IOSR Journal of Computer Engineering (IOSR-JCE)
e-ISSN: 2278-0661, p- ISSN: 2278-8727Volume 10, Issue 5 (Mar. - Apr. 2013), PP 65-74
www.iosrjournals.org

Scalable Transaction Management on Cloud Data Management Systems

[1] Salve Bhagyashri, [2] Prof. Y.B.Gurav
[1] Department of Computer Science, PVPIT COE Pune.
[2] Assistant Professor Department of Computer Science, PVPIT COE Pune.

Abstract: *Cloud computing is a model for enabling convenient, on -demand network access to shared pool of configurable computing resources that can be rapidly provisioned and released with minimal management effort . As the problem of building scalable trans-action management mechanisms for multi-row transactions on key-value storage systems . For that purpose we develop scalable techniques for transaction management by using the snapshot isolation (SI) model. This SI model can give non-serializable transaction executions, we investigate two conflict detection techniques for ensuring serializability under SI. As getting importance of scalability, we investigate system models 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 models and demonstrate their scalability under the scale-out paradigm of Cloud com-putting platforms. In the first system model, all transaction management functions are executed in a fully decentralized manner by the application processes. The second model is based on a hybrid approach in which the conflict detection techniques are implemented by a dedicated service.*
Keywords- *cloud data, scalable transaction, SI model , serialization , decentralization.*

I. Introduction

It has been considered 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 HBase, have been shown to be scalable in large scale applications. However, 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 [3], G-store [4] provide transactions only over a particular group of entities. However, many applications such as online, online auction services, shopping stores ,financial services, while requiring high scalability and availability, still need certain strong transactional consistency guarantees. These two classes of systems, relational and NoSQL based systems, represent two opposite points in scalability versus functionality space.

In this paper, we are giving scalable system models for providing multi-row serializable transactions by using *snapshot isolation (SI)* [5]. The attractive part of snapshot isolation model is for scalability, [5], since transactions read from a snapshot, the reads are never blocked due to write locks, thereby providing more concurrency. Due to this we focused on two aspect. First, we investigate scalable models for transaction management on key-value based storage systems. For this transaction support is based on decoupling transaction management from the storage service and then integrating it with the application-level processes. Due to this we present and evaluate two system models for transaction management. The first model 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. 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 key-value based Cloud storage. The second model 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 model*.

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* [5], [6]. Various techniques have been proposed to avoid serialization anomalies in SI [7], [8], [9]. Some of these techniques [7], [8] 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.. 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 our approach, the commit protocol is performed by individual application processes in various steps, and the entire se-quence 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 par-tially executed sequence of commit/abort actions on behalf of another process which is suspected to be failed.

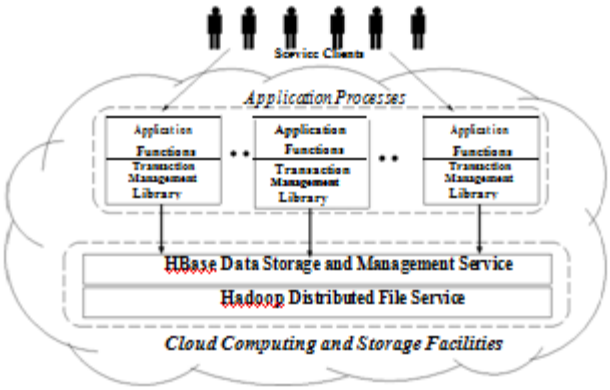

Figure 1. Decentralized and Decoupled Transaction Management Model

Any number of processes may initiate the recovery of a failed transaction, and such concurrent recovery actions should not cause any inconsistencies.

The problem of providing multi-row transactions on NoSQL Cloud databases has been recently addressed by several researchers. ElasTraS [11] supports multi-row trans-actions only over a single database partition and provides restricted *mini-transaction* semantics over multiple partitions. CloudTPS [12] provides a design based on a replicated transaction management layer which provides ACID transactions over multiple partitions, but with the assumption that transactions access only a small number of data items. In [13] an approach based on decoupling transaction management from the data storage is presented, however it requires a central transaction component. Recently, other researchers have also proposed decentralized transaction management approaches [14], [15], however, they do not ensure serializability. The work presented in [15] 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 models for providing multi-row transactions using snapshot isolation (SI) on NoSQL Cloud databases. Furthermore, we extend SI based transaction model to support serializable transactions on key-value based Cloud storage systems. 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. Contrary to conventional understanding, we demonstrate that multi-row transactions can be supported in a scalable manner, through application-level transaction management techniques pre-sented. The strong consistency guarantee 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 presented here, the utility of key-value based Cloud data management systems can be extended to applications requiring strong transac-tional consistency.

II. Background: Snapshot Isolation Model

Snapshot isolation (SI) based transaction execution model is a multi-version based approach utilizing the optimistic concurrency control concepts. When a transaction T_i com-mits, it is assigned a commit timestamp TS_c^i, and it 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, whatever is the data item modified by it, a new version is created the timestamp which value equal to TS_c^i. When a transaction T_i's execution starts, it gets 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, therefore, a transaction 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 Tj 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. Anti-dependency [16] between two concurrent transactions T_i and T_j is a *read-write (rw)dependency,*

denoted by $T_i \xrightarrow{rw} T_j$, 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 [5], [6]. Fekete et al. [6] have shown that a non-serializable execution must always involve a cycle in which there are two consecutive *anti-dependency* edges of the form

$$T_i \xrightarrow{rw} T_j \xrightarrow{Rw} T_k,$$

In such situation, there exists a *pivot* transaction [6] with both incoming and outgoing *rw* dependencies. In the above example, T_j is the pivot transaction. Several techniques [7], [8], [9], [17] have been developed utilizing this fact to ensure serializable transaction execution, in the context of traditional RDBMS. Utilizing these theoretical foundations, we investigate the following two approaches for implementing serializable transactions on key-value based storage systems.

• ***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 [7].

• ***Cycle Detection Approach****:* In this approach a trans- during its execution, transaction is aborted only when a dependency cycle detected during the transaction commit protocol. This approach is conceptually similar to the technique [9] 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. 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 commit-ted transactions in the dependency graph. Such committed transactions are called as *zombies* in [9]. 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 support-ing 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 con-flicts 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.

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.

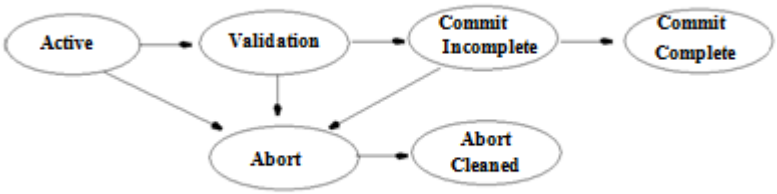

Figure 2. Transaction Protocol Phases

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 *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.

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: *GT S* (global timestamp) which is the latest commit timestamp assigned to a transaction and *ST S* (stable timestamp), *ST S* ≤ *GT S*, which is the largest timestamp such that all transactions with commit timestamp up to this value have completed their commit protocol. When a new transaction is started, it uses the current STS value as its snapshot timestamp $T\,S_s$. 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 *ST S* counter and allocates transaction-ids to transactions.

Validation: The SI model requires checking for *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)* [6] rule to resolve conflicts, i.e. the transaction that acquires the lock first wins.This way validation is performed before acquiring the commit timestamp. In contrast, the *first-committer-wins (FCW)* rule requires acquiring commit timestamps prior to validation.

Data Management Model: We maintain the following information for each transaction in the system: transaction-id (*tid*), snapshot timestamp ($T\,S_s$), commit timestamps $T\,S_c$, write-set information, and current status. This information is maintained in a table named *TxnTable* in the global storage. 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 application data items and write locks. Since we adopt the eager update model, uncommitted versions of data items also need to be maintained in the global storage. For these versions we can not use the transaction's commit timestamp as the version timestamp because 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 times-tamp. These version timestamps then need to be mapped to the transaction commit timestamp $T\,S_c$ when transaction commits. This mapping is stored by writing *tid* in a column named *committed-version* with version timestamp as $T\,S_c$. A column named *wlock* is used to acquire a write 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 $T\,S_s$ from *TimestampSer-vice*. It then inserts in the *TxnTable* an entry: <*tid*, $T\,S_s$, *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 for any subsequent read operations on that item within the transaction. 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, $T\,S_s$]. 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.

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 *T xnT able* 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 *V validation* in *T xnT able* 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

Algorithm 1 Execution Phase for transaction T_i

Start Phase:

1:$tid_{ii} \leftarrow$ get a unique tid from TimestampService
2:$T\,S_s \leftarrow$ get current$_i$ $ST\,S$ value from TimestampService
3:insert tid, $T\,S_s$ information in $T\,xnT\,able$.

Active Phase: Read *item*:

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

1: $tid_R \leftarrow$ read value of the latest version in the range [0, $T\,S_s^i$] of "committed version" column for *item* 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

its recovery by some other transaction. The transaction per-forms 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. On acquiring all locks, it proceeds to the *CommitIncomplete* phase.

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→tid* mappings in the *committed-version* column in the *StorageT able* for the items in its write-set and changes its status to *CommitCom-plete*. At this point the transaction is committed. It then notifies its completion to *TimestampService* and provides its commit timestamp $T\,S_c^i$ to advances the *ST S* counter. The *ST S* counter is advanced to $T\,S_c^i$ 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 *ST S* counter is advanced to $T\,S_c^i$. On abort, a transaction releases the acquired locks and deletes the item versions created.

Cooperative Recovery: When a transaction T_i is waiting for the resolution of the commit status of some other trans-action 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 sus-pects 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. These cooperative recovery actions are also triggered, when

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

1: update status to *V alidation* in *TxnTable* provided *status = Active*
2: insert write-set information in *T xnT able*
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**
8: else retry from step 4
9: **else**
10: acquire lock on *item* by writing tid_i in lock column
11: **end if**
12: **:end row level transaction**]
13: **end for**

CommitIncomplete phase:
1: update status to *CommitIncomplete* in the *TxnTable* provided *status = V alidation*
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 TimestampSer-vice to advance *ST S*

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**

the *ST S* 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 *ST S* and *GT S* exceeds beyond some limit.

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

The cycle prevention approach aborts 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. One way of doing this is to record for each item version the *tids* of the trans-actions 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 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. A transaction acquires (releases) a read lock by incrementing (decrementing) the value in a column named *rlock*.

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 $T S_c^i$ to TimestampSer-vice to advance *ST S*

A writer transaction checks for the presence of a read lock 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 *StorageT able*, 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.

The commit phase execution based on this approach is presented in Algorithm 3. The *ww* conflict check is performed as done in the 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 $T S_c^i$. 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 depen-dencies among transactions, i.e. *rw* (incoming and outgo-ing), *wr*, and *ww* (with non-concurrent committed transactions) dependencies. We maintain the *dependency serializa-tion graph (DSG)* [6] , in the global storage. For detecting dependencies, we record in *StorageT able* (in a column named 'readers'), for each version of an item, a list of transaction-ids that have read that item version.

We include an additional phase called *DSGupdate*, which is performed before the *V alidation* 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 *V alidation* phase, the transaction checks for dependency cycle(s) involving itself, by traversing the outgoing edges starting from itself. If a cycle is detected, then 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.

V. Service-Based Model

In the decentralized scheme discussed above, the conflict detection is performed by the application processes them-selves 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 trans-action 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 ser-vice maintains the read and write sets information (only the row keys for items) of committed transactions. A trans-action 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 se-rializability 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 [12], [13]. The other transac-tion 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, this 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. This is because, the service receives only one request per transaction and it needs to access only the in-memory data structures for conflict detection.

VI. Conclusion

We have presented here a fully decentralized transaction management model and a hybrid model for supporting snapshot isolation as well as serializable transactions for key-value based cloud storage systems. We investigated here two conflict detection approaches for ensuring serializabil-ity. 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, however, its scalability could be limited by the capacity of the conflict detection service. On the other hand 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. In summary, our work demonstrates that serializable transactions can be supported in a scalable manner in cloud computing systems.

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. Computer. Syst.*, vol. 26, no. 2, pp. 1–26, 2008.

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

[3] J. Baker, C. Bond, J. Corbett, J. J. Furman, A. Khorlin, 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.

[4] 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.

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

[6] 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.

[7] 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.

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

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

[10] N. Chohan, C. Bunch, C. Krintz, and Y. Nomura, "Database-Agnostic Transaction Support for Cloud Infrastructures," in *IEEE CLOUD'11*, pp. 692 –699.

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

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

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

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

[15] 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.

[16] A. Adya, B. Liskov, and P. E. O'Neil, "Generalized isola-tion level definitions," in *International Conference on Data Engineering ICDE*, 2000, pp. 67–78.

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