HBase++
Extending HBase With Client-Centric Consistency Guarantees for Geo-Replication

Alexandre Filipe Campos
alexandre.t.campos@tecnico.ulisboa.pt

Instituto Superior Técnico
INESC-ID

Abstract. Motivated by requirements of the growing social web applications, the necessity to store data across several data centres has become even more important in order to bring data as close to the users as possible. A plethora of non-relational databases raised in recent years in order to address this issue due to their ability to scale out. But in the presence of wide distances, to deliver high-availability and data consistency is not possible in the presence of partitions. Several works have have been developed that try to reach the panacea of consistency and availability, at times sacrificing consistency in other to provide a better service or sacrificing latency to provide strongly consistent data, without taking into account that not all data needs the same consistency and not all data needs to be readily available. To tackle this problem, we propose a consistency model that will reason about data through user defined divergence bounding values, causal relationships and transactional enforcement within the data centre, in order to provide urgency to data in a geo-replication scenario.

1 Introduction

With the growth of the Internet and social web applications, and as larger amounts of information need to be processed and managed, more data centres are being deployed each year. As users are spread across the globe it becomes relevant, to deploy those data centres as close as possible to the most amount of users, in order to avoid the large latency imposed by round trips across the globe. Amazon has shown that web users are sensitive to latency [45]: even a 100ms increase in latency causes measurable revenue losses. The focus therefore shifts, from simply having a well built distributed system that can serve from within a data centre, replicating data through several machines, but to be able to propagate and manage data through machines in various numbers of data centres. This, as said, brings data closer to the users and allows to maintain the system available even in the presence of full data centre outages due to various reasons (e.g. natural catastrophes).

Cloud storage systems, such as the currently popular class of NoSQL data stores, have been designed in accordance to this scenario, as they, in contrast to relational database management systems (RDMS), can scale out horizontally to thousands of machines. Due to data not being tightly coupled with several relations, it becomes easier to partition. Many of the NoSQL systems provide a relaxed form of consistency in order to provide high availability, while others strive for strong consistency and sacrifice latency. Other systems go even further and focus on the added feature of geo-replication, providing a system to manage data on a worldwide scale. This comes with inherent trade-offs as the already present problem of performance versus consistency, is highly accentuated due to the high communication latencies between data centres.

Choosing between either an eventual consistent store and a strongly consistent one becomes an arduous task, unless your data falls into a very specific category. As in most problems that need a data store solution, not all data is the same, and it could benefit from different consistency degrees. To address this problem, various works have been developed with the intent of providing support for developers, when deploying a geo-replicated data store, by offering consistency models in-between those opposite extremes. Some go even further as to offer dynamically different levels of consistency,

to be defined at development time and others try to change the degrees of consistency at runtime.

HBase[1] is the open source version of Bigtable [11], and supports geo-replication in a eventually consistent manner between different clusters. Thus, one can not predict accurately enough how and when replication takes place, or ensure a given level of quality of service for delivering data to remote master replicas. Also, HBase alone offers only ACID guarantees on single row operations, making it impossible to group up certain related operations for geo-replication. Currently, there are some projects that try to provide transactional support for HBase, such as Omid.[2]

Given the current context, we propose a consistency model based on Vector Field Consistency (VFC) [44, 51] and further advancing on the work developed in the Distributed Systems Group [21, 42]. This model intends to enhance the geo-replication mechanism present in HBase, making it aware of the urgency of certain pieces of data updates, and giving them priority over less urgent data updates, taking into account operations incorporated into transactions and their causal relations.

The remainder of this report is organized as follows. Section 2 will summarize the objectives of this work. Section 3 contains the analysis of the state of the art, describing related works and a concluding analysis of them. In Section 4, we present the architecture of the proposed solution, followed by (Section 5) the methodology to evaluate said solution. Finally, in Section 6, we end the report with some concluding remarks.

2 Objectives

As stated, the overall aim of this work is to enhance the geo-replication mechanism present in HBase, making it aware of the urgency of certain pieces of data. In other words, we intend to improve the currently deployed eventual consistent model to be eventual consistency with notion of quality-of-data. With that said we can specify our objectives accordingly:

- Analyse the current state of the art in commercial NoSQL data stores, as well as academic work and proprietary stores that deepen the quality of consistency models.
- Make use of Omid to provide the basic underlying transactional support for HBase.
- Adapt the VFC model to allow users to define consistency bounds on data in the storage. These bounds, will be used to define which data urgency for each data item with relation to others, and will be placed as metadata information along or inside the HBase tables depending on the granularity of the enforcement.
- Incorporate the tracking of causal relationships between operations inside the HBase cluster explicitly and implicitly as possible.
- Develop a scheduling algorithm to define the ordering of operations for geo-replication. This algorithm will take into account the user defined bounds, as to reason about which operations are more urgent, as well as define the order of geo-replication, in a way that does not break causal relations.
- Define metrics to infer cost of geo-replication in terms of queue disruption to assist the scheduling algorithm to make choices with the least cost possible. This will also help developers to reason about data that might be too strictly bounded.
- Develop a geo-replication protocol that enforces the order provided by the scheduling algorithm.
- Evaluate the proposed solution in qualitative, quantitative and comparative terms.

[1] http://hbase.apache.org
[2] http://yahoo.github.io/omid/

3 Related Work

This section addresses the related work we consider more relevant to the proposed work. We start by describing the main data models used in NoSQL stores [25, 43](Section 3.1) and their main choices for architecture (Section 3.2). Followed by the analysis of the various consistency models used (Section 3.3) and non-functional requirements (Section 3.4). Finally, we describe the current benchmarking tools used for such systems (Section 3.5).

3.1 Data Model

One of the main differences from relational databases that defines NoSQL stores is for the most part the non-relation data model, which can be split into four groups as follows.

Column Family Stores. A data model pioneered by Google with their release of Bigtable [11] (Figure 1), it presents data as tables which consist of rows, columns and cells (that are the intersection of row and column coordinates). All columns belong to a column family providing another level of grouping, also, they need to be predefined and can not be changed at runtime. Rows can be byte arrays and all rows in the table are sorted by their row key. Finally, cells are uninterpreted byte arrays and are and versioned. Although the name table is used, there is little relation to the table format used in relational databases, being the main difference the storage of null values.

In contrast, a column family only stores a value if it is needed and that is why the tables are described as "sparse" (as in sparse arrays used in many fields of scientific computing). Therefore to access an existing value, one must provide a 3-tuple, containing a row key, column family, column and optionally a timestamp (default behaviour returns the most recent). Some open source systems, like HBase[3], HyperTable[4], Accumulo[5], who followed this model and most core architectural design choices, are also designated are as "Bigtable clones". Accumulo takes this model a bit further by providing an extra attribute to cells in order to offer cell level access control. Cassandra [31] also follows this data model, although it does not support versioning; it added the possibility to store, query and update collections as cell values as of version 2.0.

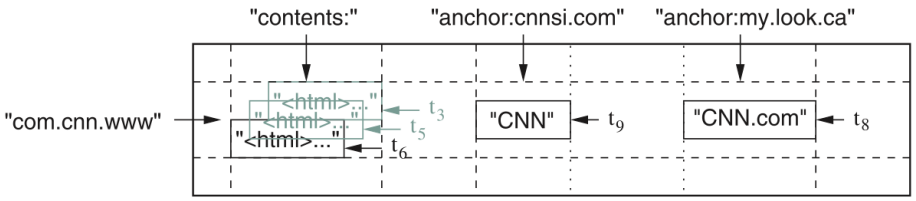


Fig. 1: A slice of an example table that stores Web pages [11].

Document Stores. The approach adopted by these stores is to incorporate into a JSON[6] or JSON like document the key value concept. Every document contains a unique key that identifies it explicitly. In contrast to key value and column family stores, the values stored in the document are not opaque to the system, support various data types and queries ranging multiple attributes of different types, providing a rich model for developers. Some examples of systems who follow this model are RethinkDB[7], CouchDB[8], MongoDB[9] adds the possibility to group documents under a Collection ID.

[3] http://hbase.apache.org
[4] http://hypertable.com
[5] http://accumulo.apache.org/
[6] http://www.json.org
[7] http://www.rethinkdb.com
[8] http://couchdb.apache.org
[9] http://www.mongodb.org

Also instead of JSON, they decided to use BSON,[10] which consists of a binary-enconded serialization of JSON-like documents and extends to support some other types that are not present in JSON. Both Riak[11] and MongoDB provide links between documents which can be used to model graph like relationships between objects.

Graph Based Stores. The basis of existence for these stores is the necessity to model heavily linked data with many relationships. Nodes are represented by objects (for most JSON is used) and edges are represent attribute links between objects. The graph like structure allows the possiblity to not have duplicate information, and provides faster join queries since they require only a simple graph traversal. Examples of such systems are Neo4j[12] and OrientDB.[13] Twitter also developed their graph database, FlockDB[14] to model following relationships in their social network, which focuses on one-hop relations instead of multi-hop as the aforementioned stores.

Key Value Stores. The most simple and least structured of the data models, the key-value model, offers a map where each value is addressed by a single key, being this key the only way to access the value. In most stores that adopt this model, such as Dynamo [17] and on of its clones Voldemort[15], the value is an uninterpreted byte array. Redis[16] provides a few options besides the basic uninterpreted byte array, such as the ability to store lists, sets, sorted sets and hashes. HyperDex [20] supports the ability to relate a key to zero or more attributes, this is to support their novel data organization technique: hyperspace hashing; which enables searches to be placed on a subset of the total number of attributes. Google's Spanner [15] also extends this model with temporal multi-version on each value, coming closer to a Bigtable-like model.

3.2 Distributed Architecture

From an architectural point of view, we can split the approaches into the two following categories: the first, being architectural design decisions system designers make for one instace of the system which is expect to be running inside a data centre; the second, is based on geo-replication where two or more instances of the store are running separated by wide area network (WAN) and must communicate over great distances to maintain data synchronized as best as possible.

Intra Data Centre.

Partitioning. The great triumph of NoSQL data stores is the ability to scale horizontally, something relational databases struggle to achieve. As the system store scales out, data needs to be further partitioned across all nodes and replicated. Range based partitioning consists of dividing tables horizontally by the row key, as these partitions are usually the basic element of load balancing. Systems like Bigtable and its clones rely on a master server to keep the metadata information, mapping data to each data server, and also rely on a highly available locking service [26, 10] to employ cluster membership. Since column family data models can be partitioned more efficiently, these data stores are more suitable for huge datasets than document stores. Document stores like MongoDB and RethinkDB, although they do not follow the column family data model, also employ range based partitioning to shard documents across servers.

On the other hand there are the Dynamo like systems that use consistent hashing [29] to map keys to servers and partition data across the entire cluster, therefore not needing to rely on any master. Neo4j does not support any type of partitioning. Hyperdex also relies on consistent hashing, although not considered a dynamo clone.

Replication. Bigtable and clones, in order to guarantee availability in the presence of failures, rely on data replication being handled by an underlying storage system [23, 46] that places that data

[10] http://bsonspec.org/
[11] http://basho.com/riak/
[12] http://www.neo4j.org
[13] http://www.orientdb.org
[14] https://github.com/twitter/flockdb
[15] http://www.project-voldemort.com
[16] http://redis.io

in a configurable number of servers (default is 3). There is an effort to store the data in the same physical server, in order to provide faster access; besides the initial physical server, data is usually replicated once in the same rack and another time in another rack, in order to survive an entire rack failure. Hyperdex also relies on an underlying storage system analogous to HDFS.

Dynamo relies on consistent hashing to replicate and distribute N copies of each value around a cluster composed of any number of physical machines. Each key k is assigned to a coordinator node, that is in charge of the replication of the data items that fall within its range. In addition to locally storing each key within its range, the coordinator replicates these keys at the N-1 clockwise successor nodes in the ring. OrientDB follows along these lines as it partitions data based on hashing, but it replicates in synchronous mode where it waits for full replication, before responding to the client, or asynchronous where it responds after the update has been written in the primary. This could be seen as a Master/Slave architecture, as supported by the following. In Neo4j and ChouchDB, updates on the master can be optimistically propagated to slaves, and writes on the slaves must be synchronized with the master. CouchDB employs master/slave or master/master architecture and relies on incremental replication between replicas. In Redis, RethinkDB and MongoDB the primary (master), receives all write operations, and secondary instances (slaves), apply operations from the primary, so that they have the same data set and serve only reads.

Inter Data Centre.
Support for multi data centre replication, and awareness is often a high-end feature to achieve, but most systems do not even take it into consideration, as it is not part of the design objectives of the store. We analyse those who do want to achieve this, in a specific way so that geo-replication is differentiated from replication within a data centre.

HBase has three modes of geo-replication: master-master when two clusters propagate updates to each other; master-slave when a primary cluster batches updates to one or more slave clusters whom only can serve reads; and cyclical, similar to master-master but with more than two clusters involved, where updates go around all clusters in a circular manner. Inside HBase each RegionServer will be in charge of replicating their updates to the target cluster and will do so with the help of Zookeeper, which will maintain the logs to be replicated and the offset. In case of a failure another RegionServer can take over the failed server's updates for replication.

Riak basis its geo-replication on a master-slave type architecture where one cluster acts as the master and replicates data to slave clusters in two possible modes: fullsync mode, where a complete periodical synchronization occurs between primary and secondary cluster(s); in realtime mode, where continual, incremental synchronization occurs - replication is triggered by new updates. Only one node (one on each cluster) will be in charge of geo-replication. Although Riak is an open-source project, geo-replication as a feature is considered a project of its own named Riak Enterprise, which is closed source. Cassandra is similar but the server that receives each write in one data centre (after replicating it to N-1 nodes in its cluster), propagates the write asynchronously to its twin server in the other cluster.

RethinkDB allows clients to define replication and acknowledgement settings on per data centre basis. For example, it has the ability that a write must not be declared successful until it has been applied in at least two replicas in each data centre.

MongoDB provides some features that allow the customization the behaviour of a sharded cluster or replica set deployment be more geo-replication aware, as the servers in one of the clusters can only act as slaves.

3.3 Consistency Models

The CAP conjecture was introduced by Brewer in 2000 [9] and later confirmed by Gilbert and Lynch [24] as a theorem. The theorem states that only two out of the three following properties can be guaranteed simultaneously: Consistency, Availability, and Partition tolerance.

- Consistency (C) equivalent to having a single up-to-date copy of the data.

- High availability (A) of that data (for updates).
- Tolerance to network partitions (P).

Twelve years later in [35], Brewer revisits explaining that the 2 out of 3 view is misleading as partitions are rare, there is little reason to forfeit C or A when the system is not partitioned. Also, as the choice between C and A can occur many times within the same system at very fine granularity, not only can subsystems make different choices, but the choice can change according to the operation or even the specific data or user involved, as we will see further, there are several systems that acknowledge this. Finally, all three properties are more of a continuous spectrum than discrete or even binary.

Abadi [1], reached similar conclusions defining a new acronym: PALEC. Defined as follows: if there is a partition (P), how does the system trade off availability and consistency (A and C); else (E), when the system is running normally in the absence of partitions, how does the system trade off latency (L) and consistency (C). Even so, the so far presented NoSQL stores still follow the CP/AP definition.

Before we analyse each of the system in terms of consistency we reference Amazon's CTO (Werner Vogels)[17] article for the definition of eventual and strong consistency from a client's perspective:

- **Strong consistency.** After an update completes, any subsequent access will return the updated value.
- **Eventual consistency.** This 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. 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.

Bigtable, HBase, Accumulo, RethinkDB, OrientedDB and Hypertable employ a mixture of multi-version concurrency control (MVCC) with locking, as writes use row-level (document level for document stores) locking to update to a new version, and reads will try to read the most recent version and require no locking.

Hyperdex provides linearisability as all key operations are totally ordered and will be seen in the same order by all clients. It employs a novel technique, called value-dependent chaining, to provide strong consistency and fault tolerance in the presence of concurrent updates.

MongoDB implements locks on a document level basis and provides consistent reads at the master. Neo4j relies on locking aswell with read committed isolation level. Redis provides optimistic locking with watch primitives on keys that are not supposed to be tampered with during the transaction. These systems provide strong consistency, with the exception of HBase geo-replication protocol as it is eventually consistent.

CouchDB also makes use of MVCC and when conflicts arise, between two replicas, one of versions will be considered the latest. Riak and Voldemort rely on optimistic locking, as well as vector clocks to reason about causality and staleness of stored values; this enables clients to always write to the database in exchange for consistency conflicts being resolved at read time by either application or client code. Dynamo and Cassandra use timestamps (no MVCC supported) to determine the most-recent value when doing read requests, there is no concurrency control as they rely on overlapping quorums for consistency. These systems are considered eventually consistent as there is a possibility of stale reads.

Through the rest of this section we will analyse mostly academic work that focuses on the line that separates CP and AP, as works try to come close to the panacea of consistency and availability. Some try to work up form highly available eventual consistent stores to higher consistency while others try to work from high latency consistent stores towards lower latency. There are systems that

[17] http://www.allthingsdistributed.com/2008/12/eventually_consistent.html

rather focus on offering multiple models of consistency for different data, while others reduce the data to be held in the data store to a specific subset, that in specific predefined scenarios can always be merged.

Eventual Consistency

In this section we analyse systems that build up from eventually consistent stores to offer stronger guarantees.

Probabilistically Bounded Staleness for Practical Partial Quorums [5]. By analysing latency-consistency trade-off, in the context of quorum-replicated data stores, the authors introduce a consistency model (Probabilistic Bounded Staleness - PBS) for Dynamo-style systems, which revolves around predicting the staleness of partial quorums, which do not guarantee the intersection of the read quorum and write quorum in all occasions. This is leveraged in a way that although it does not always guarantee strict consistency, a target percentage of the time, satisfactory values are returned. PBS thus describes the probabilities in terms of versions (k-staleness), time (t-visibility) and a combined model of the two. K-staleness is the probability of returning a value within a bounded number of versions. T-visibility is the probability that a read operation, starting t seconds after a write has committed, will indeed observe the latest value of a data item. This t captures the expected length of the window of inconsistency. Combining the two determines the probability that a read will return a value no older than k versions stale, if the last write has committed at least t seconds ago.

Quality-of-Service for Consistency of Data Geo-replication in Could Computing [21, 42] is a system implemented on top HBase that provides support for geo-replication by providing a unified cache view across different clusters. It relies on a consistency model based on [44, 51], and enables the definition and dynamic enforcement of multiple consistency degrees over different groups of data, within the same application, across very large scale networks of cloud data centres.

The basis, as stated, of this consistency model relies on defining three dimensional vectors to be associated with data objects, where each dimension holds a numerical scalar that defines the maximum divergence of that value. The constraints are defined as time (θ), sequence (σ), value (ν). These values are not fixed, as developers can define intervals to allow the system as it reasons about access patterns, to relax or restrict the bound. As bounds get close to being broken the priority of the associated values for geo-replication increases, resulting in high priority data having stricter bounds and low priority data have relaxed bounds.

The unified cache view will hold frequently used database items and items within the same locality group (i.e., pre-fetch columns of an hot row). Also it will keep track of items that need to be replicated and handle the whole replication process across clusters. The system constantly checks bandwidth and can trigger data replication and synchronization on low bandwidth utilization periods, even if consistency constrains do not impose it. Concurrent updates in different data centres rely on the deterministic rule of last-writer wins to enforce convergence.

Causal Consistency

Causal consistency provides useful semantics for developers as it guarantees that effects of operations are observed only after their causes: participants will not see data unless its dependencies are also seen. This is very useful for modern web applications in such use cases as preserving a comment section structure (replies will not appear before their parents), or changing the privacy settings of an album and then posting a picture you would only want to be available to certain friends.

Lamport's work [32], formally introduced causal happens-before relations as follows:

- **Thread-of-Execution.** An operation performed by a thread is causally after all of its previous ones.
- **Reads-From.** An operation that reads a value is causally after the operation that wrote the value.
- **Transitive-Closure.** If operation a is causally after b, and b is causally after c, then a is causally after c.

A convergence requirement [38, 36] for causal consistency abiding systems (also known as causal+) is an important feature due to the lack of liveness guarantees given by causal consistency alone; as an example, it is possible to satisfy causal consistency by never propagating writes between agents. Convergence guarantees that if updates cease (or for a given log or queue prefix), all agents will converge to the same state. Possible conflicts must be resolved in a deterministic way (e.g. last writer wins).

Peter Bailis et al. [3] observe that causal consistency is the strongest consistency model achievable in the presence of network partitions, and provide insight to the potential dangers of scaling causal consistency to multiple data centres as well as present a solution to the identified problems.

- **Throughput and Visibility Latency**. If new versions of data are generated faster than they can be applied (because of the throughput limit across data centres is exceeded), visibility latency will increase indefinitely due the formation of unstable queues, as writes must wait for their dependencies to arrive at that data centre. This is further aggravated by the convergence requirement, as all data centres must apply all writes locally.
- **(Not) Scaling Throughput and Datacentres**. As stated, convergence requires all-to-all data centre replication which limits write throughput, so the write throughput is limited by the slowest data centre. Adding more data centres will not help in making the overall system faster.
- **Potential Histories and Cluster Capacity**. Since all new writes causally depend on all writes (versions) that could have influenced it, potential causality graphs have large fanout and depth, limiting local apply-capacities and, accordingly, maximum global throughput.

It is proposed that instead of tracking potential causality (all possible influences) and having deep, wide and overall very large graphs, one can reduce the size of the causality graph by tracking only application-specified causal dependencies (explicit causality). It would be up to the application to add to each write, a set of dependencies that determine its happens-before relations. Each new write depends on few others which leads to faster checks and decrease in metadata size; also, concurrency increases due to explicit causality results in several smaller, disjoint graphs. Semantically unrelated updates are disconnected, so there is more independence and parallelism between sets of writes.

Following up on their work, the authors developed *Bolt-on Causal Consistency* [4], which is a shim layer to put on top of eventual consistent stores (as a client library) to provide convergent causal consistency. Tested on top of Cassandra, the underlying data store is expected to provide liveness, replication, durability, and convergence while the shim layer will provide safety. In their implementation, there are various assumptions, such as the presence of an eventual consistent data store (ECDS) with a known merge function for overwrites (e.g. last writer wins) and version information. Also the ECDS does not store multiple versions of data and there are no callbacks from the ECDS notifying clients of new writes.

Both possibilities of potential and explicit causality tracking are offered. As the bolt-on layer is not guaranteed to see all writes, attempting to set causal consistency for multiple writes on the same data item will fail. The shim layer will hold a local storage and a dependency graph, and cannot block waiting for causal dependencies that are either delayed or may never arrive due to partitions. This local store is defined as holding a consistent set of writes that can be read from and added to at any time without violating safety constraints. This "consistency" is defined with the notion of causal cuts. Being a set of writes, the dependencies for each write in the causal cut should either:

i.) be in the cut,
ii.) happen-before a write to the same key that is already in the cut, or
iii.) be concurrent with a write to the same key that is already in the cut.

The algorithm then can be defined as follows: before revealing a write w to a client, a shim needs to ensure that w, along with the set of writes in its local store is a causal cut. Writes will go through to the ECDS after being placed in the shim's local store and forming a causal cut. Reads can be done in one of two ways: the first being returned from the local store and having a background

thread checking the ECDS for new versions to updated the local store; the second is a pessimistic approach, where shims attempt to read the latest value from the ECDS and cover it. This requires synchronously testing the values dependencies against the local store and, if the dependency checks fail, attempting to also read the dependencies themselves from the ECDS.

Eiger [37], the follow up system to *COPS* [36], is a geo-replicated key-value storage system built on top of Cassandra. It provides read-only, write-only transactions and causal+ consistency. They claim to be the only geo-replicated data store available to support 3 out of 3 on the following properties: low latency, a rich column-family data model, and stronger consistency semantics: consistency guarantees stronger than the weakest choice (eventual consistency) and support for atomic updates and transactions.

Their previous work on COPS provides low latency and some stronger semantics such as causal+ consistency and read-only transactions but neither a rich data model nor write-only transactions.

Inside the data centre linearisability is provided, and across data centres, since linearisability, serializability and sequential consistency are incompatible with the low latency requirement, Eiger offers causal+ consistency (last writer wins convergence). Unlike COPS, causal dependencies are tracked on operations instead of values, which increases efficiency significantly.

Also a choice is made to track one-hop dependencies, a slightly larger superset of nearest dependencies, which have a shortest path of one hop to the current operation. They consider this increase in memory requirements by the client worth it, since this voids the need to store dependency information on the servers.

The client library is an important portion of the system as it tracks causality and attaches dependencies to write operations on a per-user basis, mediates access to nodes in the local data centre, and executes the read and write transaction algorithms.

Operation Transformation

SPORC [22] is a framework for collaborative services, that allows offline work, keeps that confidential from the servers and also detects and recovers from servers misbehaving. To achieve these goals it relies on two underlying mechanisms: *operation transformation* and *fork* consistency*. Fork* consistency holds the purpose of detecting servers misbehaviour and consists of every client embedding in every operation they send their individual view of the history. Operation transformation hold the purpose of resolving conflicts after a fork or clients offline work, and it works by transforming operations so they can be applied commutatively by different clients, resulting in the same final state. To evaluate the system they built a key-value store that uses last-writer wins as a transformation function.

Gemini [34] is a coordination infrastructure that offers RedBlue consistency, which is based on the separation of operations into two categories (red and blue). Where red operations are strongly consistent (and possibly slow), as they are serialized with respect to each other and require immediate cross-site coordination. In contrast, blue operations execute locally faster, and are lazily replicated across clusters in an eventually consistent manner (causality is kept across different sites for all operations).

All operations will go through a process to identify if they can be blue; this process involves splitting operations into two components: generator and shadow operations. Generator operations must not have any effect on system state and are only applied at their primary site, while shadow operations must have the same effect as the original operation and are applied at all sites. The generator operation decides which state transitions should be made while the shadow operation applies the transitions in a state-independent manner. To be able to provide state convergence and invariant preservation, Gemini needs to be informed of:

i. For any pair of non-commutative shadow operations u and v, label both u and v red;
ii. For any shadow operation u that may result in an invariant being violated, label u red;

iii. Label all non-red shadow operations blue.

Scalable CRDTs [18] follows up on the work by Mihai Letia, Nuno Preguiça and Marc Shapiro [33] that orginally coined the term: Conflict-free replicated Data Types. CRDTs introduce the concept of strong eventual consistency as they were designed to avoid the problems of conflict resolution, as replicas of CRDTs are proved to converge in a self-stabilising manner without blocking client operations and without having to deal with consensus, complex conflict resolution, or roll-backs.

Two styles equivalent styles (any data type that can be implemented as a state-based object can also be implemented as an op-based object and vice versa) are used to define CRDTs:

- **State-based replication.** as updates are applied locally, modifying the replica' state, replicas send their local state to other replicas which merge it with their own state. Convergence is achieved by by eventually delivering the updates to all replicas.
- **Operation-based replication.** Instead of transmitting the entire state, only updates are propagated. Causal relations are kept in the propagation.

There are various replicated set designs (G-Set, 2P-Set, U-Set, PN-Set, OR-Set), and to improve the OR-Set specification first by making state-based OR-Set specification to transfer only deltas between replicas instead. To achieve higher scalability *sharding* is introduced which consists of dividing the OR-Set into disjunctive subsets through several machines. Garbage collection is also added as it is proposed to remove expiring tuples from the sets after a specified time interval without breaking lookup semantics on the OR-Set. This means causal relations must be preserved over replicas as update tuples must expire in the same order as they were originally inserted.

Transactional

Walter [47], is a key-value store that implements a consistency model called parallel snapshot isolation (PSI), which is a variant of snapshot isolation. It makes use of preferred sites and counting sets (similar to commutative data types), that reconcile multiple concurrent writes by merging conflicting versions. Within a site (data centre), hosts observe transactions according to a consistent snapshot and across sites PSI enforces causal ordering and not a global ordering of transactions, allowing the system to replicate transactions asynchronously across sites.

To avoid conflicts, as stated, Walter relies on preferred sites which consist of defining a site for each object where writes to that object can be committed without checking for conflicts with other sites. If all objects within a transaction belong to the same site, and are being applied at that site, the transaction commits very quickly. Another mechanism is conflict-free counting set (cset) objects, which consist of objects that regardless of the number of updates in different sites never generate write-write conflicts.

In the presence of non-cset objects and updates at non-preferred sites, PSI must detect write-write conflicts across sites, and abort transactions in their presence. A transaction when committing at its site, must check if some write-conflicting transaction has committed at that site, or if a transaction who already committed at another site is currently propagating to this site, in order to be able to commit. With this PSI, guarantees write-write conflict detection. Another property is that a transaction reads from a snapshot established at its site. And finally, to preserve causal consistency, operations can propagate to a site s only if all transactions that committed at its site, before it started, have already propagated to s.

Percolator [41] is a lock-based, distributed approach to implement snapshot isolation on top of Bigtable, tailored for incremental data processing. Allowing the possibility of multi-row transactions, Percolator achieves this by adding two extra columns to each column: lock and write. The write column is used to store the commit timestamps, while the lock column is used to simplify write-write conflict detection by allowing the two-phase commit protocol to withhold writing into a locked column. If a transaction finds the column locked it queries the primary node of that transaction,

and will either wait, abort (in the presence of conflicts) or force the abort of whom holds the lock if it suspects a failure.

While useful to prevent massive repeated computations, by triggering incremental processing on newly updated data, the metadata held on the data store adds extra load to the servers, and the lock-based nature of the system prevents progress in the face of failures. Percolator also makes use of a timestamp oracle server that provides monotonically increasing timestamps, that must be highly available and must serve every transaction.

Transactional support in HBase was a natural follow up to Percolator, since HBase is a Bigtable clone. The first attempt was HBaseSI [53, 54] in 2010. Without depending on any centralized server, this work breaks from the Percolator implementation and uses HBase itself to store transactional management metadata in special purpose tables.

Following up this work, Padhye and Tripathi [40, 39], offered a solution based on HBaseSI that uses less metadata and provides serializability, also claiming that HBaseSI does not offer robustness in the presence of transaction failure. It uses a DSGTable to provide timestamps and active transaction information, a CommitLogTable for committed transactions and also adds columns to regular storage tables. Additionally, it is proposed the possibility of a service-based model that steps away from the decentralized model by providing a conflict detection service that maintains read and write set information as well as timestamps. This would reduce the performance overhead of write and read requests for metadata on HBase, which leads to the next work.

In 2011, Junqueira et al. [27], proposed such an architecture with ReTSO. A lock-free transaction management system that relies on a "transaction status oracle" (TSO) for managing timestamp and read and write sets. It provides snapshot isolation and still requires some metadata (commit timestamps) to be stored in HBase. It later evolved into replicating the commit timestamps to the client since they do not need to be persistently stored [7], further avoiding storage overhead inside the data store.

Currently, the open-souce implementation of the above system is Omid.[18] It relis on BookKeeper [28] for durability and relies on ZooKeeper [26], to enforce a singleton TSOServer instance operating over time. A transaction's life-cycle will start with the request for a start timestamp from the TSO. It will then access the data store and perform all the operations, reading only values whose timestamp is lower than the assigned start timestamp, and writing values with the assigned start timestamp as its version. At the end of the transaction the client submits the rows to the TSO to check for conflicts. If no conflicts arise, the transaction will commit and the commit timestamp for the affected rows will be updated on the TSO. This system can handle large amounts of transactions but it is limited by memory, which will eventually restrict the scalability of the system. A solution is currently in development to further scale Omid called MegaOmid. It will partition transactions among multiple status oracles, designated for certain areas, and will have a global status oracle for global transactions (spawning more than one area).

Further work by D. Ferro and M. Yabandeh [52], takes Omid to the next level by introducing *write-snapshot isolation*. An isolation level that has performance comparable to snapshot isolation, and yet provides serializability. It is shown how snapshot isolation unnecessarily lowers the concurrency of transactions by preventing some serializable histories and allow some non-serializable histories. And its proved that read-write conflict avoidance is necessary and sufficient to achieve serializability.

Similarly to snapshot isolation, write-snapshot isolation assigns start and commit timestamps to transactions and ensures that transactions only read from a snapshot with a commit timestamp lower than their start timestamp. Their main difference is in how conflict detection is defined, as shown:

[18] http://yahoo.github.io/omid/

	Snapshot Isolation	Write-isolation isolation
Temporal Overlap	$T_s(tx_i) < T_c(tx_j)$ and $T_s(tx_j) < T_c(tx_i)$	$T_s(tx_i) < T_c(tx_j) < T_c(tx_i)$
Spatial Overlap	both write into row r	tx_j writes into a row r and tx_i reads from row r

If both spatial and temporal overlap conditions are true, one of the transactions must abort. In short, write-snapshot isolation defines that transactions that commit during the lifetime of another transaction should not modify its read data.

H 1. r1[y] r2[z] w1[x] w2[x] c1 c2

H 2. r1[y] w1[x] c1 r2[z] w2[x] c2

For example **History 1** under snapshot isolation would abort since there is a write-write conflict, although it is in fact serializable. Under write-snapshot isolation it would be interpreted in the serial order presented in **History 2** and none of the transactions would abort.

Warp [19], is an add-on for HyperDex [20] that provides support for one-copy serializable ACID transactions spanning multiple objects (a concurrent execution of transactions in a replicated database is one-copy-serializable if it is equivalent to a serial execution of these transactions over a single logical copy of the database). It combines optimistic client side execution with linear transactions, which consist of arranging the servers, whose data is affected by the transaction, in a dynamically-determined chain and processing the transaction in a two-way pipeline. Independent transactions are unordered and overlapping transactions must be detected for conflicts. Happens-before relationships must be identified along the chain to avoid dependency cycles.

Initially the client keeps the transactional context on a local cache (reads are stored in the cache and writes are applied locally). As the client submits its transaction for commit, the keys in accessed or modified rows within the transaction are sorted in lexicographical order to determine the ordering of the server chain. Along the forward pass, overlapping transactions are identified and happens-before relations are established. If transactions overlap, read-write conflicts or write-write conflicts will make one of the transactions abort (read-read conflicts are fine). As the end of the chain is reached, the backward pass will start informing all the servers to either commit or abort.

This protocol allows multiple transactions to be pushed through the pipeline concurrently, and in the presence of faults requires a replicated state machine (such as ZooKeeper [26]) to keep track of cluster membership.

Lynx [55] is a system that also (like Warp) relies on a chaining protocol to offer transactional support but (unlike Warp) focuses on geo-replication and does not require a backward pass. This is due to the fact that analysis is done at the first hop, and thus requires a priori knowledge of transactions and static analysis to prevent non-serializable executions, which neither guarantee external consistency nor order-preserving serializability. Nevertheless, this provides Lynx with the ability to return to the client after the first hop, with the assurance that the transaction is going to be committed. This reduces user perceived latency but not the actual latency of the transaction.

Spanner [15] is Google's multi-version globally distributed database and the successor to Megastore [6]. It uses synchronized clocks to assign globally meaningful timestamps that reflect a serialization order that satisfies external consistency and offers gloablly consistent reads across the database at a timestamp. Also, data can be dynamically controlled at a fine grain by applications that can specify which data centres contain which data, and can dynamically move data across them. The key enabler for these features is the TrueTimeAPI that explicitly represents time as an interval with bounded time uncertainty (unike standard time interfaces that give clients no notion of uncertainty).

Multi Model

Here we analyse work that focus on providing different levels of consistency for differentiated

data in the same storage. Thus, exploring explore the fundamental tradeoffs between consistency, performance and availability it is argued that not all data needs the same consistency, and that multiple (sometimes dynamic) levels should be offered [48].

PNUTS/Sherpa [13] is a geographically distributed database and was designed by Yahoo! with the intent of supporting its web applications. Following the observation that web applications typically manipulate one record at a time, it focuses on applying a per-record timeline consistency model, where all replicas of a given record apply all updates to the record in the same order. This means that there is no transactional support for multiple records. Independently for each record, one of the replicas is designated as the master, based on the amount of write requests, and all updates to that record are forwarded to it. Records within the same table can therefore be mastered in different clusters.

Yahoo! Message Broker (YMB), which is a topic based pub/sub system, helps to enforce consistency, as data updates are considered to be committed when they have been published to YMB, which will asynchronously propagate them in commit order to different regions, to be applied to non-master replicas. Since all replicas may not have the latest updates, a consistency model with various levels of consistency guarantees is offered:

- **Read-any** returns a possibly stale, yet valid version of the record;
- **Read-critical(version)** returns a version newer than the requested (may imply blocking);
- **Read-latest** returns the latest version and like the above read-critical, may imply waiting for replication to take place;
- **Write:** single record write with ACID guarantees;
- **Test-and-set-write(version)**, writes only if the record version is the one stated.

Consistency Rationing in the Cloud: Pay only when it matters [30]. In this system data is divided into three consistency categories, and present them in the context of a web shop: the first (A) covers data which a consistency violation would result in large penalty costs (e.g. credit card info); the second (B) focuses on data where the consistency requirements vary over time depending on actual availability (e.g. a products stock: a product that has a very low stock needs higher consistency than one who's stock is very high) of an item: finally the third (C) covers data that temporary inconsistencies are acceptable or negligeable (e.g. logging info, users preferences). For data that falls in category A serializability is provided through pessimistic concurrency control: two-phase locking. Category C data, read-your-writes and monotonicity session consistency is provided with last-writer wins conflict resolution for non-comutative updates. Finally, category B offers a runtime adaptive approach mixing both of the above. The adaptivity in category B is based three policies:

i. **General Policy:** based on the frequency of accesses the probability of conflicts is calculated, if it breaks a certain threshold the consistency changes;
ii. **Time Policy:** when a certain temportal metric is reached the consistency changes (e.g. 5 minutes left for the auction to end);
iii. **Numeric Type Policies:** by defining or infering by acesses frequency probabilistic analysis a value threshold, where if it is broken strict consistency is employed.

Transactions are also supported and since consistency guarantees are defined on a data level, operations within a transaction can cover various categories and each operation is handled with the consistency defined for that record, but the results of joins, unions, and any other operations between A and C data provide only C guarantees for that operation.

Bismar [12] is a consistency model that proposes to change the level of consistency during run-time in a way that reduces the monetary cost (focusing on the pay-as-you-go Amazon Elastic Compute Cloud type models) while simultaneously maintaining a low fraction of stale reads. The study is focused on optimizing eventually consistent stores.

For this the authors introduce a new metric called consistency-cost-efficiency, that exposes the tight relation between the degree of achieved consistency for a given monetary cost. The monetary cost is inferred through network cost (intra and inter data centre), storage and instance cost and

combined with a probabilistic estimation of stale reads.

Pileus [50], is a key-value store with a range of consistency choices between eventual and strong consistency and offers the developers the choice to define service level agreements (SLA) that incorporate both consistency as well as latency. Their target applications are those that tolerate relaxed consistency but, nevertheless, benefit from improved consistency.

Rank	*Consistency*	*Latency*	*Utility*
1.	read-my-writes	300 ms	1.0
2.	eventual	300 ms	0.5

Fig. 2: A shopping cart type application SLA [50]

All puts and gets are within the context of a session, and all puts are strictly ordered at a primary site to avoid inter-site conflicts, making it the authority for strong consistency responses.

In Figure 2, is an example of such SLA, where the developer can define and variable amount of subSLA's, ranking them in preference. Each subSLA defines a consistency-latency pair where the consistency value can have one of the following values: strong, eventual, read-my-writes, monotonic, bounded(t), causal. The utility of a subSLA is a number that allows applications to indicate its relative importance and along with network monitoring and storage node information gathered by the Client, will help making a decision on which SLA to attempt to satisfy and which storage nodes to contact.

Transaction support is also offered in Pileus [49] with snapshot isolation where the snapshot being read is determined based on consistency choice. One of the primary sites is chosen as the coordinator of the transaction. The client, at the end of the transaction, will send all its operations to the coordinator which will then send each put to the respective responsible primary-site. Each site will respond with their highest timestamp and the coordinator will decide the transactions interval ([initTS, endTS]), and send it to all primarys. Afterwards the primarys will analyse the interval and respond informing if there are any conflicts or not, and only in the absence of conflicts will the transaction commit.

Others

MeT [16] is an elastic system that autonomously manages the addition and removal of nodes and also reconfigures them heterogeneously according to workload patterns with the objective to have better performance and resource usage. By heterogeneously it means not all nodes are configured equally. Focusing on HBase, the authors show that, in order to have an elastic data store, simply adding or removing nodes is not sufficient but it is instead necessary to take into account the current workload together with the cluster state, and adapt the cluster accordingly, adding/removing nodes or altering HBase configuration parameters such as block cache size and memstore size

This is achieved by the use of a middle-ware, comprising a Monitor component that gathers important statistics of the running cluster and periodically passes them to the Decision Maker component that will reason over them and decide if the current cluster configuration is acceptable or not. It passes the decision to the Actuator who will enforce the new decided configuration in the running cluster.

3.4 Benchmarks

Benchmarking Cloud Serving Systems with YCSB [14] is a benchmarking tool that was designed to fit all NoSQL data stores, so one can compare them to each other, with the help of its various workloads emulating different types of applications.

Their focus is to evaluate data store on performance and scaling elastically. In the performance field the objective is to observe the behaviour under heavy load situations, mainly throughput and latency. In the scaling field, the objective is to check the impact on performance, as more machines are added to the system. It is expected that with double the load and machines the performance is the same, also increasing the number of machines in the middle of a workload still should be able to improve performance.

The workload client must make random choices between the available operations (insert, read, scan, update) and what records to access when generating load. To govern these decisions, YCSB has the following built-in distributions:

- **Uniform:** Choose an item uniformly at random. For example, when choosing a record, all records in the database are equally likely to be chosen.
- **Zipfian:** Choose an item according to the Zipfian distribution. For example, when choosing a record, some records will be extremely popular (the head of the distribution) while most records will be unpopular (the tail).
- **Latest:** Like the Zipfian distribution, except that the most recently inserted records are in the head of the distribution.
- **Multinomial:** Probabilities for each item can be specified. For example, we might assign a probability of 0.95 to the Read operation, a probability of 0.05 to the Update operation, and a probability of 0 to Scan and Insert. The result would be a read-heavy workload.

Combining the different distributions with the choices of which operations to perform, and what records to access, they arrived at the following workloads:

Workload	Operations	Record selection	Application example
A—Update heavy	Read: 50% Update: 50%	Zipfian	Session store recording recent actions in a user session
B—Read heavy	Read: 95% Update: 5%	Zipfian	Photo tagging; add a tag is an update, but most operations are to read tags
C—Read only	Read: 100%	Zipfian	User profile cache, where profiles are constructed elsewhere (e.g., Hadoop)
D—Read latest	Read: 95% Insert: 5%	Latest	User status updates; people want to read the latest statuses
E—Short ranges	Scan: 95% Insert: 5%	Zipfian/Uniform*	Threaded conversations, where each scan is for the posts in a given thread (assumed to be clustered by thread id)

Fig. 3: Workloads in the core YCSB package [14].

Where workload E, uses the Zipfian distribution to choose the first key in the range, and the Uniform distribution to choose the number of records to scan.

TPC benchmarks[19] define transaction processing and database benchmarks with the objective to disseminate objective and, verifiable performance data to the industry. TPC-C and TCP-E are the most relevant benchmarks to this work.

TPC-C is an on-line transaction processing benchmark (OLTP) that simulates a complete environment where a population of terminal operators executes transactions against a database. It also provides a metric (tpm-C) that does not just measure a few basic computer or database transactions, but measures how many complete business operations can effectively be processed per minute.

PyTPCC[20] is a Python-based framework of the TPC-C OLTP benchmark for NoSQL systems. The framework is designed to be modular, to allow for new drivers to be written for different systems. As of the writing of this report, they support several commercial NoSQL stores.

TPC-E is also an OLTP workload but focuses on modelling the activity of a brokerage firm that must manage customer accounts, execute customer trade orders, and be responsible for the interactions of customers with financial markets.

[19] http://www.tpc.org
[20] https://github.com/apavlo/py-tpcc

RUBiS[21] is an open-source, auction site prototype modelled after eBay.com that is used to evaluate application design patterns and application servers performance scalability. It implements the core functionality of an auction site: selling, browsing and bidding. Two workload mixes are provided, one based on a browsing mix made up only of read-only interactions, and another workload that includes 15% read-write interactions. Although RUBiS implementation is based on PHP using a local MySQL database system, it has been successfully adapted to fit other stores (L-RuBiS [55]).

Analysis and Discussion

In this section (Section 3) we began by discussing the state of the art of NoSQL data stores (Figure 4). First we analysed the possible data models available in NoSQL stores and how efficient the wide column model is due its sparse nature, by avoiding to store null values. Document stores allow for various attributes to be associated with a key, therefore making possible more complex queries. Also Document stores are breaking the boundary towards Graph stores, as all the presented stores support some type of mechanism to allow linkage between documents. OrientDB is a self proclaimed graph store but the graph "nodes" are JSON objects, as used in most document stores.

		CAP	Concurreny Control	Partioning	Replication	Geo-Replication
Document	Riak	AP	Vector clocks	Hash based	Hash based (virtual nodes)	yes
	MongoDB	CP	DB level locks	Range Based	Master-Slave	yes
	RethinkDB	CP	MVCC + locks	Range Based	Master-Slave	yes
	CouchDB	AP	MVCC	Hash based	Master-Master	no
Wide Column	Cassandra	AP	-	Hash based	Hash based (virtual nodes)	yes
	HBase	CP	MVCC + locks	Range Based	Distributed Storage System	yes
	HyperTable	CP	MVCC + locks	Range Based	Distributed Storage System	no
	BigTable	CP	MVCC + locks	Range Based	Distributed Storage System	no
	Accumulo	CP	MVCC + locks	Range Based	Distributed Storage System	no
Key value	Redis	CP	Optimistic locking	Hash based	Master-Slave	no
	Hyperdex	CP	value-dependent chaining	Hash based	Distributed Storage System	no
	DynamoDB	AP	-	Hash based	Hash based	no
	Voldemort	AP	Vector clocks + optimistic locking	Hash based	Hash based	no
Graph	Neo4j	CP	Locks	-	Master-Slave	no
	OrientDB	CP	MVCC + locks	Hash based	Master-Master	no

Fig. 4: NoSQL data stores analysis table.

We saw how each store partitioned their data, as this concept falls into basically two categories: range based and hash based. The advantage of hash based technique is that it comes with a inherent load balancing capability, which is further accentuated if coupled with hash based replication with virtual nodes. The downside is that there is no guarantee that related data ends up in the same server. Systems that rely on range based partitioning usually require some sort of extra service to map data to servers.

Finally we saw that only a few commercial systems provide data centre awareness. We also analysed them in terms of consistency and concurrency control, as most systems rather suffer from some storage overhead by providing multiple versions of the same, in order to implement stronger consistency. Others rely simply on locking mechanism. Others refrain from locking and accept possibility of conflicts either resolved by deterministic functions or by letting clients deal with them.

After having gone deeper into the consistency topic, as we saw different approaches that balanced either several consistency levels or focus on a certain niche. We analysed how to improve eventual consistent systems through the use a probabilistic model to predict the staleness of partial quorums and through the use of vector field consistency to provide quality of data features.

[21] http://rubis.ow2.org/

Work on causal consistency provided insight into how this model provides very useful semantics for developers but requires deterministic functions to resolve conflicts and an all-to-all replication to provide convergence. We also saw how tracking potential causality can become a very heavy task with large amounts of metadata, and as a solution, systems could opt to track application-defined explicit causality as well as if you add data centres to a multi data centre causal system, it does not help the system scale. In addition, we covered a system that demonstrated how to add causality to an eventual consistent store through a shim layer, and a system that went further and implemented it in a geo-replicated store.

Where other system rely on the simplistic last writer wins strategy to resolve conflicts, we address systems that resort to use conflict-free data types and avoid conflicts altogether but limiting the type of data the store can hold. The system Gemini instead, resorted to strict consistency but searched in every operation the possibility to transform it into a commutative operation are therefore apply the operation locally without the worry of conflict resolution.

We then looked at systems that enforce strong consistency across multiple data centres, such as Walter, Spanner and Lynx. Where the first relied on counting sets and preferred sites to avoid transactional conflicts. Spanner relies on Paxos and its TrueTimeAPI to enforce the serial order and external consistency. Lynx used a chaining protocol that offers a sometimes false sense of speed and does not offer external consistency.

To help our decision with transactional support for HBase, we started by analysing the solution used for Bigtable (Percolator), and followed up through various systems that made use of special purpose tables inside HBase to coordinate transactional metadata. Through our analysis, we concluded this brought too much storage overhead. Omid takes metadata away from the storage and provides faster transactions, although the TSO can be seen as a possible bottleneck; the development of MegaOmid is addressing that issue. Although Omid is an incomplete and ongoing project, we see that as a motivation to contribute to a renowned open-source project run by Yahoo!, as the same can be said about HBase.

Finally, multiple consistency models systems were analysed starting with Yahoo's PNUTS, that provides different calls for the different consistency levels. Also we saw how a system alters between strict and session based consistency using policies that somehow can be related to vector field consistency. Bismar makes adjustments based on monetary cost in the cloud. Pileus also relates, as it uses SLAs with utility metrics. All the aforementioned multi model systems relate to our proposal, as some type meta information is either inferred or user defined to regulate the different consistency levels.

4 Proposed Architecture

In this section we will present the overall architecture for HBase++, each of its main components and how they cooperate. The first subsystem/component addresses how to enforce serializable transactions and track causality within the cluster (Section 4.1). Afterwards we will detail the scheduling algorithm for geo-replication (Section 4.2), and finally we present the geo-replication protocol and how it enforces the scheduling order provided by the algorithm across clusters (Section 4.3).

4.1 Architecture Overview

Our goal is to provide the user the ability to associate bounding values, inspired by and enhancing the VFC^3 consistency model [21, 42], on data stored in HBase and therefore defining maximum divergence of the constraints:

- **Time (θ).** Specifies the maximum time a replica can be without being updated with is latest value. Considering $\theta(o)$ provides the time (e.g., seconds) passed since the last replica update of object o, constraint κ_θ, enforces that $\theta(o) < \kappa_\theta$ at any given time.
- **Sequence (σ).** Specifies the maximum number of updates that can be applied to an object without refreshing its replicas. Considering $\sigma(o)$ indicates the number of applied updates, this sequence constraint κ_σ enforces that $\sigma(o) < \kappa_\sigma$ at any given time.

- **Value (ν).** Specifies the maximum relative difference between replica contents or against a constant (e.g., top value). Considering $\nu(o)$ provides that difference (e.g., in percentage), this value constraint κ_ν enforces that $\nu(o) < \kappa_\nu$ at any given time. It captures the impact or importance of updates on the objects internal state.

These constraints, coupled together with the transactional enforcement within the cluster and causal relation tracking, will determine the creation of a schedule for the geo-replication of operations towards slave clusters. This way break from the inferred serial order provided currently by the eventual consistent enforcement, and the underlying transactional system, but it is one that upholds the constraints defined by the VFC^3 model and causal relations between the groups of operations. This provides the ability to give urgency of replication to more important data and relax other data which its global appearance might not be so urgent. In essence it allows awarding different quality of service to different data without breaking application semantics and the typical consistency assumptions developers make over accessed data. With this model, it will also be possible to infer a specific cost of replication for each operation, while enforcing the desired semantics. In Figure 5 we present a high-level view of the architecture of this project.

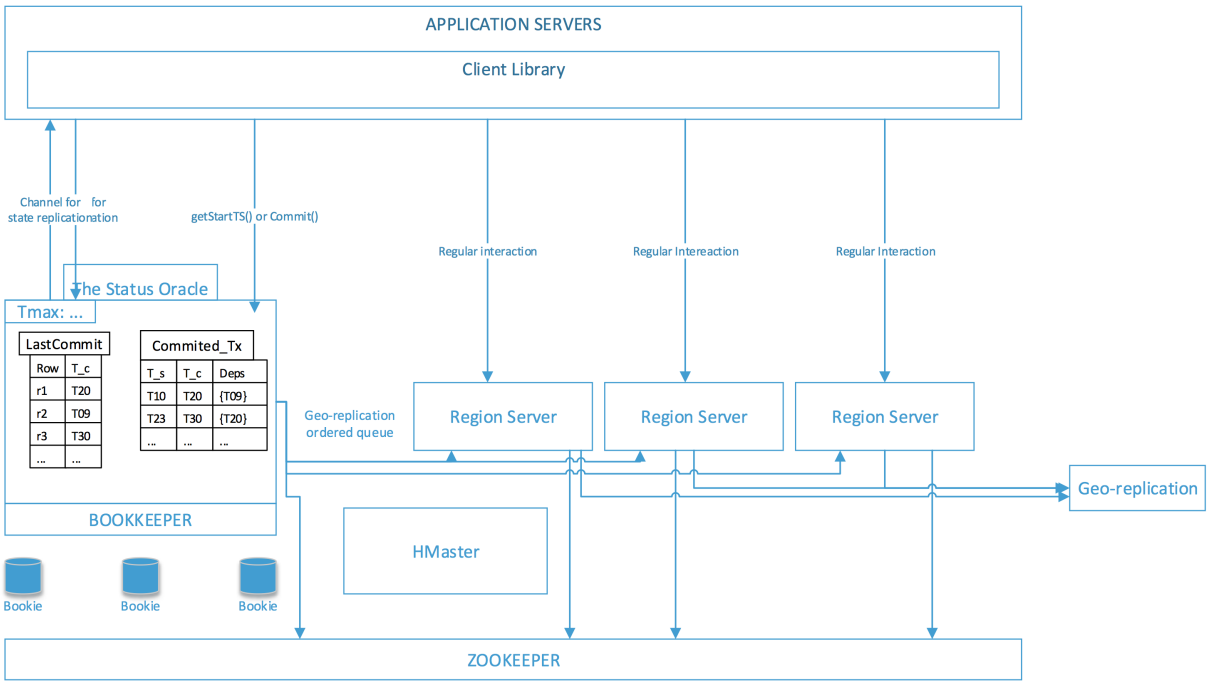


Fig. 5: Cluster architecture

The client library will communicate with the transaction Status Oracle (TSO) to coordinate their transactions, as transactions commit clients also submit the dependencies and progress on the bounding constraints. The TSO will reason about the serial order of operations and reMASTorder it to oblige to the constraints but without breaking causal relations. As the order is defined, the TSO pushes the queue to Region Servers in order for them to follow up with replicating the operations to the other cluster. Each Region Server will look to the group of operations set to replicate, check which of them are they responsible for, and propagate them.

Transactional Enforcement and Causality Tracking. As described in section 3.3, Omid[22] is a system that provides transactional support for HBase with either snapshot isolation [8] or write-snapshot isolation [52] which guarantees serializability; it also relies on BookKeeper [28] for durability. We will leverage this as an underlying system for our transactions and extend it to track causal relations between client operations.

To add causal tracking to this system we simply need to add the transactions read and write set to the list of committed transactions already saved in the TSO. With this method we can track every potential causal relation between groups of operations. Another option with less storage and memory requirements is to track only explicit application defined causal relations where clients will, for each transaction, submit the nearest one-hop distance dependency [37] in the context of its session.

So in this second mode the client library, must maintain a causality graph where for each new transaction, submitted in a session, they will also submit the nearest one-hop dependency. In addition, they must also submit the constraint values, which are placed next to data they have accessed in the HBase tables as metadata, as is explained in the following section.

Bounding Values. Inside HBase we will store the user defined bounding values and associate them with the data whose constraints they are meant to represent. Four levels of granularity are considered here, ordered from the one with least storage and metadata overhead requirements to the most costly. The first level, although very coarse it can fit some developer needs and requires the least storage requirements, is to associate a single bounding vector to an entire table. In Figure 6 is shown the second level, where different values for each row in a table are defined, making all columns in that row have the same bounds. Represented in Figure 7 is the next level of granularity that defines the values on a per column-family basis, covering all columns inside a column-family. In Figure 8, we show the highest demanding level but the most fine-grained where the bounding values can be defined differently for each cell leading to a column of metadata for each column.

	cf1:col1	cf1:col2	cf2:col1	cf2:col2	cf_vfc:col_vfc
row1	value	value	value	value	$(\theta, \sigma, v)_1$
row2	value	value	value	value	$(\theta, \sigma, v)_2$

Fig. 6: Different bounds for each row across all the column families (cf).

	cf1:col1	cf1:col2	cf1:col_vfc	cf2:col1	cf2:col2	cf2:col_vfc
row1	value	value	$(\theta, \sigma, v)_1$	value	value	$(\theta, \sigma, v)_3$
row2	value	value	$(\theta, \sigma, v)_2$	value	value	$(\theta, \sigma, v)_4$

Fig. 7: Different bounds for each row inside each column family (cf).

	cf1:col1	cf1:col1_vfc	cf1:col2	cf1:col2_vfc	cf2:col1	cf2:col1_vfc	cf2:col2	cf2:col2_vfc
row1	value	$(\theta, \sigma, v)_1$	value	$(\theta, \sigma, v)_3$	value	$(\theta, \sigma, v)_5$	value	$(\theta, \sigma, v)_7$
row2	value	$(\theta, \sigma, v)_2$	value	$(\theta, \sigma, v)_4$	value	$(\theta, \sigma, v)_6$	value	$(\theta, \sigma, v)_8$

Fig. 8: Different bounds for each cell.

The basic storage requirements for each constraint will be five integers being the first four two pairs, where each pair will represent value and sequence constraints. The second value of the pair will be the bounding value and the first the progress towards it. The 5th integer will be the time constraint.

[22] http://yahoo.github.io/omid/

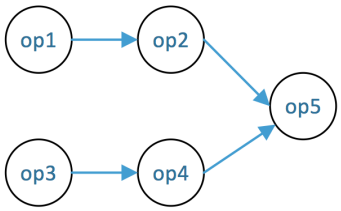
4.2 Scheduling & Cost Inference

First we must define how the earliest deadline first algorithm will reason about each operation's individual constraint values, and how it will define the constraint values of a transaction based on all of its operations. Considering all bound values are defined for individual operations, the bound closest to being broken will define that operation's level of urgency. The same logic will be applied to a transaction: when looking at all of its operations, the one that implies the most urgency will define the urgency of the entire transaction and enclosed operations, regardless of the presence of very relaxed urgency operations in the transaction (Figure 9).

Transaction (Tx)	Tx1		Tx2		
Operation (Op)	Op1-1	Op1-2	Op2-1	Op2-2	Op2-3
Op. Constraint	[0.8, 7:7, 13:20]	[0.5, 0:9, .]	[0.3, 2:5, .]	[0.7, ., 5:60]	[., 1:3, 40:40]
Most strict constraint	Op1-1		Op2-3		

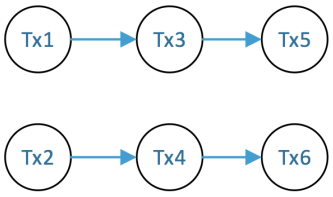
Fig. 9: Operations whose bounding constraints have reached threshold hold the strictest constraint and represent the transaction. Undefined bound are represented by a dot.

The scheduling algorithm will reason broadly as follows: for all transactions it will identify the operations with the highest constraints and will define that constraint as the transaction constraint. It will then start from the transaction with the highest constraints, and place next in queue, its yet non propagated causal dependencies. Between two independent causal dependencies of that transaction, the highest constraint one will be placed first in queue. This will basically be a Breadth-First Search where child ordering decision will be based on highest constraint value as it is shown in Figure 10. Ordering of transactions is the same (Figure 11).



Serial Order	1, 2, 3, 4, 5
Constraint Order	5, 4, 2, 1, 3
Queue Order	1, 3, 4, 2, 5

Fig. 10: The serial order presents operation 1 as the 1st applied operation. In terms of constraints operation 5 has the highest constraints but as causal order can not be broken, its queue order will not change. Independent pairs of operations will be moved according to constraints of Figure 9



Serial order	1, 2, 3, 4, 5, 6
Constraint order	4, 3, 1, 2, 5, 6
Queue order	2, 4, 1, 3, 5, 6

Fig. 11: A scheduling example that focus on two independent sets of dependent transactions.

As transactions are ordered with relation to other independent transactions, the idea is to satisfy all the constraints with the least cost possible. We define cost as the following:

- **Operation move cost.** The number of operations that had to be delayed in order for this operation to be propagated earlier (e.g. if an operation is moved from the 10th position to the 2nd, it has an associated cost of 8, as it causes the delay of 8 operations).
- **Total move cost.** The total cumulative number of queue positions of the moves made.

This means we intend to minimize the amount of "moves" in the queue, done to operations with no causal relation, by stepping away the least possible from the serial order, but yet maintaining the constraints satisfied. Transactions can somewhat be viewed as a single operation due to the strictest constraint operation being the one that represents the bounding values of the transaction, but the cost of a move will be different as we move a transaction we are moving a certain amount of operations together.

4.3 Geo-Replication

As stated, the ordered queue is pushed to the Region Servers so they can have the necessary info on the order of the updates to propagate. For this to work correctly, the TSO will have to make a decision about the head of the queue and define a final ordering that will not be changed, so it can be removed from the queue and safely pushed to Region Servers. This group of operations comprising the head will be called an *epoch*. The amount of updates to include in an epoch will be decided according to the size of the operations and their urgency to manage the lag and batching efficiently. For example if bounds are very close to being broken it is accepted that a smaller epoch is conceived so constraints are not broken. It is also accepted to delay somewhat the geo-replication (without breaking constraints) to create a larger epoch and augment the amount of updates in one commit protocol enforcement.

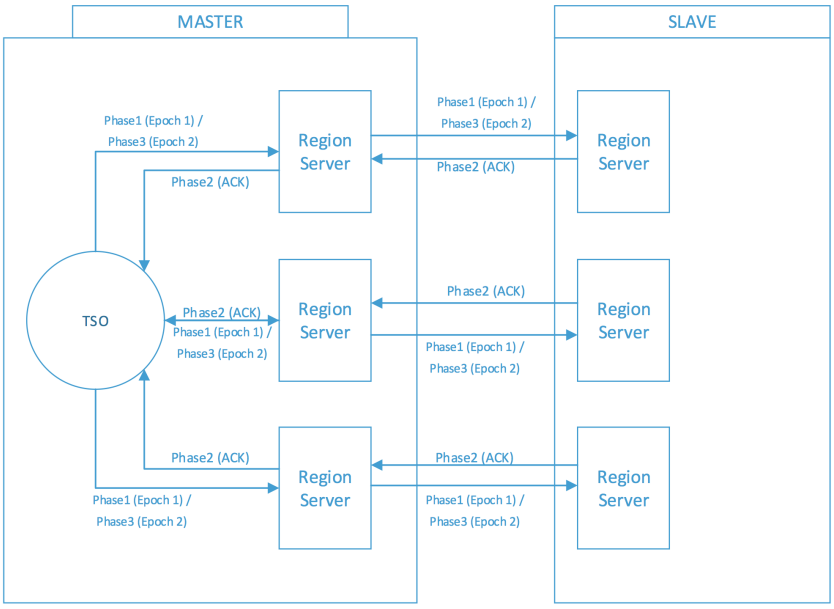


Fig. 12: Geo-replication commit protocol

In the commit protocol (Figure 12) the TSO will act as a coordinator. It pushes the new epoch boundary to the Region Servers (RS), and each one of them will analyse the epoch, pick out the updates they are in charge of, and propagate them to the slave cluster (phase 1). The RS on the slave cluster will acknowledge the reception and respond to the RS on the master (phase 2), similarly to the voting phase in two-phase commit but negative responses are not possible. On the master cluster the TSO will gather all responses and send commit command along the new epoch (phase

3/phase 1 of the next round). This is very similar to the two-phase commit protocol but with a few improvements inspired by Sinfonia's [2] mini-transactions, and where replies are always positive and serve only to ensure consensus and global knowledge of epoch boundaries.

5 Proposed Evaluation

To evaluate our solution, we will focus on cost of the new added features in terms of how they will affect the performance and scalability of the system, versus the benefits these new features bring. We will make use of the PyTPCC HBase adaptation for transactional workloads and the YCSB benchmark.

- The correctness of the system must be evaluated, as to make sure causal relations are respected as the ordering algorithm acts. Another factor is the correctness of the commit protocol as to making sure the no later epoch is exposed before an earlier one, and also the behaviour in the presence of failures, when this is not achieved, quantify the extent and/or magnitude of the set of affected operations (with the metrics of Section 4.2 or with error measurements using geometric norms.

- The effectiveness of the scheduling algorithm to uphold the user specified constraints with the least possible cost, but it must be taken into account as above that certain constraint combinations will be simply impossible to serve. For example, in a worst case scenario where every operations has very strict constraints and no relaxed operations to leverage, the systems best option would be to simply default to serial order. We can thus assess the cost for each level of correctness and quality demands.

- In terms of causality tracking and constraint metadata we must evaluate the memory, storage and network overhead. Also the delay induced by the geo-replication commit protocol, for the propagation of epochs.

- The full behaviour of the system will be analysed in terms of performance and scalability. By emulating several clients, we plan to evaluate how the system will in terms of memory requirements in the TSO, identifying the point where queues growth speed will overcome the ability to propagate updates to slave data centres. We will also compare the performance of our geo-replication to the default HBase mode and balance the latency our enforcements will bring against the added features.

6 Conclusion

We now wrap up the report with some concluding remarks, which we intend on being a light overview of what was studied and referenced, and also of what we hope to achieve with our work, namely the architecture and its goals.

We started this document by describing and classifying current commercial NoSQL stores, by their most defining features as we later transitioned through the most relevant feature to our work, we analysed how system coped with the inherent struggle between consistency and latency, showing different types of consistency models as well as modes who incorporated multiple models that were either static or could transition dynamically according to various criteria.

We then presented our proposal to improve HBase's eventual consistent geo-replicated approach with data urgency with data urgency that abides to causal relations, by proposing a new scheduling algorithm. It will incorporate data bounding values and causal relations into its reasoning as it disrupts serial order, as well as define a geo-replication protocol to enforce such ordering. Finally, we presented the metrics that will support the evaluation of our solution.

References

[1] Daniel Abadi. Consistency tradeoffs in modern distributed database system design: Cap is only part of the story. *Computer*, 45(2):37–42, 2012.

[2] Marcos K Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 159–174. ACM, 2007.

[3] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. The potential dangers of causal consistency and an explicit solution. *Proceedings of the Third ACM Symposium on Cloud Computing - SoCC '12*, pages 1–7, 2012.

[4] Peter Bailis, Ali Ghodsi, J Hellerstein, and Ion Stoica. Bolt-on causal consistency. *ACM SIGMOD*, 2013.

[5] Peter Bailis, Shivaram Venkataraman, Michael J Franklin, Joseph M Hellerstein, and Ion Stoica. Probabilistically bounded staleness for practical partial quorums. *Proceedings of the VLDB Endowment*, 5(8):776–787, 2012.

[6] Jason Baker, Chris Bond, James Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Léon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, volume 11, pages 223–234, 2011.

[7] DGFFJ Benjamin and RM Yabandeh. Lock-free Transactional Support for Distributed Data Stores. *sigops.org*, pages 3–4.

[8] Hal Berenson, Elizabetb O Neil, Patrick O Neil, and Sybase Corp. A Critique of ANSI SQL Isolation Levels. pages 1–10, 1995.

[9] Eric A Brewer. Towards robust distributed systems. In *PODC*, page 7, 2000.

[10] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.

[11] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[12] Houssem-Eddine Chihoub, Shadi Ibrahim, Gabriel Antoniu, and Maria S Perez. Consistency in the Cloud: When Money Does Matter! *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, pages 352–359, May 2013.

[13] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, August 2008.

[14] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. *Proceedings of the 1st ACM symposium on Cloud computing - SoCC '10*, page 143, 2010.

[15] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google?s globally-distributed database. In *Proceedings of OSDI*, volume 1, 2012.

[16] Francisco Cruz, Francisco Maia, Miguel Matos, Rui Oliveira, João Paulo, José Pereira, and Ricardo Vilaça. Met: Workload aware elasticity for nosql. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 183–196. ACM, 2013.

[17] Giuseppe Decandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. *SOSP*, pages 205–220, 2007.

[18] Andrei Deftu and Jan Griebsch. A scalable conflict-free replicated set data type. In *Distributed Computing Systems (ICDCS), 2013 IEEE 33rd International Conference on*, pages 186–195. IEEE, 2013.

[19] Robert Escriva, Bernard Wong, and E G Sirer. Warp: Multi-Key Transactions for Key-Value Stores. *hyperdex.org*.

[20] Robert Escriva, Bernard Wong, and Emin Gün Sirer. Hyperdex: A distributed, searchable key-value store. *ACM SIGCOMM Computer Communication Review*, 42(4):25–36, 2012.

[21] S Esteves, J Silva, and L Veiga. Quality-of-service for consistency of data geo-replication in cloud computing. *Euro-Par 2012 Parallel Processing*, pages 285–297, 2012.

[22] Ariel J Feldman, William P Zeller, Michael J Freedman, and Edward W Felten. SPORC : Group Collaboration using Untrusted Cloud Resources.

[23] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, October 2003.

[24] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51–59, 2002.

[25] Robin Hecht and Stefan Jablonski. NoSQL evaluation: A use case oriented survey. *2011 International Conference on Cloud and Service Computing*, pages 336–341, December 2011.

[26] Patrick Hunt, Mahadev Konar, Flavio P Junqueira, and Benjamin Reed. Zookeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, volume 8, pages 11–11, 2010.

[27] Flavio Junqueira, Benjamin Reed, and Maysam Yabandeh. Lock-free transactional support for large-scale storage systems. *2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 176–181, June 2011.

[28] Flavio P Junqueira, Ivan Kelly, and Benjamin Reed. Durability with bookkeeper. *ACM SIGOPS Operating Systems Review*, 47(1):9–15, 2013.

[29] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 654–663. ACM, 1997.

[30] Tim Kraska, Martin Hentschel, Gustavo Alonso, and Donald Kossmann. Consistency rationing in the cloud: Pay only when it matters. *Proc. VLDB Endow.*, 2(1):253–264, August 2009.

[31] Avinash Lakshman and P Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 2010.

[32] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[33] Mihai Letia, Nuno M. Preguiça, and Marc Shapiro. Crdts: Consistency without concurrency control. *CoRR*, abs/0907.0929, 2009.

[34] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 265–278, Berkeley, CA, USA, 2012. USENIX Association.

[35] MR List. CAP Twelve Years Later: How the" Rules" Have Changed. *infoq.com*, (February):23–29, 2012.

[36] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. Don t Settle for Eventual : Scalable Causal Consistency for Wide-Area Storage with COPS Categories and Subject Descriptors. pages 1–16.

[37] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. Stronger semantics for low-latency geo-replicated storage. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI?13), Lombard, IL*, 2013.

[38] Prince Mahajan, Lorenzo Alvisi, and Mike Dahlin. Consistency, availability, and convergence. *University of Texas at Austin TR-11-22 (May)*, 2011.

[39] V Padhye and A Tripathi. Scalable Transaction Management with Serializable Snapshot Isolation on HBase. 2012.

[40] Vinit Padhye and Anand Tripathi. Scalable Transaction Management with Snapshot Isolation on Cloud Data Management Systems. *2012 IEEE Fifth International Conference on Cloud Computing*, pages 542–549, June 2012.

[41] Daniel Peng and Frank Dabek. Large-scale Incremental Processing Using Distributed Transactions and Notifications. *OSDI*, 2006:1–15, 2010.

[42] ÁG Recuero, S Esteves, IID Lisboa, and L Veiga. Quality-of-Data for Consistency Levels in Geo-replicated Cloud Data Stores. *gsd.inesc-id.pt*.

[43] Sherif Sakr, Anna Liu, Daniel M Batista, and Mohammad Alomari. A Survey of Large Scale Data Management Approaches in Cloud Environments. *IEEE Communications Surveys & Tutorials*, 13(3):311–336, 2011.

[44] Nuno Santos, Luís Veiga, and Paulo Ferreira. Vector-field consistency for ad-hoc gaming. *Middleware 2007*, 4834:80–100, 2007.

[45] Eric Schurman and Jake Brutlag. The user and business impact of server delays, additional bytes, and http chunking in web search. In *Presentation at the O?Reilly Velocity Web Performance and Operations Conference*, 2009.

[46] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, May 2010.

[47] Yair Sovran, Russell Power, Marcos K Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles - SOSP '11*, page 385, 2011.

[48] Doug Terry. Replicated data consistency explained through baseball. Technical report, Technical Report MSR-TR-2011-137, Microsoft Research, 2011.

[49] Douglas Terry, Vijayan Prabhakaran, and Ramakrishna Kotla. Transactions with Consistency Choices on Geo-Replicated Cloud Storage. *research.microsoft.com*, 1234.

[50] Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 309–324, New York, NY, USA, 2013. ACM.

[51] Luís Veiga, André Negrão, Nuno Santos, and Paulo Ferreira. Unifying divergence bounding and locality awareness in replicated systems with vector-field consistency. *Journal of Internet Services and Applications*, 1(2):95–115, August 2010.

[52] Maysam Yabandeh and Daniel Gómez Ferro. A critique of snapshot isolation. *Proceedings of the 7th ACM european conference on Computer Systems - EuroSys '12*, page 155, 2012.

[53] Chen Zhang and Hans De Sterck. Supporting multi-row distributed transactions with global snapshot isolation using bare-bones HBase. *2010 11th IEEE/ACM International Conference on Grid Computing*, pages 177–184, October 2010.

[54] Chen Zhang and Hans De Sterck. Supporting multi-row distributed transactions with global snapshot isolation using bare-bones hbase. In *Grid Computing (GRID), 2010 11th IEEE/ACM International Conference on*, pages 177–184. IEEE, 2010.

[55] Yang Zhang, Russell Power, Siyuan Zhou, Yair Sovran, Marcos K Aguilera, and Jinyang Li. Transaction chains: achieving serializability with low latency in geo-distributed storage systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 276–291. ACM, 2013.

A Planning

Here we present our planning schedule for the implementation of the solution.

Planning		
Tasks	**Details**	**Duration**
Design Software Solution	- Deepen HBase code study - Deepen Omid code study - Design the solution	February (2 weeks) February (1 week) February (1 week)
Implementation of the Solution	- Extend Omid with causality tracking - Extend HBase with to support the storage of bounding values - Implement the scheduling algorithm - Implement the geo-replication protocol	March (3 weeks) March (1 week) Abril May
Development of the Evaluation System	- Extend PyTPCC - Extend YCSB	June (2 weeks) June (2 weeks)
Testing and Performance measurements	- Evaluate the implemented solution	July (2 weeks)
Thesis Final Report Writting	- Write thesis report	July (2 weeks) - August
Review and Submission	- Report review and submission	July (2 weeks) - August
Documentation	- Document design decisions, code, tests and notes for final document	January - September
Weekly / Bi-weekly meetings	- Report and analyse progress	January - September