Performance Analysis of a Tree-Based Consistency Approach for Cloud Databases

Md. Ashfakul Islam[1], Susan V. Vrbsky[2] and Mohammad A. Hoque[3]
Department of Computer Science
The University of Alabama
Tuscaloosa, AL 35487, USA
[1]mislam@crimson.ua.edu; [2]vrbsky@cs.ua.edu; [3]mhoque@cs.ua.edu;

***Abstract*—Cloud storage service is currently becoming a very popular solution for medium-sized and startup companies. However, there is still no suitable solution being offered to deploy transactional databases in a cloud platform. The maintenance of ACID (Atomicity, Consistency, Isolation and Durability) properties is the primary obstacle to the implementation of transactional cloud databases. The main features of cloud computing: scalability, availability and reliability are achieved by sacrificing consistency. While different forms of consistent states have been introduced, they do not address the needs of many database applications. In this paper we present a tree-based consistency approach, called TBC, that reduces interdependency among replica servers to minimize response time of cloud databases and to maximize the performance of those applications. Experimental results indicate that our TBC approach trades off availability and consistency with performance.**

Keywords- Database as a Service, interdependancy, tree-based consistency, auto scaleup, response time, update requests

I. INTRODUCTION

Cloud computing is becoming a very prevalent word in industry and is receiving a large amount of attention from the research community. Cloud computing provides on-demand access to computing, eliminating the need for users to maintain computing resources. Cloud computing shifts the location of various resources to the network to provide basic components, such as software, storage, CPUs, and network bandwidth as a service, by specialized service providers at a low unit cost. Users of these services need not worry about scalability and backups because the available resources are virtually infinite, and failed components are replaced without any service interruption or data loss.

Data applications are potential candidates for deployment in a cloud computing platform, especially when the amount of working data changes rapidly. Cloud computing allows the demand for cloud resources to be elastic, thereby, allowing cloud architectures to solve some of the following key difficulties faced in auto scale (automatic incremental) data processing [1]. Cloud computing eliminates the difficulty in acquiring the amount of resources required for on-demand data processing. Cloud computing can distribute and coordinate an on-demand workload on several servers and provision another server for recovery in case of server failure. It can auto scale up and down based on dynamic workloads. Lastly, cloud computing eliminates the difficulty in getting rid of all those resources when the job is done.

Transactional data management is the heart of the database industry. Nowadays, almost all business transactions are conducted through transactional data management applications. These applications typically rely on guaranteeing the ACID (Atomicity, Consistency, Isolation and Durability) properties provided by a database and they are fairly write-intensive. Many existing solutions for cloud databases are applicable only to analytical databases, which do not have strong guarantees about the ACID properties. The main challenge to deploy transactional data management applications on cloud computing platforms is to maintain the ACID properties without compromising the main feature of cloud platform scalability.

Data availability and durability are the main principles of cloud vendors. Any kind of data loss or service unavailability can destroy their business reputation. Service availability and data durability can be achieved by a certain number of replicas of the data distributed over different geographical areas [2]. Amazon's S3 cloud storage service replicates data across 'regions' and 'availability' zones so that data and applications can persist even in an entire location black out.

A consistent database must remain consistent after the execution of a sequence of update (Write) operations to the database. Any kind of inconsistent state of the data can lead to significant damage, especially in financial applications and inventory management, which is unacceptable. Most of the time consistency is sacrificed to maintain high availability and scalability in the implementation of transactional applications in cloud platforms. To maintain strong consistency in such an application is very costly in terms of performance.

When data is replicated over a wide area, maintaining consistency is complicated and time consuming issue. Some techniques make tradeoff between the consistency and response time of an update request. The authors in [8, 9, 10] develop a model for transactional databases with eventual consistency, in which updated data becomes "eventually" consistent. Other approaches use data versioning to keep a reasonable amount of delay. However, data versioning and compromised consistency are not favorable for transactional databases.

In this paper we present an approach for transactional cloud databases that does not sacrifice data consistency for performance. The remainder of this paper is organized as follows. In Section II we describe related work. In Section III

we present our consistency approach and provide an analysis of its performance in Section IV. Conclusions and future work appear in Section V.

II. Related Work

The first consistency model for databases presented in 1979 [3] provided the fundamental principle of database replication and a number of techniques to achieve consistency. Any replication appears as only one database to the user, so a data value is not returned until all replica copies can return the same value.

The consistency model of databases remained relatively static until the increase in databases that were highly distributed and replicated over a network. The CAP theorem is presented in [7], states that at most two out of three properties can be achievable from data Consistency, data Availability, and data Partitions. In a cloud platform, data is usually replicated over a wide area. As a result, only data consistency and data availability remain between which a system can choose. Cloud vendors always focus on high availability, so the 'C' (consistency) part of ACID is left to be compromised. We propose to minimize this compromise.

As an alternative, the eventual consistency model is presented in [4]. Different types of consistency are presented in this model, including the typical strong consistency, whereby after an update operation all subsequent access will return the updated value, and weak consistency, in which the system does not ensure that subsequent accesses will return the updated value. The focal point of this model is eventual consistency, which is a specific form of weak consistency in which the system ensures that if no updates are made to the object, all subsequent access will "eventually" return the last updated value. A number of variations of eventual consistency are proposed. Eventual consistency is adopted by many analytical data management solutions in clouds. We consider only strong consistency in our approach.

The authors in [5] develop a model of how to use Eventual Consistency in a transactional database. They propose that the degree of consistency may differ by trading-off between cost, consistency and availability. A new transaction paradigm named Consistency Rationing is also presented in [5] that allows designers to define different levels of consistency guarantees on the data and automatically switches levels of consistency guarantees at runtime. While the authors propose this new model for trading off consistency and performance, they do not discuss how to achieve this.

Additional research needs to be done to address the issue of consistency of transactional databases in a cloud. In this next section we propose an approach that involves maintaining consistency by minimizing the inconsistency and maximizing the performance in an efficient manner.

III. Consistency Approach

According to many researchers [6], services offer by cloud vendors are very suitable for small, startup companies especially those who are not ready yet to spend a large amount of capital to deploy their own infrastructure. Walker et al. [6] present an equation to allow a company to decide whether to deploy their own infrastructure or lease resources from a cloud. According to his research, a medium-size enterprise should always lease resources from a cloud. One single server is able to serve the regular workload of a medium-sized company's transactional database. But to ensure high reliability of transactional databases and the high availability offered by cloud vendors, the database should be replicated over multiple sites. Cloud vendors should also be prepared to auto scale up to handle an increase in workload.

In this paper we present a consistency approach for transactional cloud databases for medium-sized companies where a typical database can be handled by a single server, and reliability and consistency are the main concern. We note that a Write operation updates a data item and differs from an append-only approach that can be resolved by creating another version of a data item. We base our new approach on the approach proposed in [11], which considered only the reliability of the server and network path. Our new extended tree-based consistency approach, called TBC, is different in the following ways. First, we consider multiple performance factors. Second, we also limit the maximum number children that each parent can have in the tree to minimize the effect of interdependency on performance. Lastly, no performance analysis of the work in [11] was given and we present performance results in this paper.

A. System Description

Our TBC strategy works as follows. We assume that there are multiple replicas of data in the database, each associated with a different replica server. Based on information maintained about each replica server, a tree is constructed that maximizes the performance. The tree is constructed so that certain replicas (nodes in the tree) can be identified as copies available for a Read operation that are always consistent. Similarly, a node in the tree is designated as responsible for ensuring consistency after a Write operation.

Our TBC strategy requires the inclusion of the following two components in a cloud database system: the Controller and Database Replicas. (See Figure 1.)

i) Controller: There may be two or more controllers in this system. The tasks of the controller are to: build a tree with the given criteria, maintain periodic communication with all replicas, handle server failure, integrate and synchronize additional or recovered servers with the system, and collect and maintain service logs for future tree building. It also makes decisions about database partitioning and load balancing.

ii) Replica servers: There are several replicas of the data in this system and each is stored at a different replica server. Replica servers store the data and perform all operations required to complete the transaction and any other database operations. One replica is selected from among the replicas based on some criteria as the primary replica to serve update requests from the user.

B. Performance factors

Building a tree from available replica servers necessitates

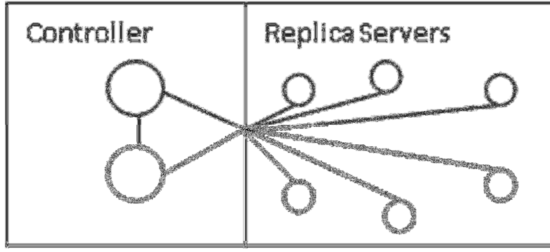


Figure 1: Communication between Controller and Replicas

considering various features of our system in order to evaluate the servers. Our main goal is to maximize performance, so we have to find out the main causes behind performance degradation. A cloud computing infrastructure is almost always built with heterogeneous components. Heterogeneous characteristics, such as the time required for disk update, workload of the server, reliability of the server, time to relay a message, reliability of network, or the network load of a cloud infrastructure can cause enormous performance degradation. We introduce a new evaluation metric, called PEM (Performance Evaluation Metric), which takes all these factors into consideration. Each performance factor of PEM has a weight factor that indicates its importance relative to the other factors. Weight factors can have positive or negative values based on their impact on performance. The performance factor is multiplied by its weight factor to capture its effect on the response time of an update request. A larger PEM indicates better performance. Obviously, the importance factor can vary per system. If the response time of an update request in a system depends on *n* performance factors, then Equation (1) describes the formula for calculation of the evaluation metric PEM for that system, where pf_j is the jth performance factor and wf_j is the corresponding weight factor of the jth performance factor.

$$PEM = \sum_{j=1}^{n} (pf_j * wf_j) \qquad (1)$$

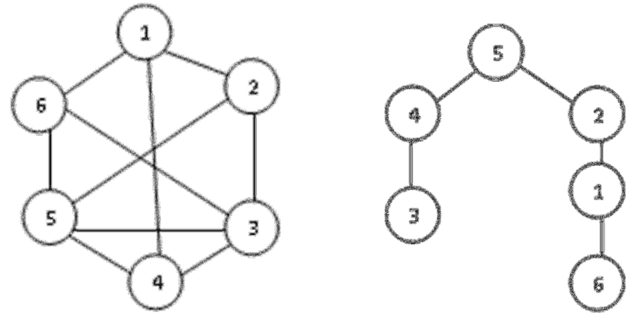
C. Building the Tree

It is the controller's responsibility to calculate the evaluation metric PEM and prepare the consistency tree accordingly. The tree building process requires the following steps.

i) Preparing the connection graph: The controller will first prepare the weighted connection graph G (V, E). Each replica server is a vertex, and therefore, is a member of set V. All direct connections among replica servers are considered as edges and members of set E. The matrix M_j contains the performance factor pf_j of each edge (connection path) in G and the vector R_j contains the performance factor pf_j for each vertex (server) in G. Figure 2(a) illustrates a connection graph with six vertices. In figures 2(c) – 2(f) the performance factors in M_1 and R_1 are associated with pf_1 (path reliability) and M_2 and R_2 are associated with pf_2 (time delay). The values of pf_j for the edges and vertices are predefined. Weight factors $wf_1 = 1$ and $wf_2 = -.02$ are used in this example.

ii) Selecting the root of the tree: The controller will calculate PEM values for servers from the performance factors (pf) of servers and corresponding weight factors (wf), and chooses as the root the server who has the maximum value. This decision can be affected by the load balancing of the system which will be explained in Section III. F.

iii) Preparing the consistency tree: The consistency tree will be prepared from the weighted connection graph. The controller will apply Dijkstra's single source shortest path algorithm with some modification ($O(n^2)$). Dikjstra's algorithm is modified by imposing a constraint on the maximum number of children (*Max_Child*) for a parent node. The maximum number of children that a parent node can have depends on the tradeoff between reducing interdependency to minimize the response time of an update operation and maximizing the consistency and reliability of a database. The root of the tree (returned by first call to Max_PEM) will be selected as the single source. Algorithms 1-2 below will find the suitable path to every replica server to maximize performance and all paths together will form the consistency tree. Similar to Dijkstra's Relax function, our Relax function tries to find a path better than the current best path. If any node u reaches the *Max_Child* limit, the Reweight function will find the alternative best path for those nodes of Q whose immediate parent node is u. Using the values in figures 2(c) - 2(f), we set the *Max_Child* = 2 for our example. After applying the algorithm, node 5 is chosen as the root of the tree. The resulting tree appears in Figure 2(b).



(a): Interconnection Graph (b): Consistency tree

1	.9	0	.8	0	.9
.9	1	.8	0	.9	0
0	.8	1	.9	.6	.7
.8	0	.9	1	.9	0
0	.9	.6	.9	1	.7
.9	0	.7	0	.7	1

(c): Path Reliability M_1

0	25	0	20	0	15
25	0	26	0	17	0
0	26	0	15	24	20
20	0	15	0	19	0
0	17	24	19	0	22
15	0	20	0	22	0

(d): Time Delay M_2

.98	.98	.91	.93	.99	.96

(e): Path Reliability R_1

50	20	60	30	10	30

(f): Time Delay R_2

Figure 2: Example of a tree calculation

The granularity of the replicas upon which the tree is based can be the entire database, a table or a subset of tables in the database. Similarly, a tree can be recalculated periodically to reflect any changes in the performance factors or its recalculation can be triggered by a specific event, such as a failure or exceeding some threshold.

```
Algorithm 1 : Modified_Dijkstra (G, V, M, R, W)
// G is connection graph, V is vertex set
//M is 3D array, all pf values of each edge
//R is 2D array of pf values of each vertex
// W is 1D array of wf values regarding all pf
s ← Max_PEM (V, R, W)       // return node with max PEM
Initialization(V, s)          // similar to Dijkstra's algorithm
S ← Φ             // set of nodes whose best path are determined
Q ← V             // priority queue of nodes
while Q isn't empty
    u ← Max_PEM (Q, R, W)
    S ← S U u
    remove u from Q
    num_child[ Π [u]] ← num_child[ Π [u]] + 1
    if num_child[Π [u]] = Max_Child then
       Reweight(Π [u], S, Q)       // Π is the immediate parent
    for each vertex v ε Adj[u]
       Relax(u, v, M, R, W)      //update best path estimation
    end for
end while
END Modified_Dijkstra
```

```
Algorithm 2: Reweight (p, S, Q)
for each v ε Q and  Π [u] = p
     for each pf
          if pf to be minimized then
               P[pf][v] ← ∞
          else pf to be maximized then
               P[pf][v]  ← 0
     end for
     for each u ε S and u ≠ p and u ε Adj[v]
          if num_child[u] < Max_Child then
               Relax(u, v, M, R, W)
     end for
end for
END Reweight
```

D. Update Operation

The controller informs all replica servers about its immediate descendants. Each replica is responsible for maintaining the updates of its own descendants. Each update operation has a unique sequence number associated with it. Each replica stores two state flags:

i) Partially consistency flag: The last updated operation sequence number is stored as the partially consistent flag. A partially consistent flag is set using a top-down approach. In other words, if we traverse from a node (replica server) to the root of the tree, for a particular update, all these nodes have the same sequence number stored as their partially consistent flag.

ii) Fully consistent flag: A fully consistent flag is set by a bottom-up approach. A fully consistent flag is also an update operation sequence number. It indicates that any node that is a descendant of a sub-tree also has the same update sequence number stored as its fully consistent flag.

When the root receives an update request, it will store the request in a request queue. The update process at the root continuously monitors the queue. When the replica server is available, an update request is dequeued to initiate an update operation.

An update operation is done using the following four steps.

1. An update request will be sent to all children of the root
2. The root will continue to process the update request on its replica
3. The root will wait to receive confirmation of successful updates from all of its immediate children
4. A notification for a successful update will be sent from root to the client

The root will update its partially consistent flag with the corresponding update sequence number after completion of its update operation.

Update operations at non-root nodes are handled by two different processes. After receiving an update request one process initiates the required steps to update it, notifies its parent and stores the update sequence number as its partially consistent flag. The other process stores an update request in a queue, sends the request to all its children, waits until all of its children reply with a successful update before its sends the next update request from queue.

Storing the fully consistent flag is initiated by leaf nodes. When a leaf node is done with an update request, it stores the sequence number as both the partially consistent flag and fully consistent flag. It then informs its parent about storing its fully consistent flag. All intermediate nodes will notify their parent when they receive notification from all of their children.

E. Maintaining Consistency

A request for a Read to the data will always return a value from either the replica that is the root node or one of the immediate descendants of the root node, as these values will always reflect the most recent updates. The structure of the tree itself determines how many replicas can serve in this capacity. If all nodes are immediate descendents of the root node, then this is considered the *classic approach*. In the classic approach, all replicas must be updated before any Read can occur, so the response time is increased. Fewer immediate descendents of the root node will result in a decrease in response time for updates, but increase the workload on the root node and its immediate descendents.

The root node of the tree is responsible for making sure all of the replicas in the tree are updated once a Write operation is issued. Hence, in our approach, we consider one replica as responsible for update operations and several replicas to manage durability and reliability. In the next section we discuss increasing the number of replicas responsible for update operations.

F. Auto Scale up

One of the key features of a cloud platform is to prevent the user from having to worry about the variable nature of the workload. Cloud systems should be able to handle an increased workload automatically. If the workload exceeds the threshold limit of the root node, the controller takes the initiative for handling the excess load. This process is done by following steps:

i) Controller uses the results from the evaluation of the PEM metric to identify the server that scored the second best among all replica servers as the next possible root. Another tree is then built by same process described in Section III.C.

ii) Controller makes a decision about which portion of an update request should be shared with another root. Obviously, an update request on the same data or highly related data must not be sent to a different root. To address consecutive Read/Write requests, the data can be partitioned based on such factors as: relationships between the data, nature of the data, importance of the data and response time tolerance.

As the load continues to increase, the controller follows the same steps as long as the load exceeds the total capacity of the current system. In that case, the required number of new nodes will be launched. Details of that process and partitioning for serializability will be included in next stage of our research.

G. Failure Recovery

We assume any server or communication path between two servers can go down at any time. It is the responsibility of the controller to handle such a situation. There are two types of situations:

i) Primary server is down: The controller maintains continuous communication with the primary server. If the controller is able to determine that the primary server is down, it will communicate with the immediate descendant of the root server concerning its partially and fully consistent flag. If both flags are the same, the controller will choose a new root from all servers. If both flags are not the same, then the controller queries the servers to find a server with the latest updates. It is an immediate descendent which has the same sequence value as the controller's for its partially consistent flag. From among these latest updated servers the controller will find the root. The connection graph is then reconfigured with all available servers and the consistency tree is built using the strategy described in Section III.C.

ii) Other server or communication path is down: If the unresponsive behavior of an immediate descendant is reported at any time by an intermediate node, the controller tries to communicate with that server. The controller will fail to communicate with the server if it is down. The connection graph is then reconfigured without that server, the consistency tree is built with the same root and all servers are informed about the new tree structure. In the event the communication path is down, the controller can still communicate with the server via another path. The controller will then reconfigure the connection graph including the server and build the consistency tree as described in Section III.C.

The TBC approach reduces interdependency between replicas because of the fact that a replica is only responsible for its immediate descendants. When a replica is ready to perform an update, it only has to inform its immediate descendants, receive acknowledgements from them and update its data. Hence, the transaction failure risk due to interdependency is reduced regardless of the network load, bandwidth and other performance factors.

IV. EXPERIMENTAL RESULTS

A. Experimental Environment

We have performed TBC experiments on a green cluster called *Sage*, built at the University of Alabama. All 9 nodes of sage are composed of an Intel D201GLY2 mainboard with 1.2 GHz Celeron CPU, 1 Gb 533 Mhz RAM, 80 Gb SATA 3 hard drive. All nodes are connected to a central switch on a 100 Megabit Ethernet in a star configuration, using the D201GLY2 motherboard on-board 10/100 Mbps LAN. The operating system is Ubuntu Linux 7.10. Shared memory is implemented through NFS mounting of the head node's home directory.

A standalone java program runs on each cluster to perform the update operation, which is defined as a certain amount of delay in execution. When the server receives an update request, it will wait a certain amount of time t before sending a reply indicating a successful update, where $t = op + wl$ (op represents actual time required for a disk operation and wl represents a delay due to the workload of the disk and operating system). A uniform random variable is used in the calculation of wl. The time duration between two consecutive update operations is determined by a Poisson random distribution. All times are measured in Milliseconds.

In our experiments, every 20 ms a new update request will arrive with an arrival rate of $\lambda = 0.1$. The default update time op is 10 ms. The workload overhead wl is uniformly distributed between 0 and 50. The fastest server is five times faster than the slowest server (server heterogeneity).

B. Performance Metric

In our experiments, we consider a thousand update requests sent by clients for each case. A comparison between the TBC approach and the classic approach described in Section III is the main focus of our experiments. We measure the elapsed time between when an update request is sent and when the primary server sends a reply to the client indicating a successful update. We compute the average response time in order to make the comparison between the classic approach and our TBC approach, as well as to determine the effect of some of the system factors on performance.

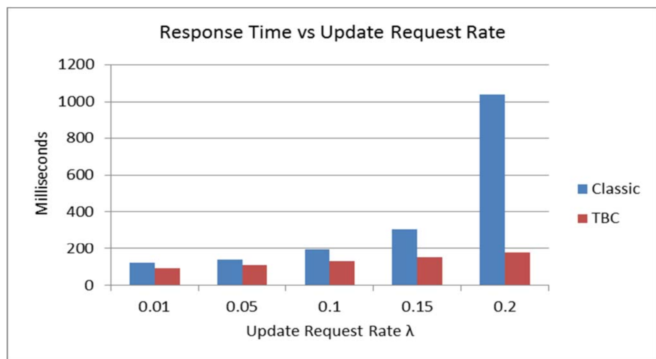
C. Effect of update request rate

We first study the effect of different arrival rates of update requests by changing the value of λ. We use the other parameter values as default values. As shown in Figure 3(a), the response time increases as the request arrival rate increases

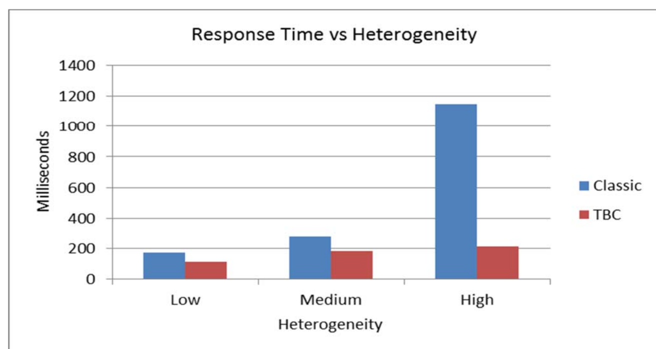
from 0.01 to 0.2 for both approaches. The TBC approach ranges from a low of 92 for λ = 0.01 to a high of 179 for λ = 0.2. The classic approach ranges from a low of 121 for λ = 0.01 to a high of 1040 for λ = 0.2. The performance of the TBC approach increases linearly, while the increase in the classic approach is approaching exponential.

D. Effect of heterogenity of servers

In our next experiment we vary the heterogeneity of servers by considering three categories of heterogeneity: low, medium and high. To model low heterogeneity, all replica servers are similar in response time, whereby the fastest server is two times faster than the slowest server. In order to model medium heterogeneity, the fastest server is five times faster than the slowest server. For high heterogeneity, the fastest server is nine times faster than the slowest server. Obviously, the fastest server was selected as the primary server in the classic approach and as the root in the TBC approach. The tree was formed according to algorithm described in Section III.C.



(a): Effect of update request rate



(b): Effect of heterogeneity of servers

Figure 3: Effect on Response Time

Figure 3(b), illustrates that as the degree of heterogeneity of the servers increases, the response time increases. The results also demonstrate that despite the degree of heterogeneity of the servers, the TBC approach has a much faster response time which ranges from a low of 112 to a high of 212. The classic approach response time ranges from 173 to 1147. The response time of the TBC approach doubles from the low to high heterogeneity, while the response time of the classic approach increases more than five times. Though we did expect the response time of the classic approach to be higher than the TBC for high heterogeneity and a high arrival rate (Figure 3(a)), we did not anticipate the dramatic amount of this increase.

V. CONCLUSIONS AND FUTURE WORK

Maintaining consistency, availability and high throughput among replica servers is a key issue in cloud databases. Many highly concurrent systems tolerate data inconsistency across replicas to support high throughput. However, in many systems, it is still important to maintain data consistency. Highly unreliable systems can face transaction failures for interdependency among replica servers, but for some of these systems it remains important to maintain data consistency despite such failures. In this paper we have proposed a TBC approach that reduces the interdependency among replica servers. Experimental results indicate that our TBC approach trades off consistency and availability with performance. The TBC approach performed well in terms of response time despite the heterogeneity of the servers or the arrival time.

The goal of our work is to provide low cost solutions to ensure data consistency in the cloud. Our next plan is to implement abort, commit, and rollback protocols in our TBC approach for transactional data management in clouds. Finding suitable values for the importance factors is also included in our future work. Minimizing the inconsistency window length will also be addressed.

VI. REFERENCES

[1] Jinesh Varia. Cloud Architectures. White paper of Amazon. 2008.
[2] Daniel J. Abadi. Data Management in the Cloud: Limitations and Opportunities. Data Engineering 2009
[3] B. G. Lindsay P. G. Selinger C. Galtieri J. N. Gray R. A. Lorie T. G. Price F. Putzolu B. W. Wade. Notes on Distributed Databases. July 1979.
[4] Werner Vogels. Eventually Consistent. Communications of the ACM, 2009.
[5] Tim Kraska Martin Hentschel Gustavo Alonso Donald Kossmann. Consistency Rationing in the Cloud: Pay only when it matters. VLDB 2009.
[6] Edward Walker, Walter Brisken, Jonathan Romney. To Lease or not To Lease From Storage Clouds. IEEE Computer April 2010.
[7] Eric Brewer. *Towards Robust Distributed Systems.* Annual ACM Symposium on Principles of Distributed Computing. July 2000.
[8] Matthias Brantner, Daniela Florescuy, David Graf, Donald Kossmann, Tim Kraska. Building a Database on S3. SIGMOD 2008
[9] Zhou Wei, Guillaume Pierre, Chi-Hung Chi. Scalable Transactions for Web Applications in the Cloud. Euro Par 2009.
[10] Sudipto Das, Divyakant Agrawal, Amr El Abbadi. ElasTraS: An Elastic Transactional Data Store in the Cloud. HotCloud'09
[11] Md. Ashfakul Islam and Susan V. Vrbsky, “A Tree-Based Consistency Approach for Cloud Databases,” 2nd IEEE International Conference on Cloud Computing Technology and Science, CloudCom 2010.