lab 2. Cassandra Architecture

In this lab, we will discuss the architecture behind Apache Cassandra in detail. We will discuss how Cassandra was designed and how it adheres to the **Brewer's CAP** theorem, which will give us insight into the reasons for its behavior. Specifically, this lab will cover:

- Problems that Cassandra was designed to solve
- Cassandra's read and write paths
- The role that horizontal scaling plays
- How data is stored on-disk
- How Cassandra handles failure scenarios

This lab will help you to build a good foundation of understanding that will prove very helpful later on. Knowing how Apache Cassandra works under the hood helps for later tasks around operations. Building high-performing, scalable data models is also something that requires an understanding of the architecture, and your architecture can be the difference between an unsuccessful or a successful cluster.

Why was Cassandra created?

Understanding how Apache Cassandra works under the hood can greatly improve your chances of running a successful cluster or application. We will reach that understanding by asking some simple, fundamental questions. What types of problems was Cassandra designed to solve? Why does a relational database management system (RDBMS) have difficulty handling those problems? If Cassandra works this way, how should I design my data model to achieve the best performance?

RDBMS and problems at scale

As the internet grew in popularity around the turn of the century, the systems behind internet architecture began to change. When good ideas were built into popular websites, user traffic increased exponentially. It was not uncommon in 2001 for too much web traffic being the reason for a popular site being slow or a web server going down. Web architects quickly figured out that they could build out multiple instances of their website or application, and distribute traffic with load balancers.

While this helped, some parts of web architectures were difficult to scale out, and remained bottlenecks for performance. Key among them was the database. This problem was further illustrated by Dr. Eric Brewer and Dr. Armando Fox, in their 1999 paper Harvest, Yield, and Scalable Tolerant Systems (http://lab.mscs.mu.edu/Dist2012/lectures/HarvestYield.pdf). This paper was key in describing how large-scale systems would have to be built in the future, in that achieving only two of consistency, availability, and [network]partition tolerance were possible at any one time. The points discussed in Brewer and Fox's paper would go on to be commonly referred to as Brewer's CAP theorem.

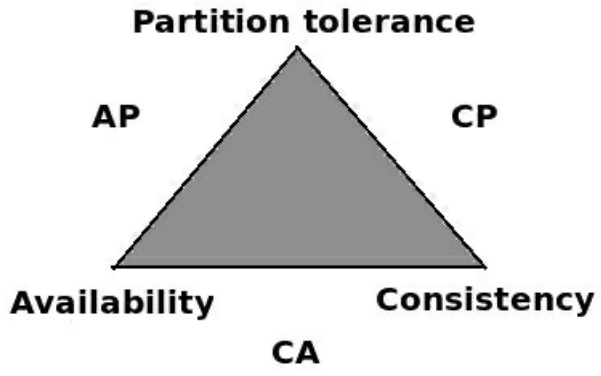
This is significant, because when a relational database receives so much traffic that it cannot serve requests, there is little that can be done. Relational databases are both highly-available and consistent, at the expense of not being tolerant to network interruptions (partitions). This is ultimately due to their single-instance, monolithic design.

Perhaps the only way to give relational databases more resources to work with was to scale vertically. This meant adding things such as RAM, CPU cores, and disks to the existing machine instance. While vertical scaling helped to alleviate some of the problems presented by the web scale, it was limited by how many resources the single server architecture could support. It wouldn't be long before vertical scaling was revealed as a band-aid, and the problem would soon present itself again.

NoSQL databases largely came about as new and innovative ways to solve this problem. The key idea was that data could be distributed across multiple instances—providing more resources to a distributed dataset, simply by adding more instances. This practice became known as horizontal scaling. This design required selecting network partition tolerance as an early design choice. Whether more importance was to be placed on data consistency or high availability was largely driven by whichever problem the system architects were trying to solve.

Cassandra and the CAP theorem

Many of the design ideas behind Apache Cassandra were largely influenced by Amazon Dynamo. It embraced partition-tolerance to be able to scale horizontally when needed, as well as to reduce the likelihood of an outage due to having a single point of failure. Cassandra also prefers to serve its data in a highly-available fashion, and embraces the concept of eventual consistency:



A graphical representation of Brewer's CAP theorem, using sides of a triangle to represent the combination of the different properties of consistency, availability, and partition tolerance

When data is written, it is replicated to multiple instances. During this process, it is possible that one (or more) of the replica writes could fail. Reads for this data would then return stale or dirty results. However, one of the design choices for Cassandra was that the possibility of serving stale data outweighed the risk of a request failing. This clearly gives Cassandra a CAP designation as an AP database.

Cassandra's ring architecture

An aspect of Cassandra's architecture that demonstrates its AP CAP designation is in how each instance works together. A single-instance running in Cassandra is

known as a **node**. A group of nodes serving the same dataset is known as a **cluster** or **ring**. Data written is distributed around the nodes in the cluster. The partition key of the data is hashed to determine it's **token**. The data is sent to the nodes responsible for the token ranges that contain the hashed token value.

Note

The consistent hashing algorithm is used in many distributed systems, because it has intrinsic ways of dealing with changing range assignments.You can refer to *Cassandra High Availability* by Strickland R. (2014), published by fenago.

The partition key (formerly known as a **row key**) is the first part of `PRIMARY KEY` , and the key that determines the row's token or placement in the cluster.

Each node is also assigned additional, ancillary token ranges, depending on the replication factor specified. Therefore, data written at an RF of three will be written to one primary node, as well as a secondary and tertiary node.

Partitioners

The component that helps to determine the nodes responsible for specific partitions of data is known as the partitioner. Apache Cassandra installs with three partitioners. You can refer to Apache Cassandra 3.0 for DSE 5.0, *Understanding the Architecture*. Retrieved on 20180404 from: https://docs.datastax.com/en/cassandra/3.0/cassandra/architecture/archTOC.html. Partitioners differ from each other in both efficiency and data-distribution strategy.

ByteOrderedPartitioner

`ByteOrderedPartitioner` sorts rows in the cluster lexically by partition key. Queried results are then returned ordered by that key. Tokens are calculated by the hexadecimal value of the beginning of the partition key.

When we discuss data modeling and CQL later on, the ability to order a result set by a partition key may seem like a good idea. But there are several problems with this partitioner. Foremost among them is that data distribution typically suffers.

RandomPartitioner

`RandomPartitioner` was the default partitioner prior to Apache Cassandra 1.2. Tokens are calculated using a MD5 hash of the complete partition key. Possible token values range from zero to *(2^127^) - 1*. For additional information you can refer to Saha S. (2017). *The Gossip Protocol - Inside Apache Cassandra*. Retrieved on 20180422 from: https://www.linkedin.com/pulse/gossip-protocol-inside-apache-cassandra-soham-saha/

Murmur3Partitioner

`Murmur3Partitioner` is an improvement in efficiency over `RandomPartitioner` , and is currently the default partitioner. The **murmur3** hashing algorithm is more efficient, because Cassandra does not really need the extra benefits provided by a cryptographic hash (MD5). Possible token values range from *-(2^63^)* to *(2^63^)-1*.

Important points about Apache Cassandra's partitioners:

- A partitioner is configured at the cluster level. You cannot implement a partitioner at the keyspace or table level.
- Partitioners are not compatible. Once you select a partitioner, it cannot be changed without completely reloading the data.
- Use of `ByteOrderedPartitioner` is considered to be an anti-pattern.
- `Murmur3Partitioner` is an improvement on the efficiency of `RandomPartitioner` .

`Murmur3Partitioner` is the default partitioner, and should be used in new cluster builds.

Note

`ByteOrderedPartitioner` and `RandomPartitioner` are still delivered with Apache Cassandra for backward compatibility, providing older implementations with the ability to upgrade.

Single token range per node

Prior to Apache Cassandra 1.2, nodes were assigned contiguous ranges of token values. With this approach, token ranges had to be computed and configured manually, by setting the `initial_token` property in `cassandra.yaml` . A possible token distribution (using `Murmur3Partitioner`) with single token ranges assigned per node is as follows:

Node # Start token End token 0 `5534023222112865485` `-9223372036854775808` 1 `-9223372036854775807` `-5534023222112865485` 2 `-5534023222112865484` `-1844674407370955162` 3 `-1844674407370955161` `1844674407370955161` 4 `1844674407370955162` `5534023222112865484`

Table 2.1: An example of the token range assignments for a five-node cluster, where each node is assigned responsibility for a single, contiguous token range

With one node responsible for a single, contiguous token range, nodes would follow each other in the ring. When rows are written to the node determined by the hashed token value of their partition key, their remaining replicas are written to ancillary ranges on additional nodes. The additional nodes designated to hold secondary ranges of data are determined by walking the ring (or data center, depending on replication strategy) in a clockwise direction:

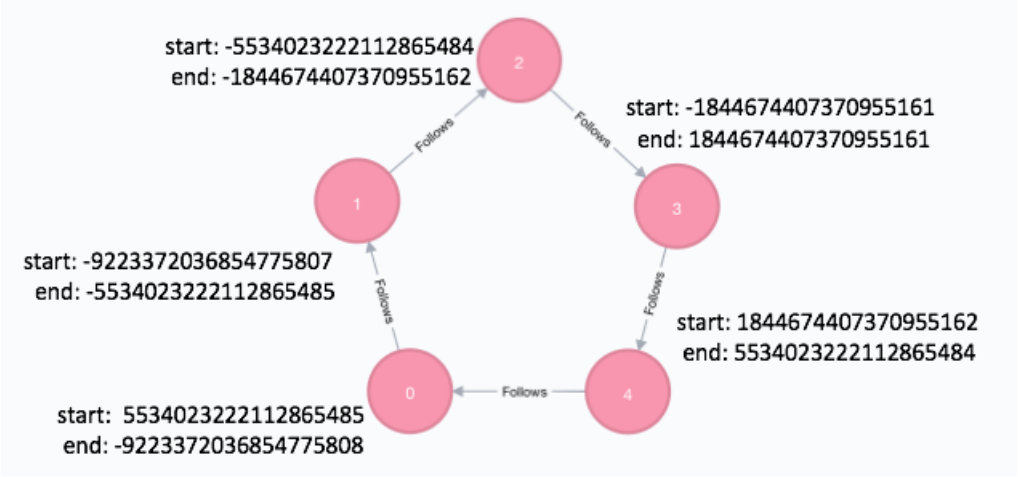


Figure 2.2: A five-node cluster (non-vnodes) illustrating how one node follows another based on their start and end token ranges

For example, consider the `hi_scores_by_game` table from the previous lab. As that table uses `game` as its partition key, data stored in that table will be written to a node based on the hash of the value it has for `game` . If we were to store data for the `Pacman` game, its hashed token value would look like this:

```
cassdba@cqlsh:fenago&gt; SELECT game, token(game)
    FROM hi_scores_by_game WHERE game=Pacman;

 game        | system.token(game)
-------------+----------------------
      Pacman | 4538621633640690522
```

The partition key value of `Pacman` hashes to a token of `4538621633640690522` . Based on the figures depicting the preceding five-node cluster, node four is primarily responsible for this data, as the token falls between `1844674407370955162` and `5534023222112865484` .

Now let's assume that the `fenago` keyspace has the following definition:

```
CREATE KEYSPACE fenago WITH replication =
  {'class': 'NetworkTopologyStrategy', 'ClockworkAngels': '3'}
  AND durable_writes = true;
```

If all five of the nodes are in the `ClockworkAngels` data center, then the nodes responsible for the two additional replicas can be found by traversing the ring in a clockwise manner. In this case, node zero is responsible for the second replica of the `Pacman` high score data, and node one is responsible for the third replica.

Note

`NetworkTopologyStrategy` also takes logical rack definition into account. When determining additional replica placement, this strategy will attempt to place replicas on different racks, to help maintain availability in the event of a rack failure scenario.

Vnodes

Newer versions of Apache Cassandra have an option to use virtual nodes, better known as **vnodes**. Using vnodes, each node is assigned several non-contiguous token ranges. This helps the cluster to better balance its data load:

Node # Token ranges 0 `79935 to -92233, -30743 to -18446, 55341 to 67638` 1 `-92232 to -79935, -61488 to 61489, 43043 to 55340` 2 `-79934 to -67638, -43041 to -30744, 67639 to 79935` 3 `-67637 to -55340, 61490 to 18446,18447 to 30744` 4 `-55339 to -43042, -18445 to -61489, 30745 to 43042`

Table 2.2: An example of the token range distribution for a five-node cluster using vnodes. Each node is assigned responsibility for three, non-contiguous token ranges. Smaller numbers were used in this example for brevity.

While the application of virtual nodes may seem more chaotic, it has several benefits over single, manually-specified token ranges per node:

- Provides better data distribution over manually-assigned tokens.
- More numerous, multiple token ranges per node allows multiple nodes to participate in bootstrapping and streaming data to a new node.
- Allows for automatic token range recalculation. This is useful when new nodes are added to the cluster, to prevent the existing nodes from needing to be reconfigured and bounced (restarted).

Note

Cassandra 3.0 uses a new token-allocation algorithm that allows the distribution of data to be optimized for a specific keyspace. Invoking this algorithm is only possible when using vnodes and `Murmur3Partitioner` . It requires that a keyspace be specified for the `allocate_tokens_for_keyspace` property in the

`cassandra.yaml` file. You can refer to Apache Software Foundation (2016). Documentation: *Adding, replacing, moving and removing nodes,* retrieved on 20180418 from: http://cassandra.apache.org/doc/latest/operating/topo_changes.html.

Cassandra's write path

Understanding how Cassandra handles writes is key to knowing how to build applications on top of it. The following is a high-level diagram of how the Cassandra write path works:

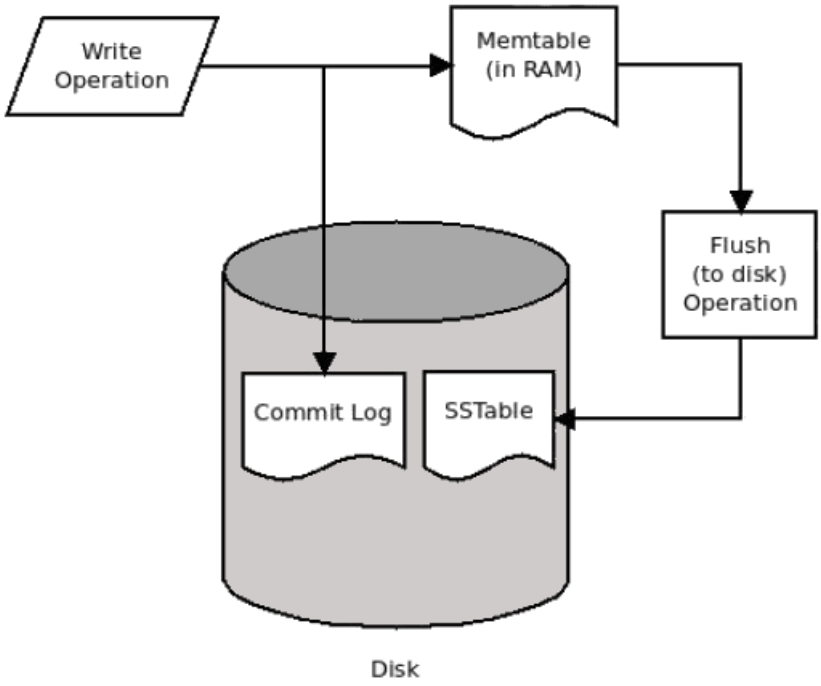


Figure 2.3: An illustration of the Cassandra write path, showing how writes are applied both to in-memory and on-disk structures

When a write operation reaches a node, it is persisted in two places. There is an in-memory structure known as a **memtable**, which gets the write. Additionally, the new data is written to the commit log, which is on-disk.

Note

The commit log is Cassandra's way of enforcing durability in the case of a *plug-out-of-the-wall* event. When a node is restarted, the commit log is verified against what is stored on-disk and replayed if necessary.

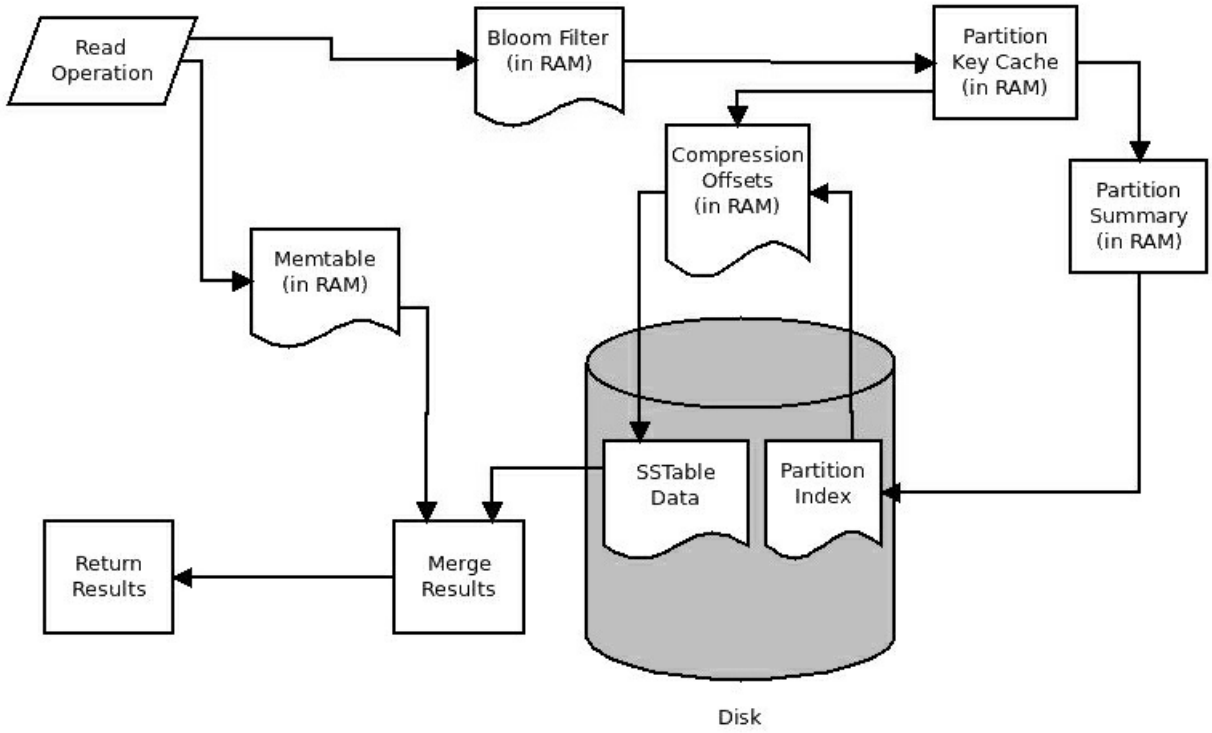
Once a flush of the memtable is triggered, the data stored in memory is written to the sorted string table files (**SSTables**) on-disk. The data is sorted by token value in memory, and then written to disk in sequential order. This is why Cassandra is known for its log-based, append-only storage engine.

Some quick notes about the write path:

- Data is sorted inside SSTables by token value, and then by clustering keys.
- SSTables are immutable. They are written once, and not modified afterward.
- Data written to the same partition key and column values does not update. The existing row is made obsolete (not deleted or overwritten) in favor of the timestamp of the new data.
- Data cannot be deleted. Deletes are considered to be writes, and are written as structures called **tombstones**.

Cassandra's read path

The Cassandra read path is somewhat more complex. Similar to the write path, structures in-memory and on-disk structures are examined, and then reconciled:



An illustration of the Cassandra read path, illustrating how the different in-memory and on-disk structures work together to satisfy query operations

As shown in the preceding figure, a node handling a read operation will send that request on two different paths. One path checks the memtables (in RAM) for the requested data.

If row-caching is enabled (it is disabled by default), it is checked for the requested row, followed by the bloom filter. The bloom filter (Ploetz, et-al 2018) is a probability-based structure in RAM, which speeds up reads from disk by determining which SSTables are likely to contain the requested data.

If the response from the bloom filter is negative, the partition key cache is examined. Unlike the row cache, the partition key cache is enabled by default, and its size is configurable (it defaults to the smaller value of either 5% of available heap or 100 MB). If the requested partition key is located within the partition key cache, its result is then sent to the compression offset. The compression offset is a dictionary/map that contains the on-disk locations for all partition data (SSTables).

If the partition key cache does not contain the requested key, the request is routed to the partition summary instead. There the requested key is examined to obtain a range of partitions for the key, which is then sent to the partition index. The partition index contains all of the partition keys which the node is responsible for. This, in conjunction with the resultant compression offset, will lead to the appropriate SSTable(s) on-disk. The appropriate values are then read, merged with any results obtained from the memtables, and the resultant dataset is returned.

Note

Some structures in the read path are predicated on a negative search result from a prior step. Requesting data that does not exist still consumes compute resources, and may even exhibit higher read latencies than queries for data that does exist.

Some quick notes about the read path:

Data is read sequentially from disk, and will be returned in that order. This can be controlled by the table's definition. Rows can persist across multiple SSTable files. This can slow reads down, so eventually SSTable data is reconciled and the files are condensed during a process called compaction. When data is read, its obsoleted or deleted values are read as well, and disregarded. Therefore data models allowing for multiple in-place writes and/or deletes will not perform well.

On-disk storage

When rows are written to disk, they are stored in different types of file. Let's take a quick look at these files, which should be present after what we did in previous lab.

SSTables

First of all, let's `cd` over to where our table data is stored. By default, keyspace and table data is stored in the `data/` directory, off of the `$CASSANDRA_HOME` directory. Listing out the files reveals the following:

```
cd data/data/fenago/hi_scores-d74bfc40634311e8a387e3d147c7be0f
ls -al
total 72
drwxr-xr-x  11 aploetz aploetz  374 May 29 08:28 .
drwxr-xr-x   6 aploetz aploetz  204 May 29 08:26 ..
-rw-r--r--   1 aploetz aploetz   43 May 29 08:28 mc-1-big-CompressionInfo.db
-rw-r--r--   1 aploetz aploetz  252 May 29 08:28 mc-1-big-Data.db
-rw-r--r--   1 aploetz aploetz   10 May 29 08:28 mc-1-big-Digest.crc32
-rw-r--r--   1 aploetz aploetz   16 May 29 08:28 mc-1-big-Filter.db
-rw-r--r--   1 aploetz aploetz   27 May 29 08:28 mc-1-big-Index.db
-rw-r--r--   1 aploetz aploetz 4675 May 29 08:28 mc-1-big-Statistics.db
-rw-r--r--   1 aploetz aploetz   61 May 29 08:28 mc-1-big-Summary.db
-rw-r--r--   1 aploetz aploetz   92 May 29 08:28 mc-1-big-TOC.txt
```

The `mc-1-big-TOC.txt` file is the table of contents file for the directory. Listing its contents essentially shows the same list of files as we have previously. Some of the other files match up with parts of the Cassandra read path:

- `mc-1-big-Index.db` : Primary key index (matches partition keys to their data file positions)
- `mc-1-big-Summary.db` : Partition summary
- `mc-1-big-CompressionInfo.db` : Compression offsets
- `mc-1-big-Filter.db` : Bloom filter
- `mc-1-big-Digest.crc32` : Contains a **Cyclic Redundancy Check** (**CRC**) or **checksum** value for the uncompressed file chunks

The last file is `mc-1-big-Data.db` , which contains the data for the table. Listing out the contents of the file with a system tool, such as cat, will be cumbersome to read. So we will examine this file using some of the tools that come with Apache Cassandra.

How data was structured in prior versions

In Apache Cassandra versions 0.4 to 2.2, SSTable data was structured by row key. A row key kept a map of columns underneath it with their values. These columns were sorted by their column key. If you consider the key/value map structure from Java, it's easy to think of it as a map of a map.

As an example, consider the following CQL query from the previous lab. Recall that it retrieves video game high-score data for a player named `Connor` :

```
cassdba@cqlsh:fenago&gt; SELECT * FROM hi_scores WHERE name='Connor';
 name   | game        | score
--------+-------------+--------
 Connor |     Frogger |   4220
 Connor |       Joust |  48850
 Connor | Monkey Kong |  15800
 Connor |      Pacman | 182330
(4 rows)
```

This CQL query and result set make sense in the way it is shown, mainly for those of us who have a background in relational databases. But this is not how the data is stored on-disk. In Apache Cassandra versions 1.2 to 2.2, CQL is simply a layer that abstracts the SSTable structure. If you were using Apache Cassandra 2.1 or earlier, this data could be viewed with the command-line interface tool, also known as `cassandra-cli` :

```
bin/cassandra-cli 192.168.0.101 -u cassdba -pw flynnLives
Connected to: "Rush" on 192.168.0.101/9160
Welcome to Cassandra CLI version 2.1.13
The CLI is deprecated and will be removed in Cassandra 2.2.  Consider migrating to cqlsh.
CQL is fully backwards compatible with Thrift data; see http://www.datastax.com/dev/blog/thrift-to-cql3
Type 'help;' or '?' for help.
Type 'quit;' or 'exit;' to quit.
[cassdba@unknown] use fenago;
Authenticated to keyspace: fenago
[default@fenago] get hi_scores['Connor'];
Using default limit of 100
Using default cell limit of 100
-------------------
RowKey: Connor
=&gt; (name=Frogger:, value=, timestamp=1527640087988234)
=&gt; (name=Frogger:score, value=000000000000107c, timestamp=1527640087988234)
=&gt; (name=Joust:, value=, timestamp=1527640087990162)
=&gt; (name=Joust:score, value=000000000000bed2, timestamp=1527640087990162)
=&gt; (name=Monkey Kong:, value=, timestamp=1527640087986113)
=&gt; (name=Monkey Kong:score, value=0000000000003db8, timestamp=1527640087986113)
=&gt; (name=Pacman:, value=, timestamp=1527640087984003)
=&gt; (name=Pacman:score, value=000000000002c83a, timestamp=1527640087984003)
1 Rows Returned.
```

A few things to note here:

- We defined the partition key as the `name` column, which does not seem to be present here. But the get query works by simply retrieving data for `RowKey` of `Connor`.
- The `game` column was defined as our clustering key. Again, `game` itself is not present here, but we can see that the score recorded for each game seems to be keyed by the game's title (such as `name=Frogger:score, value=000000000000107c, timestamp=1527640087988234`).
- The CLI informs us that one row has been returned. As far as the storage layer is concerned, all cells (column keys and their values) constitute a single row.

The most important thing to remember from this example is that all data underneath a particular partition key is stored together. This is an important lesson that we will draw upon heavily in the next lab.

How data is structured in newer versions

In versions of Apache Cassandra from 3.0 on, the underlying data is structured slightly differently. The Cassandra storage engine was rewritten as a part of this release, providing several benefits.

Consider the CQL query used as the basis for the prior example, where we queried our the `hi_scores` table for the player named `Connor`. If we were to look at the underlying data using the `sstabledump` tool, it would look something like this:

```
tools/bin/sstabledump data/data/fenago/hi_scores-73a9df80637b11e8836561ec0efea4b2/mc-1-big-Data.db
[ {
   "partition" : {
     "key" : [ "Connor" ],
     "position" : 0
   },
   "rows" : [
     {
       "type" : "row",
       "position" : 44,
       "clustering" : [ "Frogger" ],
       "liveness_info" : { "tstamp" : "2018-05-29T12:28:07.988225Z" },
       "cells" : [
         { "name" : "score", "value" : 4220 }
       ]
     },
     {
       "type" : "row",
       "position" : 44,
       "clustering" : [ "Joust" ],
       "liveness_info" : { "tstamp" : "2018-05-29T12:28:07.990283Z" },
       "cells" : [
         { "name" : "score", "value" : 48850 }
       ]
     },
     {
       "type" : "row",
       "position" : 66,
       "clustering" : [ "Monkey Kong" ],
       "liveness_info" : { "tstamp" : "2018-05-29T12:28:07.986579Z" },
       "cells" : [
         { "name" : "score", "value" : 15800 }
       ]
     },
     {
       "type" : "row",
       "position" : 94,
       "clustering" : [ "Pacman" ],
       "liveness_info" : { "tstamp" : "2018-05-29T12:28:07.984499Z" },
       "cells" : [
         { "name" : "score", "value" : 182330 }
       ]
     }
   ]
 }
```

It's important to note from this JSON output that the data for `Connor` is divided into a hierarchy of three distinct sections: `partitions` , `rows` , and `cells` . Whereas before there were simply partitions containing cells of data, rows are now a first-class citizen in the Cassandra world. In this way, CQL is no longer an abstraction, but rather a fairly accurate representation of how the data is stored. This will be discussed in more detail in the next lab.

Note

Even with the new storage engine, it is still important to note that all data underneath a particular partition key is stored together.

Additional components of Cassandra

Now that we have discussed the read and write paths of an individual Apache Cassandra node, let's move up a level and consider how all of the nodes work together. Keeping data consistent and serving requests in a way that treats multiple machines as a single data source requires some extra engineering. Here we'll explore the additional components which make that possible.

Gossiper

Gossiper is a peer-to-peer communication protocol that a node uses to communicate with the other nodes in the cluster. When the nodes gossip with each other, they

share information about themselves and retrieve information on a subset of other nodes in the cluster. Eventually, this allows a node to store and understand state information about every other node in the cluster.

Each node's gossiper runs once per second (Saha 2017) and initiates communication with up to three other nodes. The gossip protocol first runs when the node is attempting to join the cluster. As a new node contains no information on the current network topology, it must be preconfigured with a list of current live nodes or `seeds` to help it start the communication process.

Note

We will discuss the concept of seed nodes in lab 4, *Configuring a Cluster*. The most important thing to remember about them, is that they help new nodes discover the rest of the cluster's network topology when joining. All nodes are considered to be peers in Cassandra, and the seed-node designation does not change that.

You can view the currently perceived gossip state of any node using the `nodetool` utility on any node. The following command demonstrates what this looks like with a three-node cluster:

```
 bin/nodetool gossipinfo

/192.168.0.101
 generation:1524418542
 heartbeat:462
 STATUS:14:NORMAL,-9223372036854775808
 LOAD:205:161177.0
 SCHEMA:10:ea63e099-37c5-3d7b-9ace-32f4c833653d
 DC:6:ClockworkAngels
 RACK:8:R40
 RELEASE_VERSION:4:3.11.2
 RPC_ADDRESS:3:192.168.0.101
 NET_VERSION:1:11
 HOST_ID:2:0edb5efa-de6e-4512-9f5d-fe733c7d448c
 RPC_READY:20:true
 TOKENS:15:&lt;hidden&gt;
/192.168.0.102
 generation:1524418547
 heartbeat:249
 STATUS:14:NORMAL,-3074457345618258603
 LOAD:205:49013.0
 SCHEMA:10:ea63e099-37c5-3d7b-9ace-32f4c833653d
 DC:6:ClockworkAngels
 RACK:8:R40
 RELEASE_VERSION:4:3.11.2
 RPC_ADDRESS:3:192.168.0.102
 NET_VERSION:1:11
 HOST_ID:2:404a8f97-f5f0-451f-8b72-43575fc874fc
 RPC_READY:27:true
 TOKENS:15:&lt;hidden&gt;
/192.168.0.103
 generation:1524418548
 heartbeat:248
 STATUS:14:NORMAL,3074457345618258602
 LOAD:205:54039.0
 SCHEMA:10:ea63e099-37c5-3d7b-9ace-32f4c833653d
 DC:6:ClockworkAngels
 RACK:8:R40
 RELEASE_VERSION:4:3.11.2
 RPC_ADDRESS:3:192.168.0.103
 NET_VERSION:1:11
 HOST_ID:2:eed33fc5-98f0-4907-a02f-2a738e7af562
 RPC_READY:26:true
 TOKENS:15:&lt;hidden&gt;
```

As you can see, the gossip state of each node contains a wealth of information about it. Things such as status, data load, host ID, and schema version are readily visible. This is a good way to quickly gain an understanding of one node's view of the cluster.

Note

Remember, each node maintains its own view of the cluster's gossip state. While it should be the same on all nodes, it sometimes is not. It is entirely possible that some nodes can see certain nodes as down when other nodes do not. The `nodetool gossipinfo` command can be a valuable tool when troubleshooting firewall or network issues.

Snitch

Cassandra uses a component known as the snitch to direct read and write operations to the appropriate nodes. When an operation is sent to the cluster, it is the snitch's job (Williams 2012) to determine which nodes in specific data centers or racks can serve the request.

Most snitches that ship with Apache Cassandra are both data center- and rack-aware. That is, they are capable of viewing (predetermined) groups of nodes in the cluster as **logical data centers**. Additionally, the data centers can be further divided into logical racks.

Note

The term **logical** is used here because the data centers may or may not be physically separate from each other. That is, nodes in a single physical data center can be split into smaller data centers, and the snitch will enforce data-replication for each separately.

The replication of data per keyspace is defined in the keyspace definition using the configuration properties of `NetworkTopologyStrategy` . This can be seen in the following example keyspace definition:

```
CREATE KEYSPACE fenago WITH replication = {
   'class': 'NetworkTopologyStrategy',
   'ClockworkAngels': '3',
   'PermanentWaves': '2'
};
```

In the previous example, we'll assume the use of `GossipingPropertyFileSnitch` , and that we have a number of nodes in our cluster assigned to our `ClockworkAngels` data center and some assigned to our `PermanentWaves` data center. When a write operation occurs on any of the tables stored in the `fenago` keyspace, the snitch will ensure that three replicas of that data are written to the `ClockworkAngels` data center. Likewise, two replicas of the data will be written to the `PermanentWaves` data center.

As previously mentioned, the nodes in a data center can be further divided into racks. Let's assume that our `ClockworkAngels` data center has six nodes. To maintain the maximum amount of availability possible, we could divide them up like this:

```
 Data center: ClockworkAngels
Node             Rack
========================
192.168.255.1    East_1
192.168.255.2    West_1
192.168.255.3    West_1
192.168.255.4    East_1
192.168.255.5    West_1
192.168.255.6    East_1
```

Note

Nodes in a data center are ordered by primary token range, not necessarily by IP address. This example was ordered by IP for simplicity.

Let's assume that our write has a partition key that hashes out to a primary range owned by `192.168.255.2` , which is in the `West_1` rack. The snitch will ensure that the first replica is written to `192.168.255.2` . Since our keyspace has an RF of three, the snitch will then walk the ring (clockwise by token ranges) to place the remaining replicas. As there are multiple racks, it will try to spread the replicas evenly across them. Therefore, it will place the second replica on `192.168.255.3` , which is in a different rack. As both racks are now covered from an availability perspective, the third replica will be placed on the next node, regardless of rack designation.

Note

When working with physical data centers, entire racks have been known to fail. Likewise, entire availability zones have been known to fail in the cloud. Therefore, it makes sense to define logical racks with physical racks or **Availability Zones** (**AZs**) whenever possible. This configuration strategy provides the cluster with its best chance for remaining available in the event of a rack/AZ outage. When defining your cluster, data center, and rack topology, it makes sense to build a number of nodes that evenly compliment your RF. For instance, with an RF of three, the snitch will spread the data evenly over a total of three, six, or nine nodes, split into three racks.

Phi failure-detector

Apache Cassandra uses an implementation of the **Phi Failure-Accrual** algorithm. The basic idea is that gossip between nodes is measured on a scale that adjusts to current network conditions. The results are compared to a preconfigured, static value (`phi_convict_threshold`) that helps a node decide whether another node should be marked as down. `phi_convict_threshold` will be discussed in later labs.

Tombstones

Deletes in the world of distributed databases are hard to do. After all, how do you replicate nothing? Especially a key for nothing which once held a value. Tombstones are Cassandra's way of solving that problem.

As Cassandra uses a log-based storage engine, deletes are essentially writes. When data is deleted, a structure known as a **tombstone** is written. This tombstone exists so that it can be replicated to the other nodes that contained the deleted replica.

Discussion about tombstones tends to be more prevalent in Cassandra application-developer circles. This is primarily because their accumulation in large numbers can be problematic. Let's say that data is written for a specific key, then deleted, and then written and deleted three more times, before being written again. With the writes for that key being stored sequentially, SSTable data files that back that key will contain five possible values and four tombstones. Depending on how often that write pattern continues, there could be thousands of tombstoned and obsoleted stored across multiple files for that key. When that key is queried, there is essentially an accumulation of garbage that must be overlooked to find the actual, current value.

Accumulation of tombstones will result in slower queries over time. For this reason, use cases that require a high amount of delete activity are considered an anti-pattern with Apache Cassandra.

Note

The Apache Cassandra configuration has settings for warnings and failure thresholds regarding the number of tombstones that can be returned in a query.

Tombstones are eventually removed, but only after two specific conditions are met:

- The tombstone has existed longer than the table's defined `gc_grace_seconds` period
- Compaction runs on that table

The `gc_grace_seconds` stipulation exists to give the tombstone ample time to either replicate, or be fixed by a repair operation. After all, if tombstones are not adequately replicated, a read to a node that is missing a tombstone can result in data ghosting its way back into a result set.

Hinted handoff

When a node reports another as down, the node that is still up will store structures, known as **hints**, for the down node. Hints are essentially writes meant for one node that are temporarily stored on another. When the down node returns to a healthy status, the hints are then streamed to that node in an attempt to re-sync its data.

Note

As storage is finite, hints will be stored for up to three hours, by default. This means that a down node must be brought back within that time frame, or data loss will occur.

With Apache Cassandra versions older than 3.0, hints were stored in the hints table of the system keyspace. The problem with this approach is that once the stored hints were replayed, they were then deleted. This frequently led to warnings and failures due to tombstone accumulation. Apache Cassandra version 3.0 and up moved to a file-based system of hint storage, thus avoiding this problem so long as the `hints` directory has disk space available.

Compaction

As previously mentioned, once SSTable files are written to disk, they are immutable (cannot be written to again). Additional writes for that table would eventually result in additional SSTable files. As this process continues, it is possible for long-standing rows of data to be spread across multiple files. Reading multiple files to satisfy a query eventually becomes slow, especially when considering how obsolete data and tombstones must be reconciled (so as not to end up in the result set).

Apache Cassandra's answer to this problem is to periodically execute a process called **compaction**. When compaction runs, it performs the following functions:

- Multiple data files for a table are merged into a single data file
- Obsoleted data is removed
- Tombstones that have outlived the table's `gc_grace_seconds` period are removed

There are two different compaction strategies available with Apache Cassandra: `SizeTieredCompactionStrategy` and `LeveledCompactionStrategy` . These will be detailed in a later lab. Compaction is configured on a per-table basis, and its configuration includes thresholds for determining how often compaction is run.

Note

Cassandra does a good job of figuring out when compaction needs to be run. The best approach is to let it do what it needs to do. If it should be run sooner, then the configuration should be altered on the table in question. Forcing compaction (via the command line) to run outside of its configured thresholds has been known to drastically delay future compaction times.

Repair

Sometimes data replicas on one or more Cassandra nodes can get out of sync. There are a variety of reasons for this, including prior network instability, hardware failure, and nodes crashing and staying down past the three-hour hint window. When this happens, Apache Cassandra comes with a repair process that can be run to rectify these inconsistencies.

Note

The repair process does not run on its own, and must be executed manually. Tools such as Reaper for Apache Cassandra allow for some automation, including scheduling cluster repairs.

Conditions that can cause data inconsistencies essentially do so by introducing factors that increase the entropy of the stored replicas. Therefore, Apache Cassandra employs an **anti-entropy** repair mechanism, which uses a binary Merkle tree to locate inconsistent replicas across different nodes. Discrepancies are then rectified by streaming the correct data from another node.

Merkle tree calculation

Merkle trees are created from the bottom up, starting with hashes of the base or leaf data. The leaf data is grouped together by partition ranges, and then it is hashed with the hash of its neighboring leaf to create a singular hash for their parent. Parents are hashed together with neighboring parents (to create their parents), until (Kozliner 2017) all have been combined into a single hash, known as the root. Based on the hash value of the root for a token range, the Merkle trees of different nodes can be quickly compared.

Note

A Merkle tree calculation can consume high amounts of resources (CPU, disk I/O) on a node, which can result in temporary query timeouts as the node continues to try to serve requests. For this reason, it is best to run repairs during times when client requests are low. The Merkle tree calculation part of the repair processes can be monitored with `nodetool compactionstats`, where it will display as a **validation compaction**.

Let's walk through a quick example. Assume that we've added high scores for more users, we're running a repair on the `hi_scores` table, and we're only concerned about the token range of `-9223372036854775807` to `-7686143364045646507`. We'll query our the `hi_scores` table for our users whose token value for `name` (our partition key) falls within that range:

```
cassdba@cqlsh&gt; SELECT DISTINCT name,token(name) FROM fenago.hi_scores
   WHERE token(name) &gt;= -9223372036854775807
     AND token(name) &lt;= -7686143364045646507;

 name   | system.token(name)
--------+----------------------
 Connor | -8880179871968770066
    Rob | -8366205352999279036
    Dad | -8339008252759389761
   Ryan | -8246129210631849419

(4 rows)
```

Note

Executing a range query on a partition key is only possible when also using the `token` function, as shown previously.

Now let's compute a hash (MD5 is used for this example) for all data within each partition:

Connor: h(dCo) = c13eb9350eb2e72eeb172c489faa3d7f

Rob: h(dRo) = 97ab517e5418ad2fe700ae45b0ffc5f3

Dad: h(dDa) = c9df57b434ad9a674a242e5c70587d8b

Ryan: h(dRy) = 9b961a89e744c86f5bffc1864b166514

To build a Merkle tree for this data, we start by listing each partition of hashed data separately, as leaf nodes. Then we will backtrack toward our root node by applying the hash function, `h()`, while combining our data recursively:

Figure 2.5: An example Merkle tree calculation for a node's token range containing four partitions of data. This illustrates how they are hashed and combined to compute the hash of the root (hCRDR).

Once the data for the partitions is hashed, it is then combined with its neighbor nodes to generate their parent node:

hCoRo = h(h(dCo) + h(dRo)) = 3752f8e5673ce8d4f53513aa2cabad40

hDaRy = h(h(dDa) + h(dRy)) = 394de4983b242dcd1603eecc35545a6d

Likewise, the final hash for the root node can be calculated as *Root node = hCRDR = h(hCoRo + hDaRy) = 7077d9488b37f4d39c908eb7560f2323*.

We can then perform this same Merkle tree calculation on another node (which is also responsible for the token range of *-9223372036854775807* to *-7686143364045646507*). If all of the underlying data is consistent (and affirms our state of anti-entropy), they should all respond with a root node hash of *7077d9488b37f4d39c908eb7560f2323*. If one or more of them does not, the up-to-date node then prepares to stream a subset of the affected range of data to fix the inconsistency.

Streaming data

Once discrepancies are identified via Merkle trees, data files are streamed to the node containing the inconsistent data. Once this streaming completes, the data should be consistent across all replicas.

Note

Repair streams consume network resources that can also interfere with a node's ability to serve requests. Repair progress can be monitored using the `nodetool netstats` command.

Read repair

Read repair is a feature that allows for inconsistent data to be fixed at query time. `read_repair_chance` (and `dclocal_read_repair_chance`) is configured for each table, defaulting to 0.1 (10% chance). When it occurs (only with read consistency levels higher than `ONE`), Apache Cassandra will verify the consistency of the replica with all other nodes not involved in the read request. If the requested data is found to be inconsistent, it is fixed immediately.

Do note that read repairs do not occur when read consistency is set to `ONE` or `LOCAL_ONE`. Also, reads at a consistency level of all will force a read repair to happen 100% of the time. However, these read repairs do incur latency, which is why they are typically set to low percentages.

Note

Sometimes a single partition or key will be all that is inconsistent. If you find yourself in this situation, forcing a read repair is a simple task via cqlsh. If the number of inconsistent rows is a manageable number, you can fix them by setting your consistency level to all, and querying each of them.

Security

One criticism of NoSQL data stores is that security is often an afterthought. Cassandra is no exception to this view. In fact, Apache Cassandra will install with all security features initially disabled. While clusters should be built on secured networks behind enterprise-grade firewalls, by itself this is simply not enough. There are several features that Cassandra provides that can help to tighten up security on a cluster.

Authentication

Apache Cassandra allows for user authentication to be set up. Similar to other Cassandra components, users and their (encrypted) credentials are stored in a table inside the cluster. Client connections are required to provide valid username/password combinations before being allowed access to the cluster. To enable this functionality, simply change the `authenticator` property in the `cassandra.yaml` file from its default (`AllowAllAuthenticator`) to `PasswordAuthenticator`:

```
authenticator: PasswordAuthenticator
```

Note

Once you have created your own DBA-level superuser account for administrative purposes, you should never use the default cassandra user again. This is because the Cassandra user is the only user that has special provisions in the code to query the `system_auth` tables at a consistency level higher than `ONE` (it uses `QUORUM`). This means that the default Cassandra user may not be able to log in if multiple nodes are down.

When user authentication is enabled, Cassandra will create the `system_auth` keyspace. When using `PasswordAuthenticator`, `CassandraRoleManager` must also be used.

Authorization

With authentication set up, additionally enabling user authorization allows each user to be assigned specific permissions to different Cassandra objects. In Apache Cassandra versions 2.2 and up, users are referred to as **roles**. With authorization enabled, roles can be assigned the following permissions via CQL:

- `ALTER`: Change table or keyspace definitions
- `AUTHORIZE`: Grant or revoke specific permissions to other users
- `CREATE:`: Create new tables or keyspaces
- `DESCRIBE`: Retrieve definitions of a table or keyspace
- `DROP`: Delete specific tables or keyspaces
- `EXECUTE`: Run functions in a specific keyspace
- `MODIFY`: Run `DELETE`, `INSERT`, `UPDATE`, or `TRUNCATE` commands on a specific table
- `SELECT`: Query a specific table
- `ALL PERMISSIONS`: No restricted access on specific tables or keyspaces

The following are some examples:

```
GRANT SELECT ON KEYSPACE item TO item_app_readonly;
GRANT MODIFY ON KEYSPACE store TO store_app_user;
GRANT ALL PERMISSIONS ON KEYSPACE mobile TO mobile_admin;
```

Managing roles

As role-management is part of the user security backend, it requires `PasswordAuthenticator` to be enabled before its functionality is activated. The role system allows you to assign permissions to a role, and then `GRANT` access to that role for another user/role. As mentioned in the previous section, role management is new as of Apache Cassandra 2.2, and can greatly simplify user-management and permission-assignment.

Here is an example:

```
CREATE ROLE cassdba WITH SUPERUSER=true AND LOGIN=true AND PASSWORD='bacon123';
CREATE ROLE supply_chain_rw WITH LOGIN=true AND PASSWORD='avbiuo2t48';
```

Client-to-node SSL

Enabling client-to-node **Secure Socket Layer** (**SSL**) security has two main benefits. First, each client connecting to the cluster must present a valid certificate that matches a certificate or **Certificate Authority** (**CA**) in the node's Java KeyStore. Second, upon successful connection (and cert validation), all traffic between the client and node will be encrypted.

Node-to-node SSL

Enabling node-to-node SSL security is designed to prevent a specific avenue of attack. A node will not be allowed to join the cluster, unless it presents a valid SSL certificate in its Java KeyStore and the Java TrustStore matches with the other nodes in the cluster. When this level of security is active, it will encrypt communication between the nodes over port `7001` .

Note

Node-to-node SSL may seem unnecessary, but without it, an internal attacker could join a rogue node to your cluster. Then once it has bootstrapped, it will shut the node down and its data directories will contain at least some of your data. If you don't enable node-to-node SSL, this can be done without having to know any of the admin passwords to authenticate to the cluster.

Summary

In this lab, we discussed many aspects of Apache Cassandra. Some concepts may not have been directly about the Cassandra database, but concepts that influenced its design and use. These topics included Brewer's CAP theorem data-distribution and- partitioning; Cassandra's read and write paths; how data is stored on-disk; inner workings of components such as the snitch, tombstones, and failure-detection; and an overview of the delivered security features.

This lab was designed to give you the necessary background to understand the remaining labs. Apache Cassandra was architected to work the way it does for certain reasons. Understanding why will help you to provide effective configuration, build high-performing data models, and design applications that run without bottlenecks. In the next lab, we will discuss and explore CQL, and explain why it has certain limitations, and how to unleash its power.