Cassandra

How to set-up a Cassandra Cluster:

To execute the examples of this tutorial, you'll need a running Cassandra cluster. You can get this up and running quickly by using Docker.

Cassandra is built for scale, and some features only work on a multi-node Cassandra cluster, so let’s start one locally.

For Linux and Mac, run the following commands:

⚠ **The commands are meant to be understood all on a single line of code**

RUN ONLY IN UNIX, MacOS PLATFORMS

```
# Run the first node and keep it in background up and running
docker run --name cassandra-1 -p 9042:9042 -d cassandra:3.7
INSTANCE1=$(docker inspect --format="{{ .NetworkSettings.IPAddress }}"
cassandra-1)
echo "Instance 1: ${INSTANCE1}"

# Run the second node
docker run --name cassandra-2 -p 9043:9042 -d -e
CASSANDRA_SEEDS=$INSTANCE1 cassandra:3.7
INSTANCE2=$(docker inspect --format="{{ .NetworkSettings.IPAddress }}"
cassandra-2)
echo "Instance 2: ${INSTANCE2}"

echo "Wait 60s until the second node joins the cluster"
sleep 60

# Run the third node
docker run --name cassandra-3 -p 9044:9042 -d -e
CASSANDRA_SEEDS=$INSTANCE1,$INSTANCE2 cassandra:3.7
INSTANCE3=$(docker inspect --format="{{ .NetworkSettings.IPAddress }}"
cassandra-3)
echo "Instance 3: ${INSTANCE3}"
```

The startup process can take few minutes...

RUN ONLY IN WINDOWS PLATFORMS

- It is strongly recommended to use PowerShell -

```
# Run the first node and keep it in background up and running
docker run --name cassandra-1 -p 9042:9042 -d cassandra:3.7
$INSTANCE1=$(docker inspect --format="{{ .NetworkSettings.IPAddress }}"
cassandra-1)
echo "Instance 1: ${INSTANCE1}"

# Run the second node
docker run --name cassandra-2 -p 9043:9042 -d -e
CASSANDRA_SEEDS=$INSTANCE1 cassandra:3.7
$INSTANCE2=$(docker inspect --format="{{ .NetworkSettings.IPAddress }}"
cassandra-2)
echo "Instance 2: ${INSTANCE2}"

echo "Wait 60s until the second node joins the cluster"
sleep 60

# Run the third node
docker run --name cassandra-3 -p 9044:9042 -d -e
CASSANDRA_SEEDS=$INSTANCE1,$INSTANCE2 cassandra:3.7
$INSTANCE3=$(docker inspect --format="{{ .NetworkSettings.IPAddress }}"
cassandra-3)
echo "Instance 3: ${INSTANCE3}"
```

The startup process can take few minutes...

Verify:

You can verify if everything is done and ready by executing a Cassandra utility tool called nodetool via docker exec on a node:

```
$ docker exec cassandra-3 nodetool status
```

```
Datacenter: datacenter1
=======================
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address     Load       Tokens       Owns (effective)  Host ID
Rack
UN  172.17.0.3  112.69 KiB  256          68.7%             bb5ef231-0dd2-4762-
a447-806a45f710ac  rack1
UN  172.17.0.2  107.96 KiB  256          68.3%             d7392374-8daa-4292-
b724-cb790b0ee6ad  rack1
UN  172.17.0.4  93.93 KiB  256          63.0%             386d094f-5483-4945-
a1a7-2bb3975d6167  rack1
```

NOTE:

"UN" means Up and Normal. Here, all 3 nodes are running and healthy.

Connect and use CQLSH:

In this tutorial we will send lots of queries to Cassandra. ***Is recommended starting a new shell and connecting to one node using*** CQLSH***.*** Here's how to start a cqlsh shell in Docker:

CQLSH is a command-line shell for interacting with Apache Cassandra databases using the Cassandra Query Language (CQL). It allows users to execute CQL statements and perform various database operations such as creating and modifying keyspaces, tables, and indexes. Cqlsh also provides features such as auto-completion, syntax highlighting, and paging of query results. It is included as part of the Cassandra distribution and is typically installed on the same machine as the Cassandra cluster.

```
$ docker exec -it cassandra-1 cqlsh

Connected to Test Cluster at 127.0.0.1:9042.
[cqlsh 5.0.1 | Cassandra 3.7 | CQL spec 3.4.2 | Native protocol v4]
Use HELP for help.
cqlsh>
```

The first query we will execute is:

```
cqlsh> DESCRIBE keyspaces;

system_traces  system_schema  system_auth  system  system_distributed
```

The response shows all the existing keyspaces. Keyspaces group tables and are similar to a database in a traditional relational database system. In other systems, groups of certain items are also known as namespaces.

Before you begin creating tables and inserting data, first create a keyspace in your local datacenter, which should replicate data 3 times:

```
cqlsh> CREATE KEYSPACE learn_cassandra
  WITH REPLICATION = {
   'class' : 'NetworkTopologyStrategy',
   'datacenter1' : 3
  };
```

A keyspace with a replication factor of 3 using the NetworkTopologyStrategy was created. The strategy defines how data is replicated in different datacenters. This is the recommended strategy for all user created keyspaces.

Why should you start with 3 nodes? It's recommended to have at least 3 nodes or more. One reason is, in case you need strong consistency, you need to get confirmed data from at least 2 nodes. Or if 1 node goes down, your cluster would still be available because the 2 remaining nodes are up and running.

You don't need to fully understand this yet. After reading through the rest of this tutorial, things should be more clear.

Now, all the nodes are up and healthy. You have a 3-node Cassandra setup listening on ports 9042, 9043, and 9044 for client requests. This is a realistic setup for a small cluster.

How data are organized:

The concept of primary keys is more complex in Cassandra than in traditional databases like MySQL. In Cassandra, the primary key consists of 2 parts:

1. a mandatory **partition key** (in brackets)
2. an optional set of **clustering columns**.

Consider the following table:

```
Table Users | Legend: p - Partition-Key, c - Clustering Column

country (p) | user_email (c)  | first_name | last_name | age
----------------------------------------------------------------
US          | john@email.com  | John       | Wick      | 55
UK          | peter@email.com | Peter      | Clark     | 65
UK          | bob@email.com   | Bob        | Sandler   | 23
UK          | alice@email.com | Alice      | Brown     | 26
```

Together, the columns: "*country*" and "*user_email*" make up the primary key, infact in the table creation we have to put both:

```
cqlsh>
CREATE TABLE learn_cassandra.users_by_country (
    country text,
    user_email text,
    first_name text,
    last_name text,
    age smallint,
    PRIMARY KEY ((country), user_email)
);
```

where "*country*" is the PARTITION KEY and "*user_email*" the CLUSTERING COLUMN

NOTE:

The partition key is a subset of the columns that is used to distribute data across the nodes in a Cassandra cluster. Data with the same partition key value is stored on the same node or set of nodes. Meanwhile *"clustering column"* is used to define the order in which data is stored within a partition. A table may have multiple clustering columns, and each column can be sorted in either ascending or descending order. Queries that involve the clustering columns are efficient as long as they include the partition key value in the WHERE clause, since this allows Cassandra to locate the correct node to query.

Little example: Let's say you have a table that stores data about people, and the partition key is "country". This means that all data for people in the same country will be stored together on the same node or set of nodes. However, within each country, you want to sort the data by "age" email, or other…in descending order.

To achieve this, you can use "country" as the partition key and "age" as the clustering column. This will ensure that data for people in the same country is stored together, and within each country, the data is sorted by age.

Now we have to fill the table with some data:

```
cqlsh>
INSERT INTO learn_cassandra.users_by_country
(country,user_email,first_name,last_name,age)
  VALUES('US', 'john@email.com', 'John','Wick',55);

INSERT INTO learn_cassandra.users_by_country
(country,user_email,first_name,last_name,age)
  VALUES('UK', 'peter@email.com', 'Peter','Clark',65);

INSERT INTO learn_cassandra.users_by_country
(country,user_email,first_name,last_name,age)
  VALUES('UK', 'bob@email.com', 'Bob','Sandler',23);

INSERT INTO learn_cassandra.users_by_country
(country,user_email,first_name,last_name,age)
  VALUES('UK', 'alice@email.com', 'Alice','Brown',26);
```

You can remove data with this syntax:

```
DELETE FROM learn_cassandra.users_by_country WHERE country = 'UK' AND
user_email = 'alice@email.com';
```

Or if you don’t want remove all the row you can specify the field:

```
DELETE age FROM learn_cassandra.users_by_country WHERE country = 'UK' AND
user_email = 'bob@email.com';
```

It's important to note that when you delete a row in Cassandra, the data is not immediately removed from disk. Instead, a tombstone is created for the deleted data, and the data is removed during a later compaction process. This is because Cassandra is optimized for high write throughput, and immediate data removal could lead to performance issues.

Little query to visualize data:

```
cqlsh>

  SELECT * FROM learn_cassandra.users_by_country WHERE country='UK';
```

Consistency:

if you want strong consistency but low performance:

```
cqlsh>

   CONSISTENCY ALL;

   SELECT * FROM learn_cassandra.users_by_country WHERE country='UK';
```

If you don't need to be strongly consistent, you can reduce the consistency, gaining performance:

```
cqlsh>

   CONSISTENCY ONE;

   SELECT * FROM learn_cassandra.users_by_country WHERE country='UK';
```

Compaction:

For every write operation, data is written to disk to provide durability. This means that if something goes wrong, like a power outage, data is not lost.

The foundation for storing data are the so-called SSTables. SSTables are immutable data files Cassandra uses to persist data on disk.

You can set various strategies for a table that define how data should be merged and compacted. These strategies affect read and write performance:

·   SizeTieredCompactionStrategy is the default, and is especially performant if you have more writes than reads,

·   LeveledCompactionStrategy optimizes for reads over writes. This optimization can be costly and needs to be tried out in production carefully

·   TimeWindowCompactionStrategy is for Time-series data

The strategies define when and how compaction is executed. Compaction means rearranging data on disk to remove old data and keep performance as good as possible when more data needs to be stored. Compaction is necessary to reduce the number of SSTables and to eliminate duplicate data, making data access faster and more efficient. During compaction, Cassandra reads data from the old SSTables, merges it with the data in the new SSTable, and writes the merged data to a new SSTable.

-By default, tables use the SizeTieredCompactionStrategy-

In every case is recomended to define the compaction strategy explicitly during table creation of your new table:

```
cqlsh>

CREATE TABLE learn_cassandra.users_by_country_with_leveled_compaction (

    country text,

    user_email text,

    first_name text,

    last_name text,

    age smallint,

    PRIMARY KEY ((country), user_email)

) WITH

  compaction = { 'class' :  'LeveledCompactionStrategy'  };
```

Let’s check the result:

```
cqlsh>
```

```
   DESCRIBE TABLE learn_cassandra.users_by_country_with_leveled_compaction;

CREATE TABLE learn_cassandra.users_by_country_with_leveled_compaction (

    country text,

    user_email text,

    age smallint,

    first_name text,

    last_name text,

    PRIMARY KEY (country, user_email)

) WITH CLUSTERING ORDER BY (user_email ASC)

    AND bloom_filter_fp_chance = 0.1

    AND caching = {'keys': 'ALL', 'rows_per_partition': 'NONE'}

    AND comment = ''

    AND compaction = {'class':
'org.apache.cassandra.db.compaction.LeveledCompactionStrategy'}

    AND compression = {'chunk_length_in_kb': '64', 'class':
'org.apache.cassandra.io.compress.LZ4Compressor'}

    AND crc_check_chance = 1.0

    AND dclocal_read_repair_chance = 0.1

    AND default_time_to_live = 0

    AND gc_grace_seconds = 864000

    AND max_index_interval = 2048

    AND memtable_flush_period_in_ms = 0

    AND min_index_interval = 128
    AND read_repair_chance = 0.0

    AND speculative_retry = '99PERCENTILE';
```