot timestamp. In the absence of such a counter, the burden of finding the latest snapshot would be on each transaction process.

Eager Updates vs Lazy Updates: One of the design choices that we face is when should a transaction write its updates to the HBase storage. In the eager update approach, a transaction would write its updates to HBase during its execution phase, whereas in the lazy approach all writes would be performed during the execution of the commit protocol. It should be noted that the commit protocol can only be executed after the transaction acquires its commit timestamp. In the lazy update approach, the execution of the commit protocol can take arbitrary long time based on the size of the data-items to be written. A long commit phase of a transaction would potentially delay the commit decisions of other concurrent and conflicting transactions that have higher commit timestamps, affecting the transaction throughput and scalability. In the eager update approach the data is flushed to HBase prior to the commit protocol execution, thereby reducing the execution time for the commit protocol. Also, the transactions can perform their writes in overlapped manner during the execution phase. The eager update policy is important because we do not make any assumptions about the size of the write-set of the transactions. Also it facilitates the roll-forward of a transaction that fails during its commit, since its updates would be already present in the HBase storage. However, implementing eager update policy raises issues related to maintaining uncommitted versions in HBase. For such data versions, we can not use the transaction's commit timestamp as version ids because it is not known at the time of writing. Therefore, in the commit protocol these data versions need to be mapped to the transaction's commit timestamp. Also, ensuring the isolation property requires that such uncommitted versions should not be visible until the transaction commits.

Detection of Conflicts for SI Transactions: The SI model requires detection of *ww* conflicts among concurrent transactions, and a method to resolve conflicts by allowing only one of the conflicting transactions to commit. In order to detect conflicts in the decentralized execution model, there are two possible options for maintaining the write-set information. One way is to maintain this information on per transaction basis in the stable storage. Another way is to maintain for each item the id of the transaction which updates it. The first approach is more suitable for the *first-committer-wins (FCW)* rule for conflict detection and resolution. In this rule, a transaction commits only if no other concurrent transaction with a smaller commit timestamp conflicts with it. With this rule conflict checking can only be performed by a transaction after acquiring its commit timestamp. This enforces a sequential ordering on conflict checking based on the commit timestamps and it would force a younger transaction to wait for the progress of all the older transactions, thereby limiting concurrency. In contrast, the *first-updater-wins (FUW)* approach can be easily supported by maintaining for each item the updater transaction's id. With this approach, among the conflicting transactions, the one that records this information first is considered the winner. The FUW approach appears more desirable because the conflict detection and resolution can be performed before acquiring the commit timestamp, thus reducing the execution time of the commit protocol.

tid	TSs	TSc	write-set	out-edge	status
tid_i	*t1*	*t2*	list of *row-ids*	1→ out-edge-tid1 2→ out-edge-tid2	Commit Complete
tid_j					
tid_k					

Figure 4: DSGtable Structure

Detection of Conflicts for Serializable SI Transactions: For ensuring serializability, we need to track the data dependencies among the transactions and maintain the *dependency serialization graph* (DSG) [15], in the shared storage so that transactions can concurrently update and access it for detecting cycles. For detecting dependencies, in addition to maintaining the write-sets for SI transactions, as noted above, we also need to record for every transaction its read-set. One way is to maintain in the shared storage this information for each data item, i.e. for every data item version, maintain the list of the transactions which read that version. Based on this information, during the commit protocol execution, a transaction needs to detect such dependencies, record this information in a dependency graph structure in the shared storage, and then execute the cycle checking protocol. For pruning the DSG, we use a separate pool of 'cleanup' processes to remove unnecessary transactions.

5. PROTOCOL DESCRIPTION

In this section we provide detailed description of the protocol for snapshot-based transaction execution and cycle-checking protocol which guarantees the serializability. The protocol also ensures that the transaction commits atomically, i.e. the changes made by a transaction are made visible atomically and only after the transaction commits.

5.1 Tables and Data Structures

We maintain the following information for each transaction in the system: transaction-id (tid), snapshot timestamp (TS_s), commit timestamps TS_c, write-set, current status, and dependency information. This information is maintained in a table named *DSGtable*. The structure of DSGtable is given in Figure 4. In this table, tid is the row-key of the table and other items are maintained as columns. For a given transaction, its dependency information is maintained in the form of a list of $tids$ of transactions that have an incoming dependency edge from the given transaction. Elements in this list are maintained as multiple versions of the column 'out-edge', instead of maintaining a single vector data item, to support concurrent insertions.

row key	data related columns	committed version	readers	lock
row1	*tid1* → <*object data*> *tid2* → <*object data*> *tid3* → <*object data*>	*ts1* → <*tid2*> *ts2* → <*tid3*> *ts3* → <*tid1*>	*ts1* → <[..]> *ts2* → <[..]> *ts3* → <[..]>	(only one version per row) <*owner-tid*>
row2				
row3				

Figure 5: StorageTable Structure

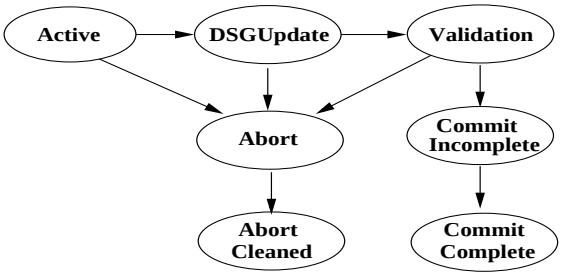


Figure 6: Transaction Protocol Steps

For each application-specific table, hereby referred as *StorageTable*, we maintain the information related to committed versions and write locks as shown in Figure 5. A StorageTable may have various application specific columns which contain the application data. A transaction writes a new version of a data item with the version timestamp as its *tid*, and on commit it maps the new version with its TS_c by writing its *tid* in the *committed-version* column with version timestamp as TS_c. For example, in Figure 5 'row1' contains three committed versions created by transaction with tids *tid*1, *tid*2, and *tid*3 with commit timestamps *ts*3, *ts*1, and *ts*2, respectively. The column named *readers* is used to store, for every version, the list of tids of transactions that read that particular version. This information is maintained to detect *rw* dependencies. The *lock* column maintains one version per row and is used by the update transactions to acquire write locks on the rows for FUW rule. When unlocked, the lock column is set to -1. To acquire the lock on a row, an update transaction atomically writes its *tid* to the lock column provided that it is not already locked, i.e. it is set to -1. A transaction may access multiple StorageTables during its execution. Additionally, a *CommitLogTable* is maintained as $< TS_c$, *tid*, *status* $>$ with TS_c as row-key to record the commit status.

5.2 Transaction Protocol

A transaction T_i goes through a series of steps during its execution as shown in Figure 6. The transaction starts by acquiring *tid* and snapshot timestamp TS_s^i. In the *Active* phase it performs reads and writes of data items in the *StorageTable* in HBase. When it reaches the commit point it enters the *DSGupdate* phase and starts the commit protocol. In the DSGupdate phase, the transaction checks for *ww* conflicts and if there is no conflict it finds the dependencies with other transactions. In the *Validation* phase, it checks for any dependency cycles with other transactions. If there is no cycle then the transaction commits and enters *CommitIncomplete* phase. In this phase, the transactions commits its versions by entering its $TS_c \rightarrow tid$ mappings in the *committed version* column in the StorageTable. After the *CommitIncomplete* phase is completed, the transaction is *CommitComplete* and it advances the STS value. During the start of every phase, the transaction updates its status in *DSGtable* to indicate its progress. The status change operation is performed atomically and conditionally, i.e. permitting only the state transitions shown in Figure 6. Whenever a transaction T_i waits for the resolution of any other transaction T_j upon a conflict, it periodically checks its status and if the status is not changed within some specified timeout, T_i suspects T_j has failed and marks T_j as aborted, provided that T_j has not reached *CommitIncomplete* phase, and then T_i proceeds to the next step in the commit protocol. If T_j has reached *CommitIncomplete* phase then it is immediately rolled-forward by completing its *CommitIncomplete* phase. The roll-back of a transaction marked as aborted, i.e. when it has not reached *CommitIncomplete* phase, is performed lazily since it does not cause any interference.

Protocol 1 Notations used in protocol description

tid_i : tid of transaction T_i
W_i: set of all item-ids written by transaction T_i
R_i: set of all items-ids read by transaction T_i
D_i: set of dependency edges $T_m \rightarrow T_n$ s.t $i = m \vee i = n$
L_i: set of all item-ids for which locks are acquired

Protocol 2 Transaction begin protocol executed by T_i

1: $tid_i \leftarrow$ get a unique tid value
2: $TS_s^i \leftarrow STS$
3: insert(DSGtable, $<tid_i, TS_s^i, status = Active>$

Protocol 3 Read and Write protocol executed by T_i

Read:
1: **if** *item-id* is present in write-set **then**
2: *item* ← read from local buffer
3: **else**
4: $tid_R \leftarrow$ read value of the latest version of "committed version" in the range $[0,TS_s^i]$) from StorageTable
5: *item* ← read data item with version tid_R
6: append tid_i to the list of readers for this version
7: $R_i \leftarrow R_i \cup$ *item-id*
8: **end if**

Write:
1: write *item* to StorageTable with version tid_i
2: $W_i \leftarrow W_i \cup$ *item-id*

Start phase: In the *Start* phase, the transaction executes the begin protocol shown in Protocol 2. When a new transaction T_i is started, it obtains its *tid* and TS_s. The transaction then inserts in DSGtable an entry: $<tid, TS_s, status = Active>$ and proceeds to Active phase.

Active phase: During the Active phase, a transaction performs read and write operations as shown in Protocol 3. For a write operation, following the eager update approach,

the transaction creates a new version in the StorageTable using *tid* as the version timestamp. The transaction also maintains its own writes in a local buffer to support *read-your-own-writes* consistency. A read operation for the data items not contained in the write-set is performed by first obtaining, for the given data item, the latest version of the *committed version* column in the range $[0, TS_s]$ in the corresponding StorageTable. This gives the *tid* of the transaction that wrote the latest version of the data item according to the snapshot. The transaction then reads data specific columns using this *tid* as the version timestamp. The transaction appends its *tid* to the list maintained in the *readers* column of the corresponding version.

Protocol 4 DSGupdate phase executed by transaction T_i

1: update status to *DSGupdate* in DSGtable provided *status* = *Active*
2: insert W_i in *DSGtable*
3: **for all** *item-id* $\in W_i$ **do**
4: atomic-read [*writer* $\leftarrow$ get *tid* of the writer of the version x for *item-id* with timestamp $> T_s^i$; *lock_owner* $\leftarrow$ get *tid* of the lock owner for *item-id*]
5: **if** *writer* $\neq$ *null* **then** abort()
6: **if** *lock_owner* = *null* **then**
7: *locked* $\leftarrow$ acquire lock on *item-id*
8: **if** *locked* **then** $L_i \leftarrow L_i \cup$ *item-id* **else** go to line line 10
9: **else**
10: *lock_owner* $\leftarrow$ get *tid* of the lock owner for *item-id*
11: **if** *lock_owner* $< tid_i$ **then** abort()
12: **else** wait for commit status of *lock_owner*, if it commits then abort, else go to line 4
13: **end if**
14: **end for**
15: **for all** *item-id* $\in R_i$ **do**
16: $T_{next} \leftarrow$ get *tid* of the writer of the immediately following version for *item-id*
17: $D_i \leftarrow D_i \cup \{T_i \overset{rw}{\rightarrow} T_{next}\}$
18: $T_{prev} \leftarrow$ get *tid* of the writer of the version of *item-id* read by T_i
19: $D_i \leftarrow D_i \cup \{T_{prev} \overset{wr}{\rightarrow} T_i\}$
20: **end for**
21: **for all** *item-id* $\in W_i$ **do**
22: set $S_r \leftarrow$ get *tids* of the reader(s) of the immediately preceding version of *item-id*
23: **for** $T_r \in S_r$ **do** $D_i \leftarrow D_i \cup \{T_r \overset{rw}{\rightarrow} T_i\}$ **end for**
24: $T_w \leftarrow$ get *tid* of the writer of the the immediately preceding version of *item-id*
25: $D_i \leftarrow D_i \cup \{T_w \overset{ww}{\rightarrow} T_i\}$
26: **end for**
27: insert D_i in *DSGtable*

DSGupdate phase: The protocol for DSGupdate phase is shown in Protocol 4. The transaction updates its status and inserts W_i in *DSGtable* for recovery purpose. If the transaction has modified one or more items then it first checks for *ww* conflicts (line 3-14). To ensure the FUW rule, the transaction sequentially checks for each data item in its write-set if either a newer version (i.e. with timestamp greater than TS_s^i) is created or a lock is present. A newer version indicates a *ww* conflict with a committed transaction, hence the transaction in the DSGupdate phase aborts immediately. The check for newer version and presence of the lock is performed as an atomic read operation to ensure correctness when some other transaction concurrently installs its version and releases the lock. If no new version and no lock is present, then it tries to acquire the lock on the item. If some transaction has already acquired the lock then it performs the following check to avoid deadlocks and livelocks: if the lock is owned by a transaction with a smaller *tid* then it aborts, otherwise it waits (line 12). If the conflicting transaction commits then it aborts, otherwise it continues execution from line 4. If there is no conflict then the transaction performs dependency check (line 15-27). To find *wr* dependencies, the transaction finds, for every item version in its read-set, the the *tid* of the transaction which wrote that version. Similarly, to derive outgoing *rw* anti-dependencies, it finds the *tid* of the transaction which either wrote the immediately following version for that data item or is holding lock on that data item, if any. For every item in the write-set of the transaction, it finds the *tids* of the reader(s) and writer of the immediately preceding version. This gives the incoming *rw* edge(s) and the *ww* edge. The dependency edges are then inserted in the DSGtable. Since our purpose is to find the directed cycles irrespective of the type of edges, we only insert an outgoing edge $T1 \rightarrow T2$, if there is at least one edge of type *rw*, *wr* or *ww* from T1 to T2.

Validation phase: The transaction first changes its status to Validation in DSGtable, and then acquires its TS_c . It then checks for any cycle including itself in DSG, by performing a depth-first search starting at its node. In the search, it considers only the transactions which are in Validation, CommitComplete, or CommintIncomplete phase. It ignores any transaction with larger TS_c, encountered during the search. If it detects a cycle and all the transactions involved in the cycle are already committed, then it aborts. The cycle checking is performed concurrently and in non-blocking manner by the transactions. Due to the ordering based on TS_c when two concurrent transactions are involved in the same cycle, only one of them would abort. If the transaction detects a cycle with one or more transactions still in the *Validation* phase then it waits for their status to resolve. If a transaction does not detect a cycle, or if any transaction involved in the cycle aborts, then it proceeds further.

Protocol 5 Validation phase executed by transaction T_i

1: update status to $Validation$ in DSGtable provided *status* = *DSGupdate*
2: $TS_c^i \leftarrow$ get GTS timestamp
3: update TS_c^i in DSGtable
4: check for cycle with all T_j s.t. $TS_c^j < TS_c^i \wedge status(T_j) \in \{Validation, CommitIncomplete, CommitComplete\}$
5: **if** there is a cycle with all committed transactions **then** abort()
6: **while** there is a cycle with any transactions in $Validation$ stage **do**
7: wait for their commit status, if all of them commit then abort
8: **end while**

Commit Incomplete phase: Once the transaction passes the validation phase and decides to commit, it atomically changes its status to CommitIncomplete provided it is not already aborted. Once the transaction updates its status to CommitIncomplete, any failure after that point would result

in roll-forward of the transaction. The transaction now inserts the $ts \rightarrow tid$ mappings in the StorageTables for all the data items in its write-set. The transaction now changes its status to *CommitComplete*. At this point the transaction is committed. It then advances the STS counter to its TS_c provided there are no gaps, i.e. status of all the older transactions are resolved as committed/aborted. After this point the updates made by the transaction would be visible to any subsequent transaction.

Protocol 6 CommitIncomplete phase executed by transaction T_i

```
1: update status to CommitIncomplete in DSGtable provided status = Validation
2: for all item-id ∈ W_i do
3:     insert TS_c^i → tid_i mapping in StorageTable
4:     release lock on item-id
5: end for
6: update status to CommitComplete in DSGtable
7: advanceSTS()
```

Abort phase: A transaction may either abort itself due to conflicts with concurrent and committed transactions or it can be aborted by some other transaction due to failure suspicions. After a transaction is aborted it advances the STS and performs the cleanup such as releasing the locks, deleting the data item versions etc.

Protocol 7 Abort protocol for transaction T_i

```
1: Abort:
1: update status to Abort in DSGtable
2: advanceSTS()
3: for all item-id ∈ W_i do
4:     delete created version of item-id
5:     if item-id ∈ L_i then release lock on item-id
6: end for
```

5.3 Correctness

The correctness of the decentralized transaction management protocol described above is proved with respect to two aspects: the correctness of the commit protocol in ensuring ACID properties with respect to the snapshot isolation model and the correctness of the cycle-checking protocol to ensure serializability.

Atomicity: To prove the atomicity property, we prove that the following two properties hold:
(A) The status of a transaction from "uncommitted" to "committed" is changed atomically.
(B) Updates made by any transactions are not visible until the transaction is "committed".
In our protocol a transaction is committed only when it changes its status to *CommitComplete* which is performed as a single-row atomic operation. So property (A) is satisfied. Any updates made by an uncommitted transaction, though actually written to the database, are not visible until the transaction inserts the $ts \rightarrow tid$ mapping in the table. Although this process is not atomic, any subsequent transaction would only see these updates when the STS counter is advanced, which only happens after the transaction has inserted all the mappings and is *CommitComplete*. In case of crashes of server processes, if the process crashed before the transaction reached the *CommitIncomplete* stage, then the transaction would not have entered its mappings and hence its updates are not visible to any transaction. In case of crashes during *CommitIncomplete*, the transaction would be always rolled-forward to CommitComplete. The process of advancing STS ensures that all the transactions up to that timestamps are stable, i.e. committed or aborted. And hence no transaction would see the updates of any uncommitted transaction, therefore property (B) is also satisfied.
Consistency: As long as the atomicity property is satisfied, which is proved above, and transactions do not write inconsistent data, the *consistency* property is satisfied. Consistency with respect to serializability is proved below.
Isolation: As proved above, the transaction reads from a consistent snapshot and updates made by any transactions do not affect the read operations of any concurrent transaction. The updates of each transaction are versioned with its *tid*, so updates made by two concurrent transactions are stored as different versions, without overwriting each others update. The *ww* conflict checking procedure in the *DSGupdate* phase described above (Protocol 4) ensures that only one of the two (or more) concurrent and conflicting transaction commits. Thus *isolation* property is satisfied.
Durability: HBase provides the guarantee that any updates made are persistent. Since all the updates made by a transaction are always written to the HBase before committing, the *durability* property is also satisfied.
Serializability: As discussed in Section 3, serializability is ensured if there are no cycles amongst the committed transactions. Here, we prove that the cycle checking protocol ensures serializability under concurrent executions by multiple transactions. With regards to detecting dependencies, we prove the following property: If there is a dependency between two transactions $T1 \rightarrow T2$, then it is detected by at least one transaction. Note that a transaction always marks its read and write sets before performing the dependency check. Suppose there is dependency $T1 \rightarrow T2$ of type of *wr* or *ww*, then it means T1 has committed before T2 started. So this dependency will be detected by T2 during its *DSGupdate* phase, as the version created by T1 must be present at that time. If the dependency is anti-dependency, then T1 and T2 are concurrent. If T1 performs its dependency check for the conflicting item after T2 inserted its mapping or acquired the lock on that data item then T1 will detect this dependency. If T1 performs the check before, then it must have already marked its reads in the StorageTable before T2 proceeds towards its dependency check, thus T2 will detect the dependency. Thus if there is a dependency of any type, then at least one of the two transactions will detect it. To prove the correctness of cycle-checking protocol, we prove the following properties:
(C) If a cycle is present, at least one transaction will detect it.
(D) If a cycle is present, at least one transaction involved in the cycle will abort.

Consider a cycle of a set of transactions and let T_j be the transaction in this cycle with the largest commit timestamp. According to our protocol, since a transaction ignores edges with any younger transaction, this cycle can only be detected by T_j. Since the detection and insertion of dependency information (Protocol 4 line 16-25) is performed before obtaining the commit timestamp and, as proved above, a dependency is detected by at least one transaction, we can

conclude that at the time T_j performs its *Validation* phase, all the dependencies in the cycle are inserted. Hence T_j will detect the cycle. Thus property (C) is satisfied. If the commit status of one or more transaction is not known, then T_j will wait. If any unresolved transactions abort, then property (D) is satisfied. If all of them commits, then T_j will abort. If any of the transactions fails during *Validation* phase, then it would be aborted. Thus properties (C) and (D) are satisfied which indicates that if a cycle is present, at least one transaction would abort thereby breaking the cycle and ensuring serializability.

6. IMPLEMENTATION ISSUES AND OPTIMIZATIONS

In this section we discuss various implementation issues and alternative implementation approaches that we evaluated for achieving better performance.

6.1 Timestamps and Transaction Id

Our first approach for storing and managing timestamps, i.e. STS and GTS, and *transaction id* values was based on using a HBase table to store these counter values. Transactions would use the single row-level read-and-modify atomic operations provided by HBase to atomically increment a counter value. The rationale behind this approach was to leverage the reliability provided by HBase for storage and management of these counters. However, in our experiments we realized that read and increment operations on these counter values using HBase are expensive and it limits the maximum achievable throughput. Therefore, we implemented a separate timestamp service for issuing timestamps. Requesting timestamp values from such a service is relatively lightweight operation compared to the approach of using a HBase table and such a service can serve a significantly large number of requests. With respect to allocating transaction ids, we faced a load balancing problem. The *DSGtable* uses transaction id as the key, and in HBase a table is split among the HBase nodes based on sequential range of keys. Thus, at any given time, the concurrently running transactions would access only one or few HBase nodes making them the bottleneck. Therefore, we adopted the approach of allocating random but unique transaction ids to achieve proper load balancing. To implement this, a transaction, in its begin protocol, would contact the timestamp service which would issue a random but unique id. An alternative approach could be that a transaction would assume a random id and check in the *DSGtable* for existence of this id. However, we use timestamp service for better performance. Ideally, the STS, GTS and transaction ids could be maintained by using separate timestamp servers for each one as these values are not dependent on each other, however, in our experiments we observed that a single timestamp server could handle a large transaction workload.

6.2 Conflict Detection Service

Section 5 presents a protocol in which the conflict detection, i.e. checking for *ww* conflicts and serialization conflicts, is performed by the server processes themselves. The conflict detection is basically a sequence of operations performed by the server processes using the transactional metadata, i.e. read and write sets of transactions, which is stored in HBase. This induces additional overhead due to the load induced by the read and write operations to HBase. As observed in our experimental evaluations, which is further discussed in Section 7, these overheads can cause increase in transaction completion time. Therefore, for better performance in terms of transactional latency and throughput, we evaluated an approach of using a dedicated conflict detection service. In the discussion that follows, we would refer to the approach of using such a service as *service-based approach* and the approach presented in Section 5 as *fully decentralized approach*. In the service-based approach, a conflict detection service would maintain information about read and write sets information of transactions and would detect conflicts among concurrent transactions. Note that, this dedicated service is only for the purpose of conflict detection and not for the entire transaction management, as done in [30, 24]. The other transaction management functions, such as getting the appropriate snapshot, maintaining uncommitted versions, and ensuring the atomicity and durability of updates when a transaction commits are performed by the application level processes. The transaction protocol using this approach is presented below in Protocol 8. In the commit protocol, a transaction would send a request to this service along with its snapshot timestamp, commit timestamp and read-write set information, i.e data item identifiers, to check if it conflicts with any previously committed transaction.

Protocol 8 Transaction protocol executed by T_i

```
1: execute begin transaction protocol
2: execute Active phase protocol
3: insert W_i in DSGtable
4: TS_c^i ← obtain GTS value from timestamp service
5: send request to conflict detection service with TS_c^i, TS_s^i,
   W_i, R_i
6: if response = commit then
7:    execute CommitIncomplete phase protocol
8: else
9:    execute Abort protocol
10: end if
```

The conflict detection service would first check if any data item in the transaction's write-set is modified by any committed concurrent transaction. If not, then based on the transaction's read and write sets, it would derive the dependencies with already committed transactions and check if the transaction is going to create any cycle or not. If transaction has no write-write conflict and it does not create any dependency cycle then the service would send the response as 'commit' otherwise it would send 'abort'. Before sending the response, it logs the transaction read-write set information and the commit status in the HBase. This write-ahead logging is performed for recovery purpose. The scalability and reliability of this service are important aspect to consider. The service would maintain the information required for conflict detection, i.e the read-write set information and DSG graph structure, in memory for better performance. This information is *soft-state* and can be recovered upon failure from the information logged in HBase. For scalability and availability, the service can be implemented using a group of servers and the data items can be partitioned across servers. The requests for conflict detection can be distributed across servers based on the read-write sets of transactions. If a transaction's read-write sets span across multiple servers then a protocol similar to the two-phase commit

can be used to detect if transaction conflicts with any previously committed transaction. For pruning the DSG, each conflict detection server would perform pruning of the subset of the DSG maintained locally by the server. However, we observe that the scaling requirement of this service would be significantly moderate compared to the scaling requirement of storage service as the workload and request processing requirements of this service are significantly lower compared to the transactional workload. The service would receive only one request per transaction and would only need to access the in-memory data structures for conflict detection. We observe that a single conflict detection server is sufficient for handling a large transactional workload as discussed in the next section.

7. EVALUATIONS

In this section, we present the correctness and scalability evaluations of the proposed approach. In correctness evaluation, our goal is to validate the correctness of decentralized transaction execution protocol in guaranteeing serializable ACID transactions. For scalability evaluations, we are primarily interested in scalability of transaction throughput through scale-out model. We perform comparative evaluation of the service-based approach and fully decentralized approach. We also evaluate the cost of ensuring serializability compared to basic snapshot isolation.

7.1 Correctness

To evaluate the correctness, we created a synthetic benchmark to generate serialization anomalies such as write skew and read-only anomalies. The benchmark simulates a joint account withdrawal process. It includes three type of transactions: deposit transactions, withdraw transactions, and balance inquiry transactions. A joint account includes two accounts. The balance of an individual account can be negative as long as total balance of the joint account is non-negative. A withdraw transaction randomly selects n joint accounts for withdrawal. For each selected joint account, it reads the balance values of the individual accounts. If the total balance is greater than 0, then it withdraws the total amount from one of the two accounts which is selected randomly. Similarly, a deposit transaction randomly selects n joint accounts and for each selected joint account it deposits some amount into one of the accounts randomly. A balance inquiry transaction randomly selects k joint accounts and reads the total balance of each joint account. Under serializable execution, the total balance of any joint account should be non-negative. Thus if a balance inquiry transaction reads a negative total balance, then it indicates serialization anomaly. Additionally, at the end of the benchmark run we check the total balance of each joint account to ensure correctness. We used this synthetic benchmark to validate correctness of our protocol.

7.2 Scalability Evaluation

For scalability and performance evaluations we used TPC-C benchmark [11] to perform evaluations under a more realistic workload. However, our implementation of TPC-C specifications differs in following ways. Since our primary purpose is to measure the transactional throughput we did not emulate terminal I/O. Also, to generate more load, we did not induce wait times in between transactions. HBase does not provide relational database features such as foreign keys and secondary indexes. For secondary indexes, we created another index table, and for composite primary keys, such as in the case of ORDER table, we created the row-keys by concatenating the specified primary keys. Moreover, since the transactions specified in TPC-C benchmark do not create serialization anomalies under SI, as observed in [15], we implemented the modifications suggested in [8], which basically adds one more transaction type 'CreditCheck' to create serialization anomalies.

The primary criterion used in these evaluations is the maximum transactional throughput achieved, measured in terms of committed transactions per minute (tpmC), under a given configuration. We evaluated the scalability in terms of achieving increased throughput by scaling-out the system using a 30 node cluster. Figure 7 shows the throughput scalability under scaling-out of the system with 1, 2, 6, 10, and 14 Hbase nodes. In all these configurations, we used one timestamp server, allocated same pool of nodes for HBase region servers and HDFS data nodes, and allocated roughly the same number of nodes for server processes to generate enough transaction load. In the evaluation of *service-based approach*, we used one conflict detection server. From Figure 7 we can observe that, incremental scalability through scale-out model is achieved in both service-based and fully decentralized approaches. Under 14-node configuration, the service-based approach achieves roughly 18 thousand transactions per minute. This is approximately 8-fold increase from one-node configuration and 5-fold increase from 2-node configuration. The throughput achieved with fully decentralized approach is lower compared to the service-based approach, however, it shows incremental scalability under scaling-out of the system. We observed that the conflict detection service is relatively light-weight and hence the request processing throughput of the conflict detection server is significantly higher compared to the transaction throughput. We performed microbenchmarking of this service using a single server running on a machine with 4 cores with 2.4GHz CPU and 4GB memory. We observed that it could serve more than 8000 requests per second. Thus, we can see that such a single server would be sufficient for transactional workload of approximately 400K transactions per minute, which, by rough extrapolation, could be supported by HBase cluster of 200 nodes. One could even achieve a higher throughput by using a higher-end machine.

Figures 8 and 9 show the 90 percentile values of transaction response times as the transaction load is increased under 14-node configuration for service-based and fully decentralized approaches, respectively. This data is shown for NewOrder transactions only because TPC-C has many varying-sized transactions and the NewOrder transactions represent a typical mid-weight, read-write transactions which are executed frequently and have stringent response time constraints. From these figures, we observe that in fully decentralized approach, the response times for transactions are higher compared to the service-based approach (although the values are within the acceptable response times bound specified in TPC-C, which is 2 second). This is due to the additional overhead of decentralized conflict detection protocol.

7.3 Cost of Serializability

Another important aspect that we wanted to evaluate was the cost of supporting serializability. We observed that by

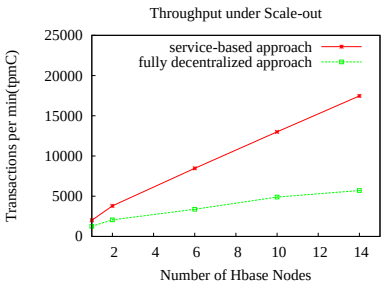


Figure 7: Transaction throughput under scaling-out of system

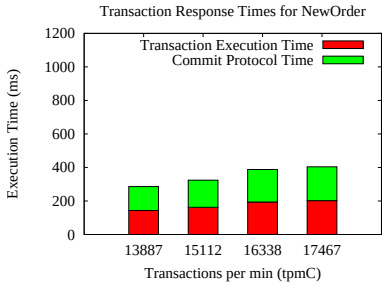


Figure 8: Response Times for NewOrder transactions using service-based approach

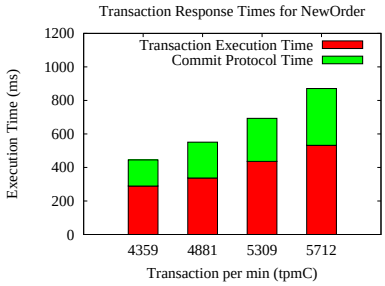


Figure 9: Response Times for NewOrder transactions using fully decentralized approach

following the database population and scaling requirements in TPC-C specifications, transactions rarely have serializability conflicts. Therefore for more contention, we only used one warehouse and populated the database accordingly. We were interested in observing the relative number of aborts due to serialization anomalies compared to aborts due to SI, i.e. *ww* conflicts. We observed that out of total number aborts approximately 13% are due to serialization anomalies. However the cost of supporting serializability, in terms of increased response times, in the fully decentralized approach is significant due to the additional overhead with regards to maintaining dependencies. We observed that majority of overhead is due to additional writes to record read set information and deriving and inserting dependencies. This would not be the case in service-based approach since the dependency information is maintained by the conflict detection service and not in the HBase and hence overhead of deriving dependencies and detecting cycles is relatively very small.

7.4 Cooperative Recovery

For evaluation of the cooperative recovery aspect, we programmed transactions to stall at random points in the commit protocol with some probability and measured the time required to recover the stalled transactions. It should be noted that the recovery of a stalled/failed transaction would be triggered if it blocks the progress of any other transaction due to conflicts or if the STS could not be advanced because of the failed transaction. One issue in this regard was how to set the timeout value for detecting failures. We used smaller timeout values (200 ms) in order to minimize the wait times of transactions. We also used a separate 'garbage-collection' process to expedite recovery of failed transactions. This process would periodically (every 10 seconds) check for failed transactions and perform their recovery. The roll-forward of a transaction takes 50-100 ms. When a stalled transaction does not block any other transaction, it can still delay advancement of STS due its unresolved status. For this we programmed the transactions to check for such failed transactions if the STS gap increases above certain limit (set to 10). We observed that recovery of such transactions was performed within 1 second. For failed transactions which did not reach $CommitIncomplete$, the roll-back of such transactions was performed within 15 seconds on average.

8. CONCLUSION AND DISCUSSION

This paper proposes an approach for decoupled and decentralized transaction management in which the transaction management functions and related recovery actions are decoupled from the storage service. The rationale behind this approach is to eliminate necessity of any distributed synchronization amongst storage nodes for distributed transactions so that the storage service can be scaled independently. We present an approach in which the transaction management functions are performed entirely by the application processes executing the transaction. We present a decentralized transaction commit protocol which implements this model. Additionally, we also present and evaluate an alternative approach which uses a dedicated service for conflict detection among concurrent transactions and to ensure serializability. This alternative design presents a hybrid approach in order to achieve higher transaction throughput and lower response times. Our experimental evaluations show that in both the approaches one can achieve scalability of transaction execution through scaling-out of the system. The approach of using dedicated conflict detection service provides higher transaction throughput and lower transaction response times. Using this approach is more desirable for applications which have stringent response time constraints for transactions. We observe that even a single conflict detection server could support a large transaction workload. However, this approach requires managing and ensuring reliability of another service. In contrast, in fully decentralized approach the transaction management functions are entirely executed by the application processes. This approach uses HBase as a shared storage for storing and accessing transactional metadata such as read-write sets and dependency information. Due to this, in this approach transaction response times are significantly higher due to the overhead of additional requests to HBase. Therefore, this approach may not be suitable for applications with stringent transaction response time constraints. However, for applications with more relaxed response time requirements, this approach is more desirable since it does not depend on any dedicated service, apart from HBase for storage. Our model is also based on cooperative recovery in which the recovery of a partially executed transaction due to process crashes is performed by other application level processes. We have demonstrated in this paper the benefit of this approach in ensuring that any failed transaction does not block progress of other transactions.

Since the basic snapshot isolation model does not guarantee serializability, we present a technique for identifying

dependency cycles among transactions. Our technique precisely aborts only the transaction which lead to serialization anomalies. Ensuring serializability induces overhead of maintaining and tracking dependencies among transactions. In the SI model, the probability of serialization errors depends on the workload and amount of contention among concurrent transactions. In our experiments, we observed that under realistic workload such as TPC-C benchmark, serialization anomalies are typically rare. For applications that can tolerate such anomalies, basic snapshot isolation would be more desirable as it would provide better throughput.

9. REFERENCES

[1] A. Adya, B. Liskov, and P. E. O'Neil. Generalized isolation level definitions. In *ICDE'00*, pages 67–78, 2000.

[2] M. K. Aguilera, A. Merchant, M. A. Shah, A. C. Veitch, and C. T. Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. *ACM Trans. Comput. Syst.*, 27(3), 2009.

[3] Amazon. Amazon simpledb, http://aws.amazon.com/simpledb/.

[4] Apache. Hbase, http://hbase.apache.org/.

[5] J. Baker, C. Bond, J. Corbett, J. J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, pages 223–234, 2011.

[6] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *Proceedings of ACM SIGMOD'95*, pages 1–10. ACM, 1995.

[7] 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.

[8] M. J. Cahill, U. Röhm, and A. D. Fekete. Serializable isolation for snapshot databases. *ACM Trans. Database Syst.*, 34:20:1–20:42, December 2009.

[9] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.*, 26(2):1–26, 2008.

[10] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1:1277–1288, August 2008.

[11] T. P. Council. TPC-C benchmark . Available at URL http://www.tpc.org/tpcc.

[12] S. Das, D. Agrawal, and A. E. Abbadi. G-store: a scalable data store for transactional multi key access in the cloud. In *Proceedings of the 1st ACM Symposium on Cloud Computing* , SoCC'10, pages 163–174, 2010.

[13] S. Das, D. Agrawal, and A. El Abbadi. ElasTraS: an elastic transactional data store in the cloud. In *Proceedings of the 2009 Conference on Hot Topics in Cloud Computing*, HotCloud'09. USENIX Association, 2009.

[14] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41:205–220, October 2007.

[15] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30:492–528, June 2005.

[16] A. Fekete, E. O'Neil, and P. O'Neil. A read-only transaction anomaly under snapshot isolation. *SIGMOD Rec.*, 33:12–14, September 2004.

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

[18] H. Jung, H. Han, A. Fekete, and U. Roehm. Serializable Snapshot Isolation for Replicated Databases in High-Update Scenarios. In *VLDB*, 2011.

[19] B. Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-r, a new way to implement database replication. In *PVLDB'00*, pages 134–143, 2000.

[20] B. Kemme and G. Alonso. Database replication: a tale of research across communities. *PVLDB*, 3(1):5–12, 2010.

[21] D. Kossmann, T. Kraska, and S. Loesing. An evaluation of alternative architectures for transaction processing in the cloud. In *SIGMOD '10*, pages 579–590. ACM, 2010.

[22] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6:213–226, June 1981.

[23] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44:35–40, April 2010.

[24] J. J. Levandoski, D. B. Lomet, M. F. Mokbel, and K. Zhao. Deuteronomy: Transaction support for cloud data. In *CIDR*, pages 123–133, 2011.

[25] Y. Lin, K. Bettina, M. Patiño Martínez, and R. Jiménez-Peris. Middleware based data replication providing snapshot isolation. In *ACM SIGMOD'05*, pages 419–430, 2005.

[26] 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.

[27] D. B. Lomet, A. Fekete, G. Weikum, and M. J. Zwilling. Unbundling transaction services in the cloud. In *CIDR*, 2009.

[28] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. OSDI'10, pages 1–15, 2010.

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

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

[31] C. Zhang and H. D. Sterck. Supporting multi-row distributed transactions with global snapshot isolation using bare-bones HBase. In *GRID*, pages 177–184, 2010.