Scalable Transaction Management with Snapshot Isolation for NoSQL Data Storage Systems

Vinit Padhye and Anand Tripathi
Department of Computer Science
University of Minnesota Minneapolis,
55455, Minnesota, USA.

***Abstract*—We address the problem of building scalable transaction management mechanisms for multi-row transactions on key-value storage systems, which are commonly termed as NoSQL systems. We develop scalable techniques for transaction management utilizing the snapshot isolation (SI) model. Because the SI model can lead to non-serializable transaction executions, we investigate two conflict detection techniques for ensuring serializability. To support scalability, we investigate system architectures and mechanisms in which the transaction management functions are decoupled from the storage system and integrated with the application-level processes. We present two system architectures and demonstrate their scalability under the scale-out model of cloud computing platforms. In the first system architecture all transaction management functions are executed in a fully decentralized manner by the application processes. The second architecture is based on a hybrid approach in which the conflict detection functions are performed by a dedicated service. We perform a comparative evaluation of these architectures using the TPC-C benchmark and demonstrate their scalability. keyword**

***Keywords*-Transaction Management, Scalable Services, Cloud Data Management Systems.**

I. Introduction

The cloud computing platforms enable building scalable services through the scale-out model by utilizing the elastic pool of computing resources provided by such platforms. Typically, such services require scalable management of large volumes of data. It has been widely recognized that the traditional database systems based on the relational model and SQL do not scale well [1], [2]. The NoSQL databases based on the key-value model such as Bigtable [1] and HBase [3], have been shown to be scalable in large scale applications. However, unlike traditional relational databases, these systems typically do not provide multi-row serializable transactions, or provide such transactions with certain limitations. For example, HBase and Bigtable provide only single-row transactions, whereas systems such as Google Megastore [4] and G-store [5] provide transactions only over a particular group of entities. These two classes of systems, relational and NoSQL based systems, represent two opposite points in the scalability versus functionality space. For certain applications, such as web search, email, and social networking, such limited support for transactions in key-value based storage models has been found to be adequate. However, many applications such as online shopping stores, online auction services, financial services, while requiring high scalability and availability, still need certain strong transactional consistency guarantees. For example, an online shopping service may require ACID (atomicity, consistency, isolation, and durability) [6] guarantees for performing payment operations. Thus, providing ACID transactions for NoSQL data storage system is an important problem.

We present here scalable architecture models for supporting multi-row serializable transactions for key-value based NoSQL data storage systems. Our approach is based on decentralized and decoupled transaction management where transaction management functions are decoupled from the storage system and performed by the application-level processes themselves, in decentralized manner. Figure 1 illustrates the seminal elements of this approach. A service hosted in a cloud datacenter environment is accessed by clients over the Internet. The service creates application level processes for performing service functions. These processes belong to the trusted domain of the deployed service. The application-specific data of the service is stored in a key-value based storage system in the datacenter. In our approach, all transaction management functions – such as concurrency control, conflict detection and atomically committing the transaction updates – are performed by the application processes themselves in decentralized manner. These functions are provided to the application in the form of library functions. The metadata necessary for transaction management is stored in the underlying key-value based storage.

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 our approach, the commit protocol is performed in various steps by individual application processes, and the entire sequence of steps is not performed as a single atomic action. Not performing all steps of the commit protocol as one atomic action raises a number of issues. The transaction management protocol should ensure the transactional consistency when multiple processes execute the protocol steps concurrently. 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 that is suspected to be failed. Any number of processes may initiate the recovery of a failed

This work was supported by National Science Foundation grant 0708604

transaction, and such concurrent recovery actions should not cause any inconsistencies.

Based on the decoupled and decentralized transaction management model, we develop two system architectures and demonstrate their scalability using the TPC-C benchmark [7]. The first architecture is *fully decentralized*, in which all the transaction management functions are performed in decentralized manner. The second architecture is a hybrid model in which only the conflict detection functions are performed using a dedicated service and all other transaction management functions are performed in decentralized manner. We refer to this as *service-based architecture*. In developing these architectures we utilize the *snapshot isolation (SI)* [8] model. The SI model is attractive for scalability, as noted in [8], since transactions read from a snapshot, the reads are never blocked due to write locks, thereby providing more concurrency. However, the snapshot isolation model does not guarantee *serializability* [8], [9]. Our work addresses this issue and develops techniques for ensuring serializability of SI-based transactions for NoSQL data storage systems.

Various approaches [10], [11], [12] have been proposed in the past, in the context of relational database management systems (RDBMS), to ensure transaction serializability under the SI model. Some of these approaches [10], [11] are *preventive* in nature as they prevent potential *conflict dependency cycles* by aborting certain transactions, but they may abort transactions that may not necessarily lead to serialization anomalies. On the other hand, the approach presented in [12] detects dependency cycles and aborts only the transactions necessary to eliminate a cycle. However this approach requires tracking of conflict dependencies among all transactions and checking for dependency cycles, and hence it can be expensive. We present here results of our investigation of these approaches in the context of key-value based NoSQL data storage systems.

The major contributions of our work are the following. We present and evaluate two system architectures for providing multi-row transactions using snapshot isolation (SI) on NoSQL databases. Furthermore, we extend the SI based transaction model to support serializable transactions. We demonstrate the scalability of our approach using the TPC-C benchmark. Our work demonstrates that transaction serializability guarantees can be supported in a scalable manner on key-value based storage systems. Using the transaction management techniques presented here, the utility of key-value based cloud data management systems can be extended to applications requiring strong transactional consistency.

The rest of the paper is organized as follows. In the next section we discuss the related work to present the context and significance of our contributions. In Section III, we provide an overview of the snapshot isolation model. Section IV presents the framework that we have developed for transaction management and highlight the various design issues in this framework. Section V presents our decentralized design for supporting the basic SI based transactions. In Section VI, we discuss how the basic SI model is extended to provide serializability. Section VII discusses the service-based architecture. Section VIII presents the evaluations of the scalability of the proposed techniques. Section IX evaluates the performance of the cooperative recovery mechanisms. The conclusions are presented in the last section.

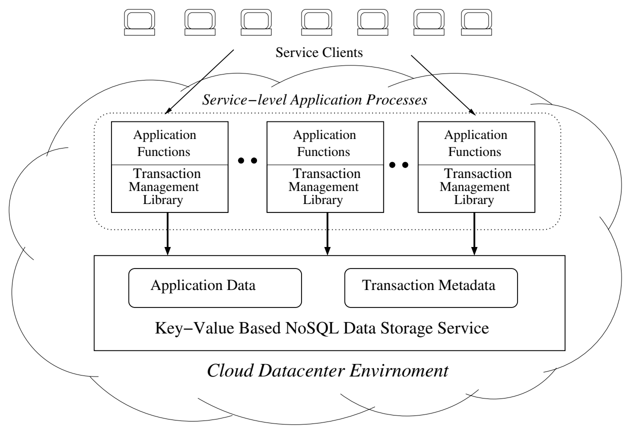
Service Clients
Service−level Application Processes
Application Functions
Transaction Management Library
Application Functions
Transaction Management Library
Application Functions
Transaction Management Library
Application Data
Transaction Metadata
Key−Value Based NoSQL Data Storage Service
Cloud Datacenter Envirnoment

Fig. 1. Decentralized and Decoupled Transaction Management Model

II. Related Work

In recent years, researchers have recognized the scalability limitations of relational databases, and to address the scalability and availability requirements, various systems based on the key-value data model have been developed [1], [13], [14], [4]. Systems such as SimpleDB [14] and Cassandra [15] provide weak consistency. Bigtable [1] and its open-source counterpart HBase [3] provide strong data consistency but provide only single-row transactions. Other systems provide transactions over multiple rows with certain constraints. For example, Megastore [4], and G-store [5] provide transactions over a group of entities. In ElasTraS [16] and VoltDB [17] ACID transactions are supported only over a single partition. Sinfonia [18] and Granola [19] systems provide restricted forms of storage level multi-item transactions. The Calvin system [20] supports ACID transactions by using deterministic ordering of transactions, but this approach requires prior knowledge of a transaction's read/write sets and it is mainly aimed for in-memory databases. CloudTPS [21] provides a design based on a replicated transaction management layer which provides ACID transactions over multiple partitions. An approach based on decoupling transaction management from data storage using a central transaction component is proposed in [22]. In contrast to these approaches, our design presents a decentralized transaction management protocol, wherein the transaction management functions are performed by the application process itself.

Other researchers have proposed techniques for providing multi-row snapshot isolation transactions [23], [24]. The Percolator system [23] addresses the problem of providing SI-based transaction for Bigtable [1]. However, it does not address the problem of ensuring serializability of transactions. Moreover, it is intended for offline processing of data. The work presented in [24] does not adequately address issues related to recovery and robustness when some transaction fails.

The problem of transaction serializability in snapshot-isolation model has been extensively studied [9], [10], [11], [12], [25]. The work in [9] 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 [26] as well as runtime detection of anomalies [10], [11], [12], [25]. Technique presented in [10], [11], [25] tend to be pessimistic and can lead to unnecessary aborts. PSSI approach [12] 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 [25], provided solutions only for centralized databases.

III. Background: Snapshot Isolation Model and Serializability Issues

Snapshot isolation (SI) based transaction execution model is a multi-version based approach utilizing optimistic concurrency control [27]. In this model, when a transaction T_i commits, it is assigned a commit timestamp TS_c^i, which is a monotonically increasing sequence number. The commit timestamps of transactions represent the logical order of their commit points. For each data item modified by a committed transaction, a new version is created with the timestamp value equal to the commit timestamp of the transaction.

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. A read operation by the transaction returns the most recent committed version up to this snapshot timestamp, therefore, a transaction never gets blocked due to any write locks.

A transaction T_i commits only if there exists no committed transaction concurrent with T_i which has modified any of the items in T_i's write-set. That means there exists no committed transaction T_j such that $TS_s^i < TS_c^j < TS_c^i$ and T_j has modified any of the items in T_i's write-set. Thus, if two or more concurrent transactions have a write-write conflict, then only one of them is allowed 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* [28] between two concurrent transactions T_i and T_j is a *read-write (rw) dependency*, denoted by $T_i \overset{rw}{\to} T_j$, implying that some item in the read-set of T_i is modified by T_j. This dependency can be considered as an incoming dependency for T_j and an outgoing dependency for T_i. This is the only kind of dependency that can arise in the SI model between two concurrent transactions. There are other kinds of dependencies, namely *write-read (wr)* and *write-write (ww)*, that can exist between two non-concurrent transactions.

Fekete et al. [9] have shown that a non-serializable execution must always involve a cycle in which there are two consecutive *anti-dependency* edges of the form $T_i \overset{rw}{\to} T_j \overset{rw}{\to} T_k$. In such situations, there exists a *pivot* transaction [9] with both incoming and outgoing anti-dependencies. Figure 2 shows an example of a pivot transaction. In this example, T_2 is the pivot transaction. In the context of traditional RDBMS, several techniques [10], [11], [12], [25] have been developed utilizing this fact to ensure serializability. We investigated the following two approaches for implementing serializable transactions on key-value based storage systems.

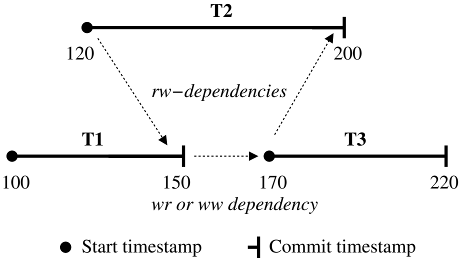

Fig. 2. Pivot transaction

- Cycle Prevention Approach: When two concurrent transactions T_i and T_j have an anti-dependency, one of them is aborted. This ensures that there can never be a *pivot transaction*, thus guaranteeing serializability. In the context of RDBMS, this approach was investigated in [10].
- Cycle Detection Approach: A transaction is aborted only when a dependency cycle is detected involving that transaction during its commit protocol. This approach is conceptually similar to the technique [12] investigated in the context of RDBMS.

The conflict dependency checks in the above two approaches are performed in addition to the check for write-write conflicts required for the basic SI model. We implemented and evaluated the above approaches in both the fully decentralized model and the service-based model. The cycle prevention approach can sometimes abort transactions that may not lead to serialization anomalies. The cycle detection approach aborts only the transactions that can cause serialization anomalies but it requires tracking of all dependencies for every transaction and maintaining a dependency graph to check for cycles.

IV. A Framework for Decentralized Transaction Management

We present here a framework for decentralized transaction management using snapshot isolation in key-value based data storage systems. Implementing SI based transactions requires mechanisms for performing the following actions: (1) reading from a consistent committed snapshot; (2) allocating commit timestamps using a global sequencer for ordering of transactions; (3) detecting write-write conflicts among concurrent transactions; and (4) committing the updates atomically and making them durable. Additionally, to ensure serializability we also need to detect or prevent serialization anomalies as discussed above.

In our approach of decoupled and decentralized transaction management, the transaction management functions described above are decoupled from the data storage systems and performed by the application processes themselves in decentralized manner. The transaction management metadata required for performing these functions is also stored in the key-value storage system. This is to ensure the reliability of this metadata and scalability of transaction management functions which require concurrent access to this metadata.

A transaction execution goes through a series of phases as shown in Figure 3. In the *active* phase, it performs read/write operations on data items. The subsequent phases are part of the commit protocol of the transaction. For scalability, our goal is to design the commit protocol such that it can be executed in highly concurrent manner by the application processes. We also want to ensure that after the commit timestamp is issued to a transaction, the time required for commit be bounded, since a long commit phase of the transaction can potentially block the progress of other conflicting transactions with higher timestamps. Thus, our goal is to perform as many commit protocol phases as possible before acquiring the commit timestamp. We discuss below the various issues that arise in utilizing this approach.

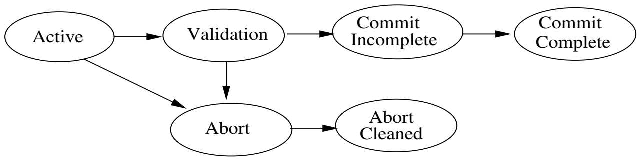

Fig. 3. Transaction Protocol Phases

A. Timestamps Management

In the decentralized model the steps in the commit protocol are executed concurrently by the application processes. Because these steps cannot be performed as a single atomic action, a number of design issues arise as discussed below. There can be situations where several transactions have acquired commit timestamps but their commitment status is not yet known. We also need to make sure that even if a transaction has made its update to the storage system, these updates should not be made visible to other transactions until the transaction is committed. Therefore, we need to maintain two timestamp counters: GTS (global timestamp) which is the latest commit timestamp assigned to a transaction, and STS (stable timestamp), which is the largest timestamp such that all transactions with commit timestamp up to this value are either committed or aborted and all the updates of the committed transactions are written to the global storage. An example shown in Figure 4 illustrates the notion of GTS and STS. In this example, STS is advanced only up to sequence number 16 because the commit status of all the transactions up to sequence number 16 is known, however, the commit status of the transaction with sequence number 17 is not yet known. When a new transaction is started, it uses the current STS value as its snapshot timestamp. We first experimented with using the key-value storage itself to store these counter values. However, we found this approach to be slow, and therefore we use a dedicated service which we refer to as *TimestampService* for maintaining these counter values.

B. Eager vs Lazy Update Model

An important design issue that arises is when should a transaction write its updates to the global key-value storage. We find two distinct approaches with different performance tradeoffs as discussed below. We characterize them as *eager* and *lazy* update models. In the eager update model, a transaction writes its updates to the global storage during its *active* phase, before acquiring its commit timestamp, whereas in the lazy approach all writes are performed after acquiring the commit timestamp.

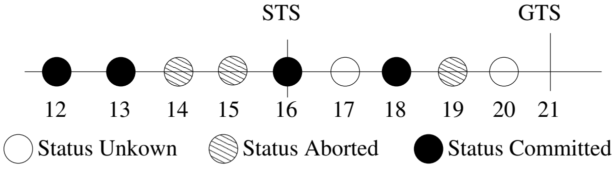

Fig. 4. STS and GTS Counters

In the lazy update approach, the time for executing the commit protocol can become arbitrarily long depending 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. This may affect transaction throughput and system scalability, but it has the advantage that the writes are performed only when the transaction is certain to commit.

In the eager update approach the data is written to the global storage during the active phase, i.e. prior to the commit protocol execution, thereby reducing the execution time for the commit protocol. Also, the transactions can perform their writes overlapped with computation during the *active* phase. The eager update scheme is attractive because its commit protocol execution time does not significantly depend on the size of the write-set of the transaction. Also, it facilitates the roll-forward of a transaction that fails during its commit, since its updates would be already present in the key-value datastore. Due to these advantages we choose to adopt the eager update model instead of the lazy update model.

Implementing the eager update model requires maintaining uncommitted data versions in the storage. For such data versions, we cannot use the transaction's commit timestamp as the version number because it is not known during the *active* phase. Therefore, in the commit protocol these data versions need to be mapped to the transaction's commit timestamp. Moreover, ensuring the isolation property requires that such uncommitted versions should not be visible until the transaction commits.

C. Transaction Validation

The SI model requires checking for write-write conflicts among concurrent transactions. This requires a mechanism to detect such conflicts and a method to resolve conflicts by allowing only one of the conflicting transactions to commit. When two or more concurrent transactions conflict, there are two approaches to decide which transaction should be allowed to commit. The first approach is called *first-committer-wins (FCW)* [27], in which the transaction with the smallest commit timestamp is allowed to commit. In this approach, 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.

This would force a younger transaction to wait for the progress of all the older transactions, thereby limiting concurrency. In contrast, in the second approach, which is called *first-updater-wins (FUW)* [9], conflict detection is performed by acquiring locks on write-set items and in case of conflicting transactions the one that acquires the locks first is allowed to commit. The FUW approach appears more desirable because the conflict detection and resolution can be performed before acquiring the commit timestamp, thereby reducing any sequential ordering based on commit timestamps and reducing the time required for executing the commit protocol. Therefore, we chose to adopt the FUW approach for conflict detection.

D. Cooperative Recovery

There are two problems that arise due to transaction failures. A failed transaction can block progress of other conflicting transactions. A failure of a transaction after acquiring commit timestamp stalls advancement of the STS counter, thereby forcing the new transactions to use old snapshot time, which may likely result in greater aborts due to *write-write* conflicts. Thus, an appropriate timeout mechanism is needed to detect stalled or failed transactions and initiate their recovery. The co-operative recovery actions for a failed transaction are triggered in two situations: (1) a conflicting transaction is waiting for the commit of a failed transaction, and (2) the STS advancement has stalled due to a failed transaction that has acquired a commit timestamp. The recovery actions in the first situation are performed by any of the conflicting transactions, whereas the failures of the second kind are detected and recovery actions are performed by any application level process or by a dedicated system level process. If a transaction fails before acquiring a commit timestamp, then it is aborted, otherwise the transaction is committed and rolled-forward to complete its commit protocol.

V. Implementation of the Basic SI Model

We first describe the metadata that needs to be maintained in the global storage for transaction management. We then describe the various steps in transaction management protocol performed by the application processes. We also describe the cooperative recovery protocol for dealing with transaction failures.

A. Storage System Requirements

We first identify the features of the key-value data storage system that are required for realizing the transaction management mechanisms presented here. The storage system should provide support for tables and multiple columns per data item (row), and primitives for managing multiple versions of data items with application-defined timestamps. It should provide strong consistency for updates [29], i.e. when a data item is updated, any subsequent reads should see the updated value. Moreover, for the decentralized architecture, we require mechanisms for performing row-level transactions involving any number of columns. Our implementation is based on HBase [3], which meets these requirements.

B. Transaction Data Management Model

For each transaction, we maintain in the global storage the following information: transaction-id (tid), snapshot timestamp (TS_s), commit timestamps TS_c, write-set information, and current status. This information is maintained in a table named *TransactionTable* in the global storage, as shown in Figure 5. In this table, tid is the row-key of the table and other items are maintained as columns. The column 'out-edges' is used to record information related to outgoing dependency edges, which is required only in the cycle detection approach. In order to ensure that the *TransactionTable* does not become the bottleneck, we set the table configuration to partition it across all the HBase servers. The data distribution scheme for HBase is based on sequential range partitioning. Therefore, if we generate transaction ids sequentially it creates a load balancing problem since all the rows in *TransactionTable* corresponding the currently running transactions will be stored only at one or few HBase servers. Therefore, to avoid this problem we generate transaction ids randomly.

For each application data table, hereby referred as *StorageTable*, we maintain the information related to the committed versions of application data items and lock information, as shown in Figure 6. An application may have multiple such storage tables. Since we adopt the eager update model, uncommitted versions of data items also need to be maintained in the global storage. A transaction writes a new version of a data item with its tid as the version timestamp. These version timestamps then need to be mapped to the transaction commit timestamp TS_c when transaction commits. This mapping is stored by writing tid in a column named *committed-version* with version timestamp as TS_c. The column 'wlock' in the $StorageTable$ is used to detect write-write conflicts, whereas columns 'rlock', 'read-ts', and 'readers' are used in detecting read-write conflicts for serializability, as discussed in the next section.

Row Key ***(Trans ID)***	**Snapshot Timestamp** ***TS***	**Commit Timestamp** ***TSc***	**Write−Set**	**Out−Edges**	**Status** ***(Trans State)***
tid1	TS = 100	TS = 150	List of Item IDs	Outgoing Dedpendency Edges	Active, Validation, Commit−Incomplete, Commited, Aborted
tid2					

Fig. 5. TransactionTable Structure

Row Key ***(Item ID)***	**Data Column**	**Commited Version**	**read−ts**	**rlock**	**wlock**	**readers**
Item1	**Value 1** **tid1** **Value 2** **tid2** **Value 3** **tid3**	**tid1** **80** **tid2** **100** **tid3** **120**	**Time Stamp**	**tid/null**	**tid/null**	List of Transaction IDs
Item2						

Fig. 6. StorageTable Structure

C. Transaction Management Protocol for Basic SI Model

A transaction T_i begins with the execution of the $start$ phase protocol shown in Algorithm 1. It obtains its transaction-id (tid) and snapshot timestamp (TS_s) from *TimestampService*. It then inserts in the *TransactionTable* an entry: <tid, TS_s, $status = active$> and proceeds to the active phase. For a write operation on an item (specified by row and column keys), following the eager update model, 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 that data item, the latest version of the *committed-version* column in the range $[0, TS_s]$. This gives the tid of the transaction that wrote the latest version of the data item according to the transaction's snapshot. The transaction then reads data specific columns using this tid as the version timestamp.

Algorithm 1 Execution Phase for transaction T_i

```
Start Phase:
  $tid_i$ ← get a unique tid from TimestampService
  $TS_s^i$ ← get current $STS$ value from TimestampService
  insert $tid$, $TS_s^i$ information in $TransactionTable$.
Active Phase:
Read $item$: /* $item$ is a row-key */
  $tid_R$ ← read value of the latest version in the range
      [0,$TS_s^i$]) of the "committed version" column for $item$
  read item data with version $tid_R$
  add $item$ to the read-set of $T_i$
Write $item$:
  write $item$ to StorageTable with version timestamp = $tid_i$
  add $item$ to the write-set of $T_i$
```

A transaction executes the commit protocol as shown in Algorithm 2. At the start of each phase in the commit protocol it updates its status in the $TransactionTable$ to indicate its progress. All status change operations are performed atomically and conditionally, i.e. permitting only the state transitions shown in Figure 3. The transaction first updates its status in the $TransactionTable$ to $validation$ and records its write-set information, i.e. only the item-identifiers (row keys) for items in its write-set. This information is recorded for facilitating the roll-forward of a failed transaction during its recovery. The transaction performs conflict checking by attempting to acquire write locks on items in its write-set as described below. If a committed newer version of the data item is already present, then it aborts immediately. If some transaction T_j has already acquired a write lock on the item, then T_i aborts if $tid_j < tid_i$, else it waits for T_j to either commit or abort. This *wait/die* scheme is used to avoid deadlocks and livelocks. The conflict checking operations for a single item, shown by lines 7-12 in the algorithm, are performed as a single atomic action using the row level transaction feature provided by HBase. On acquiring all locks, the transaction proceeds to the *commit-incomplete* phase.

Algorithm 2 Commit protocol executed by transaction T_i for Basic SI model

```
Validation phase:
 1: if $status = active$ then
 2:     $status \leftarrow validation$
 3: end if
 4: insert write-set information in $TransactionTable$
 5: for all item ∈ write-set of $T_i$ do
 6:     [ begin row level transaction:
 7:     if any committed newer version for $item$ is created then
        abort
 8:     if $item$ is locked then
 9:         if lock-holder's tid < $tid_i$, then abort else wait
10:     else
11:         acquire lock on $item$ by writing $tid_i$ in lock column
12:     end if
13:     :end row level transaction ]
14: end for
Commit-Incomplete phase:
 1: if $status = validation$ then
 2:     status ← commit-incomplete
 3: else abort
 4: $TS_c^i$ ← get commit timestamp from TimestampService
 5: for all item ∈ write-set of $T_i$ do
 6:     insert $TS_c^i \rightarrow tid_i$ mapping in the StorageTable and
        release lock on item
 7: end for
 8: status ← commit-complete
 9: notify completion and provide $TS_c^i$ to TimestampService
    to advance $STS$
Abort phase:
 1: for all item ∈ write-set of $T_i$ do
 2:     if $T_i$ has acquired lock on $item$, then release the lock.
 3:     delete the temporary version created for $item$ by $T_i$
 4: end for
```

Once T_i updates its status to *commit-incomplete*, any failure after that point would result in its roll-forward. The transaction now inserts the $ts \rightarrow tid$ mappings in the *committed-version* column in the $StorageTable$ for the items in its write-set and changes its status to *commit-complete*. At this point the transaction is committed. It then notifies its completion to *TimestampService* and provides its commit timestamp TS_c^i to advance the STS counter. The updates made by T_i become visible to any subsequent transaction, after the STS counter is advanced up to TS_c^i. If the transaction is aborted, then it releases all the acquired locks and deletes the versions it has created.

D. Cooperative Recovery Protocol

When a transaction T_i is waiting for the resolution of the commit status of some other transaction T_j, it periodically checks T_j's progress. If the status of T_j is not changed within a specific timeout value, T_i suspects T_j has failed. If T_j has reached *commit-incomplete* phase, then T_i performs roll-forward of T_j by completing the *commit-incomplete* phase of T_j using the write-set information recorded by T_j. Otherwise, T_i marks T_j as aborted, acquires the conflicting lock, and proceeds further with the next step in its own commit protocol. In this case, the cleanup actions, such as releasing other locks held by the aborted transaction and deletion of temporary versions created by the transactions, can be performed lazily if if does not block any other transaction. The cooperative recovery actions are also triggered when the STS counter cannot be advanced because of a gap created due to a failed transaction. In this case, the recovery is triggered if the gap between STS and GTS exceeds beyond some limit. These recovery actions are triggered by the *TimestampService* itself based on the gap between STS and GTS.

In the above mechanism, setting the proper timeout value is crucial. Setting a high timeout value will cause delays in detecting failures and thus potentially blocking conflicting transactions for a long time. When a transaction fails after acquiring commit timestamp, its timely recovery is crucial since it can block the advancement of STS. On the other hand, setting a low timeout value is also not desirable, since it can cause aborts of transactions that have not actually failed. We refer to these as *false aborts*. The appropriate timeout value depends on the average time taken by a transaction to complete its commit protocol. In Section IX, we present detailed evaluation of this aspect.

VI. Decentralized Model for Serializable SI Transactions

We now describe how the decentralized model for the basic snapshot isolation is extended to support serializable transaction execution using the *cycle prevention* and *cycle detection* approaches discussed in Section III.

A. Implementation of the Cycle Prevention Approach

The cycle prevention approach aborts a transaction when an anti-dependency among two concurrent transactions is observed. This prevents a transaction from becoming a pivot. One way of doing this is to record for each item version the $tids$ of the transactions that read that version and track the *read-write* dependencies. However, this can be expensive as we need to maintain a list of $tids$ per item and detect anti-dependencies for all such transactions. To avoid this, we detect the *read-write* conflicts using a locking approach. During the $validation$ phase, a transaction acquires a *read lock* for each item in its read-set. Read locks on an item are acquired in shared mode. A transaction acquires (releases) a read lock by incrementing (decrementing) the value in a column named *rlock* in the $StorageTable$.

Algorithm 3 Commit protocol for cycle prevention approach

```
Validation phase:
for all item ∈ write-set of $T_i$ do
    [ begin row-level transaction:
    read the 'committed version', 'wlock', ’rlock', and 'read-ts' columns for $item$
    if any committed newer version is present, then abort
    else if $item$ is already locked in read or write mode, then abort
    else if 'read-ts' value is greater than $TS_s^i$, then abort.
    else acquire write lock on $item$
    :end row-level transaction ]
end for
for all item ∈ read-set of $T_i$ do
    [ begin row-level transaction:
    read the 'committed version' and 'wlock' columns for $item$
    if any committed newer version is created, then abort
    if $item$ is already locked in write mode, then abort.
    else acquire read lock by incrementing 'rlock' column for $item$.
    :end row-level transaction ]
end for
execute commit-incomplete phase shown in Algorithm 2
for all item ∈ read-set of $T_i$ do
    [ begin row-level transaction:
    release read lock on $item$ by decrementing 'rlock' column
    if read-ts $< TS_c^i$ then
        read-ts $\leftarrow TS_c^i$
    end if
    :end row-level transaction ]
end for
$status \leftarrow$ commit-complete
notify completion and provide $TS_c^i$ to TimestampService to advance $STS$
```

The commit protocol algorithm for the cycle-prevention approach is presented in Algorithm 3. An anti-dependency between two concurrent transactions can be detected either by the writer transaction or the reader transaction. We first describe how a writer transaction can detect a *read-write* conflict with any other concurrent reader transaction. During the $validation$ phase, a writer transaction checks for the presence of a read lock for an item in its write-set at the time of attempting to acquire a write lock on that item. The transaction is aborted if the item is already read locked. Note that we need to detect *read-write* conflicts only among concurrent transactions to detect anti-dependencies. This raises an issue that a concurrent writer may miss detecting a *read-write* conflict if it attempts to acquire a write lock after the conflicting reader transaction has committed and its read lock has been released. To avoid this problem, a reader transaction records its commit timestamp, in a column named 'read-ts' in the $StorageTable$, while releasing a read lock acquired on an item. A writer checks whether the timestamp value written in the 'read-ts' column is greater than its snapshot timestamp, which indicates that the writer is concurrent with a committed reader transaction. A reader transaction checks for the presence of a write lock or a newer committed version for an item in its read-set to detect *read-write* conflicts. Otherwise, it acquires a read lock on the item.

During the *commit-incomplete* phase, T_i releases the acquired read locks and records its commit timestamps in the 'read-ts' column for the items in its read-set. Since there can be more than one reader transactions for a particular data item version, it is possible that some transaction has already recorded a value in the 'read-ts' column. In this case, T_i updates the currently recorded value only if it is less than TS^i_c. The rationale behind the logic for updating the 'read-ts' value is as follows. For committed transactions $T_1, T_2, ..., T_n$ that have read a particular data item version, the 'read-ts' column value for that item version would contain the commit timestamp of transaction T_k ($k \leq n$), such that T_k is the transaction with the largest commit timestamp in this set of transactions. An uncommitted writer transaction T_j that is concurrent with any transaction in the set $T_1, T_2, ..., T_n$ must also be concurrent with T_k i.e. $TS^j_s < TS^k_c$, since T_k has the largest commit timestamp. Thus T_j will detect the *read-write* conflict by observing that the 'read-ts' value is larger than its snapshot timestamp.

B. Implementation of the Cycle Detection Approach

The cycle detection approach requires tracking all dependencies among transactions, i.e. anti-dependencies (both incoming and outgoing) among concurrent transactions, and *write-read* and *write-write* dependencies among non-concurrent transactions. We maintain this information in the form of a *dependency serialization graph (DSG)* [9], in the global storage. Since an active transaction may form dependencies with a certain committed transaction, we need to retain information about such transactions in the DSG.

A challenge in this approach is to maintain the dependency graph as small as possible by frequently pruning to remove those committed transactions that can never lead to any cycle in the future. For detecting dependencies, we record in $StorageTable$ (in a column named 'readers'), for each version of an item, a list of transaction-ids that have read that item version. Moreover, for each transaction T_i, we maintain its outgoing dependency edges as the list of $tids$ of the transactions for which T_i has an outgoing dependency edge. This information captures the DSG structure.

In the transaction protocol, we include an additional phase called $DSGupdate$, which is performed before the $validation$ phase. In the $DSGupdate$ phase, along with the basic *write-write* conflict check using the locking technique discussed in Section V, a transaction also detects dependencies with other transactions based on its *read-write* sets. If a transaction T_i has outgoing dependency of any kind with transaction T_j, it records T_j's transaction-id in the 'out-edges' column in its *TransactionTable* entry. In the $validation$ phase, the transaction checks for a dependency cycle involving itself, by traversing the outgoing edges starting from itself. If a cycle is detected, then the transaction aborts itself to break the cycle.

VII. Service-based Model

We observed that the decentralized approach induces performance overheads due to the additional read and write requests to the global storage for acquiring and releasing locks. Therefore, we evaluated an alternative approach of using a dedicated service for conflict detection. In the service-based approach, the conflict detection service maintains in its primary memory the information required for conflict detection. A transaction in its commit phase sends its read/write sets information and snapshot timestamp value to the conflict detection service. We designed this service to support conflict detection for the basic-SI model and the cycle prevention/detection approaches for serializable transaction. Based on the particular conflict detection approach, the service checks if the requesting transaction conflicts with any previously committed transaction or not. If no conflict is found, the service obtains a commit timestamp for the transaction and sends a 'commit' response to the transaction process along with the commit timestamp, otherwise it sends 'abort'. Before sending the response, the service updates the transaction's commit status in the *TransactionTable* in the global storage.

The transaction commit phase executed using this approach is presented below in Algorithm 4. Note that this dedicated service is used only for the purpose of conflict detection and not for the entire transaction management, as done in [21], [22]. 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. For scalability and availability, we designed this service as a replicated service as described below.

Algorithm 4 Commit algorithm executed by T_i in Service-Based approach

```
update status to $Validation$ in TransactionTable provided
  $status = Active$
insert write-set information in TransactionTable
send request to conflict detection service with write-set
  information and $TS^i_s$
if $response = commit$ then
    $TS^i_c \leftarrow$ commit timestamp returned by the service
    execute CommitIncomplete phase as in Algorithm 2
      except for step 2
else
    execute Abort phase as in Algorithm 2
end if
```

A. Replicated Service Design

The conflict detection service is implemented as a group of processes. The data item space is partitioned across the replicas using a hashing based partitioning scheme. Thus, a service replica is responsible for detecting conflicts for a set of data items. A service replica stores, for each data item it is responsible for, the information necessary to detect conflicts for that data item. For conflict detection, each replica maintains an in-memory table called as *ConflictTable*, which contains following information for each data item: (1) commit timestamp of the latest committed transaction that has modified the item (*write-ts*), (2) commit timestamp of the latest committed transaction that has read the item (*read-ts*), (3) lock-mode (read-mode, write-mode, or unlocked), and (4) writer-tid (in case of write lock) and list of reader-tids (in case of read lock).

The validation for a transaction is performed by replica(s) responsible for the data items in the transaction's read/write sets. When a transaction's read/write sets span across more than one replica, the validation is performed by coordinating with other replicas, as described below. A client, i.e. the application process executing a transaction, contacts any of the service replicas to validate the transaction by providing its read-write set and snapshot timestamp information. The contacted service replica then acts as the *coordinator* in executing the protocol for transaction validation. The coordinator determines the replicas, called *participants*, that are responsible for the data items in the transaction's read/write sets. It is possible that the coordinator itself is one of the participants. It then performs a two-phase coordination protocol with the participants, as described below.

In the first phase, the coordinator sends *acquire-locks* request to all the participants. Each participant then checks *read-write* and *write-write* conflicts for the items it is responsible for in the following way. For a write-set item, if the *write-ts* value is greater than the transaction's snapshot timestamp then it indicates a *write-write* conflict. Similarly, if the *read-ts* value is greater than the snapshot timestamp then it indicates a *read-write* conflict. For a read-set item, there is a *read-write* conflict if the *write-ts* value is greater than transaction's snapshot timestamp. If any conflict is found, the participant sends 'failed' response to the coordinator. If there are no conflicts, then an attempt is made to acquire read/write locks on the items. We use a deadlock-prevention scheme very similar to the one used in [30]. If the participant acquires all the locks, it sends a 'success' response to the coordinator.

In the second phase, if the coordinator has received a 'success' response from all the participants, then the transaction is committed, otherwise it is aborted. The status of the transaction is updated in the *TransactionTable*. In case of transaction commit the coordinator obtains a commit timestamp from the *TimestampService* and sends a 'commit' message to all participants along with the commit timestamp. Each participant then updates the *write-ts* and *read-ts* values for the corresponding items and releases the locks. Any conflicting transaction waiting for the commit decision of this transaction is aborted. In case of abort, each participant releases the locks acquired by the aborted transaction.

For tracking the validation requests, each replica maintains two in-memory tables: a *ParticipantTable* to store information related to the validation requests for which the replica is a participant, and *CoordinatorTable* to store information for the validation requests for which the replica is the coordinator. The *ParticipantTable* maintains, for each validation request, the transaction-id, the part of the transaction's read/write sets pertaining to this participant, and lock status of each item in this set. The *CoordinatorTable* contains, for each request, the participant replicas, the read/write sets of the transaction and lock status of each item in the set, and responses received from different participants.

B. Service Fault Tolerance

The failure-detection and the group membership management is performed by the service replicas in decentralized manner. Failure-detection is performed using a heart-beat based mechanism, and for group membership management we use the protocol we developed in [31]. When a replica crashes, a new replica is started which takes over the role of the failed replica. The new replica then performs the recovery protocol as described below.

Recovering service state: When a replica fails, there may be uncompleted validation requests for which the replica is either a coordinator or a participant. The new replica needs to recover the information maintained in the *CoordinatorTable* and $ParticpantTable$. This information is *soft-state* and can be recovered from other replicas. The new replica contacts all other existing replicas in the group and obtains information regarding the pending requests for which it was either a coordinator or a participant and the lock status for the items involved in these requests. The reconstruction of *ConflictTable* is done using the read/write sets information stored in the *TransactionTable*. However, scanning the entire *TransactionTable* can become expensive, and hence to reduce this overhead the *ConflictTable* is periodically checkpointed in stable storage. Thus, only the transactions committed after the checkpoint start time need to be scanned.

Failure cases: Ensuring the correctness of the two-phase coordination protocol under replica crashes is crucial. With respect to a particular validation request, the crashed replica can either be a coordinator or a participant. In case of the coordinator crash, the client will timeout and retry the request by contacting the new replica or any other replica. There are three failure cases to consider: (1) failure before initiating the first phase, (2) failure before recording the commit status, and (3) failure after recording the commit status. In the first failure case, the client will timeout and retry the request, since none of the replicas would have any information for this request. In the second case, the new coordinator will resend the lock requests. It may happen that some locks have been already acquired, however, lock operations are idempotent, so resending the lock requests does not cause any inconsistencies. When failure occurs after recording the commit status, the new coordinator will first check the commit status and send commit or abort requests to participants accordingly. In case of the participant crash, the validation request cannot be processed until the recovery of the crashed participant is complete.

VIII. Scalability Evaluations

In our evaluations of the proposed approaches the focus was on evaluating the following aspects: (1) the scalability of dif-

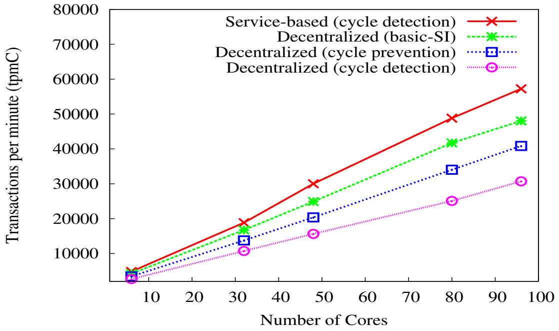

Fig. 7. Transaction throughput under the scale-out model

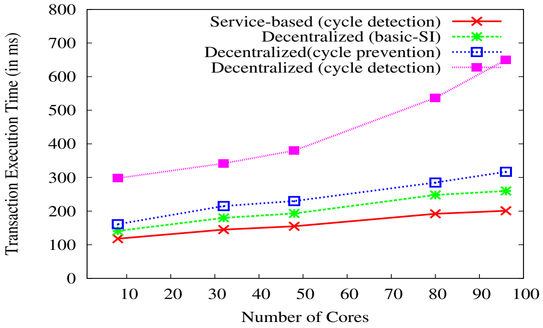

Fig. 8. Average transaction response times under the scale-out model

ferent approaches under the scale-out model, (2) comparison of the service-based model and the decentralized model in terms of transaction throughput and scalability, (3) comparison of the basic SI and the transaction serializability approaches based on the cycle-prevention and the cycle-detection techniques, (4) transaction response times for various approaches, and (5) execution times of different protocol phases.

During the initial phase of our work, we performed a preliminary evaluation of the proposed approaches to determine which approaches are more scalable and which of these need to be investigated further on a large scale cluster. This evaluation was done using a testbed cluster of 40 nodes on which the number of cores on the cluster nodes varied from 2 to 8 cores, each with 2.3 GHz, and the memory capacity ranged from 4 GB to 8 GB. The final evaluations in the later phase were conducted using a much larger cluster provided by the Minnesota Supercomputing Institute (MSI). Each node in this cluster had 8 CPU cores with 2.8 GHz capacity, and 22 GB main memory.

A. TPC-C Benchmark

We used TPC-C benchmark to perform evaluations under a realistic workload. However, our implementation of the benchmark workload differs from TPC-C specifications in the following ways. Since our primary purpose is to measure the transaction throughput we did not emulate terminal I/O. Since HBase does not support composite primary keys, we created the row-keys as concatenation of the specified primary keys. This eliminated the need of join operations, typically required in SQL-based implementation of TPC-C. Predicate reads were implemented using scan and filtering operations provided by HBase. Since the transactions specified in TPC-C benchmark do not create serialization anomalies under SI, as observed in [9], we implemented the modifications suggested in [11]. In our experiments we observed that on average a TPC-C transaction performed 8 read operations and 6 write operations.

B. Preliminary Evaluations

During each experiment, the first phase involved loading data in HBase servers, which also ensured that the data was in the memory of HBase servers when the experiment started. Before starting the measurements, we ran the system for five minutes with initial transaction rate of about 1000 transactions per minute. The purpose of this was to ‘warm-up’ the *TransactionTable* partitions in HBase servers’ memory. The measurement period was set to 30 minutes, in which we gradually increased the transaction load to measure the maximum throughput. For different cluster sizes, we measured the maximum transaction throughput (in terms of committed transactions per minute (tpmC)) and response times. In our experiments, we used one timestamp server and for the service-based model we used one validation server process.

Figure 7 shows the maximum throughput achieved for different transaction management approaches for different cluster sizes. Since there is significant node heterogeneity in our testbed cluster, we indicate the cluster size in terms of the number of cores instead of the number of nodes. This figure shows the throughput for basic-SI model to understand the cost of supporting serializability. We can observe from Figure 7 that scalability of throughput is achieved in both the service-based as well as the decentralized model. However, the service-based approach gives higher transaction throughput than the decentralized approach. As expected, the basic SI model achieves higher throughput compared to approaches for ensuring serializability. This is because of the overhead for performing additional checks required for ensuring serializability. The cycle-prevention approach provides higher throughput than the cycle-detection approach. This is because in the decentralized model the overhead of the cycle-detection approach is significant due to the overhead of maintaining dependency information in the global storage. We also compared the cycle-prevention and the cycle-detection approaches in the context of the service-based model. However, we did not observe any significant difference in the transaction throughput.

Figure 8 shows the average transaction response times for various approaches. As expected, the service-based approach gives smaller response times than other approaches. The cycle-detection approach has significant overhead. In the largest configuration, the average response time for the cycle-detection approach is more than double of the same for the cycle-prevention approach. Also, the cycle-detection approach does not scale well in terms of response times for large clusters. Therefore, we conclude that if serializability is required, it is better to use the cycle-prevention approach than the cycle-detection approach.

We also measured and compared the time taken to execute the various phases of the transaction protocol for different

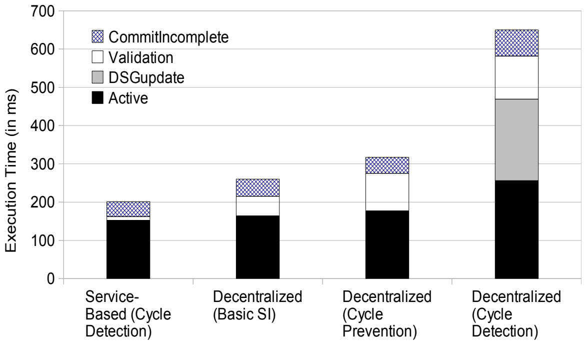

Fig. 9. Execution time for different protocol phases

approaches. Figure 9 shows the average execution times for different phases. This data is shown for the evaluations conducted with the largest (96 cores) configuration. The *validation* phase for the cycle-prevention approach takes more time (approximately by a factor of two) than the *validation* phase for the basic SI approach. In the cycle-detection approach the $DSGupdate$ phase induces a significant overhead.

The preliminary evaluations indicated that the service-based and the decentralized cycle-prevention approaches are scalable for supporting serializable transactions. Among these two approaches the service-based approach performs better. We found that the decentralized cycle-detection approach does not scale well.

C. Scalability Validations on a Large Cluster

Based on the above observations, we selected the service-based approach and decentralized cycle-prevention approach for further evaluations over a large scale cluster to validate their scalability. These evaluations were performed using the MSI cluster resources. Using this cluster, we measured maximum transaction throughput achieved for different cluster sizes. The results of these evaluations are presented in Figures 10 and 11.

Figure 10 presents the throughput scalability, and Figure 11 shows average response times for various cluster sizes. The largest cluster size used in these experiments corresponds to close to 100 nodes (800 cores). In these evaluations we enabled the synchronous logging option for HBase to ensure that the writes are durable even under crashes of HBase servers. Hence, the response times are generally higher under this setting compared to the response times shown in Figure 8 where synchronous logging was disabled. We can observe that both the approaches provide incremental scalability for large cluster sizes. The service-based architecture provides higher transaction throughput and lower response times compared to the decentralized model, confirming our earlier observations from the preliminary evaluations.

D. Scalability of Conflict Detection Service

We also evaluated the scalability of the replicated conflict detection service. In this evaluation, we were mainly interested in measuring the throughput of validation requests. For this purpose, we generated a synthetic workload as follows. A pool of clients generated validation requests for randomly selected read/write sets from an item space of 1 million items. For each request, the size of the read/write sets was randomly selected between 4 to 20 items, with half of the items being read items and half being write items. We measured the throughput as the number of requests handled by the service per second, irrespective of the commit/abort decision, since we are mainly interested in measuring the request handling capacity of the service. Figure 12 shows the saturation throughput of the service for different number of replicas. We can see that increasing the number of replicas provides sub-linear increase in throughput, for example, increasing replica size from 4 to 8 provides throughput increase by a factor of 1.35. An important thing to note here is that the saturation throughput of the conflict detection service, even with small number of replicas, is significantly higher than the overall transaction throughput of the system. For example, from Figures 12 and 10, we can see that the saturation throughput of the service with 8 replicas is approximately 24000 requests per second whereas the saturation transaction throughput with 100 nodes is approximately 5000 transactions per second. Thus, a small number of replicas for conflict detection service can suffice to handle the workload requirement of a large cluster.

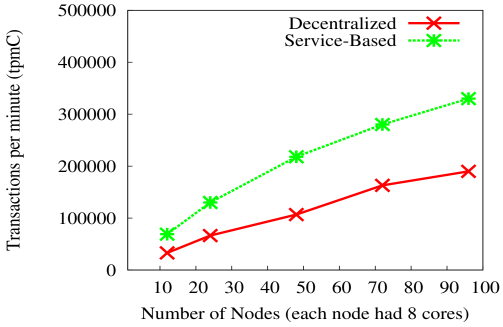

Fig. 10. Transaction throughput under the scale-out model

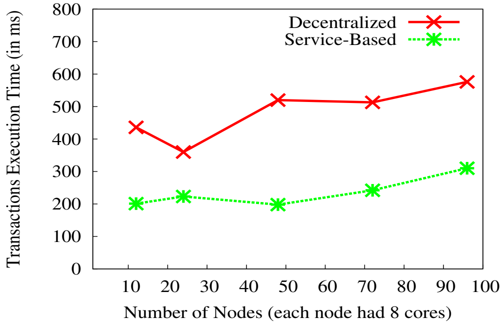

Fig. 11. Average transaction response times under the scale-out model

E. Impact of Transaction Size

Another aspect that we were interested in evaluating is the impact of transaction size, i.e. the number of reads and writes in a transaction, on various performance measures. To evaluate this impact, we created a custom benchmark as follows. We created a single table with 1 million items. The benchmark included three classes of transactions: *small* size transactions accessing 10 items each, *medium* size transactions accessing 100 items each, and *large* size transactions accessing 1000 items each. In all the three classes, half of the accessed items

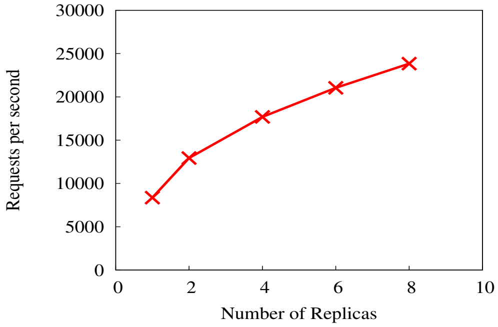

Fig. 12. Scalability of the conflict detection service

were read-set items and half were write-set items. The items to read and write were randomly selected based on uniform distribution. We performed separate evaluations for each class of transactions by generating transaction load for that class and measured the maximum throughput, average response times, and number of aborts for that transaction class. Table I shows the results of these evaluations. These evaluations were performed using a cluster of 24 nodes on the MSI platform and the decentralized model with cycle prevention approach. No failures were injected during these evaluations.

Transaction Size	Max Throughput (transactions/min)	Avg. Response Time (sec)	Percentage of Aborts
10	72196	0.31	0.4 %
100	7088	1.72	17.7 %
1000	1008	7.2	69.1 %

TABLE I
IMPACT OF TRANSACTION SIZE

From Table I, we can observe that as we increase the transaction size the maximum throughput decreases and the average response time increases. This is expected since the maximum throughput and the response time are directly proportion to the number of read/write operations in a transaction. The percentage of aborts increase with the increase in transaction size. This is primarily because under a fixed database size with increase in transaction size the likelihood of transaction conflicts increases.

IX. FAULT TOLERANCE EVALUATIONS

Our goal was to measure the performance of the cooperative recovery model. In these evaluations, our focus was on observing the following aspects: (1) impact of failure timeout values, (2) time taken to detect and recover failed transactions, and (3) impact of failures on abort rate and STS advancement.

A. Experiment Setup

We performed these evaluations on a cluster of 24 nodes on the MSI platform. We induced a moderate transaction load of approximately 50000 transaction per minute, which is lower than the saturation load observed in Figure 10 for cluster size of 24 nodes. The injection of faults was performed as follows. A transaction randomly stalls during the commit protocol execution with certain probability called as *failure probability*. The failure probability is calculated based on the desired failure rate. The failure is injected either in the *validation* phase or the *commit-incomplete* phase (after acquiring commit timestamp). We experimented with a setting of 50% failures in *validation* and 50% failures in the *commit-incomplete* phase as well as an extreme case with all failures in *validation* phase. For every injected failure, we measured the delay in detection of that failed transaction as well as time required to perform recovery actions. We also recorded information about the number of transactions that were wrongly suspected to be failed and aborted due to timeouts. We refer to such aborts as *false aborts*. One would expect that for smaller timeouts the percentage of such false aborts would be higher. The percentage of valid timeout-based aborts depends on the number of injected failures. We performed these experiments for failure rates of 0.1%, 1%, and 10% and timeout values of 50, 100, 300, and 500 milliseconds. In these evaluations, we measured following performance measures: (1) percentage of aborts due to concurrency conflicts such as *read-write* and *write-write* conflicts, referred to as *self aborts*, (2) percentage of valid timeout-based aborts and false aborts, (3) average time for failure detection and recovery, and (4) average gap between STS and GTS under failures.

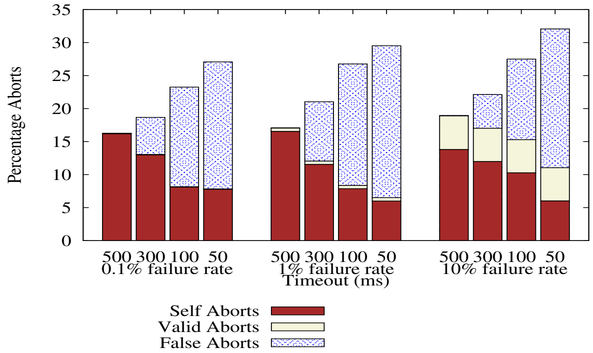

Fig. 13. Abort Statistics for Different Failure Rates (Timeout values ranging from 50ms to 500ms)

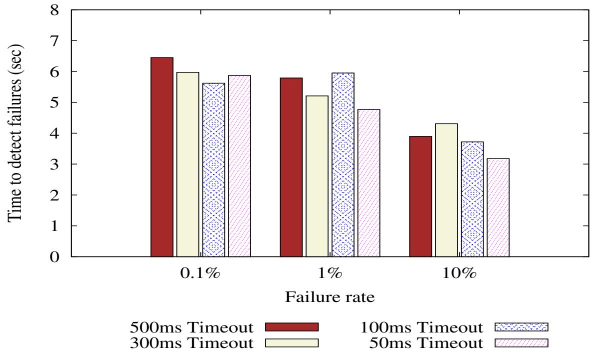

Fig. 14. Average Time to detect failures in Validation phase

B. Evaluation Results

Figure 13 shows the abort statistics for various timeout values and failure rates. This data correspond to the setting of 50% failures in the *validation* and 50% failures in the *commit-incomplete* phase. We can see that, as expected, the percentage of false aborts increases as we decrease the timeout values. The percentage of valid timeout-based aborts depends on the

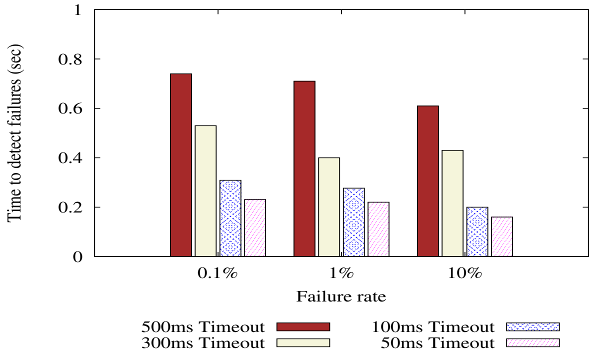

Fig. 15. Average Time to detect failures in CommitIncomplete phase

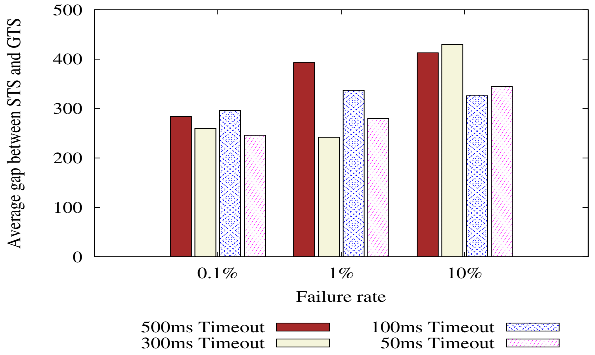

Fig. 16. Average gap between STS and GTS for different failure rates and timeout values

failure rate. Note that, even though we decrease the timeout values from 500 ms to 100 ms, the percentage of total aborts increase only by approximately a factor of two (from 15% to 30%). This is because the transactions that do not conflict with any other transactions are unaffected by the timeout values. If a transaction does not conflict with any other concurrent transaction, it would not be aborted by any transaction irrespective of the timeout values. The only problem that will arise due to failure of a non-conflicting transaction is the blocking of STS advancement if it has acquired the commit timestamp. However, in that case the transaction would be rolled-forward instead of aborting. We also performed this evaluation with the setting of all failures in the *validation* phase. We observed that under this setting also the false aborts increase with decrease in timeout values, confirming our earlier observation. Thus, the appropriate timeout value must be large enough so that the number of false aborts is kept minimum. This largely depends on the transaction response time. Therefore, the appropriate timeout value can be chosen by observing the transaction response times. One can also include autonomic mechanisms to set the timeout values by continually observing the response time values at runtime.

Figure 14 shows the data regarding average delays in detection of the failure of transactions that were failed in the *validation* phase. Figure 15 shows this data for transactions that were failed in the *commit-incomplete* phase. Since we were interested in measuring the delays in detecting valid failures, we measured this data only for the failures that were injected by the failure injection mechanism and not for the transactions that were wrongly suspected to be failed. From Figures 14 and 15, we can see that the detection delays for transactions failed in the *validation* phase are significantly higher than that for the transaction failed in the *commit-incomplete* phase. This is expected because, failure of a transaction failed in the *validation* phase will only be detected when some other transaction encounters a *read-write* or *write-write* conflict with the failed transaction, whereas failure of a transaction failed in the *commit-incomplete* phase will be detected more promptly due to non-advancement of STS. We can also observe that the failure detection delay for transactions failed in the *commit-incomplete* phase is mainly dominated by the timeout value: as we decrease the timeout value the failure detection delay decreases. We observed that the time required to perform recovery actions is independent of whether the transaction failed in the *validation* phase or the *commit-incomplete* phase. The average time to perform recovery typically ranged between 55 ms to 90 ms. We also measured the average gap between STS and GTS for various failure rates and timeout values. This data is shown in Figure 16. We can see that the average STS gap does not depend on the timeout values. However, as we increase the failure rate the average gap value typically increases due to more number transactions blocking the advancement of STS.

X. Conclusion

We have presented here a fully decentralized transaction management model and a service-based architecture for supporting snapshot isolation as well as serializable transactions for key-value based cloud storage systems. We investigated here two approaches for ensuring serializability. We find that both the decentralized and service-based models achieve throughput scalability under the scale-out model. The service-based model performs better than the decentralized model. To ensure the scalability of the service-based approach we developed a replication-based architecture for the conflict detection service. The decentralized model has no centralized component that can become a bottleneck, therefore, its scalability only depends on the underlying storage system. We also observe that the cycle detection approach has significant overhead compared to the cycle prevention approach. We conclude that if serializability of transaction is required then using the cycle prevention approach is desirable. We also demonstrated here the effectiveness of the cooperative recovery mechanisms used in our approach. In summary, our work demonstrates that serializable transactions can be supported in a scalable manner in NoSQL data storage systems.

Acknowledgement: This work was carried out in part using computing resources at the University of Minnesota Supercomputing Institute.

References

[1] 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.*, vol. 26, no. 2, pp. 1–26, 2008.

[2] 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.*, vol. 1, pp. 1277–1288, August 2008.

[3] Apache, “Hbase, http://hbase.apache.org/.” [Online]. Available: http://hbase.apache.org/

[4] 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*, 2011, pp. 223–234.

[5] S. Das, D. Agrawal, and A. E. Abbadi, “G-store: a scalable data store for transactional multi key access in the cloud,” in *Proc. ACM Symp. on Cloud Computing* , 2010, pp. 163–174.

[6] T. Haerder and A. Reuter, “Principles of transaction-oriented database recovery,” *ACM Computing Survey*, vol. 15, no. 4, pp. 287–317, December 1983.

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

[8] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O’Neil, and P. O’Neil, “A critique of ANSI SQL isolation levels,” in *Proc. of ACM SIGMOD’95*. ACM, 1995, pp. 1–10.

[9] A. Fekete, D. Liarokapis, E. O’Neil, P. O’Neil, and D. Shasha, “Making snapshot isolation serializable,” *ACM Trans. Database Syst.*, vol. 30, pp. 492–528, June 2005.

[10] M. Bornea, O. Hodson, S. Elnikety, and A. Fekete, “One-copy serializability with snapshot isolation under the hood,” in *IEEE ICDE’11*, april 2011, pp. 625 –636.

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

[12] S. Revilak, P. O’Neil, and E. O’Neil, “Precisely Serializable Snapshot Isolation (PSSI),” in *ICDE’11*, pp. 482–493.

[13] 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.*, vol. 41, pp. 205–220, October 2007.

[14] Amazon, “Amazon simpledb, http://aws.amazon.com/simpledb/.” [Online]. Available: http://aws.amazon.com/simpledb/

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

[16] S. Das, D. Agrawal, and A. El Abbadi, “ElasTraS: an elastic transactional data store in the cloud,” in *Proc. of HotCloud’09*. USENIX Association.

[17] E. P. Jones, D. J. Abadi, and S. Madden, “Low overhead concurrency control for partitioned main memory databases,” in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, 2010, pp. 603–614.

[18] 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.*, vol. 27, no. 3, 2009.

[19] J. Cowling and B. Liskov, “Granola: low-overhead distributed transaction coordination,” in *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, ser. USENIX ATC’12. Berkeley, CA, USA: USENIX Association, 2012.

[20] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi, “Calvin: fast distributed transactions for partitioned database systems,” in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, 2012, pp. 1–12.

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

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

[23] D. Peng and F. Dabek, “Large-scale incremental processing using distributed transactions and notifications,” in *USENIX OSDI’10*, 2010, pp. 1–15.

[24] C. Zhang and H. D. Sterck, “Supporting multi-row distributed transactions with global snapshot isolation using bare-bones HBase,” in *GRID*, 2010, pp. 177–184.

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

[26] S. Jorwekar, A. Fekete, K. Ramamritham, and S. Sudarshan, “Automating the detection of snapshot isolation anomalies,” in *VLDB*, 2007, pp. 1263–1274.

[27] H. T. Kung and J. T. Robinson, “On optimistic methods for concurrency control,” *ACM Trans. Database Syst.*, vol. 6, pp. 213–226, June 1981.

[28] A. Adya, B. Liskov, and P. E. O’Neil, “Generalized isolation level definitions,” in *International Conference on Data Engineering ICDE*, 2000, pp. 67–78.

[29] W. Vogels, “Eventually consistent,” *Commun. ACM*, vol. 52, pp. 40–44, January 2009.

[30] M. Maekawa, “A n algorithm for mutual exclusion in decentralized systems,” *ACM Trans. Comput. Syst.*, vol. 3, no. 2, pp. 145–159, May 1985.

[31] V. Padhye and A. Tripathi, “Building Autonomically Scalable Services on Wide-Area Shared Computing Platforms,” in *Proc. of the IEEE Symp. on Network Computing and Applications*, August 2011.

Vinit Padhye is currently a Ph.D. candidate in the Computer Science & Enginnering Department at the University of Minnesota, Minneapolis. His current research interests are in distributed systems, fault-tolerant computing, scalable and highly available systems, and cloud computing. He has received his M.S. (2011) in Computer Science from the University of Minnesota and B.E. (2005) from the University of Mumbai.

Anand Tripathi (M’76) received his M.S. (1978) and Ph.D. (1980) in electrical engineering from the University of Texas at Austin, and B.Tech in electrical engineering from the Indian Institute of Technology, Bombay, in 1972.

His research interests are in distributed systems, fault-tolerant computing, system security, and pervasive computing. He is a professor of computer science at the University of Minnesota, Minneapolis. He worked as a Scientific Officer at Bhabha Atomic Research Center, India, during 1973-75. During 1981-84 he worked as a Senior Principal Research Scientist at Honeywell Computer Science Center, Minneapolis. He joined the University of Minnesota in 1984. During 1995-97, he served as a Program Director in the Division of Computer and Communications Research at the National Science Foundation, Arlington, Virginia. He is a Fellow of IEEE and a member of the ACM. He served as an at-large member of the IEEE Computer Society Publications Board (2001-2005). He has served on the editorial boards of IEEE Transactions on Computers, IEEE Distributed Systems Online, Elsevier Journal on Pervasive and Mobile Computing, and IEEE Pervasive Computing. He was the Program Chair for the IEEE Symposium on Reliable Distributed Systems (SRDS) in 2001 and for the Second IEEE International Conference on Pervasive Computing and Communications (PerCom) in 2004. He was one of the organizers of two ECOOP Workshops on exception handling held in 2000 and 2003, and co-editor for two Springer LNCS volumes on exception handling, published in 2002 and 2006.