Scalable Transaction Management with Snapshot Isolation on Cloud Data Management Systems

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

***Abstract*—Key-value based data storage systems such as HBase and Bigtable provide high scalability compared to traditional relational databases, however, they provide only limited transactional functionality, such as single-row transactions. We address the problem of building scalable transaction management mechanisms for multi-row transactions on key-value storage 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 under SI. 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 techniques are implemented by a dedicated service. We perform a comparative evaluation of these architectures for supporting snapshot isolation and serializability using the TPC-C benchmark. Through experimental evaluations, we demonstrate that multi-row transactions can be supported with the guarantees of ACID properties in a scalable manner using the application-level transaction management techniques presented in this paper.**

I. Introduction

The Cloud computing platforms enable building scalable services through the scale-out model by utilizing the elastic pool of computing resources. 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 general multi-row transactions. For example, HBase and Bigtable provide only single-row transactions, whereas systems such as Google Megastore [4], 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 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 strong transactional consistency guarantees. For example, an online shopping service may require ACID guarantees for performing payment operations.

In this paper, we investigate scalable architectures for providing multi-row serializable transactions with *snapshot isolation (SI)* [6]. The snapshot isolation model is attractive for scalability, as identified in the past [6], since the transactions read from a snapshot, the reads are never blocked due to write locks, thereby providing more concurrency. In this regard our investigation is focused on two aspects. First, we investigate scalable architectures for transaction management on key-value based Cloud storage systems. Our approach for providing transaction support is based on decoupling transaction management from the storage service and integrating it with the application-level processes. We present and evaluate two system architectures for transaction management. The first architecture is *fully decentralized*, in which all the transaction management functions, such as concurrency control, conflict detection and atomically committing the transaction updates are performed by the application processes themselves. The general framework of this execution model is shown in Figure 1. The metadata necessary for transaction coordination such as read/write sets and lock information are stored in the underlying Cloud storage. The second architecture is a hybrid model in which certain functions such as conflict detection are performed using a dedicated service. We refer to this as *service-based architecture*.

The second aspect of our investigation is related to the level of transaction consistency and tradeoffs in providing stronger consistency models. Specifically, the snapshot isolation model does not guarantee *serializability* [6], [7]. Various techniques have been proposed to avoid serialization anomalies in SI [8], [9], [10]. Some of these techniques [8], [9] 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 technique presented in [10] 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 how these two techniques can be implemented on Cloud-based key-value databases, and present their comparative evaluation.

In realizing the transaction management model described above, the following issues need to be addressed. In central-

This work was supported by National Science Foundation grants 0834357 and 0708604

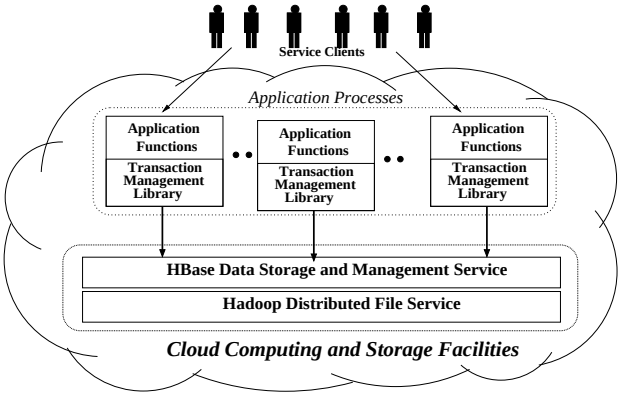

Fig. 1. Decentralized and Decoupled Transaction Management Model

ized database systems, the commit protocol is executed as an atomic operation, which performs validation and ensures that all the updates of a transaction are made durable. In our approach, such functions are performed by individual application processes in various steps, and the entire sequence of steps is not performed as a critical section. Not performing all steps of the commit protocol as one critical section raises a number of issues. 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 executions of recovery actions by multiple processes should not cause any inconsistencies.

The problem of providing multi-row transactions on NoSQL Cloud databases has been recently addressed by several researchers. Systems such as Megastore [4], G-store [5], and DAT [11] provide transactions only over a group of entities. ElasTraS [12] supports multi-row transactions only over a single database partition and provides a restricted *mini-transaction* semantics [13] over multiple partitions. CloudTPS [14] provides a design based on a replicated transaction management layer which provides general ACID transactions over multiple partitions, but with the assumption that transactions access only a small number of well-identified data items. The Deuteronomy system [15], [16] presents an approach based on decoupling transaction management from the data storage, however their design depends on a central transaction component. Recently, other researchers have also proposed decentralized transaction management approaches [17], [18], however, they do not ensure serializability. The work presented in [18] does not adequately address issues related to recovery and robustness when some transaction fails.

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 Cloud databases. Furthermore, we extend SI based transaction model to support serializable transactions under the SI model. For this we develop and evaluate two techniques based on the prevention and detection of dependency cycles. We demonstrate the scalability of our approach using the TPC-C benchmark [19]. Contrary to conventional understanding, we demonstrate that multi-row transactions can be supported in a scalable manner, through application-level transaction management techniques presented in this paper. We find that the strong consistency guarantee such as serializability can be supported in key-value based storage systems, with marginal overheads in terms of resource requirements and response times. Using the transaction management techniques and the models 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 Section II, we provide an overview of the snapshot isolation model. Section III, presents our decentralized design for supporting the basic SI based transactions. In section IV, we discuss how the basic SI model is extended to provide serializability. Section V discusses the service-based architecture. Section VI presents our evaluations of the proposed architectures and techniques. The conclusions are presented in the last section.

II. Background: Snapshot Isolation Model

Snapshot isolation (SI) based transaction execution model is a multi-version based approach utilizing the optimistic concurrency control concepts [20]. 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.

A transaction T_i commits only if none of the items in its write-set have been modified by any committed concurrent transaction T_j i.e. $TS_s^i < TS_c^j < TS_c^i$. 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* [21] between two concurrent 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. Snapshot isolation based transaction execution can lead to non-serializable executions as shown in [6], [7]. It was shown in [21] 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. [7] 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 [7] with both incoming and outgoing rw dependencies. In the above example, T_j is the pivot transaction. Several techniques [8], [9], [10], [22] have been developed utilizing this fact to ensure

serializable transaction execution, in the context of traditional RDBMS. Based on these approaches, we investigate decentralized techniques for implementing serializable transactions on Cloud-based NoSQL databases. Specifically, we consider the following two approaches.

- Cycle Prevention Approach: When two concurrent transactions T_i and T_j have an anti-dependency, abort one of them. This ensures that there can never be a *pivot transaction*, thus guaranteeing serializability. This approach can sometimes abort transactions that may not lead to serialization anomalies. In the context of RDBMS, this approach was investigated in [8].
- Cycle Detection Approach: In this approach a transaction is aborted only when a cycle of dependencies is detected during the execution of the transaction commit protocol This approach is conceptually similar to the technique presented in [10], investigated in the context of RDBMS.

The conflict dependency checks in the above two approaches are performed in addition to the check for ww conflicts required for the basic SI model. We implement and evaluate the above approaches in both fully decentralized model and service-based model.

The cycle prevention approach essentially requires checking whether an item read by a transaction is modified by any concurrent transaction or not. The cycle detection approach aborts only the transaction that can cause serialization anomalies but it requires tracking of all dependencies for every transaction and maintaining a dependency graph to check for cycles. Moreover, since an active transaction can form dependencies with a committed transaction, we need to retain information about committed transactions in the dependency graph. Such committed transactions are called as *zombies* in [10]. Also, for efficient execution, the dependency graph should be kept as small as possible by frequently pruning to remove those committed transaction that can never lead to any cycle in the future.

III. Decentralized Model for SI based Transactions

We first present our decentralized model for supporting basic SI based transactions. Implementing SI based transactions requires mechanisms for performing 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.

We first identify the essential features of the key-value storage service (referred to as the *global storage* in the rest of the paper) required for realizing our design. It should provide support for tables and multiple columns per data items (rows), and primitives for managing multiple versions of data items with application-defined timestamps. It should provide strong consistency for updates, i.e. when a data item is updated, any subsequent reads should see the updated value. Moreover, we need mechanisms for performing row level transactions. Our implementation is based on HBase, which provides these features.

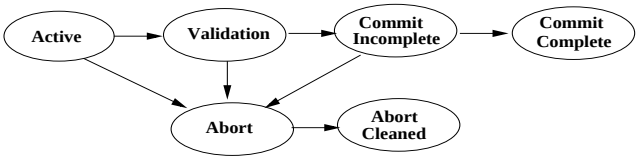

Fig. 2. Transaction Protocol Phases

In our model, a transaction goes through a series of phases during its execution as shown in Figure 2. 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 in realizing these goals and the design choices we made to address them.

Eager Updates vs Lazy Updates: One of the issues is when should a transaction write its updates to the global storage. In the eager model, updates are written during the execution (i.e. Active) phase of the transaction, whereas in the lazy model the updates are flushed to the global storage during the commit protocol. We adopt the eager update model since in the lazy model the commit time of a transaction can significantly increase if the size of the write-set is large. Moreover, the eager model also facilitates roll-forward of failed transactions since their updates are already present in the global storage.

Timestamp Management: There can be situations where a transaction has acquired the commit timestamp but it has not yet completed its commit phase. Therefore, we 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 have completed execution of their commit protocol. When a new transaction is started, it uses the current STS value as its snapshot timestamp TS_s. In the absence of such a counter, the burden of finding the correct committed snapshot would be on each transaction process during its read operations. We use a dedicated *Timestamp Service* to issue commit timestamps. This service also maintains the STS counter and allocates transaction-ids to transactions.

Validation: The SI model requires a validation phase i.e. checking ww conflicts among concurrent transactions. For decentralized conflict detection, we use a form of two-phase commit protocol using locking. A transaction in its Validation phase acquires locks on items in its write-set. We use the *first-updater-wins (FUW)* [7] rule to resolve conflicts, i.e. the transaction that acquires the lock first wins. Using the FUW rule, validation can be performed before acquiring the commit timestamp. In contrast, the *first-committer-wins (FCW)* rule requires acquiring commit timestamps to perform validation.

Data Management Model: We maintain the following information for each transaction in the system: 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 *TxnTable* in HBase. In this table, tid is the row-key of the table and other items are maintained as columns. For each application data table, hereby referred as *StorageTable*, we maintain the information related to the committed versions of data items and write locks. A *StorageTable* may have various application specific columns which contain the application data. Since we adopt the eager update model, uncommitted versions of data items need to be maintained in the global storage. For these versions we can not use the transaction's commit timestamp as the version timestamp since it is not known during the transaction execution phase. Therefore, 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. A column named $wlock$ is used to acquire exclusive write locks. We use the atomic conditional update operations provided by the HBase to atomically acquire a lock.

A. Implementation of the Basic SI Model

We now describe the transaction management protocol for implementing the basic SI model. A transaction T_i begins with the execution of the $Start$ phase protocol shown in Algorithm 1. It obtains its tid and TS_s from *TimestampService*. It then inserts in the *TxnTable* an entry: $<tid$, TS_s, $status = Active>$ and proceeds to the Active phase. For a write operation, 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 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.

When T_i finishes executing the $Active$ phase, it proceeds to its $Validation$ phase to begin the commit protocol, as shown in Algorithm 2. At the start of each phase in the commit protocol it updates its status in the $TxnTable$ to indicate its progress. All status change operations are performed atomically and conditionally, i.e. permitting only the state transitions shown in Figure 2. The transaction first updates its status to $Validation$ in $TxnTable$ 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 by some other transaction. The transaction performs conflict checking by attempting to acquire write locks on items in its write-set. 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 commit of T_j. This *Wait/Die* scheme is used to avoid deadlocks and livelocks. If T_i successfully acquires locks on all items in its write-set, it proceeds to the execution of the $CommitIncomplete$ phase.

Algorithm 1 Execution Phase for transaction T_i

Start Phase:

1: $tid_i \leftarrow$ get a unique tid from TimestampService
2: $TS_s^i \leftarrow$ get current STS value from TimestampService
3: insert tid, TS_s^i information in $TxnTable$.

Active Phase:

Read $item$: /* $item$ is a rowkey and list of column-ids */

1: $tid_R \leftarrow$ read value of the latest version of "committed version" for $item$ in the range [0,TS_s^i]) from *StorageTable*
2: read item data with version tid_R
3: add $item$ to the read-set of T_i

Write $item$:

1: write $item$ to *StorageTable* with version timestamp $= tid_i$
2: add $item$ to the write-set of T_i

In the $CommitIncomplete$ phase, T_i atomically changes its status to $CommitIncomplete$ from $Validation$. Once T_i updates its status to $CommitIncomplete$, 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 *CommitComplete*. At this point the transaction is committed. It then notifies its completion to *TimestampService* and provides its commit timestamp TS_c^i to advances the STS counter. The STS counter is advanced to TS_c^i provided there are no gaps, i.e. all the older transactions have completed execution of their commit protocol. The updates made by T_i become visible to any subsequent transaction, once the STS counter is advanced to TS_c^i. If the transaction is aborted, then it releases all the acquired locks and deletes the versions it has created.

Cooperative Recovery: 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 $CommitIncomplete$ phase, then T_i performs roll-forward of T_j by completing the $CommitIncomplete$ phase of T_j using the write-set information recorded by T_j. Otherwise, T_i marks T_j as aborted and proceeds further with its next step of the commit protocol. In this case, the rollback of T_j is performed lazily as it does not cause any interference. These cooperative recovery actions are also triggered, when the STS counter can not 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.

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

Validation phase:

1: update status to $Validation$ in *TxnTable* provided $status = Active$
2: insert write-set information in $TxnTable$
3: **for all** *item* $\in$ write-set of T_i **do**
4: [**begin row level transaction:**
5: **if** any committed newer version for $item$ is created then **abort**
6: **if** $item$ is locked **then**
7: if lock-holder's tid $< tid_i$, then **abort** else wait
8: **else**
9: acquire lock on $item$ by writing tid_i in lock column
10: **end if**
11: **:end row level transaction**]
12: **end for**

CommitIncomplete phase:

1: update status to $CommitIncomplete$ in the *TxnTable* provided $status = Validation$
2: $TS_c^i \leftarrow$ get commit timestamp from TimestampService
3: **for all** *item* $\in$ write-set of T_i **do**
4: insert $TS_c^i \rightarrow tid_i$ mapping in the *StorageTable* and release lock on *item*
5: **end for**
6: update status to $CommitComplete$ in the *TxnTable*
7: notify completion and provide TS_c^i to TimestampService to advance STS

Abort phase:

1: **for all** *item* $\in$ 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**

IV. Decentralized Model for Serializable SI Transactions

In this section, we discuss 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 II

A. Implementation of the Cycle Prevention Approach

As discussed in Section II, the cycle prevention approach is to abort a transaction when an rw dependency among two concurrent transactions is observed. This prevents any anti-dependency to form and thus no transaction can become a pivot. A straightforward way of doing this is to record for each item version the $tids$ of the transactions that read that version and track the rw dependencies. However, this can be expensive as we need to maintain a list of $tids$ per item and detect rw dependencies for all such transactions. To avoid this, we detect the read-write conflicts using a simple locking approach. During the $Validation$ phase, a transaction acquires a *read lock* for each item in its read-set. A read lock is acquired in a shared mode, i.e. the presence of read lock doesn't block other readers, but only the writers. A transaction acquires a read lock by incrementing the value in a column named *rlock* in the $StorageTable$. An acquired read lock is released by decrementing the 'rlock' column value.

A writer transaction checks for the presence of a read lock (i.e. 'rlock' column value greater than 0) to detect rw conflicts for an item in its write-set, and aborts if the item is already read locked. Note that we need to detect rw conflicts only among concurrent transactions. Therefore, a transaction releases the acquired read locks when it commits/aborts. However, this raises an issue that a concurrent writer may miss detecting an rw conflict if it attempts to acquire a write lock after the conflicting reader transaction has released the read lock. 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 if the timestamp value written in the 'read-ts' column is greater than its snapshot timestamp or not, which indicates that the writer is concurrent with a transaction that has read that particular item. 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 rw conflicts. Otherwise, it tries to acquire a read lock on the item.

The commit phase execution based on this approach is presented in Algorithm 3. The ww conflict check is performed as done in basic SI model (Algorithm 2). During the $CommitIncomplete$ 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. If some transaction has already recorded a timestamp value, then T_i updates the recorded value only if it is less than TS_c^i. Thus, for transaction $T_1..T_n$ that have read a particular data item, the 'read-ts' column value for that item would contain the commit timestamp of transaction T_k ($k \leq n$), such that commit timestamp of T_k is largest among $T_1...T_n$. An uncommitted transaction that is concurrent with any transaction from $T_1...T_n$ must also be concurrent with T_k, since T_k has the largest commit timestamp. Thus, if such a transaction attempts to write the data item, it would detect the rw conflict and abort.

B. Implementation of the Cycle Detection Approach

The cycle detection approach requires tracking all dependencies among transactions, i.e. rw (incoming and outgoing), wr, and ww (with non-concurrent committed transactions) dependencies, and checking for dependency cycles. We maintain the *dependency serialization graph (DSG)* [7], in which transactions are represented as nodes and edges represent dependencies between transactions. The dependency information and the DSG itself is maintained in the global storage. For detecting dependencies, we record in $StorageTable$ (in a column named 'readers'), for each item version, a list of transaction-ids that have read that item version.

We include an additional phase called $DSGupdate$, which is performed before the $Validation$ phase. In the $DSGupdate$ phase, along with the basic ww conflict check, a transaction also detects dependencies and records the dependency information in DSG. In $Validation$ phase, the transaction checks for dependency cycle(s) involving itself, by traversing the outgoing edges starting from itself. If a cycle is detected, then

Algorithm 3 Commit protocol for cycle prevention approach

1: **for all** *item* $\in$ write-set of T_i **do**
2: [**begin row-level transaction:**
3: read the 'committed version', 'wlock', 'rlock', and 'read-ts' columns for $item$
4: **if** any committed newer version is present, then **abort**
5: **else if** $item$ is already locked in read or write mode, then **abort**
6: **else if** 'read-ts' value is greater than TS_s^i, then **abort**.
7: **else** acquire write lock on $item$
8: **:end row-level transaction**]
9: **end for**
10: **for all** *item* $\in$ read-set of T_i **do**
11: [**begin row-level transaction:**
12: read the 'committed version' and 'wlock' columns for $item$
13: **if** any committed newer version is created, then **abort**
14: **if** $item$ is already locked in write mode, then **abort**.
15: **else** acquire read lock by incrementing 'rlock' column for $item$.
16: **:end row-level transaction**]
17: **end for**
18: perform $CommitIncomplete$ as in Algorithm 2
19: **for all** *item* $\in$ read-set of T_i **do**
20: [**begin row-level transaction:**
21: release read lock on $item$ by decrementing 'rlock' column
22: record TS_c^i in 'read-ts' provided current value of 'read-ts' is less than TS_c^i
23: **:end row-level transaction**]
24: **end for**
25: update status to $CommitComplete$ in the *TxnTable*
26: notify completion and provide TS_c^i to TimestampService to advance STS

the transaction aborts itself to break the cycle. To avoid the aborts of two concurrent uncommitted transactions involved in the same cycle, we use commit timestamps to break the ties, i.e a transaction aborts only if it detects a cycle with transactions having smaller commit timestamps. Further implementation details are provided in [23].

V. Service-based Model

In the decentralized scheme discussed above, the conflict detection is performed by the application processes themselves using the metadata, such as lock information, stored in the global storage. This induces performance overhead due to the additional read and write requests for the metadata in the global storage. These overheads can increase transaction completion time and reduce transaction throughput. Therefore, for better performance in terms of transaction latency and throughput, we evaluated an alternative approach of using a dedicated conflict detection service.

In the service-based approach, the conflict detection service maintains the read and write sets information (only the row keys for items) of committed transactions. A transaction in its commit phase sends its read and write sets information and snapshot timestamp value to the conflict detection service. We implemented basic SI validation as well as prevention and detection based approaches for serializability in the conflict detection service. 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, it sends 'commit' response to the transaction, otherwise it sends 'abort'. Before sending the response, the service logs the transaction's commit status in the global storage. This write-ahead logging is performed for recovery purpose.

Note that this dedicated service is only for the purpose of conflict detection and not for the entire transaction management, as done in [14], [16]. 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 scalability and reliability of this service are important aspects to consider. The service maintains the information required for conflict detection, such as read and write sets information, in memory for better performance. This information is *soft-state* and can be recovered upon failure from the write-set information logged in the global storage. For scalability and availability, the service can be replicated. However, based on our experiments, we observe that the scaling requirement of this service are significantly moderate compared to the scaling requirement of the storage service as the workload and request processing requirements of this service are significantly lower compared to the transactional workload. The service receives only one request per transaction and it needs to access only the in-memory data structures for conflict detection.

VI. Evaluations

In this section, we present the evaluations of the proposed approaches. In these evaluations, we are interested in the following aspects: (1) the scalability of different approaches under 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 prevention and cycle detection techniques, (4) transaction response times for various approaches, and (5) execution times of different protocol phases.

A. Experiment Setup

We used TPC-C benchmark [19] to perform evaluations under a realistic workload. However, our implementation of TPC-C specifications differs in the following ways. Since our primary purpose is to measure the transaction throughput we did not emulate terminal I/O. 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 we created the row-keys by concatenating the specified primary keys. Predicate reads were implemented using scan and filtering operations

provided by HBase. Moreover, since the transactions specified in TPC-C benchmark do not create serialization anomalies under SI, as observed in [7], we implemented the modifications suggested in [9], which basically add one more transaction type 'CreditCheck' to create serialization anomalies.

We performed these evaluations in our test cluster of 40 nodes. Following the scale-out model, we increased the number of nodes in the system and measured the maximum transaction throughput (measured in terms of committed transactions per minute (tpmC)) and response times under various system footprint sizes. We measured the maximum throughput by gradually increasing the transaction load and measured the throughput achieved before the transaction response time started increasing exponentially. In all the experiments, we used one timestamp server. In the evaluation of the service-based model, we used one conflict detection server.

B. Evaluation Results

Figure 3 shows the maximum throughput achieved under various system resource footprint sizes for different transaction management approaches. Since there is significant node heterogeneity in our test cluster, we measure the system footprint size in terms of number of cores instead of number of nodes.

Scalability of different approaches: We can observe from Figure 3 that, scalability of throughput through scale-out model is achieved in both the service-based as well as the decentralized model. For example, in case of the decentralized model with the basic SI, the largest configuration (96 cores) achieves roughly 11 fold increase compared to smallest configuration (6 cores). We measure the *scale-out factor* as the increase in the throughput (tpmC) achieved per resource unit (i.e. per core) as the system resources are increased. In other words, the scale-out factor is the slope of the throughput graphs shown in Figure 3. The value of this factor is 582.2 for service-based (cycle detection) approach, 486.6 for decentralized basic SI approach, 415.5 for decentralized cycle prevention approach, and 311.4 for decentralized cycle detection approach. This indicates that all these approaches are scalable, although the throughput gain per resource unit achieved through scaling varies for different approaches. The service-based model has the highest throughput gain per resource unit among all these approaches.

We believe that the scalability of the decentralized model would also extend beyond the largest configuration used in our experiments, provided that the underlying storage service is also scalable. This is because the entire transaction management functions are performed in concurrent manner using the metadata which is itself stored in the underlying storage service. In case of the service-based approach, the scalability of the system could be limited if the conflict detection service becomes the bottleneck. However, we believe that a single conflict detection server should be able to handle large request rates. Our prototype implementation of this service, running on a machine with 2GHz CPU with 4 cores and 4GB memory, can handle roughly 10K requests per second.

Comparative evaluation of different approaches: We first compare the throughput achieved under various approaches.

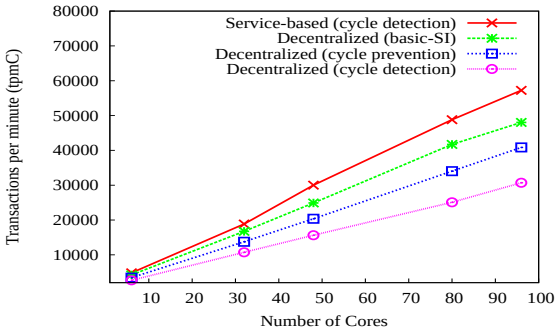

Fig. 3. Transaction throughput under the scale-out model

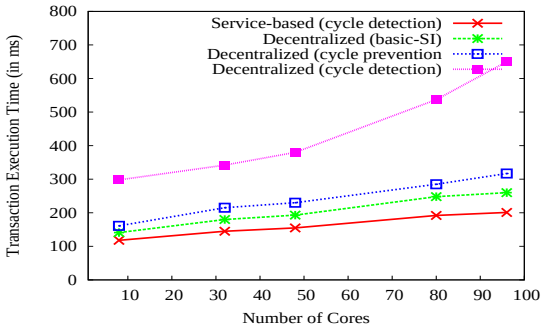

Fig. 4. Average transaction response times under the scale-out model

From Figure 3 we see that 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. The cycle prevention approach provides higher throughput than the cycle detection approach. The cycle prevention approach may abort more number of transactions compared to the cycle detection approach. However, in the decentralized model the overhead of the cycle detection approach is significant, especially due to the overhead of maintaining dependency information in the global storage. Therefore, the total number of transactions committed per unit time is smaller in case of the cycle detection approach compared to the cycle prevention approach. We also compared the basic SI approach, and the cycle prevention and the cycle detection approaches in the context of the service-based model. However, in our experiments we did not observe any significant difference in the transaction throughput mainly due to the fact that the conflict detection

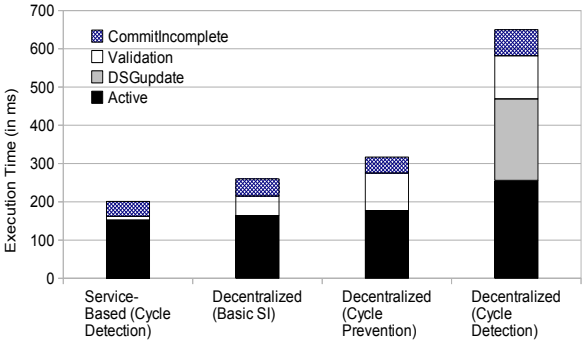

Fig. 5. Execution time for different protocol phases

service never became the bottleneck.

Figure 4 shows the average transaction response times under scale-out 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 cycle detection approach is more than double of the same for cycle prevention approach. Also, the cycle detection approach does not scale well in terms of response times under large configurations. Therefore, we conclude that if serializability is required, it is better to use cycle prevention approach than the cycle detection approach, in the context of decentralized model. We also compare the time taken to execute various phases of the transaction protocol for different approaches. Figure 5 shows the average execution times for different phases. This data is shown for the experiment conducted with the largest (96 cores) configuration. For decentralized model with basic SI, the $Validation$ and $CommitIncomplete$ takes roughly the same time. The $Validation$ phase for cycle prevention approach takes more time than the $Validation$ phase for basic SI approach due to the additional rw conflict checking. We can see that the the overhead of cycle detection approach is mainly due to the $DSGupdate$ phase which detects and stores dependency information in the global storage. The $Active$ phase also takes more time due to additional overhead of recording the read-set information in the global storage.

VII. Conclusion and Discussion

This paper presents system architectures for scalable transaction management and techniques for supporting different consistency levels for NoSQL key-value based databases. The key principle of our approach is to decouple transaction management functions from the storage service and integrate them with the application level processes. Through the experimental evaluations, we demonstrate that multi-row transactions can be supported in a scalable manner with the guarantees of ACID properties. The presented techniques and architectural approaches can be helpful to application developers for building Cloud-based applications which require such a strong transaction consistency, which is not provided by the current NoSQL Cloud databases.

Our evaluations over a 40 node cluster show that, both the decentralized and service-based architectures achieve throughput scalability under the scale-out model. In principle, since there is no bottleneck component, the scalability of decentralized model is not limited under the scale-out model, provided that the underlying storage service is scalable. The scalability of the service-based model could be limited by the saturation throughput of the conflict detection service, however it can be increased by either *scaling up* or replicating the service.

Our transaction model is based on snapshot isolation (SI). To ensure serializability of transaction executing under SI model, we present two conflict detection approaches. In our experiments, we observe that the cycle detection approach has significant overhead compared to the cycle prevention approach. Comparing the transaction throughput and response time performance of both these approaches, we conclude that if serializability of transaction is required then using the cycle prevention approach is better.

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. [Online]. Available: http://dx.doi.org/10.1145/1454159.1454167

[3] Apache, "HBase, 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 *Proceedings of the 1st ACM Symposium on Cloud Computing* , ser. SoCC'10, 2010, pp. 163–174.

[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*. ACM, 1995, pp. 1–10.

[7] 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.

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

[9] 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.

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

[11] N. Chohan, C. Bunch, C. Krintz, and Y. Nomura, "Database-Agnostic Transaction Support for Cloud Infrastructures," in *IEEE International Conference on Cloud Computing*, July 2011, pp. 692 –699.

[12] 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*, ser. HotCloud'09. USENIX Association, 2009.

[13] 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.

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

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

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

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

[18] 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.

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

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

[21] A. Adya, B. Liskov, and P. E. O'Neil, "Generalized isolation level definitions," in *ICDE'00*, 2000, pp. 67–78.

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

[23] V. Padhye and A. Tripathi, "Scalable Transaction Management with Serializable Snapshot Isolation on HBase," http://www.cs.umn.edu/research/technical_reports.php, Department of Computer Science, University of Minnesota, Twin Cities, Tech. Rep., February 2012.