NoSQL Concatenated Transactions for Numerous Application in the Cloud

Madhavamuniappan S #1, Kalimuthu M #2

#1 Department of Information Technology
SNS College of Technology, Coimbatore, Tamil Nadu, India

Abstract—NoSQL Cloud data services provide scalability and high availability properties for web applications but at the same time they sacrifice data consistency and large write back time. However, many applications cannot afford data inconsistency and large transaction commits time. Transaction Processing system is a scalable transaction manager to allow cloud database services to execute the ACID transactions of web applications, even in the presence of server failures and network partitions and it also ensure less write back time in NoSQL Database. Implementation of these approach on top of the NoSQL data base system and shows that our system takes very less Transaction Commit time compare with NoSQL database without CloudTPS. However using Berkeley DB as NoSQL database system, and we use the Virtualization technique to implement the cloud environment. In these paper used in Oracle Virtualbox software for creating the virtualization environment. The output of this work shows the transaction completion items in two different NoSQL database system and implementation transactions in one NoSQL Database system, which only take very less transaction commit time compared two second NoSQL system.

Index Terms— NoSQL, Berkeley Databases, Cloud computing, Eventual Consistency, Partitioning.

I. INTRODUCTION

Cloud computing offers the vision of a virtually infinite pool of computing storage and networking resources where applications can be scalable deployed [1]. In particular, NoSQL cloud database services such as Amazon’s Simple DB and Google’s Big table offer a scalable data tier for applications in the cloud. These systems typically partition the application data to provide incremental scalability, and replicate the partitioned data to tolerate server failures. The scalability and high availability properties of Cloud platforms however come at a cost. First, these scalable database services allow data query only by primary key rather than supporting secondary-key or join queries.

Second, these services provide only weak consistency such as eventual data consistency: any data update becomes visible after a finite but un-deterministic amount of time. As weak as this consistency property may seem, it does allow building a wide range of useful applications, as demonstrated by the commercial success of Cloud computing platforms. However, many other applications such as payment and online auction services cannot afford any data inconsistency. While primary key-only data access is a relatively minor inconvenience that can often be accommodated by good data structures, it is essential to provide transactional data consistency to support the applications that need it.

NoSQL Cloud data stores provide scalability and high availability properties for web applications, but at the same time they sacrifice data consistency and also take much more time to complete the transactions within it. However, many applications cannot afford any data inconsistency. CloudTPS is a scalable transaction manager which guarantees full ACID properties for multi-item transactions issued by Web applications, even in the presence of server failures and network partitions and also reducing the write back time in the NoSQL database.

We implement this approach on top of the NoSQL data base with CloudTPS and without CloudTPS, and shows that our system with CloudTPS possess little amount of write back time compared with the system without CloudTPS. CloudTPS exploits three properties of Web applications to allow efficient and scalable operations [2]. First, we observe that in Web applications, all transactions are short-lived because each transaction is encapsulated in the processing of a particular request from a user. This rules out long-lived transactions that make scalable transactional systems so difficult to design, even in medium-scale environments.

Second, Web applications tend to issue transactions that span a relatively small number of well identified data items. This means that the commit protocol for any given transaction can be confined to a relatively small number of servers holding the accessed data items. It also implies a low (although not negligible) number of conflicts between multiple transactions concurrently trying to read/write the same data items.

Third, many read-only queries of Web applications can produce useful results by accessing an older yet consistent version of data. This allows executing complex read queries directly in the cloud data service, rather than in LTM.

Cloud TPS must maintain the ACID properties even in the case of server failures. For this, we replicate data items and transaction states to multiple LTM, and periodically checkpoint consistent data snapshots to the cloud storage.
service. Here demonstrated the scalability of our transactional
database service using a prototype implementation.

The design of the TPS [3] to guarantee the atomicity,
consistency, isolation and durability properties. Each of the
properties is discussed individually then discusses the
memberhip mechanisms to guarantee the ACID properties
even in the case of LTM failures and network partitions.

The main objective of the study is to compare the two
NoSQL data base systems with cloudTPS and without
CloudTPS, the result shows that the NoSQL data base with
cloudTPS takes very little write back time in the database in
contrast with NoSQL without CloudTPS. The second
objective of this thesis shows that NoSQL system with
CloudTPS guarantees full ACID properties for multi
transactions issued by web applications even in the presence
of server failures and network partitions. The third objective
of this study shows the important of Cloud Computing and
NoSQL data base system in today’s Computing environment.

This article is an extended version of a previous conference
paper. The additional contributions of this article are as
follows:

- We describe the membership management and
  failure recovery protocol in detail. This protocol
  maintains ACID properties in the case of machine
  failures and network partitions.
- We present the memory management mechanism,
  which prevents LTMs from memory over and
  reduces the required number of LTMs.
- We describe our prototype implementation on top of
  Berkeley DB and discuss the port of CloudTPS to
  other cloud data services.
- We further demonstrate the scalability of our
  transactional system by evaluating the performance
  of our prototype implementation on top of SimpleDB
  in the Amazon Cloud.

This article is structured as follows. how to guarantee
ACID properties even in the case of server failures and
network partitions. The implementation details of CloudTPS
and two optional optimizations for memory management and
read-only transactions.

II. RELATEDWORK

In recent years there have been numerous implementations
of distributed key-value stores, each exhibiting different mix
of performance, scalability, availability characteristics and
alternate architectures. These include Amazon Dynamo [11],
Google BigTable [5], Yahoo! PNUTS [6], HBase, and
Cassandra [15], Amazon SimpleDB, Amazon S3. These
systems use commodity hardware and exhibit scalability and
high-availability but provided lower consistency guarantees,
often limited to only eventual consistency [21].

Typically only single item transactions are guaranteed and
query capability is limited. Data is accessed using the primary
key and data scan support is limited to only a few systems like
PNUTS and SimpleDB. More recently, there have been
developments like Spinnaker [20], Windows Azure Storage
[4], Google Cloud Storage provide single item consistency
guarantees. The system design focuses on scalability,
performance and single key consistency. Spinnaker uses a
Paxos-based protocol to ensure consistency while COPS [17]
and Granola [9] use replication protocol optimizations to
achieve greater performance while supporting native multi-
item transactions. Despite these advances, the bulk of the
systems leave multi-item transactional data access to the client
application. This is prone to programmer error and the results
are often completely incorrect.

In order to address these issues, some systems have
implemented a relational database engine to provide the query
capabilities and transaction support with the raw data stored in
a distributed key-value store [3]. This is suitable for
applications that require an RDBMS for persistence with the
advantage that it provides a complete SQL interface with full
transaction support. The performance and transaction
throughput of the system is limited only by the underlying
queue implementation.

Most applications built using key value stores work well
because of the relative simplicity of the programming
interface to the data store. Many of these applications use
write-once-read-many (WORM) data access to the key
value store and function well under the eventual consistency
setting. However, there are applications that are built to run on
the same data that require better consistency across multiple
keys.

The first approach to address this issue is to implement
transactional capabilities within the data store itself. The data
store manages the storage as well as transaction management.
The Spanner [8] is a distributed data store that supports
transactions. The COPS [17] and Granola [9] implement the
distributed key-value store with a custom API to enable
applications to transactional access the data store.

Similarly, HyperDex Warp [12] is a high-performance
distributed key-value store that provides a client library that
supports linearizable transactions. The client library simulates
the access to the data items on behalf of the application using
an API provided by the data store which maintains multiple
versions of each data item. These systems support transactions
across multiple keys with a distributed, homogeneous key
value store. The focus of these systems is to build a better,
more capable distributed data store and optimize the trans-
action coordination across it. The CloudTPS [22] design uses
data store access through a transaction manager split across
multiple nodes into local transaction managers (LTM). LTM
failures are handled using transaction logs replication across
LTMs.

269
The middleware approach works well when the application is hosted in the cloud and there is a known and controlled set of data stores used by the application. They perform well in this situation and provides one programming interface to the application simplifying the data store access. However, these systems require to be setup and maintained separately. This is not suitable in our use case where individual application instances need the hands, low maintenance features of key value stores and each may have different access privileges to individual data stores.

Another way of implementing multi-key transaction support for distributed key-value stores is to incorporate the transaction coordinator into the client. We know of two implementations that use this approach. Percolator [19] implements multi-key transactions with snapshot isolation semantics [2]. It depends on a central fault-tolerant timestamp service called a timestamp oracle (TO) to generate timestamps to help coordinate transactions and a locking protocol to implement isolation. The locking protocol relies on a read-evaluate-write operation on records to check for a lock associated with each record. It does not take advantage of test-and-set operations available in most key-value stores making this technique unsuitable for client applications spread across relatively high-latency WANs. No deadlock detection or avoidance is implemented further limiting its use over these types of networks.

III. PROPOSED METHOD

The overview of our scalable transaction system implemented as a runtime library that linked to the client application. The architecture of the library is described in the library provides programming abstraction for transactions and data stores.

The implementation makes the following data store assumptions of support for single item transactions write followed by read returns the last updated version support for test and set operation conditional writes support for global read-only access.

A. System Overview

For every client issues in http requests to a Web application, which in turn issues transactions to a transaction processing system (TPS). The TPS is composed of any number of LTMs, each of which is responsible for a subset of all data items.

![Fig 1. Transaction Processing System](image1)

The Web application can submit a transaction to any LTM that is responsible for one of the accessed data items. This LTM then acts as the coordinator of the transaction across all LTMs in charge of the data items accessed by the transaction.

Local transactions managers operate on an in-memory copy of the data items loaded from the cloud storage service. Data updates resulting from transactions are kept in memory of the LTMs.

B. Data Model

In the simple key value store where data is stored as BLOBs. Operations are limited to single key–value pair at a time.

![Fig 2. Data Access Model](image2)

It is implemented as a partitioned system and targets on applications that need only weak consistency if it results in high availability. It does not provide any guarantee for isolation.

C. System Interface

The interface provided by Dynamo consists of only two operations:

- **get(key)** – This operation returns a list of objects and a context. It may return more than one object if there is no version conflict. The context returned is usually metadata information like the version of the object.
- **put(key, context, object)** – The client has to provide key, version of the object and the object for the put/write operation.
We implement transactions using the 2-Phase Commit protocol (2PC). In the first phase, the coordinator requests all involved LTMs and asks them to check that the operation can indeed be executed correctly. If all LTMs vote favorably, then the second phase actually commits the transaction. Otherwise, the transaction is aborted.

CloudTPS transactions are short-lived and access only well-identified data items. CloudTPS allows only server-side transactions implemented as predefined procedures stored at all LTMs. Each transaction contains one or more sub-transactions, which operate on a single data item each. The application must provide the primary keys of all accessed data items when it issues a transaction.

IV. IMPLEMENTATION

Concretely, a transaction is implemented as a Java object containing a list of sub-transaction instances. All sub transactions are implemented as sub-classes of the subTransaction abstract Java class. Each sub-transaction contains a unique class name to identify itself, a table name and primary key to identify the accessed data item, and input parameters organized as attribute-value pairs. Each sub-transaction implements its own data operations by overriding the run() operation. The return value of the run() operation specifies whether this sub-transaction is able to commit.

Phase 1
Step 1: Coordinator asks all participants to prepare to commit transaction Ti.
Step 2: Ci adds the records <adds prepare T> to the log and forces log to stable storage.
Step 3: sends prepare T messages to all LTMs at which T executed.
Step 4: Upon receiving message, transaction managers at determines if it can commit the transaction
Step 5: if not, add a record <no T> to the log and send >abort T message to Coordinator.
Step 6: if the transaction can be committed, then add the record <ready T> to the log.
Step 7: Force all records for T to stable storage.
Step 8: Send ready T message to Co-ordinator.
Step 9: T can be committed if co-ordinator receives a ready message from all the LTMs
Step 10: Otherwise T must be aborted.
Step 11: Coordinator adds a decision record, < commit > or < abort >, to the log and forces record onto stable storage. Once the record stable storage it is irrevocable.
Step 12: Co-ordinator sends a message to each LTMs informing it of the decision commit or abort.
Step 13: Participants take appropriate action.

Timestamp Ordering Protocol
The idea for this scheme is to order the transactions based on their timestamps [20]. A schedule in which the transactions participate is then serializable, and the equivalent serial schedule has (TO).

The algorithm associates with each database item X two timestamp (TS) values.
- Read timestamp
- Write timestamp

Here we use a Time stamp Ordering Protocol in CloudTPS to maintain the isolation property within the database.
- Time stamp ordering rule

If pi(x) and qi(x) are conflicting operations then pi(x) is processed before qi(x) if ts(Ti)<ts(Tj).
- Theorem
If To rule is enforced, then all the executions generated by the scheduler are serializable.

Proof: If T1 ....TnT1 exists in SG (H) then ts(T1) <...<ts(Tn) <ts(T1) due to rule.

- Read Operation
ri(x): if ts(Ti) <wts(x) then reject it.
Otherwise (ts (Ti)>=wts(x)), schedule ri(x) And set rts (x) = max (rts (x), ts(Ti))

- Write Operation
wi(x): if ts(Ti) <rts(x) then reject it.
If ts(Ti) <wts(x) then reject it.
Otherwise, schedule wi(x) And set wts(x) = max (wts(x), ts(Ti))

To ensure a consistent membership, all membership changes are realized through a 2PC across all available LTMs. All LTMs block incoming transactions until the new system membership has been committed consistently. In the first phase of a membership change, each LTM waits for all of its coordinated ongoing transactions to terminate, and then votes “COMMIT.” After reaching an agreement to “COMMIT,” the second phase updates the system membership and applies the new data assignment through data item replication/relocation.

Each membership change creates a new membership version attached to a monotonically increasing timestamp. Each LTM attaches the timestamp of its current membership to all of its messages. If an LTM receives a message with a higher timestamp than its own, this means that the other LTMs consider it as having failed. The concerned LTM discards its entire state and re-joins.

After each membership change, the new timestamps stored in a special “Membership” table in the cloud data service. By scanning through this “Membership” table, any
new LTM or any Web application instance can locate the currently available LTMs. One issue is that the cloud data services may return a stale membership.

However, one can contact the TPS and obtain the latest membership as long as the stale membership contains at least one LTM currently in the TPS. Any LTM may initiate a membership update if it wants to join the system or it detects the unavailability of other LTMs. This means that multiple membership updates may be issued simultaneously.

To guarantee the isolation of such updates, we use a simple optimistic concurrency control mechanism so that only one membership update can take place at a time [42]. If an LTM receives a request for a membership update before a previous one has finished, then this LTM will vote "ABORT" to the latter. To avoid continuous conflicts and aborts, LTMs may insert a random time delay before reinitiating the aborted membership update.

Evaluation:

We demonstrate the scalability of our transactional database system by presenting the performance evaluation of a prototype implementation under the workload of TPC-W [7]. TPC-W is an industry standard e-commerce benchmark that models an online bookstore similar to Amazon.com. This paper focuses on the scalability of the system with increasing number of LTMs rather than on its absolute performance for a given number of LTMs. Our evaluation assumes that the application load remains roughly constant and studies the scalability in terms of the maximum sustainable throughput under a response time constraint.

V. EXPERIMENTAL RESULTS

CloudTPS supports read-write and read-only transactions indifferently. The only difference is that in read-only transactions no data item is updated during the second phase of 2PC. Read-only transactions have the same strong data consistency property as read-write transactions, but also the same constraint: accessing well-identified data items by primary key only. However, CloudTPS provides an additional feature to support complex read-only transactions containing example range queries.

We exploit the fact that many read queries can produce useful results by accessing a consistent but possibly stale data snapshot. For example, in e-commerce Web applications, a promotion service may identify the bestseller items by aggregating recent orders information. However, it may not be necessary to compute the result based on the absolute most recent orders. We therefore introduce the concept of Weakly-Consistent Read-only Transaction (WCRT).

<table>
<thead>
<tr>
<th>Name</th>
<th>Berkeley DB</th>
<th>Mango DB</th>
<th>Oracle DB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Widely used in process key-value store</td>
<td>One of the most popular document store</td>
<td>Key value store based on Berkeley DB Java Edition</td>
</tr>
<tr>
<td>Implementation Languages</td>
<td>C</td>
<td>C</td>
<td>C,C++</td>
</tr>
<tr>
<td>Operating System</td>
<td>Android, iOS, Linux, Solaris, Windows</td>
<td>Linux, Solaris, Windows</td>
<td>HP-UX, Linux, Solaris, Windows</td>
</tr>
<tr>
<td>Partitioning</td>
<td>None</td>
<td>Shading</td>
<td>Horizontal</td>
</tr>
<tr>
<td>Replication Methods</td>
<td>Master slave replication</td>
<td>Master slave replication</td>
<td>Master slave replication Master – master replication</td>
</tr>
<tr>
<td>Map reduce</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Transactions Concepts</td>
<td>ACID</td>
<td>No</td>
<td>ACID</td>
</tr>
</tbody>
</table>

A WCRT contains any number of read operations offered by the cloud data service, such as table scans for Big table. Web applications issue WCRTs directly to the cloud data service, by passing the LTMs. All read operations of a WCRT executes on the same internally consistent but possible slightly out dated snapshot of the database.

Fig. 3. Difference between TPS and without TPS
IV. Conclusion

The need for high scalability, availability, massive data processing and distribution lead to the development and usage of NoSQL data stores. This paper examined and compared the features of NoSQL data stores Dynamo, Bigtable and Cass and redeveloped by the pioneers in industry like Amazon, Google and Apache. Most of the key value stores available today borrow heavily from Berkeley. Google’s Bigtable automatically partitions and distributes data among multiple tablet servers of a cluster. The design and implementation of Bigtable has been adopted by open source projects like Hypertable, Hbase etc. Apache Cassandra integrates the full distribution and eventual consistency of Dynamo and follows the data model of Bigtable. NoSQL databases will not replace relational databases but instead they will become a better option for certain types of applications that are distributed, involve large amounts of data and need to be highly scalable.

REFERENCES