National Conference on 'Innovative Research on Robotics, Circuits and Technology' (IRCT 2018)
Organized By : CIrcuit Branches of SCSVMV, (EEE, ECE, EIE & Mechatronics) , Kanchipuram, Tamil Nadu, India
In Assotiation with International Journal of Scientific Research in Science, Engineering and Technology
© 2018 IJSRSET | Volume 5 | Issue 1 | | Print ISSN: 2395-1990 | Online ISSN : 2394-4099

Scalable Transaction Management with Snapshot Isolation For NOSQL Data Storage System

S. Vijayaraghavan
Assistant Professor, Department of, ECE, SCSVMV, Kanchipuram, Tamil Nadu, India

ABSTRACT

Cloud computing system refers to the demand delivery of IT resources via the internet with pay as you go pricing. A cloud offers many services to the end users such as software, infrastructure and platform go on. Develop scalable techniques for transaction management utilizing the snapshot isolation (SI) model. Because the SI model can lead to non-serializable transaction executions, investigate two conflict detection techniques for ensuring serializability. To support scalability, investigate system architectures and mechanisms in which the transaction management functions are decoupled from the storage system and integrated with the application-level processes. 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. Perform a comparative evaluation of these architectures using the TPC-C benchmark and demonstrate their scalability.

Keywords: Scalable Transaction, key-value store, snapshot isolation

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. The NoSQL databases based on the key-value model such as Bigtable and HBase. Have been shown to be scalable in large scale applications. 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 and G-store provide transactions only over a particular group of entities. These two classes of systems, relational and No SQL based systems. Represent two opposite points in the scalability versus functionality space. We present here scalable architecture models for supporting multi-row serializable transactions for key value based No SQL data storage systems. The widespread popularity of Cloud computing as a preferred platform for the deployment of web applications has resulted in an enormous number of applications moving to the cloud, and the huge success of cloud service providers. Due to the increasing number of web applications being hosted in the cloud, and the growing scale of data which these applications store, process, and serve – scalable data management systems form a critical part of cloud infrastructures.

This design is suitable for applications that require transactional access to groups of keys that are transient in nature, but live long enough to amortize the cost of group formation. Our assumption is that the number of keys in a group is small enough to be owned by a single node. Considering the size and capacity of present commodity hardware, groups with thousands to hundreds of thousands of keys can be efficiently supported. Furthermore, the system can scale-out from tens to hundreds of commodity nodes to support millions of Key Groups. G-Store inherits the data model as well as the set of operations from the underlying Key-Value store; the only addition being that the notions of atomicity and consistency are extended from a single key to a group of keys.

II. OVERVIEW OF EXISTING SYSTEMS

Present here scalable architecture models for supporting multi-row serializable transactions for keyvalue 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. In this approach the multi row transaction on SQL will be facing many issues like hanging or some kind of update error and so on. So that in this proposed system will be used for NoSQL method that will contain Some kind of issues like serializability to overcome this issues implement that Snapshot isolation method will helpful for to conform serializability. A 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 cycle detection approach requires tracking all dependencies among transactions. Anti-dependencies (both incoming and outgoing) among concurrent transactions, and write-read and write-write dependencies among non concurring transactions. We maintain this information in the form of a dependency serialization graph (DSG), 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.

III. PROPOSED APPROACH

The approaches the focus was on evaluating the scalability of different approaches under the scale-out model. A comparison of the service-based model and the decentralized model in terms of transaction throughput and scalability and comparison of the basic SI and the transaction serializability approaches based on the cycle-prevention and the cycle-detection techniques .The approaches the focus was on evaluating the scalability of different approaches under the scale-out model. A comparison of the service-based model and the decentralized model in terms of transaction throughput and scalability and comparison of the basic SI and the transaction serializability approaches based on the cycle-prevention and the cycle-detection techniques .When two concurrent transactions Ti and Tj have anti-dependency, one of them is aborted. This ensures that there can never be a pivot transaction, thus guaranteeing serializability. 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. Cycle Prevention Approach is a two concurrent transactions Ti and Tj 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 . A transaction is aborted only when a dependency cycle is detected involving that transaction during its commit protocol. 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. Cycle Prevention Approach is a two concurrent transactions Ti and Tj have an anti-dependency, one of them is aborted.

IV. DESIGN

PNUTS presents a simplified relational data model to the user. Data is organized into tables of records with attributes. In addition to typical data types, "blob" is a valid data type allowing arbitrary structures inside a record, but not neces-sarily large binary objects like images or audio. (We observe that blob fields, which are manipulated entirely in application logic, are used extensively in practice.) Schemas are flexible: new attributes can be added at any time without halting query or update activity, and records are not required to have values for all attributes. The query language of PNUTS supports selection and prjection from a single table. Updates and deletes must specifiy the primary key. While restrictive compared to relational systems, single-table queries in fact provide very flexible access compared to distributed hash [12] or ordered [8] data stores, and present opportunities for future optimization by the system.Consider again our hypo-thetical social networking application: A user may update her own record, resulting in point access. Another user may scan a set of friends in order by name, resulting in range access. PNUTS allows applications to declare tables to be hashed or ordered, supporting both workloads efficently. Our system is designed primarily for online serving work-loads that consist mostly of queries that read and write single records or small groups of records. Thus, we expect most scans to be of just a few tens or hundreds of records, and optimize accordingly. Scans can specify predicates which are evaluated at the server. Similarly, we provide a "multiget" operation which supports retrieving multiple records (from one or more tables) in parallel by specifying a set of primary keys and an optional predicate, but again expect that the number of records retrieved will be a few thousand at most.

Our system, regrettably, also does not enforce constraints such as referential integrity, although this would be very desirable. The implementation challenges in a system with fine-grained asynchrony are significant, and require future work. Another missing feature is complex ad hoc queries (joins, group-by, etc.). While improving query functionality is a topic of future work, it must be accomplished in a way that does not the response-time and availability currently guaranteed to the more "transactional" requests of web applications.

V. CYCLE DETECTION APPROACH

A transaction is aborted only when a dependency cycle is detected involving that transaction during its commit protocol. 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. 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 ap-proaches 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. When two concurrent transactions Ti and Tj have anti-dependency, one of them is aborted. This ensures that there can never be a pivot transaction, thus guaranteeing serializability. 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.

VI. CYCLE PREVENTION APPROACH

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. Cycle Prevention Approach is a two concurrent transactions Ti and Tj 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 .The cycle prevention 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 concurring transactions. We maintain this information in the form of a dependency serialization graph (DSG), 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. 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, transaction records its commit timestamp, in a column named 'read-ts' in the Storage Table, while releasing 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.

VII. TIME STAMP MANAGEMENT

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 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 is known, however, the commit status of the transaction with sequence number 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 Timestamp Service for maintaining these counter values.

VIII. EXPERIMENT

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 , 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, there-by 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.

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 cooperative recovery actions for a failed transaction are triggered in two situations.

The conflicting transaction is waiting for the commit of a failed transaction, and 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.

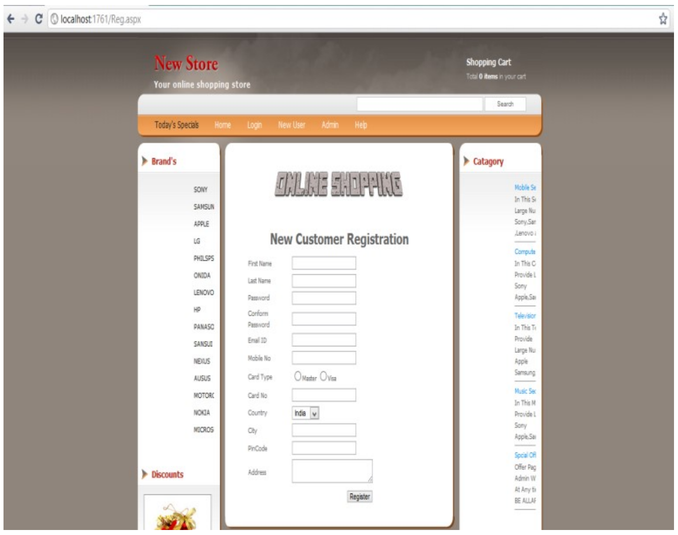


Figure 1. Registration screenshot

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. In our experiments we observed that on average a TPC-C transaction performed 8 read operations and 6 write operations.

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.

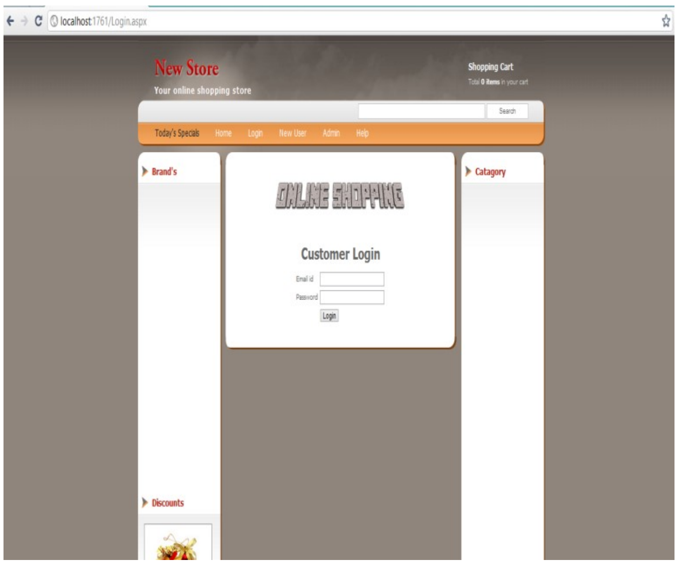


Figure 2. Login screenshot

For each transaction, we maintain in the global storage the following information: transaction, snapshot timestamp, commit time stamps , write-set information, and current status. This information is maintained in a table named Transaction Table in the global storage, as shown in Fig. 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. To ensure that the Transaction Table 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 Transaction Table 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 Storage Table, we maintain the information related to the committed versions of application data items and lock information, as shown in Fig. 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 TSc when transaction commits. This mapping is stored by writing tid in a column named committed-version with version timestamp as TSc. The column 'w lock' in the Storage Table is used to detect write-write conflicts, whereas columns 'r lock,' 'read-ts,' and 'readers' are used in detecting read write conflicts for serializability, as discussed in the next section.

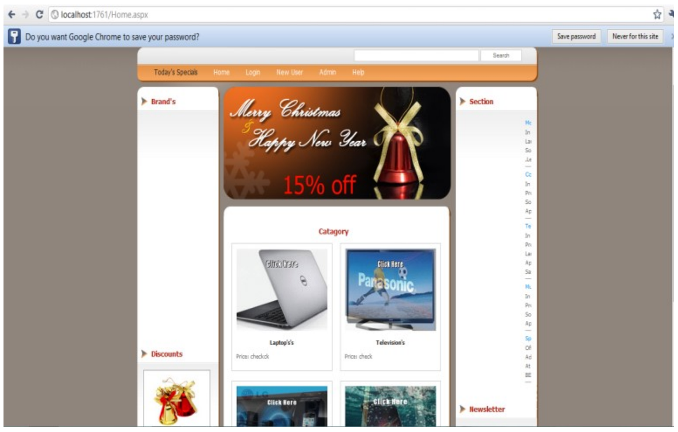
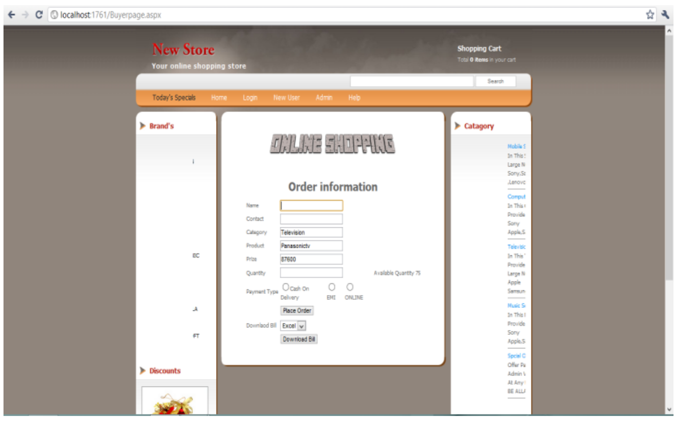


Figure 3. Home Screen Shot

Understanding Snapshot Isolation and Row Versioning. Once snapshot isolation is enabled, updated row versions for each transaction are maintained in tempdb. A unique transaction sequence number identifies each transaction, and these unique numbers are recorded for each row version.SI is an extension of multiversion concurrency control. A transaction T1 executing with Snapshot Isolation Takes snapshot of committed data at start of T1 called start- timestamp Always reads/modifies data in its own snapshot Updates of concurrent transactions are not visible to T1 T1 is allowed to commit only when another Tx t2 running concurrently has not already written the data item that T1 intends to write.

PNUTS is a hosted, centrally-managed database service shared by multiple applications. To add capacity, we add servers. The system adapts by automatically shifting some load to the new servers. The bottleneck for some applications is the number of disk seeks that can be done concurrently; for others it is the amount of aggregate RAM for caching or CPU cycles for processing queries. In all cases, adding more servers adds more of the bottleneck resource. When servers have a hard failure (such as a burnt out power supply or RAID controller failure), we automatically recover by copying data (from a replica) to other live servers (new or existing), carrying out little or no recovery on the failed server itself. Our goal is to scale to more than ten worldwide replicas, each with 1,000 or more servers. At this scale, auto-mated failover and load balancing is the only way to manage the operations load. This hosted model introduces several complications that must be dealt with. First, different applications have different workloads and requirements, even within our relatively narrow niche of web serving applications. Therefore, the system must support several different workload profiles, and be automatically or easily tunable to different profiles. For example, our master ship migration protocol adapts to the observed write patterns of different applications. Second, we need performance isolation so that one heavyweight application does not negatively impact the performance of other applications. In our current implementation, performance isolation is provided by assigning different applications to different sets of storage units within a region.



Order information

Name	
Contact	
Category	Television
Product	Panasonictv
Price	87600
Quantity	

Available Quantity 75

Payment Type ○ Cash On Delivery ○ EMI ○ ONLINE

Place Order

Download Bill Excel

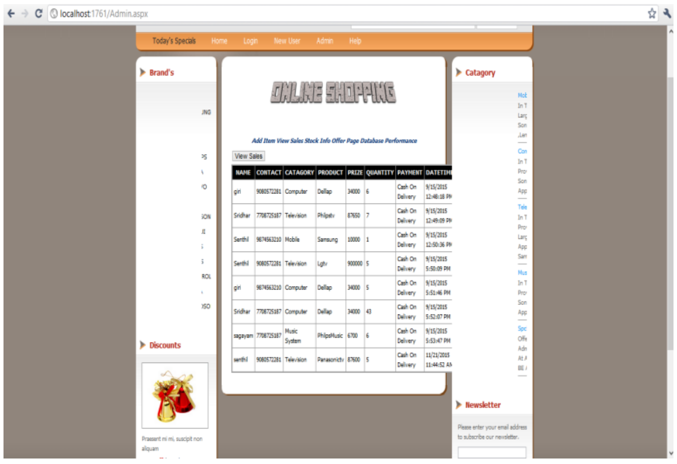
Download Bill

Figure 4. Buyer page Screen shot

Our implementation of Paxos has interesting trade in system behavior. Application servers in multiple datacenters may initiate writes to the same entity group and log position simultaneously. All but one of them will fail and need to retry their transactions. The increased latency imposed by synchronous replication increases the likelihood of conicts for a given per-entity-group commit rate Limiting that rate to a few writes per second per entity group yields insignificant conict rates. For apps whose entities are manipulated by a small number of users at a time this limitation is generally not a concern. Most of our target customers scale write throughput by shading entity groups more only or by ensuring replicas are placed in the same region, decreasing both latency and connect rate. Applications with some server \stickiness" are well positioned to batch user operations into fewer Megastore transactions. Bulk processing of Megastore queue messages is a common batching technique, reducing the conict rate and increasing aggregate throughput. For groups that must regularly exceed a few writes per second, applications can use the _ne-grained advisory locks dispensed by coordinator servers. Sequencing transactions back-to-back avoids the delays associated with retries and the reversion to two-phase Paxos when a convict is detected.

To scale throughput and localize outages, we partition our data into a collection of entity groups, each independently and synchronously replicated over a wide area. The underlying data is stored in a scalable

NoSQL datastore in each datacenter . Entities within an entity group are mutated with single-phase ACID transactions (for which the commit record is replicated via Paxos). Operations across entity groups could rely on expensive two-phase commits, but typically leverage Megastore's e_cient asynchronous messaging. A transaction in a sending entity group places one or more messages in a queue; transactions in receiving entity groups atomically consume those messages and apply ensuing mutations. Note that we use asynchronous messaging between logically distant entity groups, not physically distant replicas .All network transaction between datacenters is from replicated operations, which are synchronous and consistent. Indexes local to an entity group obey ACID semantics those across entity groups have looser consistency. See Figure 2 for the various operations on and between entity groups.



NAME	CONTACT	CATAGORY	PRODUCT	PRIZE	QUANTITY	PAYMENT	DATETIM
giri	9080572281	Computer	Dellap	34000	6	Cash On Delivery	9/15/2015 12:48:18 P
Sridhar	7708725187	Television	Philpstv	87650	7	Cash On Delivery	9/15/2015 12:49:09 P
Senthil	9874563210	Mobile	Samsung	10000	1	Cash On Delivery	9/15/2015 12:50:36 P
Senthil	9080572281	Television	Lgtv	900000	5	Cash On Delivery	9/15/2015 5:50:09 PM
giri	9874563210	Computer	Dellap	34000	5	Cash On Delivery	9/15/2015 5:51:46 PM
Sridhar	7708725187	Computer	Dellap	34000	43	Cash On Delivery	9/15/2015 5:52:07 PM
sagayam	7708725187	Music System	PhilpsMusic	6700	6	Cash On Delivery	9/15/2015 5:53:47 PM
senthil	9080572281	Television	Panasonictv	87600	5	Cash On Delivery	11/21/2015 11:44:52 A

Figure 5. View sale Screen shot

We decided to use Paxos, a proven, optimal, fault-tolerant consensus algorithm with no requirement for a distinguished master. We replicate a write-ahead log over a group of symmetric peers. Any node can initiate reads and writes Each log append blocks on acknowledgments from a majority of replicas, and replicas in the minority catch up as they are able |the algorithm's inherent fault tolerance eliminates the need for distinguished failed" state. A novel extension to Paxos, detailed in Section, allows local reads at any up-to-date replica. Another extension permits single-roundtrip writes. Even with fault tolerance from Paxos, there are limitations to using a single log. With replicas spread over a wide area, communication latencies limit overall through-put. Moreover, progress is impeded when no replica is cur-rent or a majority fail to acknowledge writes. In a traditional SQL database hosting thousands or millions of users, using a synchronously replicated log would risk interruptions of widespread impact . So to improve availability and throughput we use multiple replicated logs, each governing its own partition of the data set.

We evaluated common strategies for wide-area replication Asynchronous Master/Slave A master node replicates write-ahead log entries to at least one slave. Log appends are acknowledged at the master in parallel with transmission to slaves. The master can support fast ACID transactions but risks downtime or data loss during failover to a slave. A consensus protocol is required to mediate master ship. Synchronous Master/Slave a master waits for changes to be mirrored to slaves before acknowledging them, allowing failover without data loss. Master and slave failures need timely detection by an external system. Optimistic Replication Any member of a homogeneous replica group can accept mutations , which are asynchronously propagated through the group. Availability and latency are excellent. However, the global mutation ordering is not known at commit time, so transactions are impossible. We avoided strategies which could lose data on failures, which are common in large-scale systems. We also discarded strategies that do not permit ACID transactions. Despite the operational advantages of eventually consistent systems, it is currently too dificult to give up the read-modify-write idiom in rapid application development. We also discarded options with a heavyweight master. Failover requires a series of high-latency stages often causing a user-visible outage, and there is still a huge amount of complexity. Why build a fault-tolerant system to arbitrate mastership and failover work ows if we could avoid distinguished masters altogether.

Replicating data across hosts within a single data center improves availability by overcoming host-specific failures but with diminishing returns. We still must confront the networks that connect them to the outside world and the infrastructure that powers, cools, and houses them. Economically constructed sites risk some level of facility-wide outages [25] and are vulnerable to regional disasters. For cloud storage to meet availability demands, service providers must replicate data over a wide geographic area. In contrast to our need for a storage platform that is global, reliable, and arbitrarily large in scale, our hardware building blocks are geographically conned, failure-prone, and super limited capacity. We must bind these components into a united ensemble _bring greater throughput and reliability. To do so, we have taken a two-pronged approach for availability, we implemented a synchronous, fault tolerant log replicator optimized for long distance-links for scale, we partitioned data into a vast space of small databases, each with its own replicated log stored in a per-replica NoSQL datastore.

PNUTS presents a simplified relational data model to the user. Data is organized into tables of records with attributes.In addition to typical data types, "blob" is a valid data type, allowing arbitrary structures inside a record, but not neces-sarily large binary objects like images or audio. (We observe that blob fields, which are manipulated entirely in application logic, are used extensively in practice.) Schemas are flexible: new attributes can be added at any time without halting query or update activity, and records are not required to have values for all attributes. The query language of PNUTS supports selection and prjection from a single table. Updates and deletes must specific the primary key. While restrictive compared to relational systems, single-table queries in fact provide very flexible access compared to distributed hash or ordered data stores, and present opportunities for future optimization by the system . Consider again our hypo-thetical social networking application: A user may update her own record, resulting in point access. Another user may scan a set of friends in order by name, resulting in range access. PNUTS allows applications to declare tables to be hashed or ordered, supporting both workloads efficently. The implementation challenges in a system with fine-grained asynchrony are significant, and require future work. Another missing feature is complex ad hoc queries (joins, group-by, etc.). While improving query functionality is a topic of future work, it must be accomplished in a way that does not jeapardize the response-time and availability currently guaranteed to the more "transactional" requests of web applications. In the shorter term, we plan to provide an interface for both Hadoop, an open source implementation of Map Reduce , to pull data out of PNUTS for analysis, much as Map Reduce pulls data out of Big Table .

IX. 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 bottle neck, 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 system.

In this paper we present Megastore, a scalable, highly available datastore designed to meet the storage requirements of interactive Internet services. We use Paxos for synchronous wide area replication, providing lightweight and fast failover of individual operations. The latency penalty of synchronous replication across widely distributed replicas is more than onset by the convenience of a single system image and the operational benefits of carrier-grade availability. We use Bigtable as our scalable datastore while adding richer primitives such as ACID transactions, indexes, and queues. Partitioning the database into entity group sub-databases provides familiar transactional features for most operations while allowing scalability of storage and throughput.

X. FUTURE WORK

In the future, we would like to explore the implications of the Key Grouping protocol in the presence of analytical workloads and index structures built on Key-Value stores. We would also like to explore the feasibility of the design of G-Store using Key-Value stores such a Dynamo and PNUTS where the data store spans multiple data centers and geographical regions, and supports replication and weaker consistency guarantees of reads, and evaluate the ramifications of the weaker consistency guarantees of the data store on the consistency and isolation guarantees of transactions on groups In the concept multi row transaction using DB2 database has been more important one for all the process that scalability process to done the project work. In feature thing is all multi rows. If user at stabel mode means database will be remove the user from DataBase.

XI. 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: ADistributed Storage System for Structured Data,' ACM Trans.Comput. Syst., vol. 26, no. 2, pp. 1-26, June 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 Endowment, vol. 1, no. 2, pp. 1277-1288, Aug. 2008.

[3]. Apache, Hbase. 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 Proc. 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. Cloud Comput., 2010.

[6]. T.Haerder and A. Reuter, 'Principles of Transaction-Oriented Database Recovery,' ACM Comput. Survey, vol. 15, no. 4, pp. 287-317, Dec. 1983.

[7]. T.P. Council, San Francisco, CA, USATPC-C Benchmark.Online]. Available: http://www.tpc.org/tpcc.

[8]. H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, andP. O'Neil, 'A Critique of ANSI SQL Isolation Levels,' in Proc.ACM SIGMOD, 1995, pp. 1-10.

[9]. A. Faceted, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha,'Making Snapshot Isolation Serializable,' ACM Trans. Database Syst., vol. 30, no. 2, 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 Proc. IEEE ICDE, Apr. 2011, pp. 625-636.

[11]. M.J. Cahill, U. Rohm, and A.D. Fekete, 'Serializable Isolation for Snapshot Databases,' ACM Trans. Database Syst., vol. 34, no. 4, pp. 20:1-20:42, Dec. 2009.