TOWARDS TRANSACTIONAL DATA MANAGEMENT OVER THE CLOUD

Rohan G. Tiwari
Database Research Group, College of Computing
Georgia Institute of Technology
Atlanta, USA
rtiwari6@gatech.edu

Shamkant B. Navathe
Database Research Group, College of Computing
Georgia Institute of Technology
Atlanta, USA
sham@cc.gatech.edu

Gaurav J. Kulkarni
Database Research Group, College of Computing
Georgia Institute of Technology
Atlanta, USA
gauravjkulkarni@gatech.edu

Abstract— **We propose a consistency model for a data store in the Cloud and work towards the goal of deploying Database as a Service over the Cloud. This includes consistency across the data partitions and consistency of any replicas that exist across different nodes in the system. We target applications which need stronger consistency guarantees than the applications currently supported by the data stores on the Cloud. We propose a cost-effective algorithm that ensures distributed consistency of data without really compromising on availability for fully replicated data. This paper describes a design in progress, presents the consistency and recovery algorithms for relational data, highlights the guarantees provided by the system and presents future research challenges. We believe that the current notions of consistency for databases might not be applicable over the Cloud and a new formulation of the consistency concept may be needed keeping in mind the application classes we aim to support.**

Keywords-cloud computing; transaction management; database update; queue manager; transaction manager; transaction schedule; version; replication; Database as a Service.

I. INTRODUCTION

Data stores deployed over the Cloud have to be resistant to system and network failures and provide replica co-ordination. In this paper, we are targeting a fully replicated database. A replica refers to a complete copy of the data. Fault-tolerance and availability is ensured by replicating (or caching) frequently used data across multiple locations. The downside of this is that co-ordination of the various replicas is a cause for overhead, possibly reducing system availability. In current Cloud environments, immediate consistency is not given high priority resulting in poor support for true online transaction processing.

Most distributed data stores are scalable for large amount of data. These systems are highly available for data management but do not support general serializable transactions. Supporting serializable transactions over replicated and widely distributed systems is expensive [18, 20]. Hence, to suit the requirements of web-based applications, designers of these systems have sacrificed the ability to provide scalable transactions. These systems have stringent performance requirements and provide eventual consistency guarantees [18]. In short, these systems use algorithms and access methods which compromise on consistency guarantees for availability and scalability. Also, it is relatively easier to deploy systems over the Cloud that are comparable to traditional databases when the major proportions of operations performed on the data are analytic [1].

Examples include systems like Bigtable [10], PNUTS [11] and Dynamo [9] which are highly scalable. They use a key-value type data-store and allow single key accesses. They provide minimum consistency guarantees for multi-key accesses. However, these systems cannot match the consistency and recovery mechanisms guaranteed by traditional databases. Moreover, unlike traditional databases, these data stores do not have the concept of referential integrity (foreign keys). Other systems like Sinfonia [12] and Chubby [13] can be used in the design of scalable distributed systems. Sinfonia [12] introduces the minitransaction primitive. Lomet et al [17] and Brantner et al [14] suggest database design techniques but stop short of giving elasticity guarantees over the Cloud. ElasTras [19] does not guarantee consistency across data partitions.

In three-tier applications, the scaling of the entire software stack is limited by the database layer. This is because the inability of the database layer to scale well is a bottleneck that limits the growth of applications sizes as a whole. Applications like airline reservations and employee data systems have been the bread and butter of database industry - but these applications need an RDMS and transaction processing capabilities. If these applications are to be deployed over the Cloud, it is essential to come up with a scalable consistency model over a scalable data store for the Cloud that supports referential integrity and a transaction mix with a majority of updates. This boils down to a problem of deploying databases over the Cloud without compromising on any of their existing features.

We aim to deal with elastic and scalable transaction management when databases are deployed as a service over

the Cloud, without any loss of functionality. For DaaS (Database as a Service), a scalable transaction management paradigm is necessary; one which would work well even when the majority of the transactions involve updates. Hence, it is necessary to decide the level of consistency needed for data management over the Cloud. This consistency guarantee should span across all partitions and over all replicas. This is the notion of consistency across all partitions or **global consistency**. If the proposed goal of global consistency is achieved, a Service Level Agreement (SLA) for the model is essential to measure the performance of the model. This is imperative because the methods of data access and the demands of each application on latency would not be the same. For some applications, transactions might involve majority of accesses in a single data partition but for others this might be unpredictable and would require global synchronization. In short, this would help in determining and enumerating the classes of transactions that could be supported.

In the above DaaS scenario, we try to work a level above the data storage layer but are really tightly coupled to it. A method is required to deal with data objects being modified by the client and to reconcile different versions or to keep a single updated version. Two types of Consistency models are considered as a basis for the development of the model:

- Deferred Consistency model
- Immediate Consistency model

Immediate consistency is the real threat to availability for data stores over the Cloud. A deferred consistency model could involve the use of a write back cache or a write back data log or data versions. This model would ensure an eventually consistent distributed non-relational database as it does now. The advantage of immediate consistency is that the data objects accessed by the client would always be consistent. For example, a user may make some change to his shopping cart and this change might go to an older version of his shopping cart in the deferred model but in the real-time model this is not possible. Immediate consistency model does not need reconciliation of versions. Immediate consistency with its strong guarantees of synchronous replica consistency can be compared to the consistency guarantees offered by traditional databases.

Our work is for a fully replicated data store. We aim for a scalable, elastic model which can handle transactions for this system and also provide the desired consistency guarantees for a typical application like a hotel-room reservation system implemented on a fully replicated scalable data store.

II. General Architecture of the System

A **node** (or a replica) is a virtual machine instance. Each node has a complete copy of the data. The system is distributed across many nodes. A bunch of nodes taken together form an Interest Group. At this stage, each node has only one table without any referencing constraint. In section V, this is extended to multiple tables with referencing constraints.

Each node consists of the following components:

1. **Data Objects**: Data Objects are simply tables stored as .db files on Amazon S3. Each data file contains a single table. A table has many attributes and rows (like a matrix). Each table row has a unique row number and contains the timestamp of the transaction which performed the last update. Data Objects are fully replicated over the network. Each row in the table has an entry for the last update time (or the value of the global time stamp of the transaction which last updated this row).

2. **Update Queue**: The Update Queue is the recovery mechanism present at each node. The transaction requests from the Transaction Managers (which are server-side virtual machine instances) are added to the Update Queue at each replica. It uses Amazon's Elastic Block Storage.

3. **Queue Manager**: The Queue Manager handles the Update Queue and runs the transaction with the smallest time-stamp. After successful commit, this transaction is de-queued.

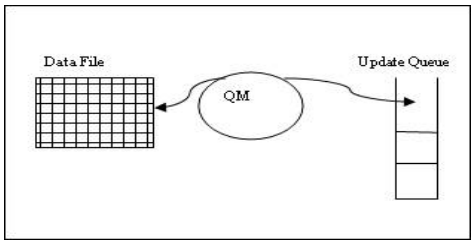

Figure 1: The components of a node (or a replica).

Other components of the system include:

Interest Group (IG): An IG is a group of replicas in which all (or a majority of replicas) have the most recent global timestamp. Hence with the current setting (one table per replica), there is only one IG. This will change in section V when referential constraints are considered.

The IG is used for servicing client requests. The IG is the core of the system and the replicas outside the IG act as backups to the replicas in the IG. IG is dynamic- any change (replica failure or replica boot up) in the network can lead to addition of replicas to the IG. The IG is reflective of the state (number of running nodes) of the network. The size of an IG has a great impact on client waiting time. The maximum size of an IG is currently kept at N/2 and the minimum size at 2, where N is the total number of working replicas in the network.

The necessary condition for an IG to exist is that it must have the minimum number of nodes. The size of the IG mainly depends on the transaction mix viz. the proportion of reads and writes. The concept of an IG will be clearer in sections III and IV. More on IG size in section VII.

Transaction Managers (TMs): The TM is a server-side virtual machine instance. A TM is very closely associated with the replicas. The TM is responsible for maintaining data related to the IG and also ensuring parallel execution of update operations on the IG nodes. One TM is designated as the **Master TM (MTM)**. TMs have meta-data regarding table references. Read/write requests from the client are received by the TMs. TMs then issue these requests to the IG following the algorithm described in section III. Before sending any transaction to any IG, a TM is required to check

if any transaction with a lower time-stamp is being processed in any other TM. If it is, then this TM must wait till its turn to send the transaction comes.

Nodes in the system are numbered starting from 0. Each TM contains a bitmap called as an **IG map**. The IG map includes an entry for each node in the system. If X denotes the IG map and i denotes an index into the IG map then:

$X_i = 1$ if node i is a part of the IG

$X_i = 0$ otherwise.

All the components of this system are deployed on the Cloud.

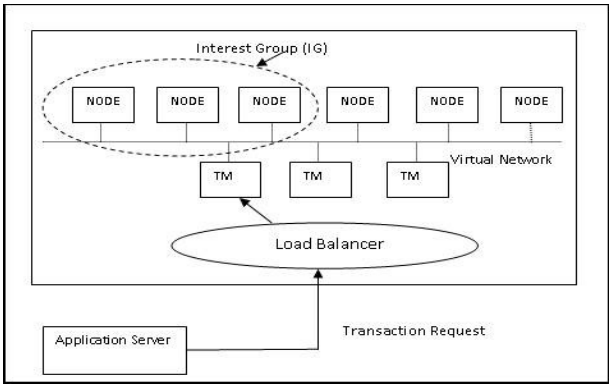

Figure 2: A snapshot of the system showing an IG and 2 TMs. There is exactly one table on each node and it is fully-replicated.

III. CONSISTENCY ALGORITHM

This section presents the consistency algorithm for a single table fully replicated across all the nodes. A few terms need to be defined:

Version: The global time stamp value assigned to the data file at a given replica.

Current replica: A replica with the most recent version of the data file. It is possible to have more than 1 current replica.

The client must submit an estimate of the transaction workload initially. This is used to determine the number of TMs and number of nodes (total number of virtual machine instances) to be launched on each hypervisor [21]. After this initial startup phase, the system scales with the transaction workload following the pay-as-you-go model. Initially, all the replicas start with the same version number- Version 0. All the IG maps are reset. The replicas are randomly grouped together into an IG by the MTM. The starting size of the IG is specified by the application developer. The MTM then sends its IG map to other TMs for synchronization.

A client request can go to any TM, thus ensuring that no TM is overwhelmed by more requests than it can service (load balancing). The TM which receives the client write request multi-casts the request to all the nodes in the IG. The TM is required to wait for acknowledgment from only the current replicas in the IG. The nodes in the IG forward the write requests to the other replicas after sending their acknowledgements to the TM.

Inside the IG, the updates are applied in parallel to the replicas. Hence, the amount of time the client has to wait is bounded by the slowest replica in the IG. This makes the write latency of the IG equal to the write latency of the slowest replica in the IG currently. Write latency of a replica for a TM is the difference between the time at which a TM sends an update to this replica and the time at which this TM receives the acknowledgement. Write latency is variable due to the dynamic nature of the IG. Suppose the IG has m nodes and the write latency of i^{th} node is t_i and the write latency of the IG is T_{IG}. Then,

$T_{IG} = max\ (t_1, t_2, t_3, \ldots, t_m)$.

The updates are performed on those replicas which are currently not 'in use'. These updates go into an Update Queue at each replica. In each replica, the Queue Manager de-queues the update with the smallest timestamp from the Update Queue. If at least one of the rows affected by this update is 'in use' then the update must wait in the Update Queue. This ensures that the updates at all replicas are applied consistently and in order. Each queue acts as a log. Unlike the write operation, there is no multi-cast needed for a read operation. The data can be read from any current replica in the IG. Since there are two types of operations - read and write, this leads to following dependencies of operations on a single table row:

Read after Write: A read coming into the IG when a write operation is being performed. The read can be issued on any replica on which this write has been performed and committed.

Read after Read: A read coming into the IG when a read operation is being performed. The read can be issued on any replica. The TM sends a read request to only one replica.

Write after Read: A write coming into the IG when a read operation is being performed. The writes can be issued on all the replicas except on which this read is being performed. The write on this replica is issued after this read is done.

Write after Write: A write coming into the IG when a write operation is being performed. The writes can be issued on all the replicas except on which this write is being performed. The write on this replica (or replicas) is issued after this write is committed.

TMs put each read/write request into the Update Queue at each replica in the IG. The Queue Manager at each replica must keep track of these dependencies while issuing the transactions from the Update Queue. This algorithm ensures that whenever a client issues a read operation, a consistent and correct value is read from the data store.

IV. RECOVERY ALGORITHM

A failure is defined as a state in which a replica cannot service any client request at all. Partial failures are not considered.

When a node (inside or outside the IG) fails, a check is run by any one TM on the network and on the IG. This checks if the IG continues to have the minimum number of nodes and establishes if more nodes can be added to the IG if it is at less than the maximum capacity. This is called **re-evaluation of the IG** and is performed by the MTM. MTM sends the new contents of the IG map to the other TMs.

More than one node can be added to the IG after re-evaluation.

The re-evaluation of IG does not mean that the system is down. While a node is getting added to an existing IG, a client operation can still be performed on the remaining nodes in the IG. If operation was a write, it can be performed on the new node added to the IG only when its addition to the IG is complete.

The algorithm presented in section III allows for a recovery scheme to handle failures. This is elaborated by considering the following possible events.

When node(s) inside the IG fails: The IG is re-evaluated. If no node can be added to the IG and the IG has less than the minimum number of nodes, then the system is unavailable and does not service any client request until a new IG is formed.

When node(s) outside the IG fails: The IG is re-evaluated.

When node(s) comes up: The IG (either existing or see if a new one can be formed) is re-evaluated. This node can become a part of the existing IG or forms a new IG.

The above scenarios can be thought of as interrupts to the system. Each of which will lead to re-evaluation of the IG.

Queue Manager: Each replica has a Queue Manager. As discussed earlier, the updates at each replica are first added to a queue, before applying to the respective table row. This queue called the Update Queue which ensures that the updates are applied in order of time. The queue also serves as a log of transactions which can be used when the replica comes up after failure.

Let there be two replicas A and B. The version of the data file at B is V_1 and that at A is V. If V_1 is the most current version of the file, then B's update queue would be empty. If it is not the most current version of the file and suppose V is the most current version of the file then V_1 would need, say, n updates to reach V or simply $V = V_1 + n$. Now, suppose B fails and is down for m units of time. In this time, the version at A would advance from V to a new version V_2. Say there were n_1 updates done during this time. Now the current version at this time is V_2 and V needs to have $(n + n_1)$ updates to become V_2 when B comes up.

Now, the question to be answered is: From where would B get the updates after it comes up? To go to the version V_1, it simply needs to de-queue all the updates in its update queue when it failed. Then to go to V_2 it needs to ask A to send it the modified tuples or all the tuples. This being an operation conducted infrequently, should not affect the performance of the system greatly. If this is done, then B does not need to use its update queue at all. The updates from the TM(s) would still be coming to the replica B and would be added to its update queue. So if B comes up at time t, then it asks any replica in the IG for the most updated data. Until the file at B comes to the current version at the time t, the Queue Manager does not apply the updates to the table at B.

When the IG fails (which implies that no node outside the IG can be added to the IG after IG re-evaluation), there is no need for the transmission of updates from other nodes as the IG nodes would already be current when they come up. A more general form of this is when the entire system fails; each replica which is not current has to do what B does in the above example. But these types of failures would be unlikely.

When a node(s) executing a data operation fails: Reads can be forwarded to another node with the current version of the replica. This is done by the TM, when it does not get any acknowledgement from this node. In case of writes, the time stamp of the transaction is read from the front of the update queue and a local undo operation is executed on the corresponding row in the table.

When a TM fails: Each Cloud node has one TM as a virtual machine instance and any number of virtual machine instances as nodes or replicas. When a TM fails, a new virtual machine instance is started and it has the same state as the failed TM. It continues processing from the time the TM had failed.

So when the system is down when either all replicas are down or there is no functional IG in the network (There might be current replicas active even without an IG in the network. This is a tradeoff between what the minimum size of an IG should be and also the maximum size. It is application dependent and it is best if the developer selects it.)

The algorithm is dynamic and reflects the current state of the network. The size of the IG is a function of the number of working nodes. As long as the IG is up and running, the client finds the system available and reading consistent values. The client latency depends on the current size of the IG currently. Since the size of the IG is dynamically dependent on the number of working replicas it is safe to say that the client latency also depends on the number of working replicas. Clients can get concurrent reads and writes which happen in parallel. When a replica in the IG fails, the IG is re-evaluated and possibly more replicas are brought into the IG hence the performance does not suffer for each IG replica failure. The replicas outside the IG (which are up and running) also keep on applying the updates but the TM does not wait for acknowledgments from them. But they are not left behind as the updates are sent to them as well. The system keeps evolving and ensures immediate consistency for the IG and eventual consistency for replicas outside the IG.

V. Extending to Multiple Tables with Referencing

Each node now has more than one table. Each table is fully replicated over each node; this means that each node has exactly the same data. The attributes of these tables can have referential constraints between them.

An update to a single row in one table may lead to updates of referenced rows in other tables. Consider a typical example of EMPLOYEE and DEPARTMENT tables. Suppose the primary key of DEPARTMENT table is deptID and it references the deptID attribute in the EMPLOYEE table. Any change in the deptID value in DEPARTMENT table must be consistent with the referenced attribute in the EMPLOYEE table. For example, suppose the deptID is changed from 4000 to 5000, then all the corresponding

EMPLOYEE tuples in the EMPLOYEE table must be updated. This is a simple example of referential integrity.

The IG concept is extended to solve this problem by creating separate IGs for each table. So, the EMPLOYEE and DEPARTMENT tables would have their own separate IG. A node can be a part of more than one IG but copies of one table constitute a single IG (See figure 3 below). In this case, when the data is updated in the EMPLOYEE IG a separate update is launched on all the replicas in the DEPARTMENT IG. But when the EMPLOYEE table is being updated (in its IG) no DEPARTMENT table to which this update has not been applied yet can be updated or read on any node This would lead to inconsistency.

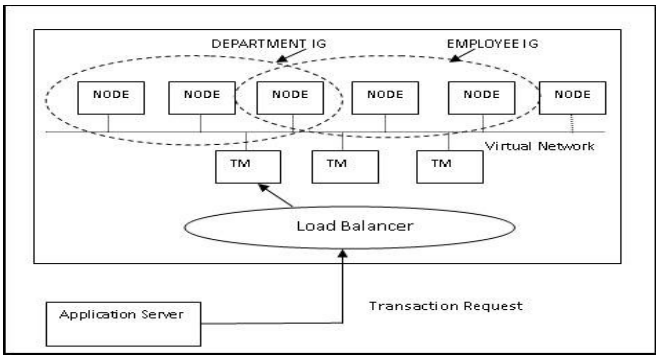

Figure 3: The system showing two IGs. Note one node is common to both the IGs.

To avoid this inconsistency an independent locking scheme is needed which would lock the tuples from the various tables that would be updated as a result of our current update. As an example, if the deptID is changed from 4000 to 5000 then all EMPLOYEE tuples and all DEPARTMENT tuples with the deptID 4000 would have to be marked locked. These tuples have to be locked by the TM before the updates are sent to the respective IGs. To make clear why this is important, assume that query updating the deptID from 4000 to 5000 is in progress and currently the department table IG is being updated and there is a query to update the salaries of all employees (in the EMPLOYEE table) with the DEPARTMENT ID 5000. There will not be any EMPLOYEE tuple with DEPARTMENT ID 5000 yet (as the the deptID from the previous update has not been applied to the EMPLOYEE table). So this query would not update anything in the EMPLOYEE table. This implies that until all the required tables are not updated no operation can be allowed to go forward on any of the related tables. It may be allowed on those tables which have already been updated. This is a performance bottleneck.

VI. Maintaining Meta-Data About Referencing Constraints in the TM

To implement the scheme discussed in section V, some additional information is stored in each TM. Each TM has meta-data containing the referencing information about tables viz. which updates are cascaded. The TM makes a **transaction schedule** (the set of updates to be sent over the network). This schedule is a map (key-value pairs) and specifies which table and row have to be updated. Suppose an update is to be made on table T_i, row R_j by transaction T_1 with timestamp t_x (timestamp is the time at which the transaction is issued by the client). The schedule for T_1 would have the corresponding key entry:

(T_i, R_j, t_x)

The value part of the map has the update query to be executed. Whenever a TM gets an update transaction, it prepares a list of tables which need to be updated because of this transaction and creates the transaction schedule. This essentially prepares the TM for sending independent updates to each of the table IGs. But these updates are still a part of the main transaction, so it is important that the TM sends these updates together in time. After these updates are sent, this works as in the normal case (the single table case).

Consider the DEPARTMENT and EMPLOYEE table example. The above approach would lead to two independent update requests to be multi-cast to two different IGs from the TM. The problem arises due to network delay- the update for DEPARTMENT goes to the DEPARTMENT table IG and is added to the Update Queue of the member nodes and similarly the update for the EMPLOYEE table goes to the EMPLOYEE table IG. Now, say a read or a write request comes in and can enter between these two updates on the time line hence is ahead in the respective update queue. This may cause a problem. This is avoided by having each TM first verify that no transaction with a smaller timestamp is being processed at another TM before sending the transaction to the replicas in an IG.

Consider the EMPLOYEE and DEPARTMENT tables' example. The TM issues two transactions with the same timestamp at the same time to two different IGs. This ensures that these transactions are consistent in time and no read/write request (as the case pointed out above) can sneak in between. The Queue Manager at each node picks up the transaction with the minimum time stamp value to run.

VII. Selecting the Minimum and Maximum Sizes of an IG

The minimum and maximum sizes of an IG play a major role in availability and performance guarantees of the system. The term size here refers to the number of virtual machines which access the data file.

When the IG has less than the minimum number of nodes, then the system does not service any client request and is unavailable. This implies that a low minimum size increases the availability. But if the minimum size is too small the system would be available for a longer time but the load on the IG would become high when the number of working nodes becomes less thereby increasing the chances of failures. If the minimum size is too large then the system is unavailable for a longer time leading to more client waiting time.

Unlike the minimum size, the maximum size of an IG has impact on the performance of the system. A large maximum size would lead to higher write latency for the client as the TMs would have to wait for more number of

acknowledgements. A small maximum size would lead to larger Update Queues at each replica and more queuing time.

As the IG itself is not static, it would be a good idea to have the minimum and maximum sizes of an IG as variable. If there is an increase in the transaction load then the maximum size can be increased to a higher value depending on the load. Similarly, the maximum can be reduced to a lower value if the transaction load decreases. A very low value of minimum size is not recommended. The minimum size can be lowered if the system is experiencing high failure rates otherwise a relatively high value of minimum size can be chosen giving a desired size of the IG. These decisions are dependent on the transaction load and are taken at the server side. Client would be required to give the initial load estimates to facilitate the initialization of minimum and maximum IG sizes. Currently, the value of minimum size is 2 and that of maximum size is N/2 where N is the total number of replicas in the system.

VIII. Future Scope

This is a work in progress. We are implementing this idea on a room reservation system as a prototype. This fully replicated system has an almost equal proportion of reads and writes transactions. We aim to measure average client waiting time for different IG sizes; costs of IG maintenance, cost efficiency of getting computing power proportional to the amount client pays and also fault tolerance which would test the recovery algorithm. We aim to conduct experiments on different IG sizes for various transaction loads which would give a good idea about what an ideal IG size would be for a given system and a given transaction load. In the future, we would like to measure the feasibility of this design for analytic workloads too and also extend the proposed design to partially replicated data stores.

IX. Conclusion

We are working on one of the limitations with Cloud computing – transactional data management. This algorithm aims to use a SQL-like interface to the data management applications over the Cloud but internally have consistency algorithms adapted to the Cloud. We are also trying to incorporate the paradigm of relational integrity into our system which is the corner stone of RDMS. We have this algorithm supporting updates to a single table or many related tables replicated over the network. We are sure this would raise questions relating to the trade-offs between performance v/s availability and also performance v/s consistency (overhead of our consistency scheme). We aim to find out a satisfactory way of deploying database applications over the Cloud and also support update operations apart from data analysis.

References

[1] Daniel Abadi, "Data Management in the Cloud: Limitations and Opportunities", IEEE Data Engineering Bulletin, vol. 32 no. 1, March 2009.

[2] Ashraf Aboulnaga, Kenneth Salem, Ahmed A. Soror, Umar Farooq Minhas, Peter Kokosielis and Sunil Kamath, "Deploying Database Appliances in the Cloud", IEEE Data Engineering Bulletin, vol. 32 no. 1, March 2009.

[3] Robert L. Grossman and Yunhong Gu, "On the Varieties of Clouds for Data Intensive Computing", IEEE Data Engineering Bulletin, vol. 32 no. 1, March 2009.

[4] Bo Peng, Bin Cui and Xiaoming Li, "Implementation Issues of A Cloud Computing Platform". IEEE Data Engineering Bulletin, vol. 32 No. 1, March 2009.

[5] Luis M. Vaquero, Luis Rodero-Merino, Juan Caceres and Maik Lindner, "A break in the clouds: towards a cloud definition", ACM SIGCOMM Computer Communication Review, vol. 39 no.1, January 2009.

[6] Dionysios Logothetis , Kenneth Yocum, "Ad-hoc data processing in the cloud", proceedings of the VLDB Endowment, vol .1 no.2, August 2008.

[7] Aaron Weiss, "Computing in the clouds", NetWorker, vol. 11 no.4, December 2007.

[8] E. Deelman, Gurmeet Singh, Miron Livny, Bruce Berriman, John Goodl, "The Cost of Doing Science on the Cloud: The Montage Example", proc. 2008 ACM/IEEE Conf. Supercomputing. High-Performance Networking and Computing, IEEE Press. 2008.

[9] D. Hastorun, M. Jampani, G. Kakulapati, A. Pilchin,S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store", in proc. 2007 ACM SIGOPS symposium on Operating systems principles,2007.

[10] Fay Chang , Jeffrey Dean , Sanjay Ghemawat , Wilson C. Hsieh , Deborah A. Wallach , Mike Burrows , Tushar Chandra , Andrew Fikes , Robert E. Gruber, "Bigtable: a distributed storage system for structured data", proc. 2006 of the 7th conference on USENIX Symposium on Operating Systems Design and Implementation, November 2006.

[11] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "PNUTS: Yahoo!'s hosted data serving platform", Technical report, Yahoo! Research, 2008.

[12] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis, "Sinfonia: a new paradigm for building scalable distributed systems:, in SOSP, 2007.

[13] M. Burrows, "The chubby lock service for loosely-coupled distributed systems", in OSDI, 2006.

[14] M. Brantner, D. Florescu, D. Graf, D. Kossmann, and T. Kraska, "Building a database on S3", in SIGMOD, 2008.

[15] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "Above the Clouds: A Berkeley view of Cloud Computing", Technical Report 2009-28, UC Berkeley, 2009.

[16] T. D. Chandra, R. Griesemer, and J. Redstone, "Paxos made live: an engineering perspective", in PODC, 2007.

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

[18] W. Vogels, "Eventually consistent", Communications of the ACM, vol. 52 no.1, 2009.

[19] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi, "Elastras: An elastic transactional data store in the cloud", in Workshop on Hot Topics in Cloud Computing, 2009.

[20] P. Helland, "Life beyond distributed transactions: an apostate's opinion", in Proc. Conference on Innovative Data Systems Research (CIDR), 2007.

[21] "Hypervisor", December 2004. [Online]. Available: http://en.wikipedia.org/wiki/Hypervisor. [Accessed: Apr. 23, 2010].