



Building scalable geo-replicated storage back ends for web application

¹Rajkumar M. Chauhan (M.Tech. C.S.E.)

¹Foreman Instructor ITI Amirgadh

Ex Asst. Professor BCA College

Palanpur, Gujarat, INDIA

Abstract: The goal of this paper is to develop a scalable geo replicated storage back ends for web applications. Web application mostly requires a storage system replicate across different sites also scalable. Generally we use traditionally database having strict consistency guarantee and programming ease, but in concern in geo replicated no SQL are scalable and efficient. For scalable and geo replicated storage for web application we have two system Walter and Lynx. Walter supports transactions and replication of data across sites. Walter have new isolation property called PSI(Parallel snapshot isolation) that allows asynchronously data replication. For implementing PSI Walter uses two techniques one is preferred sites and counting sets. Lynx is database backend for scaling web application. It support optimizing query via denormalization, secondary indexes and join view. A Distributed Transaction Chain (DTC) used to sequence transaction on different node with guarantee that all transactions execute exactly once despite failure. So many application will be developed like micro blogging, social networking. Using these methods applications are capable for providing scalability across geo replicated site.

Keywords: Walter, Lynx, PSI, DTC, micro blogging, social networking

I. Introduction

There are two major concerns when building a popular web application. First, it must be able to scale up quickly to handle a rapidly growing user base. For example, the popular photo sharing application Instagram saw a nearly exponential growth to over 30 million users in less than two years. Achieving scalability and geo-replication is not an easy task. Web applications are commonly constructed using a multi-tiered design, where application servers use a storage tier to store and share data. Scaling the application tier is easily achieved by running application servers on any machines across multiple data centers, but it is much harder to scale the storage tier and have it support geo-replication. To be scalable, a storage system must divide data into a large number of partitions spread across many machines. As applications often need to access data belonging to multiple partitions, the storage system must coordinate access across different partitions, which comes at a performance cost. The cost of such coordination increases substantially when data is replicated across geographically distant data centers with tens or hundreds of milliseconds of communication delays.

Walter: A geo-replicated transactional key-value store

Existing geo-distributed key-value stores provide no transactions or only restricted transactions. Without transactions, an application must carefully coordinate access to data to avoid race conditions, partial writes, overwrites, and other hard problems that cause erratic behavior. Developers must address these same problems for many applications. With transactions, developers are relieved from concerns of atomicity, consistency, isolation, durability, and coordination. For example, in a social networking application, one may want to remove user A from B's friends list and vice versa. Without transactions, developers must write code carefully to prevent one removal from happening without the other. With transactions, developers simply bundle those updates in a transaction.

With PSI, hosts within a site observe transactions according to a consistent snapshot and a common ordering of transactions. Across sites, PSI enforces only causal ordering, not a global ordering of transactions, allowing the system to replicate transactions asynchronously across sites. With causal ordering, if Alice posts a message that is seen by Bob, and Bob posts a response, no user can see Bob's response without also seeing Alice's original post. Besides providing causal ordering, PSI precludes write-write conflicts (two transactions concurrently writing to the same object) so that developers need not write conflict resolution logic.

To prevent write-write conflicts and implement PSI, Walter relies on two techniques: preferred sites and counting sets. In web applications, writes to an object are often made by the user who owns the object, at the site where this user logs into. Therefore, we assign each object to a preferred site, where objects can be written more efficiently. For example, the preferred site for the wall posts of a user is the site closest to the user. Preferred

sites are less restrictive than primary sites. Walter uses multi-version concurrency control within each site, and it can quickly commit transactions that write objects at their preferred sites or that use csets. For other transactions, Walter resorts to two-phase commit to check for conflicts. We found that the latter type of transaction can be avoided in the applications we built. Using Walter as the storage system, we build WaltSocial, a Facebook-like social networking application.

II. Lynx: A scalable, eventually consistent database

In Lynx, each table is split into partitions which are spread across many machines. For better fault-tolerance, Lynx can also optionally replicate data across several geographically distributed data centers. As web applications demand low latency operation, Lynx allows for three common query optimization patterns: denormalization, distributed secondary indexes, and materialized join views. These optimizations essentially pre-compute results so that a query can be satisfied by contacting just one machine, minimizing the latency and overhead for common read operations. Lynx uses a new primitive, called the distributed transaction chain (DTC), to update secondary indexes and join tables. It also exposes the DTC primitive to the application programmer to update denormalized data in related tables.

Denormalization and index/view generation impose consistency constraints among data partitions managed by different machines. One way of enforcing these constraints is to use distributed ACID transactions. But this is a heavy hammer, and comes at a high price. Distributed ACID transactions require tight coordination among machines that manage different data partitions, lengthening the tail latency of operations. When data is replicated across data centers, such coordination requires communication between data centers, further increasing operation latency.

DTC is more powerful than a persistent message queue, a popular technique for modifying data on different machines—examples of this include Amazon's Simple Queue Service and eBay's message queue among others. Compared to DTC, the queue interface is low-level and transfers the burden of handling the difficult cases onto the programmers: programmers must not only design applications to explicitly enqueue and dequeue transactions, they must also ensure that no transaction is executed more than once after failure recovery and that the arbitrary interleaving of transactions of different logical operations does not lead to inconsistencies.

III. Design and algorithms

Basic architecture

There are multiple sites numbered 1, 2, . . . Each site contains a local Walter server and a set of clients. A client communicates with the server via remote procedure calls implemented by the API library. The server executes the actual operations to start and commit transactions, and to access objects.

Walter employs a separate configuration service to keep track of the currently active sites, and the preferred site and replica set for each object container. The configuration service tolerates failures by running as a Paxos-based state machine replicated across multiple sites. A Walter server confirms its role in the system by obtaining lease from the configuration service. The lease assigns a set of containers to a referred site, and it is held by the Walter server at that site. A Walter server caches the mapping from a container to its replica sites to avoid contacting the configuration service at each access. Incorrect cache entries do not affect correctness because a server rejects requests for which it does not hold the corresponding preferred site lease.

At Server_i: // i denotes the site number
CurrSeqNoi : integer with last assigned local sequence number
CommittedVTSi : vector indicating for each site how many transactions of that site have been committed at site i
History_i [oid] : a sequence of updates of the form $\langle \text{data}, \text{version} \rangle$ to oid , where $\text{version} = \langle j:n \rangle$ for some j ,
GotVTSi : vector indicating for each site how many transactions of that site have been received by site i

Figure 1: Variables at server on each site.

Figure 1 shows the variables at the server at site i . Variable *CurrSeqNoi* has the last sequence number assigned by the server, and *CommittedVTSi*[j] has the sequence number of the last transaction from each site j that was committed at site i .

At Server_i: // i denotes the site number
operation startTx(x) $x.\text{tid} \leftarrow \text{unique transaction id}$
 $x.\text{startVTS} \leftarrow \text{CommittedVTSi}$
 return OK
operation write(x, oid, data):
 add $\langle \text{oid}, \text{DATA}(\text{data}) \rangle$ to $x.\text{updates}$;
 return OK

```

operation setAdd(x, setid, id):
    add ⟨setid, ADD(id)⟩ to x.updates;
return OK

operation setDel(x, setid, id):
    add ⟨setid, DEL(id)⟩ to x.updates;
return OK

operation read(x, oid) // if oid is locally replicated then return state of oid reflecting x.updates and all versions
in Historyi [oid] visible to x.startVTS else return state of oid reflecting x.updates, the versions in
Historysite(oid) [oid] visible to x.startVTS, and the versions in Historyi [oid] visible to x.startVTS operation
setRead(x, setid): same as read(x, oid)

```

Figure 2: Executing transactions.

To execute transactions, the server at each site i maintains a history denoted $\text{History}_i[\text{oid}]$ with a sequence of writes/updates for each object oid , where each update is tagged with the version of the responsible transaction. This history variable is similar to variable Log in the PSI specification, except that it keeps a list per object, and it has versions not timestamps. When a transaction x starts, Walter obtains a new start vector timestamp startVTS containing the sequence number of the latest transactions from each site that were committed at the local site. To write an object, add to a cset, or remove from a cset, Walter stores this update in a temporary buffer $x.\text{updates}$. To read an object, Walter retrieves its state from the snapshot determined by startVTS and any updates in $x.\text{updates}$. Specifically, for a regular object, Walter returns the last update in $x.\text{updates}$ or, if none, the last update in the history visible to startVTS . For a cset object, Walter computes its state by applying the updates in the history visible to startVTS and the updates in $x.\text{updates}$.

Scalability

Walter relies on a single server per site to execute and commit transactions, which can become a scalability bottleneck. A simple way to scale the system is to divide a data center into several "local sites", each with its own server, and then partition the objects across the local sites in the data center. This is possible because Walter supports partial replication and allows transactions to operate on an object not replicated at the site—in which case, the transaction accesses the object at another site within the same data center. We should note that PSI allows sites to diverge; to avoid exposing this divergence to users, applications can be designed so that a user always log into the same local site in the data center. Another approach to scalability, which we do not explore paper, is to employ several servers per site and replace the fast commit protocol with distributed commit.

Method

```

void start()
int commit()
int abort()
int read(Oid o, char **buf)
int write(Oid o, char *buf, int len)
Oid newid(ContainerId cid, OType otype)
int setAdd(Oid cset, Id id)
int setDel(Oid cset, Id id)
int setRead(Oid cset, IdSetIterator **iter)
int setReadId(Oid cset, Id id, int *answer)

```

Description

```

start transaction
try to commit
abort
read object
write object
get new oid
add id to cset
delete id from cset
read cset
read id in cset

```

C++ Example:

```

Tx x;
x.start();
len = x.read(o1, &buf);
err = x.write(o2, buf, len);
res = x.commit();

```

PHP Example:

```

$x = waStartTx();
$buf = waRead($x, $o1);
$serr = waWrite($x, $o2, $buf);
$res = waCommit($x);

```

Figure 3: Basic C++ API for Walter and C++ and PHP examples.

IV. Implementation

The Walter implementation has a client-side library and a server, written in C++, with a total of 30K lines of code. There is also a PHP interface for web development with 600 lines of code. The implementation differs from the design as follows. First, each Walter server uses direct-attached storage devices, instead of a cluster storage system. Second, we have not implemented the scheme to reintegrate a failed site currently, the

administrator must invoke a script manually to do that. Third, the client interface differs cosmetically due to the specifics of C++ and PHP. In C++, there is a Transaction class and operations are methods of this class. Functions read, setRead, and setReadId return the data via a parameter (the C++ return value is a success indication). setRead provides an iterator for the ids in a cset. setReadId indicates the count of an identifier in a cset. commit can optionally inform the client via supplied callbacks—not shown—when the transaction is disaster-safe durable and globally visible (i.e., committed at all sites). There is a function newid to return a fresh oid, explained below.

There are no specialized functions to create or destroy objects. Conceptually, all objects always exist and are initialized to nil, without any space allocated to them. If a client reads a never-written object, it obtains nil. Function newid returns a unique oid of a never-written object of a chosen type (regular or cset) in a chosen container. Destroying a regular object corresponds to writing nil to it, while destroying a cset object corresponds to updating its elements so that they have zero count. There are some additional functions (not shown), including (a) management functions for initialization, shutdown, creating containers, and destroying containers; and (b) functions that combine multiple operations in a single RPC to the server, to gain performance; these include functions for reading or writing many objects, and for reading all objects whose ids are in a cset. The functions to create and destroy containers run outside a transaction; we expect them to be used relatively rarely. Identifiers for containers and objects are currently restricted to a fixed length, but it would be easy to make them variable-length.

The server stores object histories in a persistent log and maintains an in-memory cache of recently-used objects. The persistent log is periodically garbage collected to remove old entries. The entries in the in-memory cache are evicted on an LRU basis. Since it is expensive to reconstruct csets from the log, the eviction policy prefers to evict regular objects rather than csets. There is an in-memory index that keeps, for each object, a list of updates to the object, ordered from most to least recent, where each update includes a pointer to the data in the persistent log and a flag of whether the data is in the cache. To speed up system startup and recovery, Walter periodically checkpoints the index to persistent storage; the checkpoint also describes transactions that are being replicated. Checkpointing is done in the background, so it does not block transaction processing. When the server starts, it reconstructs the index from the checkpointed state and the data in the log after the checkpoint.

To improve disk efficiency, Walter employs group commit to flush many commit records to disk at the same time. To reduce the number of threads, the implementation makes extensive use of asynchronous calls and callbacks when it invokes blocking and slow operations. To enhance network efficiency, Walter propagates transactions in periodic batches, where each batch remotely copies all transactions that committed since the last batch.

The protocol for slow commit may starve because of repeated conflicting instances of fast commit. A simple solution to this problem is to mark objects that caused the abort of slow commit and briefly delay access to them in subsequent fast commits: this delay would allow the next attempt of slow commit to succeed. We have not implemented this mechanism since none of our applications use slow commit.

V. Conclusion

This paper presented two storage systems that were specifically designed and built to address the requirements of large-scale web applications. Both systems, Walter and Lynx are georeplicated and employ a relaxed, yet relatively strong consistency model. The chosen consistency guarantees of both systems allow the implementations to achieve high performance at large scale, and make the development of web application on top the systems simple and rapid.

Walter is a transactional geo-replicated key-value store. A key feature behind Walter is Parallel Snapshot Isolation (PSI), a precisely-stated isolation property that permits asynchronous replication across sites without the need for conflict resolution. Walter relies on techniques to avoid conflicts across sites, thereby allowing transactions to commit locally in a site. PSI thus permits an efficient implementation, while also providing strong guarantees to applications. We have demonstrated the usefulness of Walter by building a Facebook-like social networking application and porting a third-party Twitter clone. Both applications were simple to implement and achieved reasonable performance.

Lynx targets intra-site scalability, in addition to geo-replication. It is a distributed database for building scalable web applications. Lynx supports distributed secondary indexes and materialized joins which help programmers optimize queries for low latency operation. Lynx maintains its derived tables using DTC, which executes a series of transactions at different nodes while guaranteeing fault-tolerance and correct interleaving. Lynx also exposes the DTC primitive to programmers for maintaining other types of denormalized data in the application. Lynx provides the consistency guarantee that denormalized data and derived tables are eventually consistent with each other. We have demonstrated the usefulness of Lynx by building an auction service, a microblogging and a social networking website.

References

- [1] Private data on database access patterns from a popular dating website.
- [2] Amazon simple queue service. <http://aws.amazon.com/sqs>.

- [3] A Twitter clone for the Redis key-value database. <http://retwis.antirez.com>.
- [4] Twitter turns six, 2012. <http://blog.twitter.com/2012/03/twitter-turns-six.html>.
- [5] Daniel Abadi. Replication and the latency-consistency tradeoff. DBMusings. <http://dbmusings.blogspot.com/>.
- [6] Amazon. DynamoDB: a fully managed NoSQL database service. <http://aws.amazon.com/dynamodb/>.
- [7] Wikipedia.org
- [8] Google.com

Acknowledgments

I thankful to all person who help me in this paper, special thanks to Mr. Jayesh N. Modi (Assistant Professor, MCA Dept, H.N.G. University Patan), who help me for research paper writing.