Optimistic Concurrency Control in Distributed Database Systems with low to medium contention

CompSci 516: Fall 2017
Final Report

Sahiti Bommareddy
sahiti.bommareddy@duke.edu

Naman Jain
naman.jain@duke.edu

1. Introduction

Concurrency control mechanisms are used to coordinate a system that supports transactions. Broadly speaking, there are two types of concurrency control mechanisms: pessimistic concurrency control and optimistic concurrency control techniques. Pessimistic concurrency control allows access to an object after a permission has been granted to access the object in the desired mode (shared or exclusively). Optimistic concurrency control assumes that there will be few or no conflicting actions and allows immediate access to the object. When a transaction requests a commit, it is validated using a validation algorithm. If a transaction can be serialized with other transactions in the system, the transaction is validated and committed. If it is not serializable, the transaction is aborted.

Many evolving NoSQL database services like DynamoDB, Voldemort, Riak, Cassandra and many others, stand as proof of work that optimistic concurrency control is the better option to choose over pessimistic concurrency control in low contention environment. It was very intriguing to see how these systems implemented the optimistic concurrency control bolstered with sharding , replication etc making it a reliable, resilient, scalable and highly available system.

We attempt to implement a distributed database system with optimistic concurrency control . This system ensures ACID transactions using timestamps based on loosely synchronized clocks.The system attempts to commit as many transactions as possible with conflict mitigation mechanisms. The system chooses to implement Consistency and Availability of the CAP theorem by assuming there are no network partitions.We aim to make it efficient.

Database locks are cached and manipulated at client machines with an embedded library provided by the system. This library manages communication with

the databases, the timestamping of commit requests and provides a simple CRUD API to the application.

There is full resource isolation among databases which simplifies consensus. It also means that there is no fault tolerance in the system. Global consistency, validation and transactional support are provided by the database system.

2. Related Work

A lot of work has gone into vector and logical clocks. The one that stands out is published by Leslie Lamport. Timestamp basics have been addressed to alongside the use of clocks to implement it[2]. Timestamp is a very effective way to serialize transactions and so is used in our project while using optimistic concurrency control.

Version numbers have been used in the past to implement optimistic concurrency control[3]. Although, using multi-version control was not necessary for this project.

Dynamo is interesting as it is a highly available key-value storage system. To achieve this level of availability, Dynamo sacrifices consistency under certain failure scenarios. This is because evading a network partition here is unpredictable and cannot be evaded. So, consistency is sacrificed, but only for a short time i.e., eventually consistent until the other replica databases catch up. It makes extensive use of object versioning and application-assisted conflict resolution[4].

Cassandra is a distributed storage system which provides highly available service with no single point of failure[5]. It uses optimistic concurrency control and resembles Dynamo in a lot of ways.

Slicer is Google's general purpose sharding service. It monitors signals such as load hotspots and server health to dynamically shard work over a set of servers. Its goals are to maintain high availability and reduce load imbalance while minimizing churn from moved work.[6]

A portion of research may involve combining the methods of some papers with our techniques in order to form a system which is better than each of these in some way.

3. Architecture

The system has a application side library which provides interface to interact with distributor and database servers. Distributor has metadata i.e., which server is primary for an object and mapping of entire object universe. The objects are distributed on multiple databases.

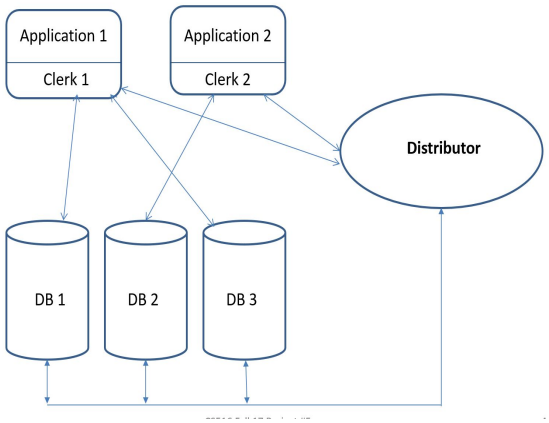

Fig.1 System Architecture

4. Algorithm for Transaction Validation done in database server

Following is the algorithm for validation checks before committing a transaction:

Threshold Check
If T.ts < Threshold then
Abort T

Checks Against Earlier transactions
For each uncommitted transaction S in history line
such that S.ts < T.ts
If (S.MOS ⋂ T.ROS is NOT NULL) then
Abort T

Current- Version Check
For each object in T. ROS
If each object's version in T.Ros is not same as version in database server or t.ts<rstamp
then
Abort T

Checks Against Later Transactions
For each transaction S in history line
such that T.ts < S.ts
If (T. ROS ⋂ S.MOS is NOT NULL) then
Abort T

5. Implementation

This system provides users with a universe of objects spread on multiple databases implemented as KV stores in our project. The mapping of these objects is stored as map in distributor, which also provides timestamps for transactions. Application has clerk to cache objects and to act as coordinator during two phase commit.

5.1 Application Side

The system creates a "Clerk" process for an application whenever the application wants to access objects. To start accessing its objects, the clerk first request distributor to provide it with the server which is primary for the required objects. Upon receiving the server map the clerk requests the necessary objects with oid from the object's primary server. The object data comes packaged with version.This data is cached on client.

When an application wants to execute , its changes are reflected in the clerk's cache, only upon successful commit will they be pushed on to the database.

The clerk is responsible for getting timestamp from distributor, packaging the transaction with its globally unique transaction id, Read set(ROS) which has values read and their version, Write set (MOS) which has new values , ts(timestamp) and sending it in prepare message to involved database servers. Thus clerk takes on the role of coordinator for two phase commit.It also sends commit message if all servers reply with prepare ok else an abort message. It receives acknowledgement from servers.

5.2 Distributor

Each object has a globally unique id called oid. Every object has a database

server as its primary responsible for the object. The oids are distributed among the databases using hashing and the server map is stored on the distributor. The distributor upon receiving a who serves message from client(with a list of oids) reply to the clerk with a map of primary servers for the oids requested.

The distributor is also responsible for generating timestamps which are needed by clerks just before the start of phase 1 of two phase commit.

5.3 Database Side

Each database manages some objects of the object universe. A database has a store to map oid, its value, version and rstamp(read timestamp). When it gets object request from a client it replies with oid, its value and version number.

Every database maintains a history of transactions sorted on their timestamp.Each transaction in history has read set , write set, a flag to indicate if it is in prepared or committed stage.

Database has cache table to indicate a mapping of objects cached on clients.

6. Transaction Execution Process

When an application starts a transaction it requests its clerk for some objects. Clerk finds the primary for the objects from distributor, fetches objects from database and caches them.

The application operates on these copies and when it is ready to commit, the clerk acquires a timestamp from distributor, packages the transaction into read set, write set and timestamp. A prepare message including this information is sent to all databases involved.

The databases validates, if it passes it is entered into database history with status P(indicating prepared) and database sends to clerk prepare ok or abort message in case of failure.

The clerk on receiving prepare ok from all participant database servers, begins phase two of 2PC by sending commit message with transaction id to participant servers. The database servers upon receiving commit message updates transaction status to C(indicating committed), updates the new value of object in their store, increment version number and set rstamp to transaction's timestamp. Once this is done we can remove committed transaction from history line. We will also truncate history before a threshold. These truncations ensure that our history is not too long and in turn reduces validation cost. Then it acknowledges the commit to client.

7. Observations

Validation process of Optimistic Concurrency Control, is costly and increases in proportion to the length of history line. However, we handled it by truncation of history and also removing committed transactions from it by using version and rstamp updations.

Also, to reduce the number of transaction restarts further, we plan to use cache invalidation mechanism.

Another observation is that the system does not work well for a high contention load which can be tackled by switching to Pessimistic Concurrency Control at the time when the system experiences high contention load.

8. Conclusion and Future work

An improvement of this system would be accommodating Pessimistic Concurrency Control for high contention load. This would be possible only when the distributor is able to identify whether the load is high contention or low contention and assign PCC or OCC accordingly.

Another improvement is Cache Invalidation. Whenever a transaction commits, the updated values are sent to the clients so that they can update their caches and the new transactions won't have to be aborted because of stale input in later validation phase.

Also, consistent hashing can be used with replication so that we have persistent objects.

9. Contributions of Project Members

We split papers between us and designed the architecture. With whiteboard in place we coded by splitting the segments.

Waqar Aqeel, another member, was also involved in the project. He added additional function in the thespian python actor model library used in the project coding, this contribution was acknowledged and added to thespian open source by its developer GoDaddy.

GitHub Link for project:
https://github.com/nj19141/CS516_Fall17_Project

10. References

[1] R. Ramakrishnan and J. Gehrke. Database Management Systems. McGraw-Hill, Inc., New York, NY, USA, 3 edition, 2003.

[2] L. Lamport,Time,clocks and the ordering of events in a distributed system, Massachusetts Computer Associates, Inc., 1978.

[3] B. Liskov, M. Day and L. Shrira, Distributed Object Management in Thor, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, USA, 1993.

[4] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall and W. Vogels, Dynamo: Amazon's Highly Available Key-value Store, Amazon.com, 2007.

[5] A. Lakshman and P. Malik, Cassandra: a decentralized structured storage system, 2010.

[6] A. Adya, D. Myers, J. Howell, J. Elson, C. Meek, V. Khemani, S. Fulger, P. Gu, L. Bhuvanagiri, J. Hunter, R. Peon, L. Kai, A. Shraer, A. Merchant, and K. Lev-Ari. Slicer:

Auto-sharding for Datacenter Applications. In Proceedings of the USENIX Conference on Operating Systems Design and Implementation, OSDI'16, pages 739–753, Berkeley, CA, USA, 2016. USENIX Association.