The Reading Report for Google Megastore

Abstract: Megastore is a storage system designed to meet the demand of today`s interactive online service. Therefore, it provides a set of high-scalable and available services. Besides, it also provides fully support for ACID semantics to motivate fast development, as well as geographical-distributed data storage system to guarantee data safety. This paper discusses the basic concepts in Megastore and its core replication algorithm. It also present deployment issues as supplementary.

Keywords: Megastore, Large database, Paxos, Distributed transaction, Bigtable

Index

1 Introduction

As today`s interactive online services are progressing at the speed of light, their natural characteristics cause huge pressure on traditional cloud computing platform at 3 aspects: **development, deployment and administration** (See Figure 1-1).

Take an example. Almost 4 months after the launch event in 2011, an online instant message named WeChat only takes 4 million users. But the number of active users grows to 800 million at the second quarter this year.

From the aspect of cloud computing service provider, in order to meet the strict requirements that originated from WeChat-like services, the cloud computing platform beneath must be highly scalable to satisfy the explosive growth in total user number. Moreover, it also requires rapid development to catch up with current trend, in order to take the lead in the competition with other services. Finally, its clients also demand low-latency service and the guarantee of availability and safety for their personal data.

But when the implementation of desired service comes into practice, situation becomes complicated. Admins want it to become easy-deployed with high-available administrative policy. Developers want it can be managed in an automatic way and keep the ACID semantics. And users want a high-available service with good experience.

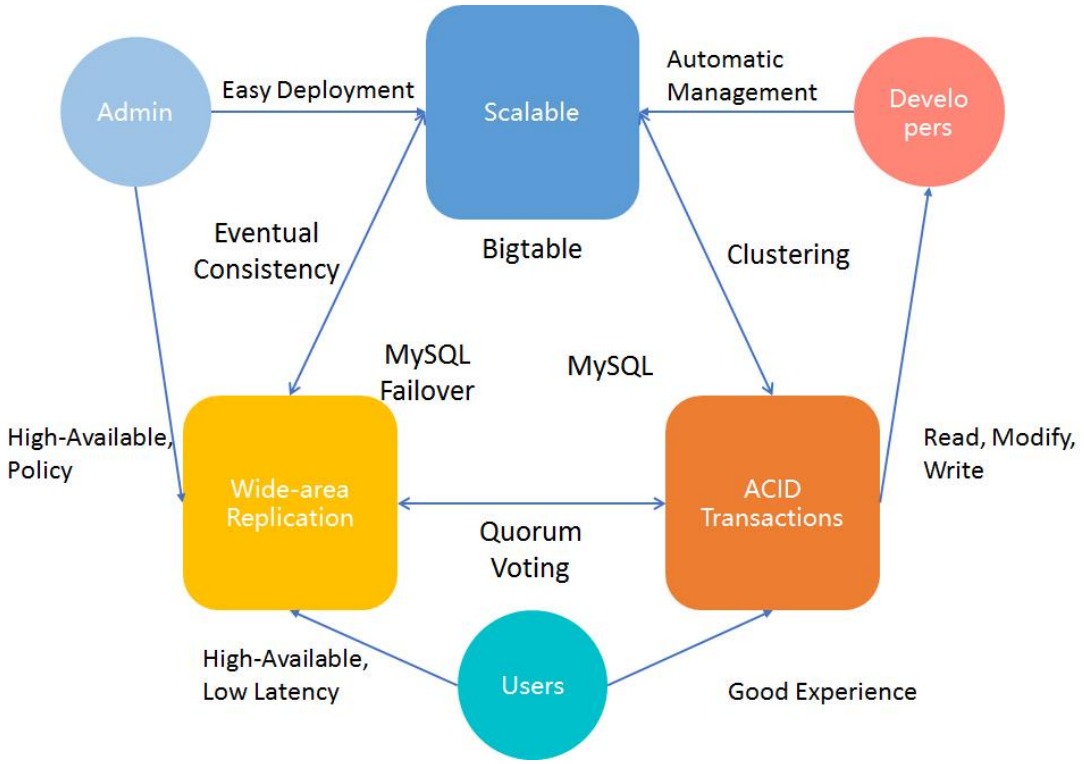

Figure 1-1 Requirements from all Stakeholders

To tackle all these seemingly conflicting requirements, we create tons of technologies. For example, we use Bigtable to improve scalability, use MySQL Failover for wide-area replication and MySQL for ACID transaction.

However, the more modules a single project has, the more complicated it will be. Therefore, an integrated platform absorbing all the benefits from Bigtable, MySQL and MySQL Failover, is necessary for the future of today`s interactive online services.

2 Availability and Scale

Unfortunately, the hardware is of limited scale and availability, and data is not always stored in the same datacenter as before. Therefore, to settle down this problem, we have to turn to system design, so that the whole system can be bound

in a proper way with minimum disadvantages.

So far, we had two practical ways to achieve it:

1. **For Availability.** We implemented a synchronous and fault-tolerant log replicator to optimized for long-distance data links (i.e., by Chubby).
2. **For Scalability.** We introduced NoSQL data storage with its own replicated log (i.e. by Bigtable) to maximize the data scale.

2.1 Replication Algorithms

Before we actually step into the door of Megastore, we should have a brief understanding of current replication algorithms, in order to get a better understanding of Megastore`s approach in replicating data through geographical distributed datacenters.

2.1.1 Traditional Algorithms

Currently, we had three kinds of traditional algorithms:

1. **Asynchronous Master/Slave**

Master maintains the write ahead log. If there are appends to the log, master will acknowledge slave through Message Queue(MQ) in parallel. But it had an obvious weakness: if the target slave is down, data loss will occur. Therefore, an additional consensus protocol is needed.

2. **Synchronous Master/Slave**

Master informs its slaves after the changes are applied. But it also has its own weakness: it needs an external system to keep the time.

3. **Optimistic Replication**

Mutations propagate through the group asynchronous and independently. As the order of propagation is unpredictable, it is impossible to implement transaction control in such algorithm.

2.1.2 Paxos

In Paxos, there are no distinguished masters and slaves. They use **vote** and catch up and to keep data consensus. This algorithm is applied in Megastore with proper modifications, and we will give a thorough explanation in later section.

2.2 Entity Group

To get a complete understanding of Megastore, we have to understand the concept of Entity Group(EG).

An entity group is an abstract concept for a set of related data stored in Megastore.

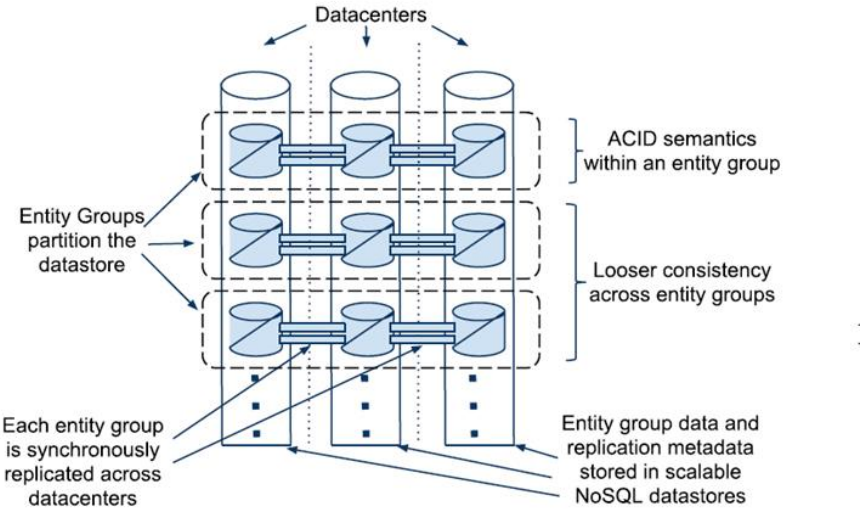

Figure 2-1 Scalable Replication

In Figure 2-1, we assume that there are three datacenters, which showed as three vertical cylinders. Also, there are many EGs, but we just show three of them

in this figure to make it simple and clear. All data are stored in different Bigtables in different datacenter physically.

Getting deep in it, we will see that, there are EG partitions, which store the different parts of data within an EG in some geographical distributed datacenters. Partitions can be the copies of a particular instances within an EG, but it also can be part of that instance, which depends on the storage dispatch algorithm in Megastore.

Within a single EG, Megastore provides full ACID semantics, while a looser consistency across EGs, as there are unpredictable interventions during cross-EG committing.

2.3 Operations Across Entity Groups

Obvious, EGs need to communicate with each other to make a practical program. Up till now, we have two ways to transport data (i.e. messages) between EGs: **Two-Phase Commit(2PC)** and **Asynchronous Message Queue(AMQ)** (See Figure 2-2).

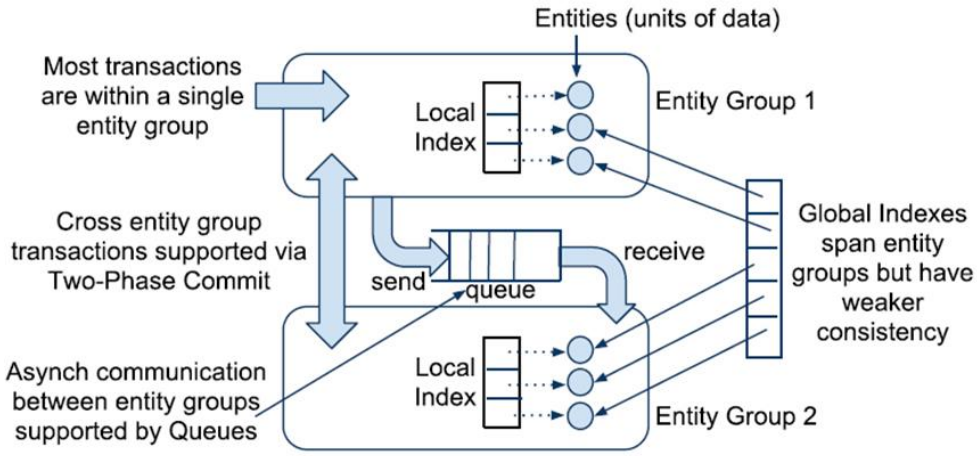

Figure 2-2 Communication between Entity Groups

But the use of 2PC and AMQ is limited within logically distributed EG, not geographical distributed replicas. The replication algorithm for geographical distributed replicas is much more complicated, we will discuss it in later sections. But we are going to detail the principles within 2PC and AMQ.

2.3.1 Two-Phase Commit

Within a single database, we prefer to use Two-Phase Commit (2PC) to reach data consensus among different replicas.

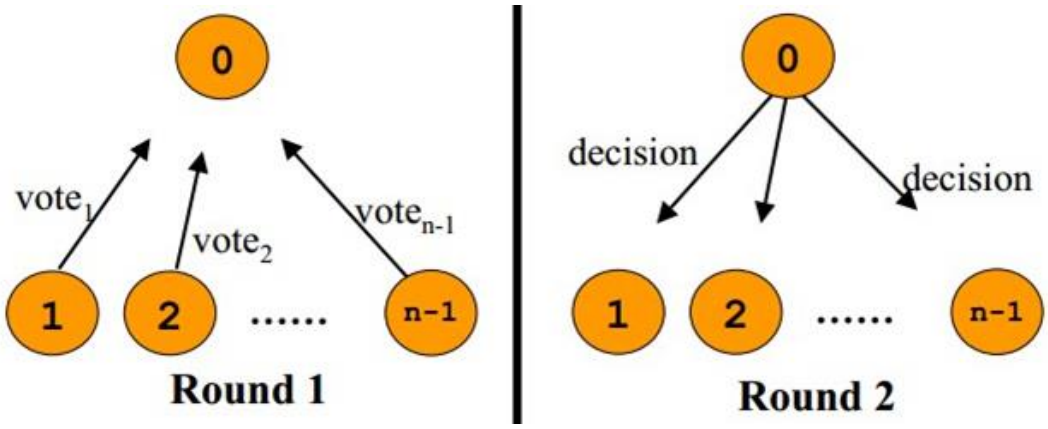

Figure 2-3 Two Phase Commit

Since 2PC require at least 2 rounds communication: one for **Voting Phase** and the other for **Commit phase** (See Figure 2-3), it is unacceptable in a geographical distributed environment with possible high latency.

2.3.2 Asynchronous Message Queue

More conveniently, we use **Asynchronous Message Queue** to achieve our goal (See Figure 2-2).

Communicating with AMQ requires less rounds. Sender only need to simply push their messages into the queue, and dispatcher will handle the rest.

2.4 Boundary of an Entity Group

To finish the final section for understanding EG, we still have to discuss another topic: how to define the boundary of an EG. If we want to make it clear, we have to understand what components can be put together in an EG first.

Take the email service as an example. Naturally, each email account is an EG: all of his emails and metadata. And emails from different accounts will be regarded as cross-EG transactions.

But when it comes to a map, the situation changes. Forming an EG by dividing the map into countries or continents will be a good choice. But since countries and continents have huge difference in size, the storage gap between EG will be wide too, which is bad for load balancing between datacenters. Therefore, we will break the map into several patches with similar size.

Hence, nearly all applications built on Megastore can find their proper ways to draw entity group boundaries, defining the boundary will not be a headache.

3 Megastore

To some extent, Megastore is a database system that had overcome traditional disadvantages of RDBMS, as well as provides a set of brand-new features which motivates the fast-development at the same time.

3.1 Assumptions and Philosophy for API Design

Before we actually start to design the new service, we have to understand the situation we are facing.

1. Batched transactions suffer lesser performance lose when using **store procedure** than using query command directly.
2. **Read dominates write**, as most transactions require database read operations.
3. It is convenient to access the value with provided key in **NoSQL storage platform** like Bigtable.

Hence, we are able to draw our new design base on these facts (See Figure 3-1).

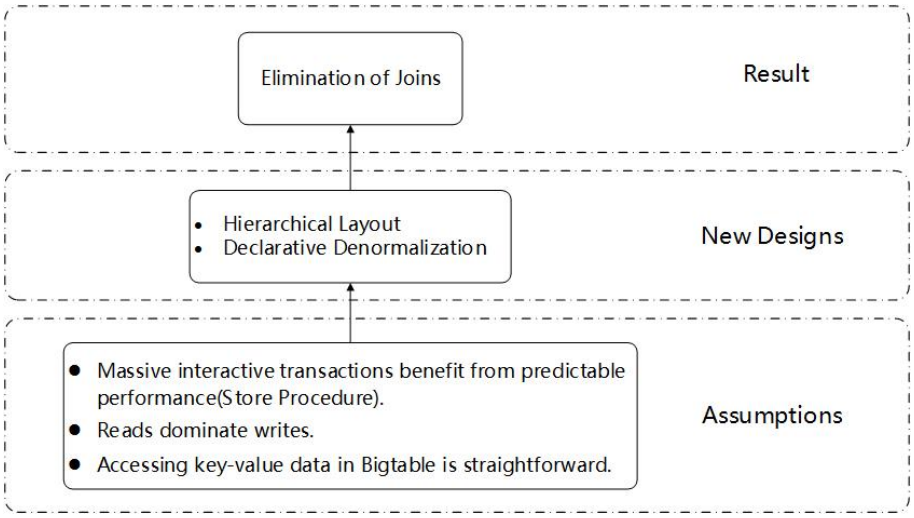

Figure 3-1 Assumptions, Philosophy and new Design

With this hierarchical layout, it is easy to construct the Megastore above Bigtable, and use the declarative demoralization to help making data storage mapped to Bigtable. Finally, the elimination of joins will be the last result, lowering the latency of service.

3.2 Data Model with Example

With a hierarchical philosophy in mind, we implemented a layered data model (See the right part of Figure 3-2).

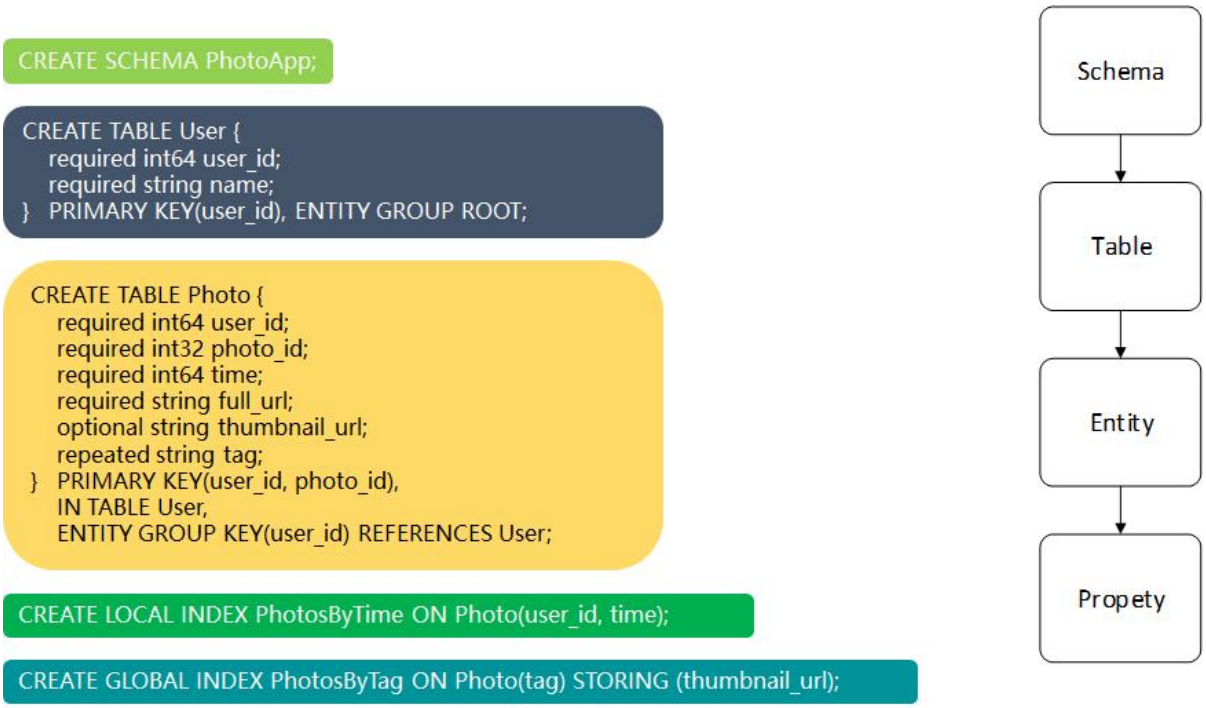

Figure 3-2 Data Model

Figure 3-2 is a data model sample which contains a 4-layer layout: **SCHEMA**, **TABLE**, **ENTITY**, **PROPERTY**. In this sample, we created a SCEMA named PhotoApp, a user TABLE under PhotoApp, some Photo table, and 2 indexes.

3.2.1 Elimination of Joins

All tables have their own Primary Keys(PKs). In table User, the PK is user_id. In table Photo, the PKs is photo_id and user_id, which refers to the user_id in table User.

Traditionally, it will result in a join operation when a query request demand for information in table Photo with a relational database. But as we want it become simple and clear, we implement it as Table 3-1 in Megastore, with some photo instances follow their root user. Therefore, when there are queries for information in table Photo, Megastore can directly get to target rows with the help of indexes. Hence, the existence of joins is eliminated.

Table 3-1 Data Sample in Megastore

Row Key	user.name	photo.time	photo.tag	photo.url	…
101	John				
101, 500		12:31:01	Dinner, Paris	…	…
101, 502		12:15:22	Betty, Paris	…	…
102	Mary				
103	Jane				
103, 19		08:32:11	Office	…	…

3.2.2 Indexes

In Megastore, we provide two types of indexes in general: **global index** and **local index**. Just as their names imply, global index is responsible for the cross-EG query, and the local index is responsible for the local query (See Figure 3-2).

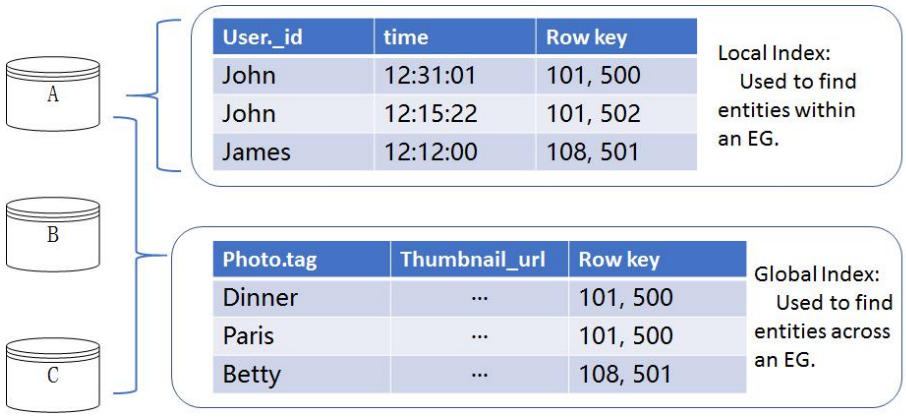

Figure 3-2 Indexes

Apart from the well-known global index and local index, Megastore also provide additional index features which can be attached on indexes.

1. **Storing Clause.** In global index PhotoByTag (See Figure 3-1), we can store a little additional information for fast retrieval. For example, we store thumbnail_url with the index itself. Therefore, when application server queries the thumbnail_urls, it is unnecessary for Megastore to search in

Photo table explicitly to get them, saving rounds of communication.

2. **Repeated Indexes.** Each repeated tag has its own index entry (See Figure 3-1). Hence, we can get to the same photo with any tags it has, boosting the performance of retrieval.
3. **Inline Indexes.** Extracting slices of info from child entities and storing it in the parent for fast access.

3.3 ACID Semantics

It is necessary for a database system to provide full support to ACID Semantics. Therefore, Megastore uses **Write Ahead Log(WAL)** and **Multi-Version Concurrency Control(MVCC)** to provides ACID semantics, as well as different types of reads to satisfy different requirements.

3.3.1 Write Ahead Log

Megastore will write it before apply the changes. It can be used for fail recovery or transaction rollback. We will detail the WAL in later section.

3.3.2 Multi-Version Concurrency Control

MVCC is the key to keep the ACID semantics in Megastore. In Megastore, different values can be stored the same Bigtable cell, with timestamps attached (See Table 3-2). Reader use timestamps to identify the latest value for target property.

Table 3-2 Sample for MVCC

Photo.tag
[(Dinner, Paris), 12:30:01], [(Father, Mother), 12:31:01]
[(Betty, Paris), 12:15:22], [(Betty), 12:16:22]

What`s more, reads and writes are isolated in this model. If a writer is appending the latest value to Bigtable, reader will fetch the one-version-older. But it is still the latest value until the writer finish.

3.3.3 Reads

In Megastore, we provide three types of reads to meet the different requirements from different application logics.

1. **Current Read.** It is always done within a single EG. Readers will fetch values before after all previous transactions are applied and informed.
2. **Snapshot Read.** Readers pick up the latest-known fully applied version, though there may be transactions waiting in AMQ.
3. **Inconsistent Read.** Readers fetch the value in Bigtable directly regardless of the log, no matter whether there are messages waiting for applying or other situations.

Their differences are listed in Table 3-3.

Table 3-3 Difference in all Reads

Type	Data Consensus	Latency
Current Read	High	High
Snapshot Read	Medium	Medium
Inconsistent Read	Low	Low

Hence, under MVCC and WAL as well as different reads, the completed lifecycle for a transaction in Megastore should looked like this:

1. **Current Read.** Use a current read to determine the next available log position.

2. **Application Logic.** Prepare the data to be written together, and designate it a latest log position. Also, batching writes to a front-end server can reduce the possibility of contention.
3. **Commit.** Client submits mutations and the replicas will use Paxos to vote a consensus value.
4. **Apply.** Write mutations that win the Paxos procedure into the Bigtable, with its own timestamp attached.
5. **Clean Up.** Clean all unnecessary values. For example, older version of an updated value.

4 Replication

In this section, we will focus on our implementation of Paxos in Megastore, which is the heart of our synchronous replication scheme. After that, we will turn to operational issues and tests on performance.

4.1 Original Paxos

It is the way to reach consensus among a distributed system on a given value by winning more than half votes. There are 2 phases in Paxos: **Prepare Phase** and **Accept Phase**.

Suppose 25 travelers need to reach a consensus about where to go with 5 additional leaders. In Prepare Phase, all travelers will send their recommendation on where to go to all leaders with timestamp attached (See Figure 4-1).

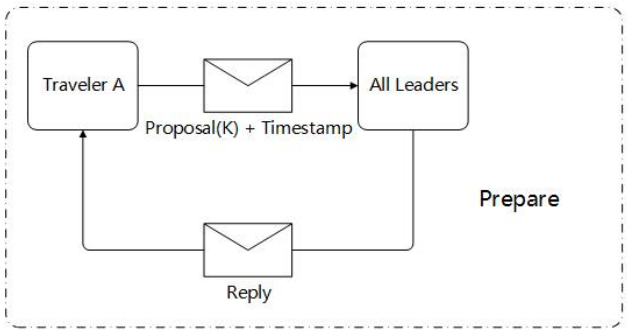

Figure 4-1 Prepare Phase

There are 2 possible results if a traveler getting into the Accept Phase (See Figure 4-2).

1. **If none of the leaders had even made a decision, this traveler will send message to all leaders with his proposal (See the left side in Figure 4-2).**
   1) If more half leaders agree and reach a consensus, this will be the decision.
   2) If it is the other situation, he has to retry from Prepare Phase.
2. **At least 1 leader had made decision (See the right side in Figure 4-2).**
   1) If more half leaders reach a consensus, this will be the decision obviously.
   2) If leaders had not reached a consensus, he will support the latest decision.

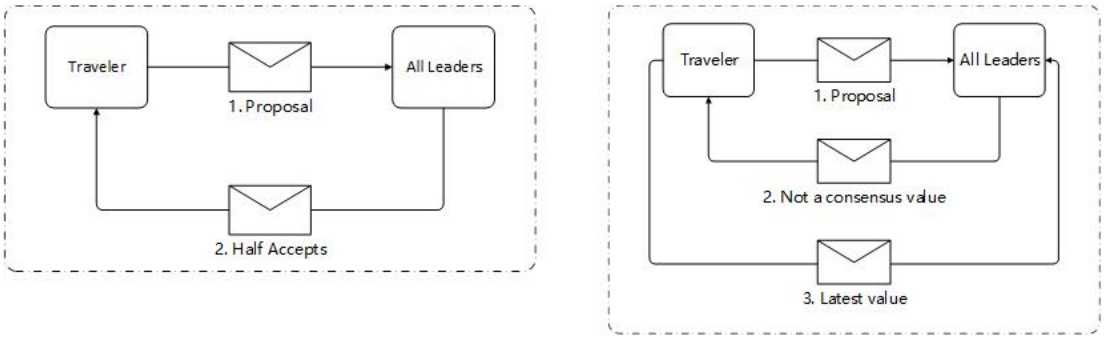

Figure 4-2 Accept Phase

Apparently, the original Paxos is ill-suited for possible high latency network, as it requires multiple rounds of communication.

4.2 Fast Read and Write in Megastore

To overcome the disadvantages brought by Paxos and implemented it in Megastore, we introduce two innovative measurements to do it.

4.2.1 Fast Read

As almost all reads are successful in most replicas, therefore, allowing **local reads** which have a lower latency and better utilization is reasonable.

Also, to make committing process simple and safe, we introduce a service named **Coordinator** (See Figure 4-3). A coordinator is a server which tracks a set of entity groups for which its replica has observed all Paxos writes. In short, a coordinator supervises all its EGs and knows all.

4.2.2 Fast Write

We also implement the **Fast Write** in Megastore with features below.

1. **Implied prepare message.** Each successful write implies a prepare message for next log position it needs to perform next write. So, one round for each subsequent writes are saved.
2. **Use the closest replica.** Select the replica with most submitting in this region to optimize the latency.

4.3 Replica Types

So far, we implement three types of replicas in Megastore: **Full Replica**, **Witness Replica**, and **Read-Only Replica**. And their differences will be listed as below.

Table 4-1 Differences between Replica Types

	Full Replica	Witness Replica	Read-Only Replica
Current Read	Yes	No	No
Log and Data Storage	All	Not-applied log, No data and indexes.	Full data snapshot
Vote	Yes	Yes	No
Usage	All	Tie breakers, Voting	Dissemination - CDN

4.4 Architecture

In Megastore, all key components are locating in different layers (See Figure 4-3).

1. **Application Layer:** Application Server, Megastore Library.
2. **Megastore Layer:** Replication Server, Coordinator.
3. **Physical Layer:** Bigtable Server with Logs and Data.

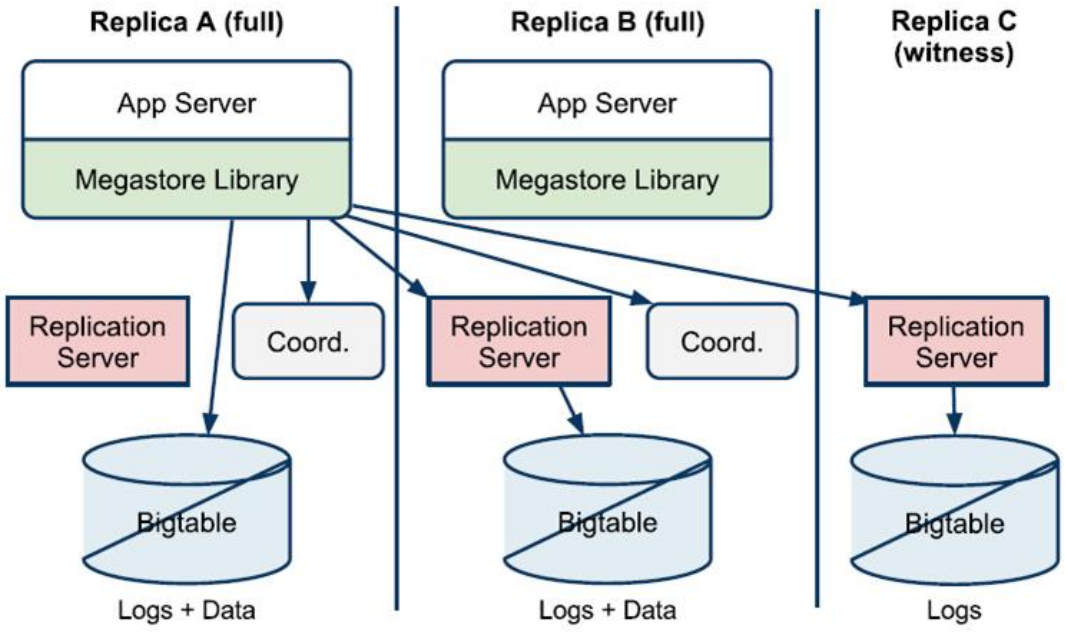

Figure 4-3 Architecture

4.5 Replicated Logs

The functioning of Megastore depends on the WAL we discussed in previous. Because there are numerous transactions submitted to Megastore at the same time, the status of WAL will be various (See Figure 4-4).

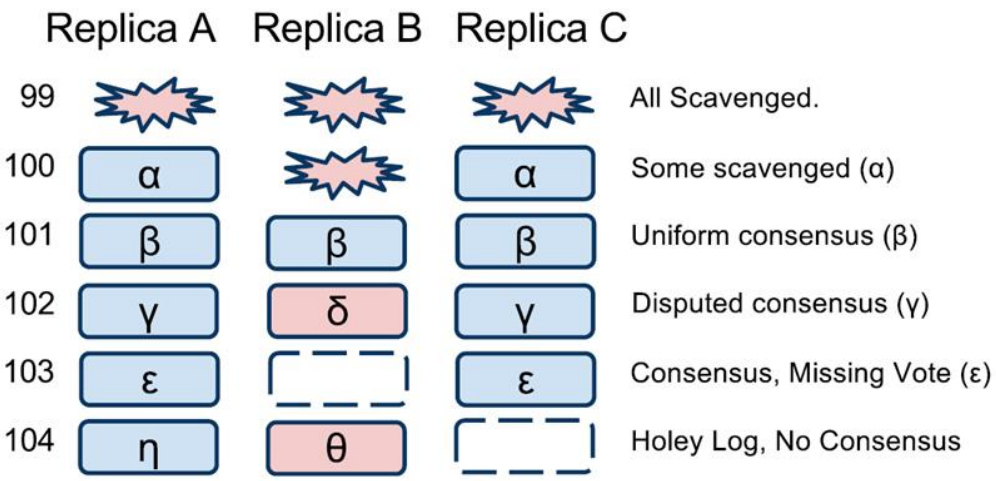

Figure 4-4 Write Ahead Log

Figure 4-4 shows 6 different status in different rows:

1. **Row 99**, all log position is outdated, therefore scavenged to save storage space.
2. **Row 100**, the scavenger is processing with one column cleared.
3. **Row 101**, all replicas reached consensus, therefore this log position should not be scavenged until there is a new write transaction.
4. **Row 102**, under the Paxos process, some replicas had not agreed on a specific value.
5. **Row 103**, there may be problem with Replica B with its missing voting.
6. **Row 104**, probably in the Prepare Phase with different votes from all replicas.

4.6 Read and Write with Paxos in Megastore

In order to take full advantage of the data consensus brought by Paxos, we made modifications on read and write algorithms to make it a highly available and low-latency service.

4.6.1 Reads

There are 5 steps for read algorithm in Megastore (See Figure 4-5):

1. **Query Local:** Determine if the entity group is up-to-date locally.
2. **Find Position:** Get the highest log position, and select the corresponding replica.
    1) If the step 1 indicates the local replica is up-to-date, get the log position and timestamp locally.
    2) Read from other replicas to determine the latest value and log position.
3. **Catchup**
    1) Get unknown value from other replica; run Paxos for any non-consensus values.
    2) Apply all consensus value, and push the state up-to-date.
4. **Validate:** Send its coordinator a message asserting itself up-to-date.
5. **Query Data**

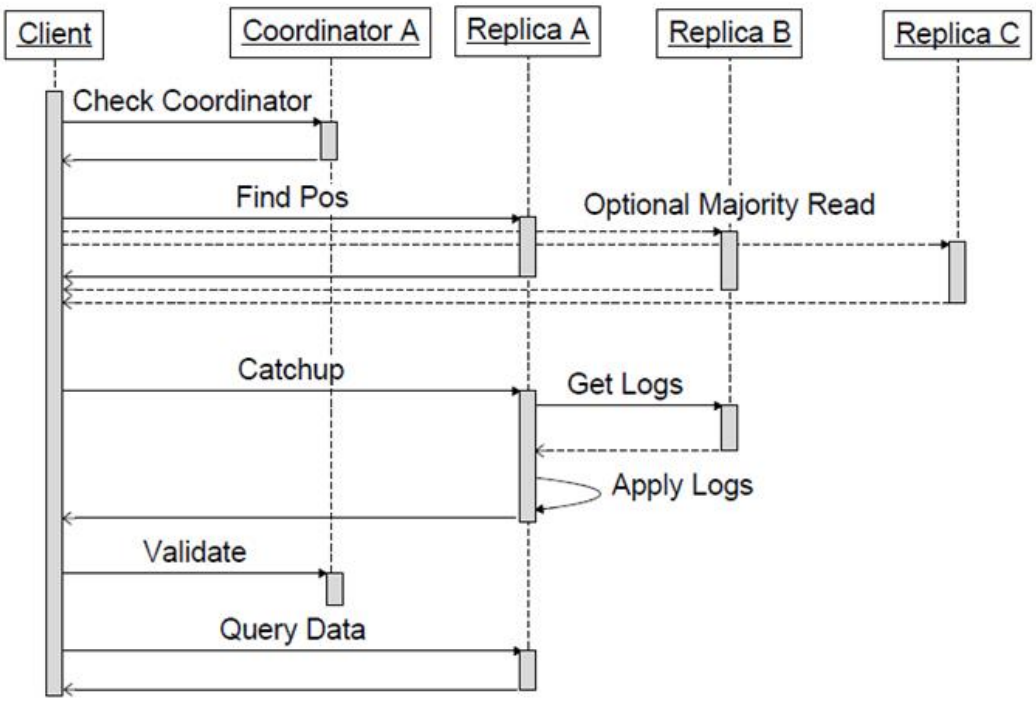

Figure 4-5 Timeline for Read Operation

4.6.2 Writes

There are 5 steps for write algorithm in Megastore (See Figure 4-6):

1. **Accept Leader:** Ask the leader to accept the value as proposal number zero.
2. **Prepare:** Run Paxos Prepare phase at all replicas [2].
3. **Accept:** Ask the rest of replicas to accept the proposal, as the Accept Phase in Paxos.
4. **Invalidate**: If a full replica doesn't accept this value, invalidate its coordinator.
5. **Apply Mutations.**

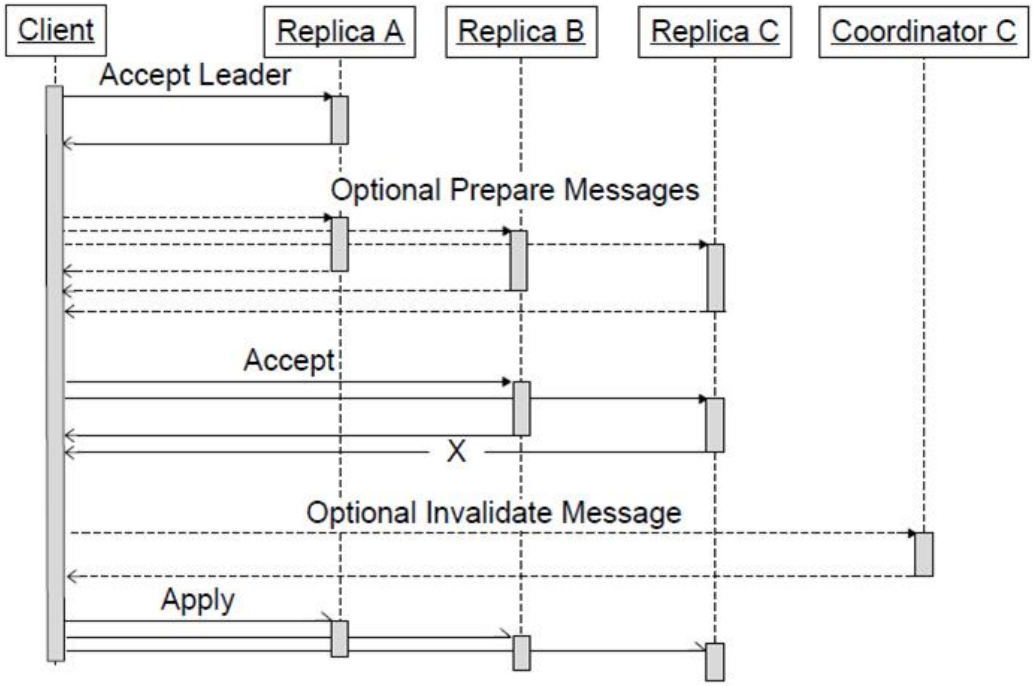

Figure 4-6 Timeline for Write Operation

4.7 Coordinator Availability

Coordinator is a simpler process than Bigtable server, but it still has the risk to crash or encounter other situations that cause it unavailable. Therefore, readers and writers have different strategies against the instability introduced by coordinator:

1. **Reader:** To process a request, a coordinator must hold a majority of its locks
2. **Writer:** Test the coordinator whether it still has locks before a writer submits its transactions.

4.8 Tests

In short, we made tests on **Availability** and **Latency**. In our test for availability, we observe an availability rate for at least 99.9% for Read/Query in more than 90%

of testee and more than 99.999% in more than 80% of the total testee all involved hosts in this test (See Figure 4-7), which proves the availability on Megastore.

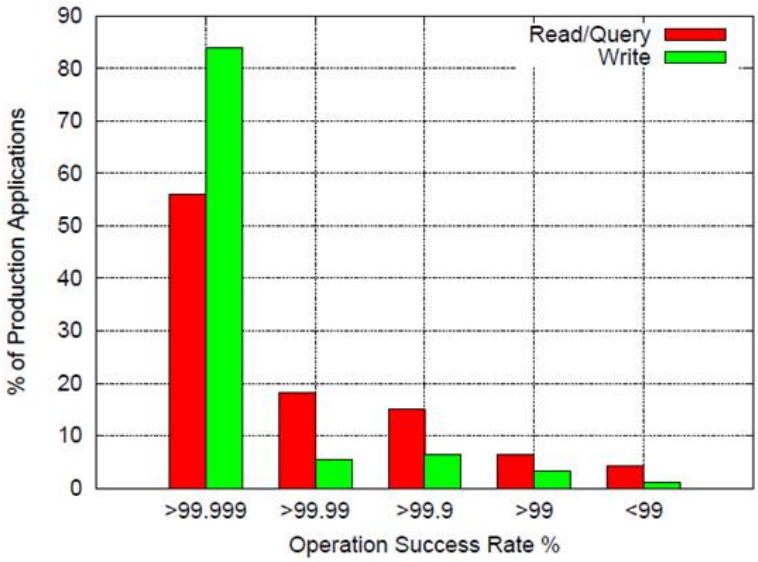

Figure 4-7 Availability

In our test for latency, we observe an average latency at 100ms for reads and 500ms for writes in more than half testee (See Figure 4-8), which proves the low-latency feature on Megastore.

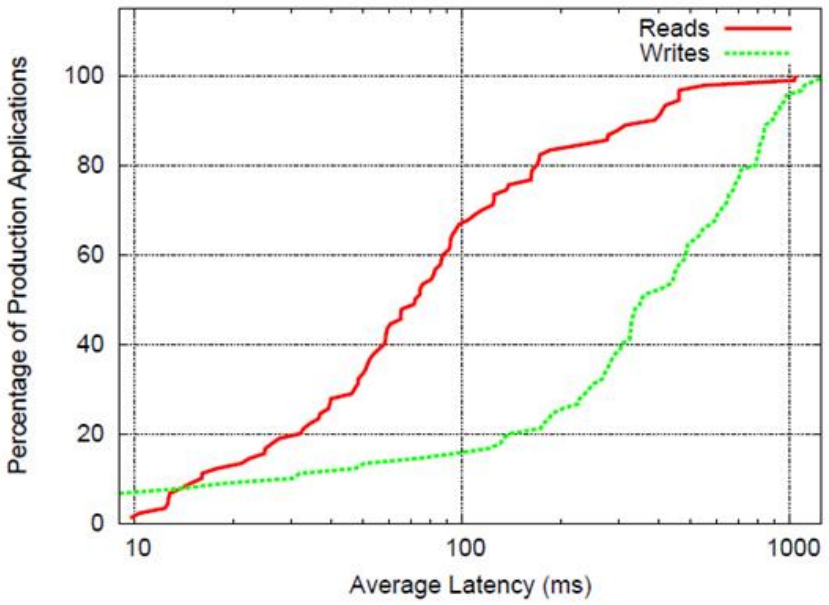

Figure 4-8 Latency

5 Conclusion

In this paper, we make introduction for the following basic concepts in Megastore: Entity Groups, Data Model and MVCC, Paxos in Megastore, Replication algorithm. Since Megastore is the origin of many distributed systems, we truly hope our we can fully understand the revolutionary concepts and theories that Megastore brings to us. One step further, keeping the motivation for learning and innovating is the fundamental power towards ultimate truth.

Reference

[1] 演员.用户数量破 8 亿 ；2016 年腾讯每天收入 3 亿[EB/OL].兴趣部落.(2016-8)[2016-06-18].

https://buluo.qq.com/p/detail.html?bid=261433&pid=7574448-1471501699

[2] Baker J, Bond C, Corbett J, et al. Megastore: Providing Scalable, Highly Available Storage for Interactive Services.[C]// CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings. DBLP, 2011:223-234.