Real World Cassandra

Jon Haddad

Table of Contents

This book will focus on the latest stable version of Apache Cassandra. I'm targeting the most current release, which at the time of this writing will be 4.0. Once the next version is released, this book will be updated.

This is an opinionated book. I will not advise using features that perform poorly in production or have so many caveats that failure is almost unavoidable. This is not the result of marketing, promises, or speculation.

This book is the culmination of everything I've learned about running Apache Cassandra in production. I have been lucky enough to have run it for several years at a startup, worked at DataStax as a technical evangelist, and while working as Principal Consultant at The Last Pickle I've helped fix and tune a wider variety of Cassandra clusters than almost anyone else on the planet. This is not a replacement for the project documentation, I don't intend to cover every possible feature and configuration options of Cassandra. My goal to help you run a production cluster.

I'd like to thank a handful of people to have made this book possible. I've made a lot of friends as the direct result of a technology choice. A special thanks to Blake Eggleston, Patrick McFadin, Nate McCall, Aaron Morton, Caleb Rackliffe, Luke Tillman, Jonathan Ellis, and everyone at The Last Pickle. Thank you to many others who I've had the privilege of working with and learning from over the years.

The Scale Problem

Chapter 1. Scaling, Old School

When I got interested in Cassandra, I was working at some startups that had tried to build scalable systems with tight SLAs and had run into some trouble. This was a point in time when using a relational database and vertical scaling was the tried and true solution.

Once we hit the limits of vertical scaling, the next step was to chip away at the advantages of the RDBMS.

1.1. Caching

The first (and easiest) problem that needed to be solved was dealing with read heavy workload. Fortunately, solving reads is usually solved by making more copies of the data and using more servers, in the form of caching.

Caching servers are easily scaled by partitioning the data among all the servers. Splitting data among caching servers is easy. This works great for read heavy workloads and scales linearly for a long time.

You've probably heard of one of the most well known caching projects of all time, Memcached. Memcached made it possible to massively scale read heavy workloads by exposing a simple api. The Memcached api only exposes a simple key/value api for setting and retrieving data. The following (crappy) code shows the basic logic:

```
def get_something(id):
    tmp = memcached.get(key)
    if(tmp):
        return tmp
    result = db.query("SELECT * from stuff WHERE id = " + id)
    memcached.put(key, result)
    return result
```

This is fine for small objects / rows, but doesn't work so well with large datasets that are joined and aggregated. How do we know what cache items to expire when a single row changes? We still have a problem with more complex workloads.

As the system grows and our SLAs remain the same, we have to start using optimizations to avoid performing the costly operations. Instead of using expensive joins, we denormalize and copy data around to avoid the costly join. Instead of aggregating on the fly we precalculate statistics. We can skip grouping hundreds of thousands (or millions) and return a handful of rows, which can be orders of magnitude faster than trying to do the calculation on the fly.

1.2. Scaling Writes

When it comes to scaling reads, this is a proven path. The difficulty arises when we have more work than the server is capable of handling. Scaling vertically works to a point, but gets very expensive and migrations are risky. Since we've already decoupled the tables from one another by

eliminating joins, the next logical step is to start separating the tables. First we move individual tables off, then usually the tables get sharded across many servers. Lots of tooling gets written, wheels reinvented.

We start thinking about partitioning our data at this point. Without joins, we can move tables to different machines without much problem. We're able to keep scaling our application at the cost of losing the ability to do arbitrary queries. It's a necessary tradeoff if we're to keep growing.

1.3. High Availability

The final challenge of scaling is availability. All of the changes I've described above are designed to help scale the database by spreading the workload from one server to many. By eliminating costly queries over large data sets we are able to keep our query times constant, and stick to our SLA. To maximize uptime, we need to replicate our data to multiple servers. Fortunately replication has been around for a while, so this is pretty easy to

1.4. Global Scale

One of the most difficult challenges that needs to be solved is working at a global (and possibly planetary) scale. Concurrency has always been a difficult problem to tackle.

Locks are one way of ensuring writes don't conflict, but the problem gets harder as datacenters become further and further apart. We don't usually need to think about the limitations of the speed of light, but when trying to coordinate potentially conflicting updates to individual rows in a database we need to start considering the round trip costs.

Cassandra is designed to address the needs of rapid growth, scaling both reads and writes while ensuring uptime.

Lots of teams have started with relational databases and as they have grown in size have adapted them to solve bigger and bigger problems. In this chapter we'll look at how these teams have addressed issues of scale with relational databases and how Cassandra has been designed with those lessons in mind.

Getting Started

This chapter will help you get started using Apache Cassandra on your local environment to learn the basics. We'll start by running a single node then graduate to running a small dev cluster.

Along the way in this book there will be exercises,

Chapter 2. Downloading The Latest Release

Here we'll download

Go to https://cassandra.apache.org/

Click download

go to latest version, click the version number to download.

Download from one the mirrors.

Once the download finishes

At the time of this writing, the latest version is 3.11.3. Untar the archive:

```
$ tar -zxvf apache-cassandra-3.11.3-bin.tar.gz
```

Go into the directory and start Cassandra:

```
$ cd apache-cassandra-3.11.3
$ bin/cassandra -f
```

This will launch Cassandra in the foreground.

Open another terminal, go into the same directory and do the following:

```
$ bin/cqlsh
```

cqlsh will start up, and as long as the Cassandra server is running you'll see something like the following:

```
Connected to Test Cluster at 127.0.0.1:9042.
[cqlsh 5.0.1 | Cassandra 3.11.3 | CQL spec 3.4.4 | Native protocol v4]
Use HELP for help.
cqlsh>
```

You are now connected to Cassandra.

Switch back to the terminal running Cassandra and press control-c to stop.

Chapter 3. Starting a Cluster with CCM

CCM is a useful tool for creating local clusters for testing.

You can get the source here https://github.com/riptano/ccm

```
pip install ccm
```

ccm uses loopback aliases to run several Cassandra instances on one machine. We'll get into the details of how this actually works when we start looking at configuration file options. For now, just set them up with this in your shell:

```
sudo ifconfig lo0 alias 127.0.0.2 up
sudo ifconfig lo0 alias 127.0.0.3 up
sudo ifconfig lo0 alias 127.0.0.4 up
sudo ifconfig lo0 alias 127.0.0.5 up
sudo ifconfig lo0 alias 127.0.0.6 up
```

Now you can create a cluster. We'll create a single DC cluster with 3 nodes running 3.11.3:

```
$ ccm create -n 3 -v 3.11.3 mc
Current cluster is now: mc
```

Now you can start your cluster:

```
$ ccm start
```

We'll be referencing this test cluster in the next chapter.

Distributed Architecture

Chapter 4. Shared Nothing Design

One of the most interesting aspects of Cassandra is that it's designed to run on cheap hardware and scales horizontally rather than vertically. That means instead of buying more and more expensive servers we buy a bunch of cheap servers. We don't rely on expensive SANs or other types of complex shared storage.

Each node should have it's own storage, and ideally be dedicated hardware. While it's possible to run Cassandra on virtual machines, usually that's just adding unnecessary overhead.

Given the choice, if you're buying your own hardware, opt for a simple design. That doesn't mean use your old Pentium desktops and 5400 RPM disks though.

It's easy to want to use epic hardware when setting up your Cassandra cluster, but it's unnecessary. We recommend aiming for 8-16 CPU threads and 32-64 GB RAM. We recommend using SSDs, they've come down a lot in price a ton and make a huge difference. Friends don't let friends use spinning rust.

4.1. Token Ring

distributed hash table

each node has one or more tokens

Chapter 5. Replication

At the [Keyspace] level.

5.1. Endpoint Snitch

```
endpoint_snitch
```

Chapter 6. How Writes Work

Write begins at a [coordinator]. Every node in the cluster acts as a coordinator, it's not a special type of node.

The [Consistency Level]

Chapter 7. How Reads Work

Chapter 8. Consistency Levels

Writes can go to any node

In the case of a write, consistency defines the number of nodes that must acknowledge (ACK) the write.

In the case of a read, consistency defines the number of nodes that are queried, and in the case of an inconsistency, a read repair is performed.

8.1. ONE and LOCAL_ONE

Only one node is required to acknowledge the write for the [coordinator] to return a success message to the client.

8.2. ALL

All nodes are required to acknowledge the write for the [coordinator] to return a success message to the client.

8.3. QUORUM and LOCAL_QUORUM

Storage Engine

Cassandra's storage engine is implemented as a Log Structured Merge Tree, which has some interesting qualities. We'll dig into that in this chapter. At the end of this chapter you should have a good understanding of the different components of the storage engine.

Incoming writes are first written to a Commit Log. This is pretty straightforward, if you've used other databases this should be familiar territory.

Once the commit log has been written to, data is then written to an in memory structure called a Memtable.

The Memtable is later flushed to disk, to a file called an SSTable. There are some additional files that might be written out to accompany the SSTable, we'll cover that in the SSTable chapter.

Let's look at each component in a little more detail.

Chapter 9. Commit Log

The commit log is probably one of the more boring aspects of a database but is essential if we want to ensure we're able to recover our writes after a node has restarted unexpectedly. Writes in the commitlog will be replayed when the node starts back up.

The commit log can be disabled at the keyspace level.

Cassandra has a change data capture feature that might be useful for certain use cases. CDC is frequently used to propogate changes to other systems. When CDC is enabled, commit log segments will be moved to a `cdc_raw` directory, which can then be processed by leveraging the `CommitLogReader`.

CDC is an advanced topic, I'll discuss it further in a later chapter. For *most* use cases there's almost always a better alternative.

Chapter 10. Memtable

Think of Cassandra as a big write-back cache. Writes are only held in memory (and in the commit log), they're not written to a permanent data structure right away. When the cache (memtable) is full, it gets flushed to disk as an SSTable and is later merged with other SSTables as part of the Compaction process.

Chapter 11. SSTable

Data files on disk.

Bloom filter: na-10-big-Filter.db

Data file na-9-big-Data.db

na-9-big-CompressionInfo.db

na-9-big-Statistics.db

Partition index na-9-big-Index.db

Chapter 12. Compaction

Compaction is the process of merging sstables. There seems to be a misconception that compaction is optional, or that it shouldn't be running. This is far from the truth. On a healthy system receiving writes, expect to see plenty of active compactions.

12.1. Compaction Strategies

Multiple compactions strategies

12.1.1. Size Tiered Compaction Strategy (STCS)

default, not great at much

buckets

12.1.2. Leveled Compaction Stategy (LCS)

Leveled

fixed size

12.1.3. Time Series Compaction Strategy (TWCS)

Built originally by Jeff Jirsa, a Cassandra committer.

Groups sstables into windows by using the max timestamp of all the cells in the sstable.

Data Modeling

Chapter 13. Keyspaces

As I discussed in the Replication chapter, the settings on the keyspace determine how data is replicated around the cluster.

13.1. Replication Strategies

13.1.1. Simple Strategy

13.1.2. Network Topology Strategy

Chapter 14. Tables

Much like a relational database, we create structured tables with schema in Cassandra to store our data.

Tables contain Rows.

Unlike a traditional relational database, we need to specify how we want our data partitioned across

What can we use as a sort date

You might recognize this syntax:

```
CREATE TABLE mytable (
    id uuid PIRMARY KEY,
    data blob
);
```

14.1. Partition Key

Partition key can be composed of one or more fields. These fields are hashed to give us a token, which cooresponds to a position in the Token Ring, and that determines the replicas that a given partition lives on.

If we're using the above `CREATE TABLE` syntax you see above, a single field has been identified as the `PRIMARY KEY`. That lone field is the partition key.

We can also use this syntax:

```
CREATE TABLE mytable (
    id uuid,
    data blob,
    PRIMARY KEY (id)
)
```

14.2. Clustering Keys

One of the most powerful aspects of Cassandra's partitions is that they can contain one or more rows. Rows in a partition are stored together, sequentially, in the order defined by the clustering keys.

```
CREATE TABLE test.mytable (
    id int,
    clust int,
    data blob,
    PRIMARY KEY (id, clust)
) WITH CLUSTERING ORDER BY (clust ASC)
```

```
CREATE TABLE comlex (
    id uuid,
    bucket int,
    data blob,
    PRIMARY KEY (id, bucket)
);
```

Partitions sorted lists of rows

```
CREATE TABLE sensor_data (
  id uuid,
  ts timestamp,
  data blob
) PRIMARY KEY (id, ts);
```

Chapter 15. Partitions

Chapter 16. Rows

Common Operations

Chapter 17. Starting a New Cluster

17.1. Initial Configuration

17.1.1. Use Gossipping Property File Snitch

Gossipping Property File Snitch, or GPFS, is used to help Cassandra understand where

Endpoint Snitch

17.1.2. Disable Dynamic Snitch

Dynamic snitch is intended to help keep the cluster performing well. Unfortunately it falls far short on its promise, and in most cases it causes the cluster to behave more erratically.

I recommend disabling dynamic snitch by setting the following in your `cassandra.yaml`:

```
dynamic_snitch: false
```

17.2. Set the Initial Tokens for the First Three Nodes

17.3. Create A Keyspace

As mentioned in the Keyspaces chapter, a keyspace contains tables and has Replication settings that control where data lives.

```
CREATE KEYSPACE myks (

) WITH replication
```

Chapter 18. Adding a Datacenter

Performance Tuning

Chapter 19. Tools

Glossary

Coordinator

The node that recieves the request from a client.