Illinois Data Science Initiative

Technical Report

Introduction to Cassandra

Author:
Bhuvan Venkatesh

April 9, 2017

Introduction to Cassandra

BHUVAN VENKATESH[1] **AND ROBERT BRUNNER**[2]

[1] *National Center For Supercomputing Applications (NCSA)*
[2] *Laboratory for Computation, Data, and Machine Learning*

Compiled April 9, 2017

Distributed computing is becoming more imporatant as computer engineering is reaching significant challenges in vertical scaling. As such, transitioning traditional infrastructure to distributed infrastrucre requires programming and thinking on the software end. One transition is moving from a traditional relational database to a truly distributed database where failure and network traffic are taken into account. Apache cassandra meets those need, and in this technical report, you will be able to compare and contrast Cassandra to traditional systems, set up a single/multi node cluster and load some sample data either through the command line interface or through a programming language API.

https://github.com/lcdm-uiuc

INTRODUCTION

Apache Cassandra is an Open Source distributed database. Cassandra is a highly available and fault tolerance database. The use cases for Cassandra are when you are familiar with the typical SQL databases, and you need to expand horizontally than vertically. There are a few quirks that naturally come moving to the distributed world as well as a few limitations but as long as they fit the project paramaters, Cassandra is an excellent alternative to chaining multiple RDBMS together.

CASSANDRA VS RDBMS

Big Table

Cassandra is a big table data store, meaning that every row in the table need not be filled out when inserting data. Big table is also geared toward

Querying Data

In Cassandra there are two main differences, you usually store data by what you will search for later. Meaning that if you are going to query based on one feature, then you create a table for that one feature with all the (non primary-key) columns describing that one thing.

For example, if you are modeling stock market data and all the querying will be done by day – maybe getting the average outlook for the day – the ideal set up would be one table indexed by date. If you need to query by company, then set up the table to be outlook by company. If you need both, it would be ideal to set up tables for each company and add on a time index to each of the tables. You can query based on non primary-key attributes, but it will impact your performance.

Data Distribution

In relational database management systems, the primary key tells the database how to optimize the store and order the btree structure physically for increased performance. The database is free to ignore this restriction and follow the clustered index.

In Cassandra, the primary key tells cassandra which node the data should go to. Cassandra computes a hash value and distributes the rows to different data nodes. These nodes actually store the data, they can go down at any time, and they can come back up at any time and Cassandra will automatically keep track of the data distribution.

USECASES

1. Frequently Changing Inventories (Highly available database with lots of writes). Cassandra is perfect due to the availibility and the write throughput. Other services could hook into Cassandra and compute analytics but if you are keeping track of multiple stocks then this is a huge benefit because all the data is on one system.

2. Tweeting sinks (Single source database). Cassandra is good for write heavy data stores where there is one source but that source updates rapidly.

3. Internet of things/Texting sinks (Multiple source database). Cassandra is good for multiple users sending data back to a centralized data store. It handles load balancing and when servers fail so these high frequency services can keep rolling.

INSTALLATION

For those of you on docker to run all you need to do is

```
$ docker pull cassandra
$ docker run cassandra
```

If not, the non-dockerized installation is below.

Prerequisites

1. Java 1.6 or Above

2. Python 2.7 or Above

First you will need to set a new user for Cassandra, letting Cassandra run under normal users is not recommended, so we create a new user

```
$ useradd cassandra # Name for the user
```

Navigate to *http : //cassandra.apache.org/download/* and select the most recent stable version of cassandra (Usually the most recent or the second most recent version) and execute

```
$ wget <url>
$ tar zxvf <file>.tar.gz
$ mv <folder> /opt/cassandra # May need root
$ export CASSANDRA_HOME=/opt/cassandra
$ chown -R cassandra $CASSANDRA_HOME
```

After this, create the directories that cassandra keeps its data in.

```
$ mkdir /var/lib/cassandra
$ chown cassandra /var/lib/cassandra
$ mkdir /var/log/cassandra
$ chown cassandra /var/log/cassandra
```

You can change this in the settings explained in a future section.

To run cassandra, switch to the cassandra user. Then, execute the cassandra executable

```
$ sudo su cassandra
$ cd $CASSANDRA_HOME
$ ./bin/cassandra -f # or if you want to run
    it in the background -d
```

And you have cassandra running! To make sure that everything is working on a different terminal

```
$ cd $CASSANDRA_HOME
$ ./bin/cqlsh
>       #If there are no errors, your single
    node cluster is up.
```

CONFIGURING MULTINODE CLUSTERS

Backing up date on previous

First, if you have a single node cluster you **must** complete these instructions or your system will be in a halfway state. Delete all the data associated with your cassandra instance. If you have important data, there are ways to back it up.

```
> ./bin/cqlsh
> COPY table_name TO '/user/you/file_name'
# After the switch to multinode you can do
> COPY table_name FROM '/user/you/file_name'
```

Ideally with large databases, you may want to put it on the hadoop file system. If you are comfortable with your data backup, delete the old datastore

```
sudo rm -rf /var/lib/cassandra/*
```

Setting up the configurations

Go to the $CASSANDRA_HOME/cassandra.yaml file. You can update the following values to set up a multi node system

```
# Find these values in the existing file
cluster_name: 'MustBeNamed'

# Some fields may be added
seed_provider:
  - class_name: org.apache.cassandra.locator.
      SimpleSeedProvider
    parameters:
         - seeds: "this_server_ip,server_ip_2
             ,..."

# Firewall Addresses
rpc_address: this_server_ip
listen_address: this_server_ip

# Snitches
endpoint_snitch: GossipingPropertyFileSnitch
```

1. cluster_name: The name of your cluster. In order for other nodes to join it must be named the **same thing** on each server.

2. seed_provider: This property is how cassandra knows about which nodes are directly connected to which. It will assume that the nodes are connected in a ring where the first ip is connected to the second, the second to the third and so on and the last is connected to the first.

3. rpc_address, listen_address: These are the two IPs that cassandra will do its communication on. These are usually localhost and default to two ports.

4. endpoint_snitch: A snitch in computing shares metadata about the cluster (the health of each node, the free space etc) by occasionally telling its neighbors through a fingertable. The GossipingPropertyFileSnitch is more fault tolerant due to randomness while the SimpleSnitch is ideally used for non-production testing.

For those on Linux systems, IPTables are the usualy firewall the kernel has up to block incoming traffic, we will need to circumvent that.

```
# Redo the following command for each
# machine connected to a single machine.

$ iptable -A INPUT \ # We are going to be
    receiving data
  -p tcp -s connected_machine_ip \ # Replace
      the IP as goes
  -m multiport --dports 7000,9042 \ # Default
      port
  -m state --state NEW,ESTABLISHED -j ACCEPT
$ service iptables-persistent restart
```

To see if it worked try the following command.

```
$ $CASSANDRA_HOME/bin/cqlsh ANY_SERVER 9042
```

INTERACTING WITH CASSANDRA

There are two usual ways of interacting with cassandra, either the cqlsh executable or drivers for a specific programming language.

The CQLSH command line is mainly for maintenance or executing queries during setup/backup/destruction. This is because it is easy to use, easy to script, and easy to check for errors. But, there are drawbacks to using it. One drawback is CQLSH will *always* connect the instance that you supply it with. Cassandra thrives on the ability for any client to connect to any node, reducing the point of failures and helps load balance the users among the cluster which decreases the probability of failures.

This is where community drivers come in. One can find drivers for many popular programming languages. One such driver for python (the highly popular datastax cassandra-python) is showcased in the code snippet below.

```
from cassandra.cluster import Cluster

cluster = Cluster(
    ['machine_ip_1', ...],
    port=...,
    )
try:
  sess = cluster.connect()
  sess.set_keyspace('...')
  rows = sess.execute('SELECT␣name␣FROM␣users␣\
␣␣␣␣␣␣␣␣␣␣␣␣␣WHERE␣email=%s', ['email1@gmail.com'])
  for user_row in rows:
      print(user_row.name)
except:
  pass
finally:
  # No Need to close!
```

The convention is that there is one cluster object that connects once at the start of the program, then there is one session per keyspace that executes queries.

The cluster object represents an object that can connect to any node in the cluster. The session object manages the session, there is no need to open or close it. And mostly, you can execute queries very easily in cassandra SQL.

OVERVIEW OF DATA MODELING

If you are familiar with more traditional Relational Database Management Systems, you are familiar with the concept of tables and joins. Imagine Cassandra as a traditional RDBMS without the ability to do JOINs and where every statement ought to have a LIMIT BY for performance (there are ways around it).

If you are otherwise not familiar with traditional data stores, cassandra stores its data in tables, much like a spreadsheet. The tables live in entities called keyspaces. Keyspaces are ways for cassandra to keep track of resillience levels in tables in cqlsh syntax here is how you create one.

```
> CREATE KEYSPACE "NAME" WITH replication =
      {
        'class': 'SimpleStrategy', # The
            default
        'replication_factor' : 3 # For example
      };
```

To start using a keyspace, do the following

```
> USE KEYSPACE "KEYSPACE"
```

To create a table it is pretty similar as well. One of the technical notes to data modeling is that you want to design your table with the idea that as much information as possible should be in one row of the table. Cassandra likes wide column stores because it cannot do JOINs with other tables. This means that it preferes to have denormalized data, repeated data and so on versus a single representation for everything. There are some basic crud operations below. This should look similar to SQL for those of you familiar.

```
> CREATE TABLE sample (
    sample_identifier text,
    sample_time timestamp,
    sample_size int,
    sample_data blob,
    PRIMARY_KEY (sample_identifier,
      sample_time) # You must have a key!
  )

> INSERT INTO sample
  VALUES ("1", toTimestamp(now()), 10, {"data"})
> DELETE FROM sample
  WHERE sample_size = 10
```

Important Cassandra Distinctions

Cassandra is a wide column store database that optimizes writes over reads. This means that writes happen very fast while reads take a while to get the value. Reads are done on a per row basis – the more rows you want to return the longer you are going to wait for your query.

But, the wide column store makes an important distinction. If we were to let's say group our data into static categories then our queries can be more efficient because it returns less rows. One example is storing a day or an hour's worth of stock ticker data in a *single row* and adding columns to the wide column whenever need be. This makes cassandra efficient because now we've decreased the number of rows by an amazing factor meaning our queries will return faster. If we don't need the extra data we can simply discard it, most of the time spent will be on gathering the data between servers. If it becomes the case that the bottleneck is transmitting the data back because of rows too wide, one can change the composition of the columns to have less data until a threshold is reached.

DATA MODELING EXAMPLE

Let's take a specific example and see it through in CQL and python. Let's say we wanted to

```
CREATE TABLE stock (
  company_name text,
  stock_time date,
  ticker set<tuple<int, float>>,
  PRIMARY KEY(company_name, stock_time)
);
```

Ideally you do not want to use the list data type because of failures that can occur when appending, instead we will store each of the ticker data as a set of tuples that represent (seconds from midnight, price). This example makes use of what is called a composite key, meaning that we require that the company_name and the stock_time be sent to the same node(s) every time. Now let's take a look at the two operations for inserting data. If we were starting the trading day new, then we could insert the following.

```
INSERT INTO stock VALUES ('AAA', dateof(now())
    , {});
```

Every time we get something new for our ticker

```
UPDATE stock SET ticker = ticker + { (TIME,
    PRICE) }
  WHERE company_name = 'AAA' AND date=dateof(
      now());
```

Since Cassandra is write optimized, this update will go through fast and reliably. Since our sets don't need to have order (we imposed that constraint through our set) we have additional reliability and speed.

If we wish to query our data we can grab the last few dates like the following

```
SELECT * FROM stock
WHERE stock_time >= '2017-01-01 00:00:00+0200'
  AND stock_time <= '2017-08-13 23:59:00+0200'
  AND company_name = 'AAA';
```

In python, it is *exactly* the same except instead of hard coding the values, you would use prepared statements. We don't have to worry about the downsides of prepared statements here because Cassandra doesn't normally cache popular query results do to it's write heavy nature.

```
stock_name = argv[1]
current_day = now()
while is_current_day(current_day):
  data = get_stockdata(stock_name)
  rows = session.execute("UPDATE stock ticker
      = ticker + { (\%s, \%s) } ...",
  ([data.time, data.price, stock_name,
      current_day]))
```

In the python case, this type of transition is most likely going to get CPU bound so running multiple scripts with different stock tickers is recommended.

TUNING PARAMS

There are many basic ways to tune Cassandra.

First, During your queries, make sure to set the appropriate consistency level. The consistency level tells the query to return succeeded or failed based on the number of nodes that get written to. The other nodes may eventually be replicated, but for your queries to return, this is the standard way. The most popular consistency levels are as follows.

1. ONE/TWO/THREE, this means the read/write only has to go to one/two/three node(s) before returning

2. QUORUM, this means the read/write has to go to half the nodes plus one. This is because for workloads with equal reads and writes, having a quorum is guaranteeing consistency.

3. ALL, this means the read/write has to go to all the nodes in the cluster (or almost all) before returning. This is usefull to pair with a ONE consistency level in the case of optimize workloads (more in the next section).

Workloads - Sessions

read heavy/write light

For read heavy/write light workloads, set the *read session* consistency to ONE with he following command in one session.

```
> CONSISTENCY ONE
```

And the write consistency to ALL with the following command

```
> CONSISTENCY ALL
```

write heavy/read light

For write heavy/read light workloads, set the consistency vice versa as above.

Balanced Workloads

For the case where you need a balanced workload, set both consistencies in the session to QUORUM, so you get consistent results in both reads and writes with high probability.

USUAL DO NOTS

Although cassandra gives you a lot of tunability parameters there are practices that should be avoided at all cost because it severly impacts the performance or the reliability of the database.

1. Do not put cassandra's data files on network attached storage

2. Try not to host cassandra's data files directly on hadoop either, cassandra has reliability built into it.

3. Try not to store your data on multiple separate tables to join later. Cassandra is part of the wide column store family meaning that

REFERENCES

https://www.digitalocean.com/community/tutorials/how-to-run-a-multi-node-cluster-database-with-cassandra-on-ubuntu-14-04
https://docs.datastax.com/en/cassandra/2.1/cassandra/dml/dml$_{config}$