Non-blocking Algorithm for Eventual Consistent Replicated Database on Cloud

Mawahib Musa Elbushra
Sudan University of Science and Technology
Department of Computer Science
Khartoum, Sudan

Jan Lindström
SkySQL Ab
Espoo, Finland
Email: jan.lindstrom {at} skysql.com

Abstract— **One of the challenges of cloud programming is to achieve the right balance between availability and consistency in a distributed database. Cloud computing environments, particularly cloud databases, are rapidly increasing in importance, acceptance and usage in major applications, which need the partition-tolerance and availability for scalability purposes, thus sacrificing the consistency side (CAP theorem). With this approach, use of paradigms such as eventual consistency became more widespread. In these environments, a large number of user's access data stored in highly available storage systems. In this research we use eventually consistent transactions with revision diagrams for visibility and arbitration to manage updates in replicates. We present a new non-blocking algorithm to implement eventual consistency using revision diagrams and fork-join data in a distributed environment. Finally, we show that the proposed method solves the problem.**

Keywords: cloud computing, eventual consistency, replication

I. INTRODUCTION

Clouds are a large pool of easily usable and accessible virtualized resources (such as hardware, development platforms and/or services). These resources can be dynamically reconfigured to adjust to a variable load (scale), allowing also for an optimum resource utilization. This pool of resources is typically exploited by a pay-per-use model in which guarantees are offered by the infrastructure provider by means of customized SLAs [21].

To achieve high scalability at low cost, cloud services are typically highly distributed systems running on commodity hardware. Here scaling just requires adding a new off the shelf server.

CAP theorem, also known as Brewer's theorem [7, 18] later formally shown by Gilbert and Lynch [11], states that it is impossible for a distributed computer system to simultaneously provide all three of the following guarantees:

- Consistency : all nodes see the same data at the same time
- Availability: a guarantee that every request receives a response about whether it was successful or failed
- Partition tolerance: the system continues to operate despite arbitrary message loss or failure of part of the system

According to the theorem, a distributed system can satisfy any two of these guarantees at the same time, but not all three.

This raises the concern about which property is the least important one, since all three properties are both desirable and expected in distributed systems [11]. Strong consistency is expected because many systems are often used together with databases where the so called ACID (Atomicity, Consistency, Isolation, and Durability) properties apply. The databases provide strong consistency by the use of transactions.

High availability is expected as distributed systems are connected to a network and whenever the network is available, the system is supposed to be available [11]. Finally, tolerance to network partitions is expected because it is a basic fault-tolerance technique. If a link goes down or a node crashes, the rest of the system should continue to work as before - even if the crash results in fewer nodes or even two separate partitions [11]. In order to completely avoid network partitioning, or at least to make it extremely unlikely, single servers or servers on the same rack can be used. Both solutions do not scale and, hence, are not suited for cloud systems. Furthermore, these solutions also decrease the tolerance against other failures. Also, to use more reliable links between the networks does not eliminate the chance of partitioning, and increases the cost significantly. Thus, network partitions are unavoidable and either consistency or availability can be achieved. As a result, a cloud service needs to position itself somewhere in the design space between consistency and availability.

Consistency requirement principle is similar to the atomicity property in ACID. Each transaction will be atomic in strictly consistent database. On the flip side, if a database is not strongly consistent, then different nodes may have different views of the same data.

Partition tolerance is achieved when a distributed system is built to "allow arbitrarily loss of messages sent from one node to another" [11]. The current demand makes it impractical to keep all data at one source. This is because when the source fails, it means the entire system becomes unavailable. Therefore, partition tolerance allows for system states to be kept in different locations. In the case of a distributed database

if it is partition tolerant then it will still be able to perform read/write operations while partitioned. If it is not partition tolerant, when partitioned, the database may become completely unusable or only available for read operations.

CAP summarizes trade-offs from decades of distributed-system designs and shows that maintaining a single-system image in a distributed system has a cost [13]. If processes within a distributed system are partitioned then updates cannot be synchronously propagated to all processes without blocking. Under partitions, a system cannot safely complete updates and hence presents unavailability to some or all of its users. Moreover, even without partitions, a system that chooses availability over consistency enjoys benefits of low latency: if a server can safely respond to a user's request when it is partitioned from all other servers, then it can also respond to a user's request without contacting other servers even when it is able to do so [1]. Sacrificing partitioning tolerance is not an option as noted on [13]. The choice is between consistency and availability.

Brewer [8] again noted that two of the three requirements can be guaranteed simultaneously if one of the requirements can be traded-off [9]. It would be possible to have every kind of combination, CP, AP and CA. But in practice, every system wants to be tolerant to partition. In fact the only question of interest regarding CAP is what do we do when some nodes are unavailable?! This is why the CA choice does not really make sense, it would be a system that does not tolerate network partition, but what would happen if a partition occurs is that it would lose availability. Therefore, the choices provided by the CAP theorem are CP and AP.

Therefore, distributed databases can either be strongly consistent or available. Consequently, most of the NOSQL-databases can only provide eventual consistency, a weaker type of strong consistency.

Eventual consistency states that in an updatable replicated database, eventually all copies of each data item converge to the same value. The origin of eventual consistency can be traced back to Thomas' majority consensus algorithm [19]. The term was coined by Terry et al. [17] and later popularized by Amazon in their Dynamo system, which supported only eventual consistency [10].

Eventual consistency is a specific form of weak consistency; the storage system guarantees that if no new updates are made to the object, eventually all accesses will return the last updated value [4]. If no failures occur, the maximum size of the inconsistency window can be determined based on factors such as communication delays, the load on the system, and the number of replicas involved in the replication scheme. The most popular system that implements eventual consistency is DNS (Domain Name System). Updates to a name are distributed according to a configured pattern and in combination with time-controlled caches; eventually, all clients will see the update [23].

Eventual consistency means that given enough time, over which no changes are performed, all updates will propagate through the system and all replicas will be synchronized. At any given time, there is no guarantee that the data accessed is consistent, therefore the conflicts have to be resolved.

While eventual consistency is easy to achieve, the current definition is not precise [23]. Firstly, from the definition it is not clear what eventually consistent database state is. A database always returning the value 42 is eventually consistent, even if 42 were never written. One possible addition would be that eventually all accesses return the last updated value thus the database cannot converge to an arbitrary value [22]. Even this new definition has another problem: what values can be returned before the eventual state of the database is reached? If replicas have not yet converged, what guarantees can be made on the data returned? In this case only possible solution would be to return last known consistent value. Problem here is how to know what version of data item was converged to same state on all replicas.

Eventual consistency is often strongly consistent. Several recent projects have verified the consistency of real-world eventually consistent stores. One study found that Amazon SimpleDB's inconsistency window for eventually consistent reads was almost always less than 500ms [24] while another study found that Amazon S3's inconsistency window lasted up to 12 seconds [2,5]. Other recent work shows results similar from Cassandra where inconsistency window is around 200ms [15].

In this paper out major motivation is to use eventual consistency as a consistency in a traditional distributed relational database system.

The rest of this paper is organized as follows. We first review the related work in Section 2. This is followed by motivating example from real word problem in Section 3. A new non-blocking eventually consistent protocol is presented in Section 4. Finally, conclusions are presented in Section 5.

II. RELATED WORK

In [6] Burckhardt et al. propose a novel consistency model based on eventually consistent transactions ordered by two order relations (visibility and arbitration) rather than a single order relation. That establishes a handful of simple operational rules for managing replicas, versions and updates, based on graphs called revision diagrams. They also prove a theorem stating that the revision diagram rules are sufficient to guarantee eventual consistency.

Apache CouchDB presented in [3], commonly referred to as CouchDB, and is an open source database that focuses on ease of use. It is a NoSQL database that store data using JSON, JavaScript as its query language using MapReduce and HTTP for an application interface. One of its distinguishing features is multi-master replication.

Unlike traditional relational database, CouchDB does not store data and relationships in tables. Instead, each database is a collection of independent documents. Each document maintains its own data and self-contained schema. An application may access multiple databases, such as one stored

on a user's mobile phone and another on a server. Document metadata contains revision information, making it possible to merge any differences that may have occurred while the databases were disconnected.

CouchDB implements a form of Multi-Version Concurrency Control (MVCC) in order to avoid the need to lock the database file during writes. Documents in CouchDB are versioned, much like they would be in a regular version control system such as Subversion. If you want to change a value in a document, you create an entire new version of that document and save it over the old one. This leads to situation where you have two versions of the same document, one old and one new.

CouchDB achieves eventual consistency between databases by using incremental replication, a process where document changes are periodically copied between servers. CouchDB's replication system comes with automatic conflict detection and resolution. When CouchDB detects that a document has been changed in both databases, it flags this document as being in conflict, much like they would be in a regular version control system.

When two versions of a document conflict during replication, the winning version is saved as the most recent version in the document's history. CouchDB preservers the other copy of the document as a previous version in the document's history. This happens automatically and consistently, so both databases will make exactly the same choice. It is up to you to handle conflicts in a way that makes sense for your application. You can leave the chosen document versions in place, revert to the older version, or try to merge the two versions and save the result.

Dynamo is a proprietary, highly available key-value structured storage system [10,16] or a distributed data store. It has properties of both databases and distributed hash tables (DHTs). It is directly exposed as Amazon DynamoDB [16], as well as being used to power other Amazon Web Services such as Amazon S3.

DynamoDB is a fast, fully managed NoSQL database service where customers can store and retrieve any amount of data, and serve any level of request traffic. All data items are stored on Solid State Drives (SSDs), and are replicated across 3 availability zones for high availability and durability.

DynamoDB tables do not have fixed schemas and each item may have a different number of attributes. Multiple data types add richness to the data model. Local secondary indexes add flexibility to the queries you can perform, without impacting performance.

Performance, reliability and security are built-in, with SSD-storage and automatic 3-way replication. Amazon DynamoDB uses proven cryptographic methods to securely authenticate users and prevent unauthorized data access.

When Amazon DynamoDB returns an operation successful response to your write request, Amazon DynamoDB ensures the write is durable on multiple servers. However, it takes time for the update to propagate to all copies. That is, the data is eventually consistent, meaning that your read request immediately after a write might not show the change.

To authors knowledge this paper is the first where eventual consistency is used on traditional RDBMS.

III. MOVIVATION

Eventual consistency as seen is mostly used on key-value or document databases. The most popular system that implements eventual consistency is DNS, the domain name system. Updates to a name are distributed according to a configured pattern and in combination with time controlled caches, eventually of client will see the update. However, eventual consistency can be used also in traditional relational database systems where exact strong consistency is not required. These kinds of systems consists e.g. game servers, social networks, document databases.

As an example using eventual consistency on traditional relational database system we use distributed online gaming containing virtual money banking consisting five branches. At each branch the local database has an object known as accounts which stores the set of accounts in that branch. There is one database copy per branch respectively as Databases A, B, C, D and E. Each account in the branch is associated with a balance, denoted accbal. Let's assume that this balance is 1000. The banking enterprise has two local integrity constraints: the balance of each account is positive, and an account is associated with only one balance. At present, the application has no global integrity constraints. The banking enterprise has a number of local and global transactions. The local transactions are Ldeposit; Lwithdraw; Ltransfer, and Laudit.

Let's assume that customer visits all branches A, B, C, D, and E and withdraws $300 and this withdraw happens before the earlier withdrawals from other branches has been replicated.

In this execution, all transactions read the same initial value, and then update the account with the new value $700. Note that as far as each transaction is concerned, the integrity constraint of non-negative accounts is not violated. However, in the interleaved execution, the overall result does not reflect the correct execution of five withdraw transactions. In particular, if the five transactions were not interleaved, the fourth of the five transactions would have violated the integrity constraints, and hence would not have been executed. Now if all transactions are replicated before customer can visit the last bank we still have a inconsistency because balance = 1000 - 300 *4= -200.This problem is referred to as the inconsistent retrieval problem.

IV. PROPOSED METHOD

Proposed non-blocking distributed commit protocol is now presented. We start presenting how transactions are first executed on master (or primary) server and then how changes are replicated to other servers in the system.

Read only transactions can be executed on any of the nodes reachable by the client. Because read only transaction does not

affect database consistency, there is no need to assign global transaction identifier for them. Therefore, only a local transaction identifier is used. In this work we assume that there are no distributed queries, i.e. query can be executed on a single database server.

All write transactions are first executed on master and when master has committed the transaction, it is replicated to the other nodes for execution. Master is normally selected to be a server nearest to the client but any of the nodes reachable can be selected as a master for this transaction.

Internally, data consistency is maintained by using a multiversion model (Multiversion Concurrency Control, MVCC [26, 20]). This means that while querying a database each transaction sees a snapshot of data (a database version) as it was some time ago, regardless of the current state of the underlying data. This protects the transaction from viewing inconsistent data that could be caused by (other) concurrent transaction updates on the same data rows, providing transaction isolation for each database session. MVCC, by eschewing the locking methodologies of traditional database systems, minimizes lock contention in order to allow for reasonable performance in multiuser environments. Additionally, standard write-a-head logging is used to guarantee durability.

When a transaction is executed for the first time on the master, the master assigns to it a global transaction ID GTID. GTID associated with each transaction. Since ID is the server's UUID, it remains constant for transactions executed on that server. GTID is represented as a pair of coordinates GTID = source_id:transaction_id, where transaction_id is globally unique. One method to implement globally unique transaction identifiers is to use Lamport's clocks [14].

GTIDs are persisted in the log as a new log record type that holds the actual identifier. The event is called Gtid_log. As such, when the group of events for a given transaction is to be written to the log, a new Gtid_log record is also written, preceding the group. Store GTID in log of the master to guarantees that no transaction is re-executed more than once and two different transactions cannot have the same GTID. (Therefore, before applying a transaction a server checks that it has not applied it before. If it has, it skips it).

First before Start executions transaction Check GTID is unique from log file, which means GTID is not found from the log. If it is found then there is no need to do anything i.e. guarantee there is no two transaction with same GTID and not execute transaction more than once

For every statement on a transaction, we check if the statements does writes (INSERT, UPDATE, or DELETE statements on SQL) and if it does master must be able to reach (send messages and receive replies) from majority of the nodes.

When transaction is read operation the transaction find snapshot of data using fork operation that allows creating a copy of data and read it. If transaction is write operation the transaction finds snapshot of data using fork operation that allow to create a copy of data to work with them in isolation from the original or main version. Check version that updated is not old snapshot or version of data by equation: current version equal to new version minus one if not true go to previous step to create current version, then check integrity constraint to guarantee the new update not violate constraint means consistent state. It's the main idea of revision go with data from consistent state to consistent state if true then commit. (2nd phase) Otherwise set transaction to wait queue and execute later.

Finally merge these update with main version using join all operation that is discussed in revision diagram then GTID is written to the log (immediately preceding the transaction itself in the log).

Then check queue if there is operation fork and execute else end transaction (still in master). Send transaction log and GTID to other node that means other node can't see version until committed in master.

Figure 1. Master algorithm:

```
1   GTID := Source_id:+next_transaction_id;
2   If GTID can be found from master's log {
3      End; /* transaction is already executed and we can skip it. */
4   }
5   For every statement Sk in the transaction T {
6    If Sk is done at least one write statement {
7      If # nodes reachable from master < (# total nodes / 2) + 1 {    /*
Majority?*/
8        Abort transaction T; /* we do not have majority */
9      }
10    }
11    public transaction fork {
12      revision r = fork { tuple }
13     If Sk is read {
14      Execute read and exit;
15     } Else If Sk is Write {
16      If revision = current version {
17       Execute transaction;
18      } Else {
19       request current version
20      }
21      If statement aborts because database would be inconsistent {
22         Abort;
23      }
24    }; /* Sk is write */
25  };  /*for executes statements on master*/
26  Join r;}
27  Write transaction operations and gtid to the log;
28  Send transaction log and gtid to other nodes for execution;
```

Now we consider how transactions are executed on a server (or node) that is not a master. There are basically two possibilities. Either transaction is read only transaction or transaction is already committed on a master and now replicated to this node. For read only transactions we only need

to provide current consistent version. There is no need to use any special protocol for this. For write transactions coming from master there is need to do additional checking presented below.

Transactions containing only read only statements can be executed on any of the nodes, but transactions containing write statements (update, insert or delete SQL-statements) needs to be executed on master. Nodes are not autonomous i.e. master makes always the decision whether the transaction commits or aborts. If the non master aborts the transaction, it requests normal REDO-log records of the transaction from the master. Then, it executes normal ARIES style recovery for that transaction.

Every node remove that transaction from queue after it checks gtid is not already executed on this node. If it is, we can ignore the transaction. Then we check if this is the next in the log file if not put transaction back to the queue (for order manner) and send REDO log request to master, master will reply with REDO logs and we will redo transaction using normal ARIES style recovery. Finally node writes transaction operations and GTID to the log.

When executing write transaction we need to check that data item versions are consistent. They must represent consistent view. If data item versions do not represent consistent view or integrity constraints would be violated after transaction execution, we send request to master for REDO logs. Master replies with REDO logs for all data items in read and write set of the transaction. These REDO logs are then executed on this node using traditional ARIES style recovery. When transaction finally commits, it will represent consistent (eventually) database state and the transaction log and GTID can be written to log followed by transaction commit log record.

```
1   Remove transaction T from Queue;
2   If received gtid from Master is found from log {
3     End; /* This transaction is already executed */
4   }
5   If received transaction_id > transaction_id +1 on log {
6     Put transaction T back to Queue; /* We need to execute
transactions in order */
7   };
8   For every statement Sk in transaction T {
9     Check that data items versions touched on Sk are consistent;
10    Execute statement;
11    If statement aborts because database would be inconsistent {
12      Abort transaction;
13      Send REDO log request to Master; /* Master will reply with
REDO logs and we will redo transaction using normal ARIES style
recovery*/;
14      End;
15    }
16  } /* End For;
17  Write transaction operations and gtid to the log;
```

Figure 2. Other nodes:

We consider banking enterprise consists of a five branches. At each branch the local database has an object known as accounts which stores the set of accounts in that branch. There is one database copy per branch respectively as Databases A, B, C, D and E. Each account in the branch is associated with a balance, denoted accbal. Let's assume that this balance is 1000. The banking enterprise has two local integrity constraints: the balance of each account is positive, and an account is associated with only one balance. At present, the application has no global integrity constraints. The banking enterprise has a number of local and global transactions. The local transactions are Ldeposit; Lwithdraw; Ltransfer, and Laudit.

Let's assume that customer visits all branches A, B, C, D, and E and withdraws $300 and this withdraw happens before the earlier withdrawals from other branches has been replicated.

In this execution, all transactions read the same initial value, and then update the account with the new value $700. Note that as far as each transaction is concerned, the integrity constraint of non-negative accounts is not violated. However, in the interleaved execution, the overall result does not reflect the correct execution of five withdraw transactions. In particular, if the five transactions were not interleaved, the fourth of the five transactions would have violated the integrity constraints, and hence would not have been executed. Now if all transactions are replicated before customer can visit the last bank we still have a inconsistency because balance = 1000 - 300 *5= -500.This problem is referred to as the inconsistent retrieval problem

Let at T1 and T2 denote two concurrently running transactions. Each transaction see, for example, that accbal(A) = 1000$ and accbal(B) = 1000$. Now T1 thinks it can withdraw 300$ from A and T2 thinks it can withdraw 300$ from B, because each transaction will not see the other transaction's change, which is the property of snapshot isolation. Thus, after all transaction has run, we have that Balance(A) = -500$, which violates the constraint.

By using algorithm when customer visit branch A is master:

1. Transaction receives GTID= A:1;
2. This transaction can't be found from master's log, thus we need to execute it.
3. All nodes are reachable thus we continue.
4. Transaction withdraws $300 from the account and account balance is $700. Thus database is consistent and transaction commits.
5. Transaction GTID and operations are written to master log.
6. Transaction is sent to other nodes for execution.

Now let's consider the situation on branch B where customer arrives before withdraw at branch A has not yet replicated. Now branch B naturally is the master and all other's including branch A are replicated nodes.

1. Transaction receives GTID = B:2 (remember that transaction identifiers are globally unique)
2. From master's log we note that there is no transactions executed thus last executed GTID on this master =B: 0. Thus we know that we are missing one update.
3. At step 12 we request a new revision to accbal, from above we know that database is not yet eventually consistent, thus we request a new current revision at step 19.
4. If revision diagram is not acyclic when transaction will be aborted at step 26.

When transaction with GTID=A:1 has been executed the transaction with GTID=B:2 can continue and in that execution the balance would have the correct value i.e. $700 and withdraw is still possible. If we continue the example, in all branches transaction either needs to wait until the transaction with previous global transaction identifier has been executed or if it is execute then we can continue. When balance is < $300 at some branch, the transaction is aborted because withdraw would violate the integrity constraint.

Now let's consider the network is partitioned such that branches A, B, and C are in network partition 1 and branches D and E in network partition 2. These partitions can't reach each other. Customer can withdraw money from any of branches at partition 1 because they form a majority but not from any of the branches at partition 2 because they do not form the majority. Customer can ask his/hers account but that may not represent current value. But the algorithm proposed makes sure that branches at partition 2 will eventually see consistent database. In this work we assume that network will eventually be fixed and all branches can see each other and that message changes between nodes can be reliable detected on link layer.

To show that proposed method produces correct consistency level, we need to establish some precise terminology and we do this similarly as in [6]. For uniformness, we require that all operations are part of a transaction and thus all operations are inside the transactions. We can describe the interaction between transactions and the database by the following three types of operations (query-update interface):

1. Updates $u \in U$ issued by the transactions
2. Pairs *(q, p)* representing a query $q \in Q$ issued by the transaction together with a response $v \in V$ by the database system.
3. The end of transaction operations issued by the transactions.

Formally, we can represent the activity as a stream of operations forming a history.

DEFINITION 1: A history *H* for a set *T* transactions and query-update interface *(Q, V, U)* is a map *H* which maps each transaction $\boldsymbol{t} \in \boldsymbol{T}$ to a finite or infinite sequence *H (c) operation* from alphabet $\sum = \boldsymbol{U} \cup (\boldsymbol{Q} \boldsymbol{x} \boldsymbol{V}) \cup \{\boldsymbol{end}\}$.

Furthermore, we need to define a program order (i.e.) the order where operations are executed on a transaction.

DEFINITION 2: *Program order*. For a given history H, we define a partial order $\prec_p$ over events in H such that $\boldsymbol{e} \prec_p \boldsymbol{e'}$ iff e appear before $\boldsymbol{e'}$ in some sequence *H(t)*.

Then we define an equivalence relation.

DEFINITION 3: *Factoring:* We define an equivalence relation $\sim_t$ over events such that $\boldsymbol{e} \sim_t \boldsymbol{e'}$ iff trans (e) = trans ($\boldsymbol{e'}$). For any partial order $\prec$ **o**ver events, we say that $\prec$ *factors* over $\sim_t$ iff for any events *x* and *y* from different transactions $\boldsymbol{x} \prec \boldsymbol{y}$ implies $\boldsymbol{x'} \prec \boldsymbol{y'}$ for any *x, y* such that $\boldsymbol{x} \sim_t \boldsymbol{x'}$ and $\boldsymbol{y} \sim_t \boldsymbol{y'}$. This induces a corresponding partial order on the transactions.

With following formalization we can specify the information about relationships between events declaratively, without referring to implementation-level concepts, such as replicas or messages. Namely, F takes as a parameter not a sequence, but an *operation context*, which encapsulates "all we need to know" about a system execution to determine the return value of a given operation.

Eventual consistency relaxes other consistency models by allowing queries in a transaction *t* to see only a subset of all transactions that are globally ordered before *t.* It does so by distinguishing between a visibility order (a partial order that defines what updates are visible to a query), and an arbitration order (a partial order that determines the relative order of updates).

DEFINITION 4: A history *H* is *eventually consistent* if there exist two partial orders $\prec_v$ (the visibility order) and $\prec_a$ (the arbitration order) over events in *H*, such that the following conditions are satisfied for all events $\boldsymbol{e_1}, \boldsymbol{e_2}, \boldsymbol{e} \subset \boldsymbol{E_H}$:

- (Arbitration extends visibility): if $\boldsymbol{e_1} \prec_v \boldsymbol{e_2}$ then $\boldsymbol{e_1} \prec_a \boldsymbol{e_2}$.
- (Total order on past events): if $\boldsymbol{e_1} \prec_v \boldsymbol{e}$ and $\boldsymbol{e_2} \prec_v \boldsymbol{e}$, then either $\boldsymbol{e_1} \prec_a \boldsymbol{e_2}$ or $\boldsymbol{e_2} \prec_a \boldsymbol{e_1}$.
- (Compatible with program order) if $\boldsymbol{e_1} \prec_p \boldsymbol{e_2}$ then $\boldsymbol{e_1} \prec_v \boldsymbol{e_2}$
- (Consistent query results): for all $(\boldsymbol{q}, \boldsymbol{v}) \in \boldsymbol{E_H}$, $\boldsymbol{v} = \boldsymbol{q}^{\#}(\boldsymbol{apply}(\{\boldsymbol{e} \in \boldsymbol{H}) \parallel \boldsymbol{e} \prec_v \boldsymbol{q}\}, \prec_a, \boldsymbol{s_0}))$. Thus query returns the state as it results from applying all preceding visible updates (as determined by the visibility order) to the initial state, in the order given by the arbitration order.
- (Atomicity): Both $\prec_v$ and $\prec_a$ factor over $\sim_t$.
- (Isolation): If $\boldsymbol{e_1} \notin \boldsymbol{committed}(\boldsymbol{E_H})$ and $\boldsymbol{e_1} \prec_v \boldsymbol{e_2}$, then $\boldsymbol{e_1} \prec_p \boldsymbol{e_2}$. That is, events in uncommitted transactions are visible only to later events by the same client.
- (Eventual delivery): For all committed transactions *t*, there exists only finitely many transactions $\boldsymbol{t'} \in \boldsymbol{T_H}$ such that $\boldsymbol{t} \not\prec_v \boldsymbol{t'}$.

.

The reason why eventual consistency can tolerate temporary network partitions is that the arbitration order can be constructed incrementally, i.e. may remain only partially determined for some time after a transaction commits. This allows conflicting updates to be committed even in the presence of network partitions. Now we are ready to prove the correctness of the proposed method.

THEOREM 1: A history *H* produced by the proposed method is eventually consistent.

PROOF:

- Compatible with program order: All transaction operations are executed on all nodes in the order they appear on program in step 9 of the proposed method. Thus, proposed method is compatible with program order.
- Arbitration extents visibility: In step 2 of the proposed method we make sure that slaves execute transactions in the same order as they are executed on master. Because we use MVCC for local concurrency control, we know that revision diagram is acyclic and a partial order. For versions we use revision diagram fork and join operations (see [6]) thus $\prec_v$ is acyclic, transitive and partial order. Thus we know that if $\boldsymbol{e_1} \prec \boldsymbol{e}$ and $\boldsymbol{e_2} \prec \boldsymbol{e}$, then either $\boldsymbol{e_1} \prec \boldsymbol{e_2}$ or $\boldsymbol{e_2} \prec \boldsymbol{e_1}$. Now define $\prec_v = \prec_a = \prec$, then if $\boldsymbol{e_1} \prec_v \boldsymbol{e_2}$ then $\boldsymbol{e_1} \prec_a \boldsymbol{e_2}$.
- (Consistent query results): We let at step 2 query to continue if and only if all preceding transactions are executed. Thus query returns the state as it results from applying all preceding visible updates.
- (Atomicity): Lets again using step 2 of the proposed method use the total order i.e. if $\boldsymbol{e_1} \prec \boldsymbol{e}$ and $\boldsymbol{e_2} \prec \boldsymbol{e}$, then either $\boldsymbol{e_1} \prec \boldsymbol{e_2}$ or $\boldsymbol{e_2} \prec \boldsymbol{e_1}$. Now define $\prec_v = \prec_a = \prec$, then we can easily see that both $\prec_v$ and $\prec_a$ factor over $\sim_t$
- (Isolation): Because local concurrency control method uses multiple versions (steps 20--24) and is compatible with program order, thus if $\boldsymbol{e_1} \notin \boldsymbol{committed}(\boldsymbol{E_H})$ and $\boldsymbol{e_1} \prec_v \boldsymbol{e_2}$, then $\boldsymbol{e_1} \prec_p \boldsymbol{e_2}$. That is, events in uncommitted transactions are visible only to later events by the same client.
- Eventual delivery: As noted if a server comes down or its network is separated from rest of servers in the cloud. The server when up and can reach the rest of servers requests log records of the missing transactions. These log records are then applied using standard ARIES method.
  When node is up it can serve only read-only transactions if the node does not belong to majority. If node is not separated, write transactions are executed only after the missing transactions are executed (step 2). Thus for every transaction *t*, there exists only finitely many transactions $\boldsymbol{t'} \in \boldsymbol{T_H}$ such that $\boldsymbol{t} \not\prec_v \boldsymbol{t'}$ ■

V. CONCLUSIONS

One of the challenges in distributed system is to achieve both consistency and availability not choose one of them. Consistency means when update is achieve will see to all node in database system at same time and it is difficult to achieve this strong consistency and guarantee availability of that system, according to that was necessary to take a look to properties of databases throughout the ACID, BASE and CAP theorem, strong consistency is main property of database, but in distributed environment there scarifies consistency to maintain availability through partition tolerance this led to need balance between availability and consistency it was eventual consistency the best solution.

To understand and build it eventual consistency model in distributed environment concurrency control is an essential issue whenever it comes to many users and shared resources. MVCC is an efficient method to let multiple processes access the same data in parallel without corrupting the data and the possibility of deadlocks. It is an alternative to the more strict lock based approaches, where every process first has to request an exclusive lock on a data item, before it can be read or updated.

For optimistic replication it is necessary to determine if multiple versions of the same data item were created in parallel. Furthermore, the database needs to know which version the newest is.

In this paper we have proposed a solution to non-blocking for eventual consistent replicated cloud database. A new algorithm was proposed as solution for this problem and validate by case study in bank example and how using MVCC that is good for read and no blocking but when the received request to update it check version to make sure that the newest version before commit. The algorithm also treated with eventual consistency in distributed environment or replica is guarantee local consistency then eventually to all other node.

One possible research direction would be implementing the proposed method inside a database management system capable of replication and measuring the availability, consistency and partitioning avoidance.

As future research, one interesting direction is to design encapsulated solutions that offer good isolation for common scenarios. Examples are use of convergent and commutative replicated data types and convergent merges for non commutative operations. Another direction is scenario-specific patterns, such as compensations and queued transactions, which can be leveraged to achieve high availability while providing consistency that applications can reason about.

This study could be also extended to find out what potential stronger consistency guarantees or isolation levels can be provided for transactions containing multiple statements.

REFERENCES

[1] ABADI, D. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. IEEE Computer vol. 45, no. 2, pp 37-42, February 2012.

[2] AMAZON. Simple Storage Service (S3). http://aws.amazon.com/s3/, Aug. 2009.

[3] ANDERSON, C. J., Lehnardt, J., and Slater, N. CouchDB: The Definitive Guide. Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472. January 2010, First Edition

[4] BAILIS, P., and Ghodsi, A. Eventual consistency today: limitations, extensions, and beyond , In communications of the ACM vol. 56, no. 5, PP. 55-63, May 2013

[5] BERMBACH, D. and Tai S. Eventual Consistency: How soon is eventual?. In Proceedings of ACM MW4SOC '11 and 6 other workshop on Service Oriented Computing, New York, December, 2011, no.1.

[6] BURSKHARDT, S., Leijen, D., Fähndrich, M., and Sagiv, M. Eventually Consistent Transactions. ESOP , pp.67-86, 2012

[7] BREWER, E: PODC keynote. http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf, 2000.

[8] BREWER, E. Towards Robust Distributed Systems, (invited Talk) Principles of Distributed Computing, Portland, Oregon, SIGOPS, And SIGACT News, July 2000.

[9] BREWER, E. CAP twelve years later: How the "rules" have changed. IEEE Computer, vol. 45, no. 2, pp. 23-29, February 2012.

[10] DECANDIA, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: Amazon's highly available key-value store. In Proceeding 21st ACM Symposium on Operating Systems Principles (SOSP), pp. 205-220, 2007.

[11] GILBERT, S., LYNCH, N. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Service; SIGACT News, v33, n2, p51-59, June 2002. http://doi.acm.org/10.1145/564585.564601

[12] GRAY, J.N., Lorie, R.A., Putzolu, G.R., and Traiger, I.L. Granularity of locks and degrees of consistency in a shared data base. In G.M. Nijssen, editor, Modelling in Data Base Management Systems, 365-294. IFIP, January 1976

[13] HALE, C. You can't sacrifice partition tolerance; Available from http://codahale.com/you-cant-sacrificepartition-tolerance.

[14] LAMPORT, L. Time, clocks, and the ordering of events in a distributed system, Commun. ACM, vol. 21, no. 7, pp. 558–565, July 1978.

[15] RAHMAN, M., Golab, W., AuYoung, A., Keeton, K. and Wylie, J. Toward a principled framework for benchmarking consistency. Workshop on Hot Topics in System Dependability, 2012.

[16] SIVASUBRAMANIAN, S. Amazon dynamoDB: a seamlessly scalable non-relational database service. In proceeding SIGMOD International Conference on Management of Data. ACM New York, NY, USA, 2012, pp. 729-730

[17] TERRY, D. B., Demers, A. J., Petersen, K., Spreitzer, M.J., Theimer, M.M., Welch, B. B. Session guarantees for Weakly Consistent Replicated Data. In proceeding of parallel and distributed information system (PDIS). IEEE, Austin, TX, 1994, pp. 140-149.

[18] THARAKAN, R. Brewers CAP Theorem on distributed systems, Scalable Web Architecture, February 14, 2010. http://www.royans.net/arch/brewers-cap-theorem-on-distributed-systems/

[19] THOMAS, R. H. A majority consensus approach to concurrency control for multiple copy databases. ACM Trans. on Database Systems, vol. 4, no. 2, pp. 180–209, June 1979.

[20] TRAIGER, I. L. Gray, J., Galtieri, C. A. and Lindsay, B. G. Transactions and consistency in distributed database systems, In ACM Transactions on Database Systems, New York, vol. 7, no. 3, pp. 323–342, September 1982.

[21] VAQUERO, L. M., Rodero-Merino, L., Caceres, J., and Lindner. M.. A Break in the Clouds: Towards a Cloud Definition. ACM SIGCOMM Computer communication Review, vol. 39, no. 1, pp50–55, January 2009.

[22] VOGELS, W. Scalable Web services: Eventually Consistent, ACM *Queue*, *vol.* 6, no. 6, pp. 14-16, October 20089.

[23] VOGELS, W. Eventually consistent, Communications of the ACM, vol. 52, no. 1, pp. 40–44, January 2009.

[24] WADA, H., Fekete, A., Zhao, L., Lee, K., A. and Liu, A. Data consistency and the tradeoffs in commercial cloud storage: the consumers' perspective. In Proceedings of the Conference on Innovative Data Systems Research . Asilomar, CA, USA, January 2011 .