JULY 15, 2021 / #CASSANDRA

The Apache Cassandra Beginner Tutorial

Sebastian Sigl

There are lots of data-storage options available today. You have to choose between managed or unmanaged, relational or NoSQL, write- or read-optimized, proprietary or open-source — and it doesn't end there.

Once you begin your search, you will end up in the universe that is database marketing. All of the vendors will tell you why their database is fantastic.

Unfortunately, it's difficult to find out when not to use a specific database, because this is not an attractive selling point.

If you know what questions to ask, you will eventually understand all the essential properties of a given system. In the end, your choice will depend on your expertise and your requirements.

In this tutorial I will introduce you to Apache Cassandra, a distributed, horizontally scalable, open-source database. Or as Cassandra users

I will share the essential gotchas and provide references to documentation. I'll also provide insights based on my experience of running Cassandra on a large scale at work, with executable examples wherever possible.

Here's an overview of everything you'll learn:

Along the way, you will learn to ask fundamental questions that will help you to chose a database that suits your needs. You'll also learn about other popular databases like Spanner, Cockroach, or FaunaDB, and how they can serve different use-cases.

Table of Contents

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.

> **Required Docker settings**
>
> Your device should have a minimum of 8GB of memory and at least 8GB of free disk space. Your Docker settings should be updated to be able to use at least 6GB of memory, or better, 8GB.
>
> To apply these suggestions, open your Docker preferences, go to Resources, and increase your memory threshold.

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:

```
# 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 }}" ca
echo "Instance 1: ${INSTANCE1}"

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

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

For Windows, run the following commands in 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 }}" c
echo "Instance 1: ${INSTANCE1}"

# Run the second node
docker run --name cassandra-2 -p 9043:9042 -d -e CASSANDRA_SEEDS=$INSTANC
$INSTANCE2=$(docker inspect --format="{{ .NetworkSettings.IPAddress }}" c
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=$INSTANC
$INSTANCE3=$(docker inspect --format="{{ .NetworkSettings.IPAddress }}" c
```

The startup process can take a few minutes.

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
=======================
```

```
UN  172.17.0.2  107.96 KiB  256          68.3%             d7392374-8daa-
UN  172.17.0.4  93.93 KiB  256          63.0%             386d094f-5483-4
```

UN means **U**p and **N**ormal. Here, all 3 nodes are running and healthy.

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

```
$ 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>
```

And to execute your first query:

```
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.

```
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.

In production, the instances would run on different machines to maximize performance.

In this tutorial, you will create tables with different settings for a to-do list application. If you want to get your hands dirty straight away, you can jump directly to the next `cqlsh` example.

Cassandra Architecture

Cassandra is a decentralized multi-node database that physically spans separate locations and uses replication and partitioning to infinitely scale reads and writes.

Decentralization

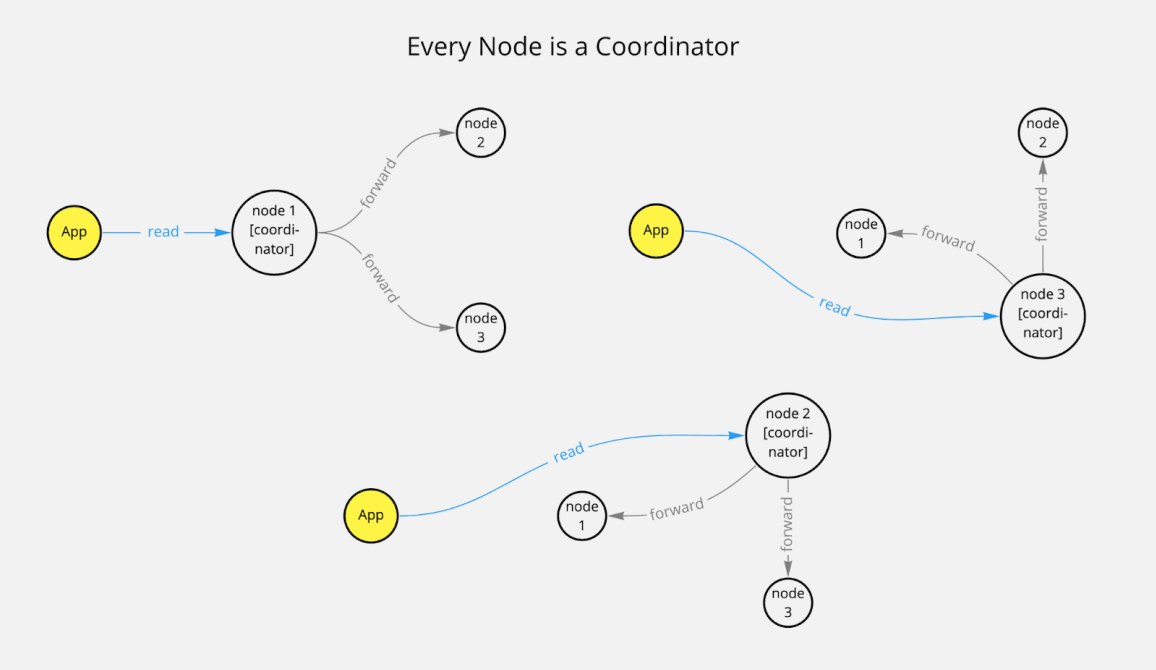
Cassandra is decentralized because no node is superior to other nodes, and every node acts in different roles as needed without any central controller. We'll get into examples of decentralization a bit later in this section.

Cassandra's decentralized property is what allows it to handle situations easily in case one node becomes unavailable or a new node is added.

Every Node Is a Coordinator

Data is replicated to different nodes. If certain data is requested, a request can be processed from any node.

This initial request receiver becomes the coordinator node for that request. If other nodes need to be checked to ensure consistency then the coordinator requests the required data from replica nodes.



Every node can be a coordinator

The coordinator is responsible for many things, such as request batching, repairing data, or retries for reads and writes.

Data Partitioning

> "[Partitioning] is a method of splitting and storing a single logical dataset in multiple databases. By distributing the data among multiple machines, a cluster of database systems can store larger datasets and handle additional requests.
>
> "How Sharding Works by Jeeyoung Kim

As with many other databases, you store data in Cassandra in a

primary key is mandatory and ensures data is uniquely identifiable by one or multiple columns.

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:

- a mandatory partition key and
- an optional set of clustering columns.

You will learn more about the partition key and clustering columns in the data modeling section.

For now, let's focus on the partition key and its impact on data partitioning.

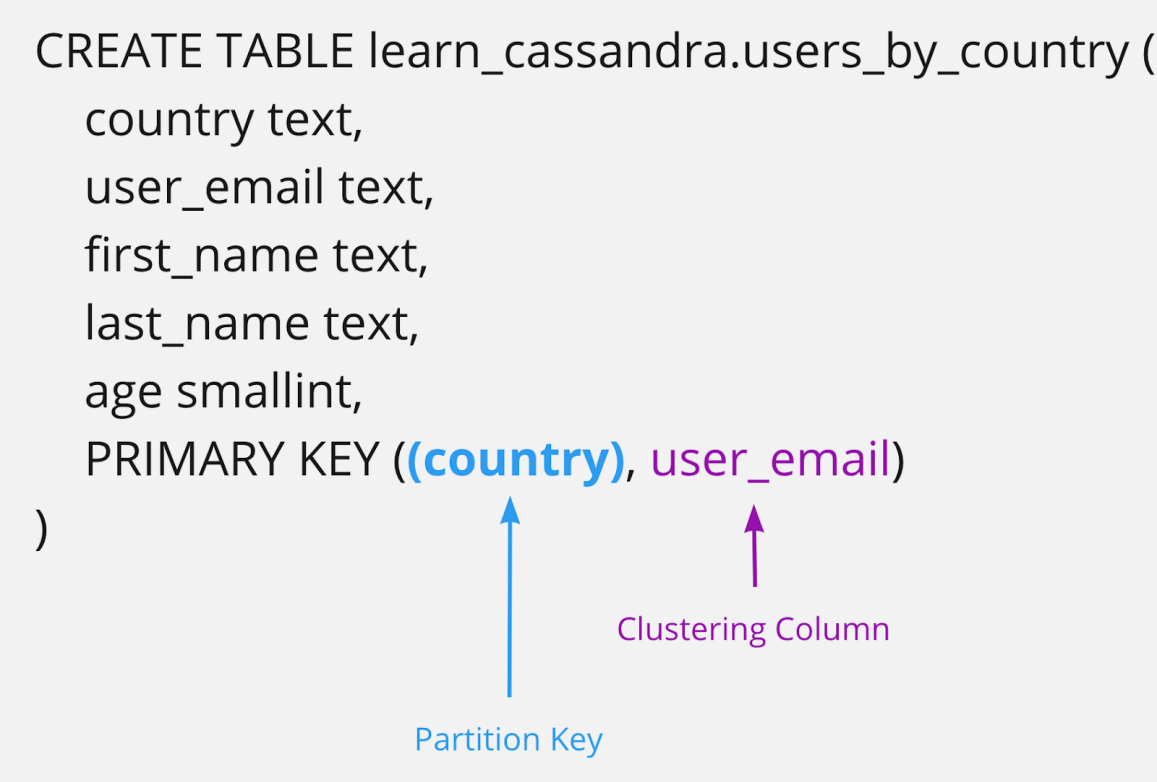
Consider the following table:

```
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 `user_email` and `country` make up the primary key.

```
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)
);
```

The first group of the primary key defines the partition key. All other elements of the primary key are clustering columns:



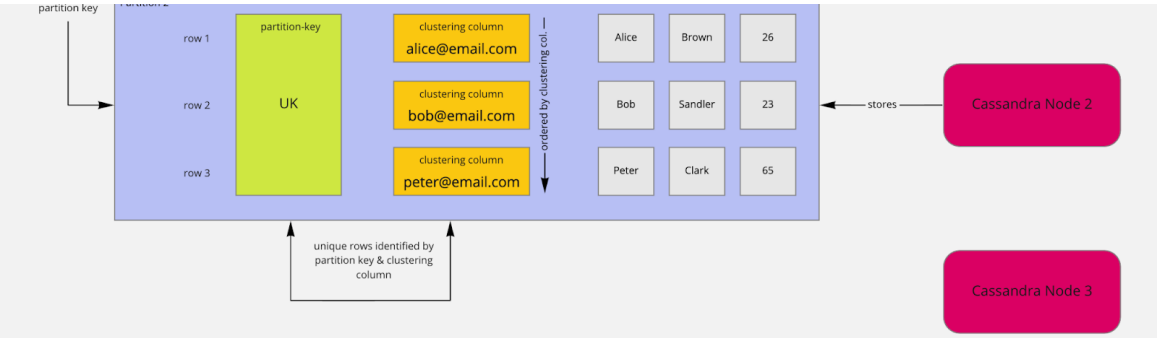
```
cqlsh>
INSERT INTO learn_cassandra.users_by_country (country,user_email,first_na
  VALUES('US', 'john@email.com', 'John','Wick',55);

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

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

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

If you're used to designing traditional relational database tables like it's taught in school or university, you might be surprised. Why would you use `country` as an essential part of the primary key?

This example will make sense after you understand the basics of partitioning in Cassandra.

Partitioning is the foundation for scalability, and it is based on the partition key. In this example, partitions are created based on `country`. All rows with the `country` `US` are placed in a partition. All other rows with the country `UK` will be stored in another partition.

In the context of partitioning, the words partition and shard can be used interchangeably.



Partitions are created and filled based on partition key values. They are used to distribute data to different nodes. By distributing data to other nodes, you get scalability. You read and write data to and from different nodes by their partition key.

The distribution of data is a crucial point to understand when designing applications that store data based on partitions. It may take a while to get fully accustomed to this concept, especially if you are used to relational databases.

Instead, think about how you read and write data and how partitioning should be done to scale horizontally.

> **What does horizontal scaling mean?**
>
> Horizontal scaling means you can increase throughput by adding more nodes. If your data is distributed to more servers, then more CPU, memory, and network capacity is available.

You might ask, then why do you even need `email` in the primary key?

The answer is that the primary key defines what columns are used to identify rows. You need to add all columns that are required to

essential when reading the data. The previously defined schema is designed to be queried by `country` because `country` is the partition key.

A query that selects rows by `country` performs well:

```
cqlsh>
  SELECT * FROM learn_cassandra.users_by_country WHERE country='US';
```

In your `cqlsh` shell, you will send a request only to a single Cassandra node by default. This is called a consistency level of one, which enables excellent performance and scalability.

If you access Cassandra differently, the default consistency level might not be one.

> **What does consistency level of one mean?**
>
> A consistency level of one means that only a single node is asked to return the data. With this approach, you will lose strong consistency guarantees and instead experience eventual consistency.
>
> We'll dive deeper into consistency levels later on.

Let's create another table. This one has a partition defined only by the `user_email` column:

```
    user_email text,
    country text,
    first_name text,
    last_name text,
    age smallint,
    PRIMARY KEY (user_email)
);
```

Now let's fill this table with some records:

```
cqlsh>
INSERT INTO learn_cassandra.users_by_email (user_email, country,first_nam
  VALUES('john@email.com', 'US', 'John','Wick',55);

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

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

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

This time, each row is put in its own partition.

This is not bad, per se. If you want to optimize for getting data by `email` only, it's a good idea:

```
cqlsh>
  SELECT * FROM learn_cassandra.users_by_email WHERE user_email='alice@em
```

If you set up your table with a partition key for `user_email` and want to get all users by `age`, you would need to get the data from all partitions because the partitions were created by `user_email`.

Talking to all nodes is expensive and can cause performance issues on a large cluster.

Cassandra tries to avoid harmful queries. If you want to filter by a column that is not a partition key, you need to tell Cassandra explicitly that you want to filter by a non-partition key column:

```
cqlsh>
SELECT * FROM learn_cassandra.users_by_email WHERE age=26 ALLOW FILTERINC
```

Without `ALLOW FILTERING`, the query would not be executed to

With conditions that don't use the partition key, are costly and should be avoided to prevent performance bottlenecks.

But how do you get all the rows from the table in a scalable way?

If you can, partition by a value like `country`. If you know all the countries, you can then iterate over all available countries, send a query for each one, and collect the results in your application.

In terms of scalability, it's worse to just select all rows, because when you use a table partitioned by `user_email`, all the data is collected in 1 request in a single coordinator.

This is OK as long as you have no performance issues.

By comparison, sending multiple requests by `country` distributes the effort to different coordinator nodes, which scales a lot better.

If you still need access to all of the data, there is an excellent integration between Spark and Cassandra that allows efficient reads and writes for massive datasets. The Spark connector for Cassandra groups your data by partition key and can execute queries very efficiently.

Replication

Scalability using partitioning alone is limited.

Consider a lot of write requests arriving for a single partition. All requests would be sent to a single node with technical limitations such

to different nodes, so called replicas, you can serve more data simultaneously from other nodes to improve latency and throughput. It also enables your cluster to perform reads and writes in case a replica is not available.

In Cassandra, you need to define a replication factor for every keyspace. At the beginning of our example, you created a keyspace with a replication factor of 3 for our default datacenter:

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

A replication factor of one means there's only one copy of each row in the cluster. If the node containing the row goes down, the row cannot be retrieved.

A replication factor of two means two copies of each row, where each copy is on a different node. All replicas are equally important; there is no primary or master replica.

As a general rule, the replication factor should not exceed the number of nodes in the cluster. However, you can increase the replication factor and then add the desired number of nodes later.

Usually, it's recommended to use a replication factor of 3 for production use cases. It makes sure your data is very unlikely to get

you can ask what information is held by the majority).

In your local cluster setup, the majority means 2 out of 3 replicas. This allows us to use some powerful query options that you will see in the next section.

Consistency Level

Now that you know about partitioning and replication, you are ready to think about consistency levels. Cassandra has a truly outstanding feature called tunable consistency.

You can define the consistency level of your read and write queries. You can check the Cassandra docs for all available settings.

Let's focus on the most popular settings and try to understand when to choose each consistency level.

Let's assume you have 3 replicas defined.

The first question you need to answer is, do you need strong consistency?

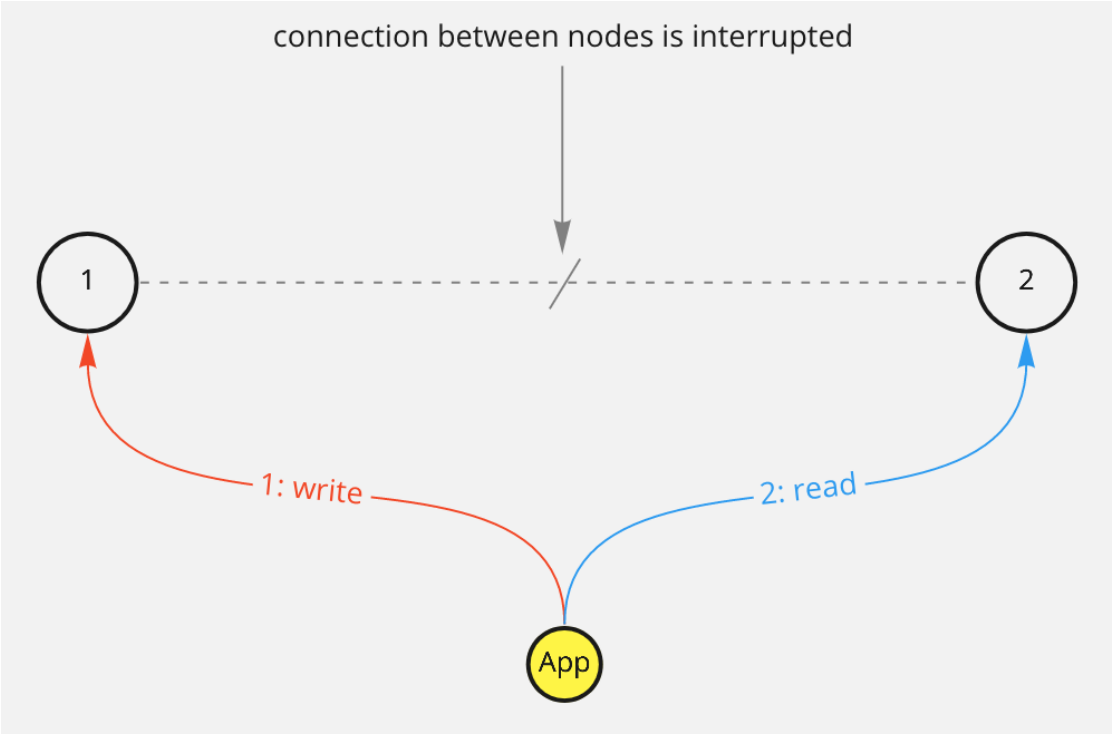
> **What does strong consistency mean?**
>
> In contrast to eventual consistency, strong consistency means only one state of your data can be observed at any time in any location.
>
> For example, when consistency is critical, like in a banking domain, you want to be sure that everything is correct. You

It all comes down to the CAP theorem. You can not be available and consistent at the same time in case of connection issues between nodes of your cluster.

Let's think through the following example:

You want to write a single value to a table. The data is replicated in 2 nodes, and the connection between the nodes is interrupted. First, a write-request is sent to node 1. Then, data is read from node 2.

How do you manage this situation?



1. Should you disallow writes to all nodes to ensure consistency? This means availability would be sacrificed to ensure

consistency and correctness

what node you read from, the answer will be different, which means sacrificing consistency over availability.

You can simplify the problem to make crucial decisions for your application: Do you want consistency or availability?

Another factor is latency. By talking to more nodes to ensure consistency, you need to wait longer to receive all nodes' responses.

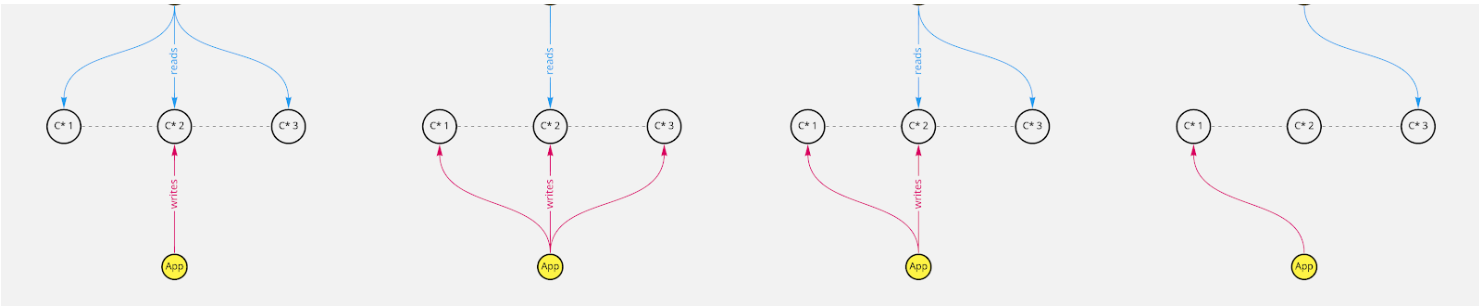
Tune for Consistency by Setting up a Strong Consistency Application

There is a very important formula that if true guarantees strong consistency:

```
[read-consistency-level] + [write-consistency-level] > [replicatio
```

> **What does consistency level mean?**
>
> Consistency level means how many nodes need to acknowledge a read or a write query.

You can shift read and write consistency levels to your favor if you want to keep strong consistency. Or you even give up strong consistency for better performance, which is also called eventual consistency:



For a read-heavy system, it's recommended to keep read consistency low because reads happen more often than writes. Let's say you have a replication factor of 3. The formula would look like this:

```
1 + [write-consistency-level] > 3
```

Therefore, the write consistency has to be set to 3 to have a strongly consistent system.

For a write-heavy system, you can do the same. Set the write consistency level to 1 and the read consistency level to 3.

You either check every node for a read to ensure all nodes have received the last updated state, or, for a write, you ensure that all nodes have written the update to their local storage. Both will make sure that data for reading and writing is correct.

This decision needs to be reflected in all the applications that access your Cassandra data because, on a query level, you need to set the required consistency level

consistency level of ALL or THREE:

```
cqlsh>
   CONSISTENCY ALL;
   SELECT * FROM learn_cassandra.users_by_country WHERE country='US';
```

If just one of your applications violates the required consistency strategy, you are quickly at the risk of either dropping consistency or pressuring the cluster more than required.

Tune for Performance by Using Eventual Consistency

If you don't need to be strongly consistent, you can reduce the consistency level for queries to 1 to gain performance:

```
cqlsh>
   CONSISTENCY ONE;
   SELECT * FROM learn_cassandra.users_by_country WHERE country='US';
```

Eventually, the data will be spread to all replicas and this will ensure *eventual* consistency. How fast data will be made consistent depends on different mechanics that sync data between nodes.

Various features can be tuned in Cassandra, like read-repairs and external processes that repair data continuously.

Optimize Data Storage for Reading

Writing data means simply appending something to a so-called commit-log.

Commit-logs are append-only logs of all mutations local to a Cassandra node and reduce the required I/O to a minimum.

Reading is more expensive, because it might require checking different disk locations until all the query data is eventually found.

But this does not mean Cassandra is terrible at reading. Instead, Cassandra's storage engine can be tuned for reading performance or writing performance.

Understanding 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

- `TimeWindowCompactionStrategy` is for Time-series data

By default, tables use the `SizeTieredCompactionStrategy`:

```
cqlsh>
   DESCRIBE TABLE learn_cassandra.users_by_country;

CREATE TABLE learn_cassandra.users_by_country (
    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 = 0.01
    AND caching = {'keys': 'ALL', 'rows_per_partition': 'NONE'}
    AND comment = ''
    AND compaction = {'class': 'org.apache.cassandra.db.compaction.SizeTi
    AND compression = {'chunk_length_in_kb': '64', 'class': 'org.apache.c
    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';
```

Although you can alter the compaction strategy of an existing table, I would not suggest doing so, because all Cassandra nodes start this migration simultaneously. This will lead to significant performance issues in a production system.

```
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_compactic

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.Levele
    AND compression = {'chunk_length_in_kb': '64', 'class': 'org.apache.c
    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
```

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.

Check out the excellent DataStax documentation about compaction for details. There may even be better strategies in the future for the performance of your use-case.

Presorting Data on Cassandra Nodes

A table always requires a primary key. A primary key consists of 2 parts:

- At least 1 column(s) as partition key and
- Zero or more clustering columns for nesting rows of the data.

All columns of the partition key together are used to identify partitions. All primary key columns, meaning partition key and clustering columns, identify a specific row within a partition.

In Cassandra, data is already sorted on disk. So if you want to avoid sorting data later, you can make sure sorting is applied as needed. This can be ensured on the table level and avoids having to sort data in the client applications that query Cassandra.

In our `users_by_country` table, you can define `age` as another

```
cqlsh>
CREATE TABLE learn_cassandra.users_by_country_sorted_by_age_asc (
    country text,
    user_email text,
    first_name text,
    last_name text,
    age smallint,
    PRIMARY KEY ((country), age, user_email)
) WITH CLUSTERING ORDER BY (age ASC);
```

Let's add the same data again:

```
cqlsh>
INSERT INTO learn_cassandra.users_by_country_sorted_by_age_asc (country,u
  VALUES('US','john@email.com', 'John','Wick',10);

INSERT INTO learn_cassandra.users_by_country_sorted_by_age_asc (country,u
  VALUES('UK', 'peter@email.com', 'Peter','Clark',30);

INSERT INTO learn_cassandra.users_by_country_sorted_by_age_asc (country,u
  VALUES('UK', 'bob@email.com', 'Bob','Sandler',20);

INSERT INTO learn_cassandra.users_by_country_sorted_by_age_asc (country,u
  VALUES('UK', 'alice@email.com', 'Alice','Brown',40);
```

And get the data by country:

```
cqlsh>
      SELECT * FROM learn_cassandra.users_by_country_sorted_by_age_asc WH

 country | age | user_email       | first_name | last_name
```

```
(3 rows)
```

In this example, the clustering columns are `age` and `user_email`. So the data is first sorted by age and then by `user_email`. At its core, Cassandra is still like a key-value store. Therefore, you can only query the table by:

- `country`
- `country` and `age`
- `country`, `age`, and `user_email`

But never by `country` and `user_email`.

After learning about partitioning, replication and consistency levels, let's head into data modeling and have more fun with the Cassandra cluster.

Data Modeling

You've already learned a lot about the fundamentals of Cassandra.

Let's put your knowledge into practice and design a to-do list application that receives many more reads than writes.

The best approach is to analyze some user stories you want to fulfill

with your table design:

Note: This is only about creating data. For now, you can delay some decisions because you want to focus on how data is read.

2. As a user, I want to list all my to-do elements in ascending order

First, you need to query by `user_email`. Create a table called `todos_by_user_email`.

You need 1 table that contains all the information of a to-do element of a user. Data should be partitioned by `user_email` for efficient read and writes by `user_email`.

Also, the oldest records should be displayed first, which means using the creation date as a clustering column. The `creation_date` also ensures uniqueness.:

```
cqlsh>
CREATE TABLE learn_cassandra.todo_by_user_email (
    user_email text,
    name text,
    creation_date timestamp,
    PRIMARY KEY ((user_email), creation_date)
) WITH CLUSTERING ORDER BY (creation_date DESC)
AND compaction = { 'class' :  'LeveledCompactionStrategy'  };
```

3. As a user, I want to share a to-do element with another user

To get all the to-dos shared with a user, you need to create a table called `todos_shared_by_target_user_email` to display all shared to-dos for the target user

But the user also wants to see the to-dos they shared with other users. This is another table, `todos_shared_by_source_user_email`.

Both tables have, according to the use-case, the required `user_email` as partition keys to allow efficient queries. Also, `creation_date` is added as a clustering column for sorting and uniqueness:

```
cqlsh>
CREATE TABLE learn_cassandra.todos_shared_by_target_user_email (
    target_user_email text,
    source_user_email text,
    creation_date timestamp,
    name text,
    PRIMARY KEY ((target_user_email), creation_date)
) WITH CLUSTERING ORDER BY (creation_date DESC)
AND compaction = { 'class' :  'LeveledCompactionStrategy'  };

CREATE TABLE learn_cassandra.todos_shared_by_source_user_email (
    target_user_email text,
    source_user_email text,
    creation_date timestamp,
    name text,
    PRIMARY KEY ((source_user_email), creation_date)
) WITH CLUSTERING ORDER BY (creation_date DESC)
AND compaction = { 'class' :  'LeveledCompactionStrategy'  };
```

This type of modeling is different than thinking about foreign keys and primary keys that you might know from traditional databases. In the beginning, it's all about defining tables and thinking about what values you want to filter and need to display.

You need to set a partition key to ensure the data is organised for efficient read and write operations. Also, you need to set clustering

Keep Data in Sync Using `BATCH` Statements

Due to the duplication, you need to take care to keep data consistent. In Cassandra, you can do that by using `BATCH` statements that give you an all-at-once guarantee, also called atomicity.

This might sound like a lot of work, and yes, it is a lot of work! If you have a table schema with many relationships, you will have more work compared to a normalized table schema.

> **What is a normalized table schema?**
>
> A normalized table schema is optimized to contain no duplications. Instead, data is referenced by ID and needs to be joined later.
>
> In Cassandra, you try to avoid normalized tables. It is not even possible to write a query that contains a join.

Batch statements are cheap on a single partition, but dangerous when you execute them on different partitions, because:

- Data mutations will not be applied at the same time to all partitions, with no isolation
- It is expensive for the coordinator node, because you have to talk to multiple nodes and prepare for a rollback if something goes wrong

- There is a batch query size limit of 50kb to avoid overloading

In general, batches are costly.

There are other ways to apply changes eventually. If you need to execute them very often, consider using async queries instead with a proper retry mechanism.

Depending on the way you access your Cassandra, the driver might already offer you retry capabilities.

Still, this approach requires thinking about what will happen if a query is never executed. If every query really needs to be executed eventually, how can you make sure that it does not get lost if your service goes down?

The topic itself needs much more time to explain, and might be the main topic of another Cassandra tutorial.

The key learning here is:

- Single partition batches are cheap and should be used
- Batches that include different partitions are expensive, and if there are a lot of reads/writes, this might be the reason why a Cassandra cluster is exhausted.

Let's create a `BATCH` statement that contains a to-do element that is shared with a user:

```
cqlsh>
```

```
  INSERT INTO learn_cassandra.todos_shared_by_source_user_email (target_u

APPLY BATCH;
```

Let's look into one of the tables:

```
cqlsh>
 SELECT * FROM learn_cassandra.todos_shared_by_target_user_email WHERE ta

 target_user_email | creation_date    | name    | source_user_email
-------------------+-----------------+--------+-------------------
bob@email.com | 2021-05-24 ...| My first todo entry |   alice@email.com
```

All the data exists and can be accessed in a performant way using all the defined tables.

Use Foreign Keys Instead of Duplicating Data in Cassandra

You might consider using foreign keys instead of duplicating data.

Traditionally, foreign keys are ID-references of an entity that are located in another table and in relational database. They guarantee that the referenced ID exists.

In Cassandra, this might feel good because you have less duplicated data. At this point, think again about why you use Cassandra. Usually

when used correctly.

Normalizing tables is against a lot of principles in Cassandra. You can reference data by ID, but keep in mind this means you need to join the data yourself. This also means reading and writing data to multiple partitions at once.

Cassandra is built for scale. If you start normalizing your schema to reduce duplication, then you sacrifice horizontal scalability.

If you still want to use foreign keys instead of data duplication, you might want to use another database. But, everything comes with trade-offs.

Instead of using Cassandra, you could use a database that sacrifices performance and availability, and gives more consistency guarantees. In cases like this, I can recommend Cloud Spanner or Cockroach DB for a scalable relational database.

Indexes in Cassandra

There are index-like features in Cassandra that can reduce the number of tables you need to maintain on your own. One feature is called secondary indexes.

I cannot recommend them because they only operate locally to a node.

Using a secondary index means talking to all nodes because the

coordinator doesn't know which nodes contain the data if you use

Materialized Views

Materialized views were designed with scalability in mind.

They make it easier to duplicate tables with different partition keys so you can query data by different column combinations. They also simplify the process of creating a new table and ensuring data integrity for mutations.

There is only one drawback — the source table's full primary key needs to be part of the materialized view's primary key, and optionally, one other column.

The columns that act as partition keys can be different.

Running a Cluster

Running a Cassandra cluster can be intense. It contains your business-critical data and is usually under heavy pressure.

I won't go into details because I am more a Cassandra user than an expert in cluster maintenance. Still, I want to share my knowledge.

Fully Managed Cassandra

Datastax started a fully managed Cassandra product called Astra. They promise a lot:

> - Start in minutes with a free tier, no credit card needed.
> - Eliminate the overhead to install, operate, and scale
>
> Cassandra clusters

> - Built on open-source Apache Cassandra™, used by the best of the internet.
> - Scale elastically — apps are viral ready from Day 1.
> - Deploy multi-cloud, multi-tenant or dedicated clusters on AWS, Azure, or GCP.
> - Ensure enterprise-level reliability, security, and management.
>
> Quoted from the Astra docs

I have no experience with their offering. But I would give it a try! Their pricing sounds reasonable.

Self-Managed Cassandra

Cassandra is built with Java. So knowing the basics of running JVM applications is very beneficial.

If you run Kubernetes, then definitely check out K8ssandra. It bundles all the helpful tools around Cassandra like:

- Stargate.io for REST, Graphql, and API Documentation
- Reaper for easier repair management
- Medusa for backups
- Metrics collector for monitoring
- Traefik for ingress

This stack of tools is fully open source and can be used without any additional monetary costs.

For developers, there is one very beneficial tool called nodetool. It can

Nodetool can also repair your data to enforce eventual consistency.

Other Learnings

Even after years of using Cassandra, there are still things to learn that let you use Cassandra more efficiently. In this section, I want to share various topics that you will experience eventually.

Data Migrations

If you have worked with other databases before, you might know database migration tools like flyway or liquibase. Since version 4.0 RC-1, there is basic liquibase support.

Additionally, the community worked on something similar with Cassandra-migration. It already supports advanced features such as leader election, for when multiple services start at the same time.

Any type of export and import can be done using DSBulk that allows loading and unloading data from and to Cassandra in CSV and JSON formats.

Tombstones

Cassandra is a multi-node cluster that contains replicated data on different nodes. Therefore, a delete can not simply delete a particular record.

For a delete operation, a new entry is added to the commit-log like for any other insert and update mutation. These deletes are called tombstones, and they flag a specific value for deletion

described in this blog post: About Deletes and Tombstones in Cassandra.

In Cassandra, you can set a time to live on inserted data. After the time passed, the record will be automatically deleted. When you set a time to live (TTL), a tombstone is created with a date in the future.

In comparison, a regular delete query is the same with the difference that the time date of the tombstone is set to the moment the delete is executed.

Let's create a tombstone by setting a TTL in seconds which basically function as a delayed delete:

```
cqlsh>
  INSERT INTO learn_cassandra.todo_by_user_email (user_email,creation_dat
```

And the data is stored like regular data:

```
cqlsh>
 SELECT * FROM learn_cassandra.todo_by_user_email WHERE user_email='john@

  user_email    | creation_date | name
----------------+---------------+--------------------
 john@email.com | 2021-05-30... | This entry should be removed soon

(1 rows)
```

```
cqlsh>
 SELECT TTL(name) FROM learn_cassandra.todo_by_user_email WHERE user_emai

 ttl(name)
-----------
        43

(1 rows)
```

After 60 seconds, the row is gone.

```
cqlsh>
 SELECT * FROM learn_cassandra.todo_by_user_email WHERE user_email='john@

 user_email | creation_date | todo_uuid | name
-----------+---------------+-----------+------

(0 rows)
```

Setting a TTL is one of many ways to create and execute tombstones.

Unfortunately, there are also others.

For example, when you insert a null value, a tombstone is created for the given cell. And as mentioned for delete requests, different types of tombstones are stored.

By default, after 10 days, data that is marked by a tombstone is freed

> **When is a compaction executed?**
>
> When the operation is executed depends mainly on the selected strategy. In general, a compaction execution takes `SSTables` and creates new `SSTables` out of it.
>
> The most common executions are:
>
> - When conditions for a compaction are true, that triggers compaction execution when data is inserted
> - A manually executed major compaction using the nodetool

Sometimes, tombstones not deleted for the following reasons:

- **Null values** mark values to be deleted and are stored as tombstones. This can be avoided by either replacing null with a static value, or not setting the value at all if the value is null
- **Empty lists and sets** are similar to null for Cassandra and create a tombstone, so don't insert them if they're empty. Take care to avoid null pointer exceptions when storing and retrieving data in your application
- **Updated lists and sets** create tombstones. If you update an entity and the list or set does not change, it still creates a tombstone to empty the list and set the same values. Therefore, only update necessary fields to avoid issues. The good thing is, they are compacted due to the new values

If you have many tombstones, you might run into another Cassandra issue that prevents a query from being executed

which is set by default to 100,000 tombstones. This means that, when a query has iterated over more than 100,000 tombstones, it will be aborted.

The issue here is, once a query stops executing, it's not easy to tidy things up because Cassandra will stop even when you execute a delete, as it has reached the tombstone limit.

Usually you would never have that many tombstones. But mistakes happen, and you should take care to avoid this case.

There is a handy operation metric that you should observe called `TombstoneScannedHistogram` to avoid unexpected issues in production.

`UPDATE`s Are Just `INSERT`s, and Vice Versa

In Cassandra, everything is append-only. There is no difference between an update and insert.

You already learned that a primary key defines the uniqueness of a row. If there is no entry yet, a new row will appear, and if there is already an entry, the entry will be updated. It does not matter if you execute an update or insert a query.

The primary key in our example is set to `user_email` and `creation_date` that defines record uniqueness.

Let's insert a new record:

```
cqlsh>
```

And execute an update with a new `todo_uuid` :

```
cqlsh>
  UPDATE learn_cassandra.todo_by_user_email SET
    name = 'Update query'
  WHERE user_email = 'john@email.com' AND creation_date = '2021-03-14 16:
```

2 new rows appear in our table:

```
cqlsh>
 SELECT * FROM learn_cassandra.todo_by_user_email WHERE user_email='john@

  user_email     | creation_date                     | name
----------------+-----------------------------------+--------------
 john@email.com | 2021-03-14 16:10:19.622000+0000 | Update query
 john@email.com | 2021-03-14 16:07:19.622000+0000 | Insert query

(2 rows)
```

So you inserted a row using an update, and you can also use an insert to update:

```
cqlsh>
  INSERT INTO learn_cassandra.todo_by_user_email (user_email,creation_dat
```

```
cqlsh>
 SELECT * FROM learn_cassandra.todo_by_user_email WHERE user_email='john@

 user_email     | creation_date            | name
----------------+--------------------------+----------------------
 john@email.com | 2021-03-14 16:10:19.62   |         Update query
 john@email.com | 2021-03-14 16:07:19.62   | Insert query updated


(2 rows)
```

So `UPDATE` and `INSERT` are technically the same. Don't think that an `INSERT` fails if there is already a row with the same primary key.

The same applies to an `UPDATE` — it will be executed, even if the row doesn't exist.

The reason for this is because, by design, Cassandra rarely reads before writing to keep performance high. The only exceptions are described in the next section about lightweight transactions.

But, there are restrictions what actions you can execute based on an update or insert:

- Counters can only be changed with `UPDATE`, not with `Insert`
- `IF NOT EXISTS` can only be used in combination with an `INSERT`
- `IF EXISTS` can only be used in combination with an `UPDATE`

Lightweight Transactions

You can use conditions in queries using a feature called lightweight transactions (LWTs), which execute a read to check a certain condition before executing the write.

Let's only update if an entry already exists, by using `IF EXISTS`:

```
cqlsh>
  UPDATE learn_cassandra.todo_by_user_email SET
    name = 'Update query with LWT'
  WHERE user_email = 'john@email.com' AND creation_date = '2021-03-14 16:

 [applied]
-----------
      True
```

The same works for an insert query using `IF NOT EXISTS`:

```
cqlsh>
  INSERT INTO learn_cassandra.todo_by_user_email (user_email,creation_dat

 [applied]
-----------
      True
```

Those executions are expensive compared to simple `UPDATE` and `INSERT` queries. Still, if it's business-critical, they are an excellent way to

Conclusion

I hope you enjoyed the article.

If you liked it and feel the need to give me a round of applause, or just want to get in touch, follow me on Twitter.

I work at eBay Kleinanzeigen, one of the world's biggest classified companies. By the way, we are hiring!

Special thanks goes to Roger Sheen, Michael de la Fontaine, Christian Baer, Thomas Uebel and Swen Fuhrmann for excellent feedback and proof-reading.

References

- Cassandra docs about replication factory
- Cassandra docs about consistency
- Compaction strategy overview
- Details on Leveled Compaction Strategy
- How materialized views work
- Known bugs with materialized views
- Start multi-node cassandra base
- Cassandra operation metrics
- How is data deleted in Cassandra
- How the Spark Cassandra connector works
- How sharding works

- Definition, history and definition of UUIDs
- Deletes and Tombstones in Cassandra
- Basic rules of Cassandra modeling
- Data Stax

Sebastian Sigl

Software engineer who loves writing software and teaching people. Since 2018, I work for eBay Kleinanzeigen that is nowadays part of Adevinta.

Trending Guides

Big O Notation
SQL Outer Join
Python For Loop
What is JavaScript?
Learn How To Code
Chrome Bookmarks
Concatenate Excel
C# String to Int
Git Switch Branch
JavaScript Splice
Model View Controller
Git Checkout Remote Branch
Insert Checkbox in Word
Find and Replace in Word
C Programming Language
HTML Link
Bayes Rule
Python Map
HTML Italics
Python SQL
HTML Bold
GraphQL VS Rest
If Function Excel
HTML List
Wav File
Subnet Cheat Sheet
String to Char Array Java
JavaScript Append to Array
Add Page Numbers in Word
JavaScript Projects

Our Nonprofit