Jepsen test for RethinkDB

Introduction

RethinkDB is a scalable document-oriented database which came out of a need to make building real-time applications easier. Apart from being one of the few NoSQL databases that allow joins, RethinkDB is a distributed database that is easy to scale. With multiple shards and replicas, the database aims to provide high availability and robust fault-tolerance.

Goal

The goal of our project is to evaluate consistency semantics of RethinkDB - an upcoming real-time open-source database. As a distributed database, RethinkDB chooses to maintain data safety over availability. Our goal in this project is to understand how strictly does RethinkDB adhere to this standard and if it indeed behaves as a strong CP system.

Consistency models

In any concurrent system that has multiple actors participating that share memory, one of the key concerns is how do writes and reads behave. More explicitly, in the absence of a global real-time clock, how is the ordering of these operations affect the outcome of the system. Hence consistency is the idea of assigning meaning to concurrent reads and writes on shared, possibly replicated, state.

For distributed systems we define the term consistency model. We say that given some rules which relate the operations and state, the history of operations in the system and their result should always follow those rules. We call these rules a consistency model. These rules give us a basic direction of what the value returned in case of every write should be. These rules can be anything. These could be very strict based on our intuition above, or these could be very weak. Like a no rule statement, where we say that every operation is permitted is also a consistency model. The principle is that once a system defines these rules, they should be adhered to, no matter what.

The strongest consistency model defined is the linearizability model. We call this model strongest because it maps most closely to our intuitive model. Effectively, linearizability enforces that once a write completes, all later reads should return the value of that write or the value of a later write. Once a read returns a particular value, all later reads should return that value or the value of a later write.

For visualization, we can assume of a shared piece of memory being accessed by several processes. We have to assume that all operations are atomic and instantaneous, that is no two operations are happening simultaneously on our shared system. If we think of such a system, we can order all our operations sequentially over a linear scale happening one after the other. In such a case, each operation will access the latest data and is consistent with its previous writes and reads. Now, in the context of distributed systems, although our assumptions may be wrong, we expect the same characteristics from a system labelled as linearizable consistent.

We can use the atomic constraint of linearizability to mutate state safely. We can define an operation like compare-and-set, in which we set the value of a register to a new value if, and only if, the register currently has some other value. Linearizability guarantees us the safe interleaving of changes.

Since, linearizability's time bounds guarantee that those changes will be visible to other participants after the operation completes this model prohibits stale reads. Each read will see some current state between invocation and completion; but not a state prior to the read. It also prohibits non-monotonic reads–in which one reads a new value, then an old one.

Linearizability is important for distributed systems and databases because of two reasons. First, it is synonymous with the term "atomic consistency". Second, it is the "C," or "consistency," in Gilbert and Lynch's CAP theorem, considered the fundamental of building any distributed system.

There are other more weaker consistency models defined such as sequential consistency, causal consistency, eventual consistency and so on. Below is a chart showing several consistency models.

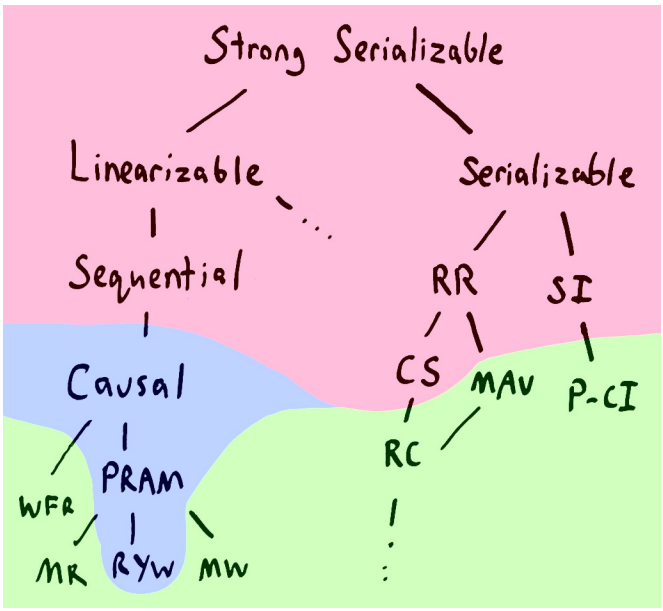

Figure 1: Tree of consistency models

RethinkDB and its consistency guarantees.

In terms of CAP theorem and quoting its documentation, RethinkDB states that it choses to maintain data consistency over availability in network partitions i.e it is **CP system**. Every shard in RethinkDB is assigned to a single authoritative primary replica. All reads and writes to any key in a given shard always get routed to its respective primary, where they're ordered and evaluated. This architecture in turn helps RethinkDB achieve its consistency goal.

If the primary replica for a table fails, as long as more than half of the table's voting replicas and more than half of the voting replicas for each shard remain available, one of those voting replicas will be arbitrarily selected as the new primary. There will be a brief period of unavailability, but no data will be lost. If the primary replica specified in a table's configuration comes back online after a failure, it will return to being the primary.

If there is a network partition, the behavior of the system from any given client's perspective depends on *which side of the netsplit* that client is on. If the client is on the same side of the netsplit as the

majority of voting replicas for the shard the client is trying to reach, it will continue operating without any problems. If the client is on the side of the netsplit with half or fewer of the voting replicas for the shard the client is trying to reach, the client's up-to-date queries and write queries will encounter a failure of availability.

Apart from the default state performance, RethinkDB also provides various configurations to tune its availability and consistency performance according to particular requirements. These configurations are based on the following three parameters.

- **Write acknowledgements** are set per table with the `write_acks` setting, either using the config command or by writing to the `table_config` system table. The default is majority, meaning writes will be acknowledged when a majority of (voting) replicas have confirmed their writes. The other possible option is single, meaning writes will be acknowledged when a single replica acknowledges it.
- **Durability** is set per table with the durability setting, again using either reconfigure or writing to the `table_config` system table. In hard durability mode, writes are committed to disk before acknowledgements are sent; in soft mode, writes are acknowledged immediately after being stored in memory. The soft mode is faster but slightly less resilient to failure. The default is hard.
- **Read mode** is set per query via an optional argument, `read_mode` (or readMode), to table. It has three possible values: *single* returns values that are in memory (but not necessarily written to disk) on the primary replica. This is the default. *majority* will only return values that are safely committed on disk on a majority of replicas. This requires sending a message to every replica on each read, so it is the slowest but most consistent. Lastly, *outdated* will return values that are in memory on an arbitrarily-selected replica. This is the fastest but least consistent.

According to the above settings based on the three parameters, various levels of availability and consistency guarantees can be achieved. However, RethinkDB claims that, we can achieve linearizability of individual atomic operations on individual documents with the following settings:

- `write_acks`: majority
- `durability`: hard
- `read_mode`: majority

Since, the write and reads are acknowledged from a majority of nodes and hard durability ensures writes in both memory and disk before a confirmation, the RethinkDB promises seem valid. The goal of our project is to validate these guarantees not in favorable conditions or under high throughput but under stressful conditions of system and network failures.

Project Description and Methodology

As stated, the goal of the project is to validate the linearizable consistency guarantees made by RethinkDB. We plan to use Jepsen for the same. Jepsen is a library written in clojure and provides the tools to simulate different testing conditions we talked about. This is written and maintained by Kyle Kingsbury and has been used this to test various distributed systems such as etcd, consul, cassandra, etc. MongoDB is another database sharing similar architecture and consistency guarantees as that of RethinkDB which has been tested with Jepsen. We aim to test RethinkDB on the same lines as MongoDB and derive help in terms of our test plan for RethinkDB.

We aim to generate a mix of read, write and CaS operations, and apply those to a five node RethinkDB cluster. Over the course of a few minutes we'll have a client perform those random read, write, and

CaS ops against the cluster, while a special process creates and resolves network partitions to induce cluster transitions. The different scenarios we would ideally like to analyze would be:

- What happens when the primary replica of the data goes down?
- What happens when there is a network partition and the primary in on the side of majority of nodes?
- What happens when there is a network partition and the primary in on the side of minority of nodes?
- Is a split brain phenomenon achievable in a RethinkDB cluster and what would be its implications?
- Can there be loss of acknowledged writes in any of the above situations?
- Conversely, can it result in any unacknowledged writes making their way to the database?

However, given the time constraints we would start answering the questions in order above and make our way deeper into the problem. At the start, our primary goal would be to evaluate the linearizability guarantees as claimed by the RethinkDB docs.

Background and Motivation

As RethinkDB makes strong claims with respect to its consistency performance, which desire to be analyzed. It has been requested by various audience in different channels and is a pending issue on github as well.

References

- Jepsen's blog posts on MongoDB and consistency models
- Highly Available Transactions: Virtues and Limitations, Peter Bailis, Aaron Davidson, Alan Fekete , Ali Ghodsi, Joseph M. Hellerstein, Ion Stoica
- Class lectures by Prof Roxana Geambasu
- RethinkDB documentation on consistency and architecture

Team Members

- Ayush Jain (aj2672)
- Prakhar Srivastav (ps2894)