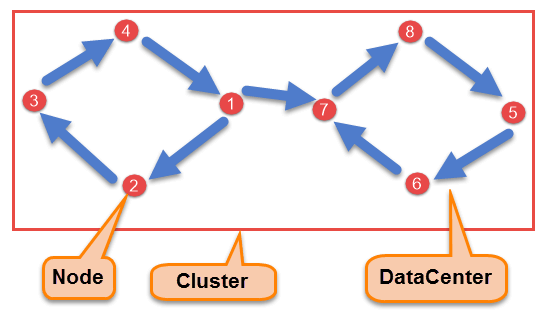
**Components of Cassandra**

There are following components in the Cassandra;

[](https://www.guru99.com/images/cassandra/021116_0524_CassandraAr1.png)

Cassandra Architecture Diagram

* **Node**

Node is the place where data is stored. It is the basic component of Cassandra.

* **Data Center**

A collection of nodes are called data center. Many nodes are categorized as a data center.

* **Cluster**

The cluster is the collection of many data centers.

* **Commit Log**

Every write operation is written to Commit Log. Commit log is used for crash recovery.

* **Mem-table**

After data written in Commit log, data is written in Mem-table. Data is written in Mem-table temporarily.

* **SSTable**

When Mem-table reaches a certain threshold, data is flushed to an SSTable disk file.

**Data Replication**

As hardware problem can occur or link can be down at any time during data process, a solution is required to provide a backup when the problem has occurred. So data is replicated for assuring no single point of failure.

Cassandra places replicas of data on different nodes based on these two factors.

* Where to place next replica is determined by the **Replication Strategy**.
* While the total number of replicas placed on different nodes is determined by the **Replication Factor**.

One Replication factor means that there is only a single copy of data while three replication factor means that there are three copies of the data on three different nodes.

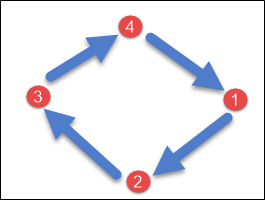
For ensuring there is no single point of failure, **replication factor must be three.**

There are two kinds of replication strategies in Cassandra.

**SimpleStrategy**

SimpleStrategy is used when you have just one data center. SimpleStrategy places the first replica on the node selected by the partitioner. After that, remaining replicas are placed in clockwise direction in the Node ring.

Here is the pictorial representation of the SimpleStrategy.

[](https://www.guru99.com/images/cassandra/021116_0524_CassandraAr2.png)

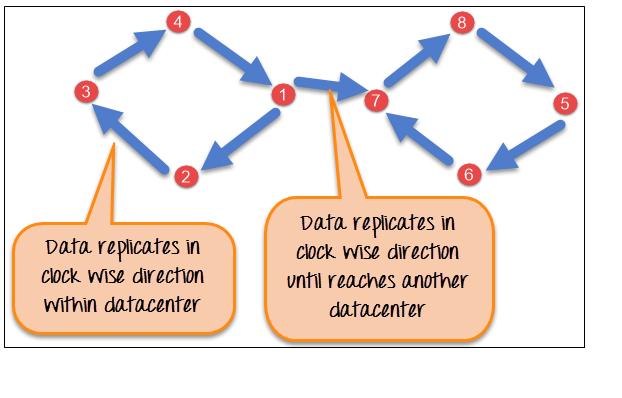
**NetworkTopologyStrategy**

NetworkTopologyStrategy is used when you have more than two data centers.

In NetworkTopologyStrategy, replicas are set for each data center separately. NetworkTopologyStrategy places replicas in the clockwise direction in the ring until reaches the first node in another rack.

This strategy tries to place replicas on different racks in the same data center. This is due to the reason that sometimes failure or problem can occur in the rack. Then replicas on other nodes can provide data.

Here is the pictorial representation of the Network topology strategy

[](https://www.guru99.com/images/cassandra/021116_0524_CassandraAr3.png)

**Write Operation**

**KeyWords:**

**Partitioning Key**— each table has a Partitioning Key. It helps with determining which node in the cluster the data should be stored.

**Commit Log**—the transactional log. It’s used for transactional recovery in case of system failures. It’s an append only file and provides durability.

**Memtable** — a memory cache to store the in memory copy of the data. Each node has a memtable for each CQL table. The memtable accumulates writes and provides read for data which are not yet stored to disk.

**SSTable** —the final destination of data in Cassandra. They are actual files on disk and are immutable.

**Compaction** —the periodic process of merging multiple SSTables into a single SSTable. It’s primarily done to optimize the read operations.

The coordinator sends a write request to replicas. If all the replicas are up, they will receive write request regardless of their consistency level.

**Consistency level** determines how many nodes will respond back with the success acknowledgment.

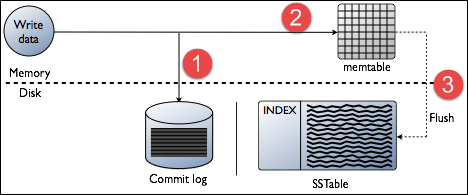
The node will respond back with the success acknowledgment if data is written successfully to the commit log and **memTable.**

For example, in a single data center with replication factor equals to three, three replicas will receive write request. If consistency level is one, only one replica will respond back with the success acknowledgment, and the remaining two will remain dormant.

Suppose if remaining two replicas lose data due to node downs or some other problem, Cassandra will make the row consistent by the built-in repair mechanism in Cassandra.

Here it is explained, how write process occurs in Cassandra,

1. When write request comes to the node, first of all, it logs in the commit log.
2. Then Cassandra writes the data in the mem-table. Data written in the mem-table on each write request also writes in commit log separately. Mem-table is a temporarily stored data in the memory while Commit log logs the transaction records for back up purposes.
3. When mem-table is full, data is flushed to the SSTable data file.

[](https://www.guru99.com/images/cassandra/021116_0524_CassandraAr4.png)

# Strategy for Writes

1. ANY — a write must succeed on any available node
2. ONE — a write must succeed on any node responsible for that row (either primary or replica)
3. QUORUM — a write must succeed on a quorum or replica nodes (replication\_factor / 2 + 1)
4. LOCAL\_QUORUM — a write must succeed on a quorum or replica nodes in the same data center as the coordinator node
5. EACH\_QUORUM — a write must succeed on a quorum of replica nodes in all data centers
6. ALL — a write must succeed on all replica nodes for a row key

**Read Operation**

**KeyWords:**

1. **Row Cache** — a memory cache which stores recently read rows (records). It’s an optional component in C\*.
2. **Bloom Filters** — helps to point if a partition key may exist in its corresponding SSTable.
3. **(Partition) Key Cache** — key cache maps recently read read partition keys to specific SSTable offset.
4. **Partition Indexes** — sorted partition keys mapped to their SSTable offsets. Partition Indexes are created as part of the SSTable creation and resides on the disk.
5. **Partition Summaries**— an off heap in memory sampling of the Partition Indexes and is used to speed up the access to index on disk.
6. **Compression Offsets** — keeps the offset mapping information for compressed blocks. By default all tables in C\* are compressed and when C\* needs to read data, it looks into the in memory compression offset maps and unpacks the data chunks.

*How does Cassandra read data?*

1. Cassandra first checks if the in-memory memtable cache still contain the data. Memtable is an in memory read/write cache for each column family.
2. If not found, Cassandra will read **all** SSTables for that Column Family.
3. To optimize reads…

* Cassandra uses bloom filter for each SSTable to determine whether this SSTable contains the key
* Cassandra uses index in SSTable to locate the data fast
* Cassandra compaction merges SSTables when the number of SSTables reaches certain threshold.
* Cassandra read is slower than write but yet still very fast

4. Cassandra depends on OS to cache SSTable files

* Do not configure Cassandra to use up most physical memory
* Some deployment configures Cassandra to use 50% of the physical memory so the rest can be used for file cache
* However, memory configuration is sensitive to data access pattern and volume

# Read Repair

Cassandra ensures that frequently read data remains consistent. Once a read is done, the coordinator node compares the data from all remaining replicas that own the row in the background. If they are inconsistent, issues writes to the out of data replicas to update the row to reflect the most recently written values.

Read repairs can be configured per column family and is enabled by default.

Read repair is important because every time a read request occurs, it provides an opportunity for consistency improvement. As a background process, read repair generally puts little strain on the cluster.

*How does a read repair work?*

When a query is made against a given key…

1. Cassandra performs a Read Repair
2. Read Repair perform a digest query on all replicas for that key. A digest query asks a replica to return a hash digest value and the timestamp for the key’s data. Digest query verify whether replica posses the same data without sending the data over the network.
3. Cassandra pushes the most recent data to any out of date replicas to make the queries data consistent again. Next query will therefore return a consistent data.

# Strategies for Reads

1. ONE — reads from the closest node holding the data
2. QUORUM — returns a result from a quorum of servers with the most recent timestamp of data
3. LOCAL\_QUORUM — returns a result from a quorum of servers with the most recent timestamp for the data in the same data center as teh coordinator node
4. EACH\_QUORUM — returns a result from a quorum of servers with the most recent timestamp in all data centers
5. ALL — returns a result from all replica nodes for a row key

# CAP Theorem

Consistency, **Availability, Partition Tolerance**

This theorem states that it’s impossible for a distributed system to have all of the 3 properties.

1. Consistency — all nodes see the same data at the same time
2. Availability — a guarantee that every request receieves a response about whether it was successful or failed
3. Partition tolerance — the system continues to operate despite arbitrary message loss or failure of part of the system

Cassandra is an AP system meaning it’s more important to be available and partition tolerant. In Cassandra you can choose between strong and eventual consistency. It can be done on a per-operation basis and for BOTH reads and writes. Also, it handles multi-data center operations.

# CQL using Consistency

Providing an argument to the CONSISTENCY command overrides the default consistency level of ONE for both read and write and configures the consistency level for the future requests.

Setting Consistency Level is done on a per query basis by adding**USING CONSISTENCY** <level> syntax as shown below.

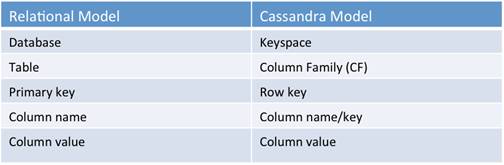
SELECT total\_purchases from SALES USING CONSISTENCY QUORUM where customer\_id = 5

UPDATE SALES USING CONSISTENCY ONE SET total\_purchases = 5000 WHERE customer\_id = 4

# Internal Data Structure

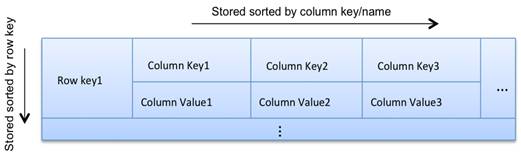
[1](https://teddyma.gitbooks.io/learncassandra/content/model/internal_data_structure.html#ref_1)The Cassandra data model is a schema-optional, column-oriented data model. This means that, unlike a relational database, you do not need to model all of the columns required by your application up front, as each row is not required to have the same set of columns.

[2](https://teddyma.gitbooks.io/learncassandra/content/model/internal_data_structure.html#ref_2)The Cassandra data model consists of keyspaces (analogous to databases), column families (analogous to tables in the relational model), keys and columns.



For each column family, don’t think of a relational table. Instead, think of a nested sorted map data structure. A nested sorted map is a more accurate analogy than a relational table, and will help you make the right decisions about your Cassandra data model.

Map<RowKey, SortedMap<ColumnKey, ColumnValue>>



A map gives efficient key lookup, and the sorted nature gives efficient scans. In Cassandra, we can use row keys and column keys to do efficient lookups and range scans. The number of column keys is unbounded. In other words, you can have wide rows.

A key can itself hold a value as part of the key name. In other words, you can have a valueless column.

**Cassandra Data Model Rules**

In Cassandra, writes are not expensive. Cassandra does not support joins, group by, OR clause, aggregations, etc. So you have to store your data in such a way that it should be completely retrievable. So these rules must be kept in mind while modelling data in Cassandra.

1. **Maximize the number of writes**

In Cassandra, writes are very cheap. Cassandra is optimized for high write performance. So try to maximize your writes for better read performance and data availability. There is a tradeoff between data write and data read. So, optimize you data read performance by maximizing the number of data writes.

1. **Maximize Data Duplication**

Data denormalization and data duplication are defacto of Cassandra. Disk space is not more expensive than memory, CPU processing and IOs operation. As Cassandra is a distributed database, so data duplication provides instant data availability and no single point of failure.

**Data Modeling Goals**

You should have following goals while modelling data in Cassandra.

1. **Spread Data Evenly Around the Cluster**

You want an equal amount of data on each node of Cassandra cluster. Data is spread to different nodes based on partition keys that is the first part of the primary key. So, try to choose integers as a primary key for spreading data evenly around the cluster.

1. **Minimize number of partitions read while querying data**

Partition are a group of records with the same partition key. When the read query is issued, it collects data from different nodes from different partitions.

If there will be many partitions, then all these partitions need to be visited for collecting the query data.

It does not mean that partitions should not be created. If your data is very large, you can’t keep that huge amount of data on the single partition. The single partition will be slowed down.

So try to choose a balanced number of partitions.

**Good Primary Key**

Let’s take an example and find which primary key is good.

Here is the table MusicPlaylist.

Create table MusicPlaylist

(

SongId int,

SongName text,

Year int,

Singer text,

Primary key(SongId, SongName)

);

In above example, table MusicPlaylist,

* Songid is the partition key, and
* SongName is the clustering column
* Data will be clustered on the basis of SongName. Only one partition will be created with the SongId. There will not be any other partition in the table MusicPlaylist.

Data retrieval will be slow by this data model due to the bad primary key.

Here is another table MusicPlaylist.

Create table MusicPlaylist

(

SongId int,

SongName text,

Year int,

Singer text,

Primary key((SongId, Year), SongName)

);

In above example, table MusicPlaylist,

* Songid and Year are the partition key, and
* SongName is the clustering column.
* Data will be clustered on the basis of SongName. In this table, each year, a new partition will be created. All the songs of the year will be on the same node. This primary key will be very useful for the data.

Our data retrieval will be fast by this data model.

**Model Your Data in Cassandra**

Following things should be kept in mind while modelling your queries.

1. **Determine what queries you want to support**

First of all, determine what queries you want.

For example, do you need?

* + Joins
  + Group by
  + Filtering on which column etc.

1. **Create table according to your queries**

Create table according to your queries. Create a table that will satisfy your queries. Try to create a table in such a way that a minimum number of partitions needs to be read.

## ****Real World Data Modelling Examples****

### **1. Facebook Posts**

Suppose that we are storing Facebook posts of different users in Cassandra. One of the common query patterns will be fetching the top ‘N‘ posts made by a given user.

Thus, **we need to** **store all data for a particular user on a single partition** as per the above guidelines.

Also, using the post timestamp as the clustering key will be helpful for retrieving the top ‘N‘ posts more efficiently.

Let's define the Cassandra table schema for this use case:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | CREATE TABLE posts\_facebook (    user\_id uuid,    post\_id timeuuid,    content text,    PRIMARY KEY (user\_id, post\_id) )  WITH CLUSTERING ORDER BY (post\_id DESC); |

Now, let's write a query to find the top 20 posts for the user Anna:

|  |  |
| --- | --- |
| 1 | SELECT content FROM posts\_facebook WHERE user\_id = "sunny" LIMIT 20 |

### **2. Gyms Across the Country**

Suppose that we are storing the details of different partner gyms across the different cities and states of many countries and we would like to fetch the gyms for a given city.

Also, let's say we need to return the results having gyms sorted by their opening date.

Based on the above guidelines, we should store the gyms located in a given city of a specific state and country on a single partition and use the opening date and gym name as a clustering key.

Let's define the Cassandra table schema for this example:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | CREATE TABLE gyms\_by\_city (   country\_code text,   state text,   city text,   gym\_name text,   opening\_date timestamp,   PRIMARY KEY (     (country\_code, state\_province, city),     (opening\_date, gym\_name))   WITH CLUSTERING ORDER BY (opening\_date ASC, gym\_name ASC); |

Now, let's look at a query that fetches the first ten gyms by their opening date for the city of Phoenix within the U.S. state of Arizona:

|  |  |
| --- | --- |
| 1  2  3 | SELECT \* FROM gyms\_by\_city    WHERE country\_code = "us" AND state = "Arizona" AND city = "Phoenix"    LIMIT 10 |

Next, let’s see a query that fetches the ten most recently-opened gyms in the city of Phoenix within the U.S. state of Arizona:

|  |  |
| --- | --- |
| 1  2  3  4 | SELECT \* FROM gyms\_by\_city    WHERE country\_code = "us" and state = "Arizona" and city = "Phoenix"    ORDER BY opening\_date DESC    LIMIT 10 |

Note: As the last query's sort order is opposite of the sort order defined during the table creation, the query will run slower as Cassandra will first fetch the data and then sort it in memory.

### **3. E-commerce Customers and Products**

Let's say we are running an e-commerce store and that we are storing the Customer and Product information within Cassandra. Let's look at some of the common query patterns around this use case:

1. Get Customer info
2. Get Product info
3. Get all Customers who like a given Product
4. Get all Products a given Customer likes

We will start by using separate tables for storing the Customer and Product information. However, we need to introduce a fair amount of denormalization to support the 3rd and 4th queries shown above.

We will create two more tables to achieve this – “Customer\_by\_Product” and “Product\_by\_Customer“.

Let's look at the Cassandra table schema for this example:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | CREATE TABLE Customer (    cust\_id text,    first\_name text,    last\_name text,    registered\_on timestamp,    PRIMARY KEY (cust\_id));    CREATE TABLE Product (    prdt\_id text,    title text,    PRIMARY KEY (prdt\_id));    CREATE TABLE Customer\_By\_Liked\_Product (    liked\_prdt\_id text,    liked\_on timestamp,    title text,    cust\_id text,    first\_name text,    last\_name text,    PRIMARY KEY (prdt\_id, liked\_on));    CREATE TABLE Product\_Liked\_By\_Customer (    cust\_id text,    first\_name text,    last\_name text,    liked\_prdt\_id text,    liked\_on timestamp,    title text,    PRIMARY KEY (cust\_id, liked\_on)); |

Note: To support both the queries, recently-liked products by a given customer and customers who recently liked a given product, we have used the “liked\_on” column as a clustering key.

Let's look at the query to find the ten Customers who most recently liked the product “Pepsi“:

|  |  |
| --- | --- |
| 1 | SELECT \* FROM Customer\_By\_Liked\_Product WHERE title = "Pepsi" LIMIT 10 |

And let's see the query that finds the recently-liked products (up to ten) by a customer named “Anna“:

|  |  |
| --- | --- |
| 1  2 | SELECT \* FROM Product\_Liked\_By\_Customer    WHERE first\_name = "Anna" LIMIT 10 |

# **CQL (CASSANDRA QUERY LANGUAGE)**

# Starting cqlsh on Linux and Mac OS X

1.Navigate to the Cassandra installation directory.

2.Start cqlsh

bin/cqlsh

If you use security features, provide a user name and password.

3.Optionally, specify the IP address and port to start cqlsh on a different node.

bin/cqlsh 1.2.3.4 9042

# Example of creating a keyspace

1. Create a keyspace

cqlsh> **CREATE** **KEYSPACE** **IF** **NOT** **EXISTS** cycling **WITH** **REPLICATION** = { 'class' : 'NetworkTopologyStrategy', 'datacenter1' : 3 };

2.Use the keyspace

cqlsh> **USE** cycling;

Creating a Table

CREATE TABLE cycling.cyclist\_alt\_stats ( id UUID PRIMARY KEY, lastname text, birthday timestamp, nationality text, weight text, height text );

Simple Primary Key

For a table with a simple primary key, Cassandra uses one column name as the partition key. The primary key consists of only the partition key in this case. Data stored with a simple primary key will be fast to insert and retrieve if many values for the column can distribute the partitions across many nodes.

cqlsh> USE cycling;

CREATE TABLE cyclist\_name ( id UUID PRIMARY KEY, lastname text, firstname text );

CREATE TABLE cyclist\_name ( id UUID, lastname text, firstname text, PRIMARY KEY (id) );

# **Composite Partition Key**

For a table with a composite partition key, Cassandra uses multiple columns as the partition key. These columns form logical sets inside a partition to facilitate retrieval. In contrast to a simple partition key, a composite partition key uses two or more columns to identify where data will reside. Composite partition keys are used when the data stored is too large to reside in a single partition. Using more than one column for the partition key breaks the data into chunks, or buckets. The data is still grouped, but in smaller chunks. This method can be effective if a Cassandra cluster experiences hotspotting, or congestion in writing data to one node repeatedly, because a partition is heavily writing. Cassandra is often used for time series data, and hotspotting can be a real issue. Breaking incoming data into buckets by year:month:day:hour, using four columns to route to a partition can decrease hotspots.

**CREATE** **TABLE** rank\_by\_year\_and\_name (

race\_year int,

race\_name text,

cyclist\_name text,

rank int,

**PRIMARY** **KEY** ((race\_year, race\_name), rank)

);

# **Using a compound primary key**

Use a compound primary key to create multiple columns that you can use to query and return sorted results.

**CREATE** **TABLE** cyclist\_category (

category text,

points int,

**id** UUID,

lastname text,

**PRIMARY** **KEY** (category, points)

) **WITH** **CLUSTERING** **ORDER** **BY** (points **DESC**);

# **Creating collections**

Cassandra provides collection types as a way to group and store data together in a column. For example, in a relational database a grouping such as a user's multiple email addresses is related with a many-to-one joined relationship between a user table and an email table. Cassandra avoids joins between two tables by storing the user's email addresses in a collection column in the user table. Each collection specifies the data type of the data held.

CQL contains these collection types:

* [set](https://docs.datastax.com/en/cql-oss/3.3/cql/cql_using/useSet.html)
* [list](https://docs.datastax.com/en/cql-oss/3.3/cql/cql_using/useList.html)
* [map](https://docs.datastax.com/en/cql-oss/3.3/cql/cql_using/useMap.html)

cqlsh> CREATE TABLE cycling.cyclist\_career\_teams ( id UUID PRIMARY KEY, lastname text, teams set<text> );

cqlsh> CREATE TABLE cycling.upcoming\_calendar ( year int, month int, events list<text>, PRIMARY KEY ( year, month) );

cqlsh> CREATE TABLE cycling.cyclist\_teams ( id UUID PRIMARY KEY, lastname text, firstname text, teams map<int,text> );

# **Creating a table with a tuple**

[Tuples](https://docs.datastax.com/en/cql-oss/3.3/cql/cql_reference/tupleType.html) are a data type that allow two or more values to be stored together in a column. A user-defined type can be used, but for simple groupings, a tuple is a good choice.

Create a table cycling.nation\_rankusing a tuple to store the rank, cyclist name, and points total for a cyclist and the country name as the primary key.

**CREATE** **TABLE** cycling.nation\_rank ( nation text **PRIMARY** **KEY**, info tuple<int,text,int> );

# **Creating a user-defined type (UDT)**

User-defined types (UDTs) can attach multiple data fields, each named and typed, to a single column. The fields used to create a UDT may be any valid data type, including collections and other existing UDTs. Once created, UDTs may be used to define a column in a table.

1.Create a user-defined type named basic\_info

**CREATE** **TYPE** cycling.basic\_info (

birthday **timestamp**,

nationality text,

weight text,

height text

);

2.Create a table for storing cyclist data in columns of type basic\_info. Use the [frozen](https://docs.datastax.com/en/cql-oss/3.3/cql/cql_reference/cql_data_types_c.html) keyword in the definition of the user-defined type column.

When using the frozen keyword, you cannot update parts of a user-defined type value. The entire value must be overwritten. Cassandra treats the value of a frozen, user-defined type like a blob.

cqlsh> CREATE TABLE cycling.cyclist\_stats ( id uuid PRIMARY KEY, lastname text, basics FROZEN<basic\_info>);

# **Inserting data into a table**

1.To insert simple data into the table cycling.cyclist\_name, use the INSERT command. This example inserts a single record into the table.

cqlsh> INSERT INTO cycling.cyclist\_name (id, lastname, firstname) VALUES (5b6962dd-3f90-4c93-8f61-eabfa4a803e2, 'VOS','Marianne');

2. If a table specifies a set to hold data, then either INSERT or UPDATE is used to enter data.

* Insert data into the set, enclosing values in curly brackets.

Set values must be unique, because no order is defined in a set internally.

cqlsh>**INSERT** **INTO** cycling.cyclist\_career\_teams (**id**,lastname,teams)

**VALUES** (5b6962dd-3f90-4c93-8f61-eabfa4a803e2, 'VOS',

{ 'Rabobank-Liv Woman Cycling Team','Rabobank-Liv Giant','Rabobank Women Team','Nederland bloeit' } );

* Add an element to a set using the UPDATE command and the addition (+) operator

**UPDATE** cycling.cyclist\_career\_teams **SET** teams = teams + {'Team DSB - Ballast Nedam'} **WHERE** **id** = 5b6962dd-3f90-4c93-8f61-eabfa4a803e2;

* Remove an element from a set using the subtraction (-) operator.

cqlsh> **UPDATE** cycling.cyclist\_career\_teams **SET** teams = teams - {'WOMBATS - Womens Mountain Bike & Tea Society'} **WHERE** **id** = 5b6962dd-3f90-4c93-8f61-eabfa4a803e2;

3.If a table specifies a list to hold data, then either INSERT or UPDATE is used to enter data.

* Insert data into the list, enclosing values in square brackets.

**INSERT** **INTO** cycling.upcoming\_calendar (**year**, **month**, **events**) **VALUES** (2015, 06, ['Criterium du Dauphine','Tour de Suisse']);

* Use the UPDATE command to insert values into the list. Prepend an element to the list by enclosing it in square brackets and using the addition (+) operator.

cqlsh> **UPDATE** cycling.upcoming\_calendar **SET** **events** = ['The Parx Casino Philly Cycling Classic'] + **events** **WHERE** **year** = 2015 **AND** **month** = 06;

4.If a table specifies a map to hold data, then either INSERT or UPDATE is used to enter data.

* Set or replace map data, using the INSERT or UPDATE command, and enclosing the integer and text values in a map collection with curly brackets, separated by a colon.

**INSERT** **INTO** cycling.cyclist\_teams (**id**, lastname, firstname, teams)

**VALUES** (

5b6962dd-3f90-4c93-8f61-eabfa4a803e2,

'VOS',

'Marianne',

{2015 : 'Rabobank-Liv Woman Cycling Team', 2014 : 'Rabobank-Liv Woman Cycling Team', 2013 : 'Rabobank-Liv Giant',

2012 : 'Rabobank Women Team', 2011 : 'Nederland bloeit' });

* Use the UPDATE command to insert values into the map. Append an element to the map by enclosing the key-value pair in curly brackets and using the addition (+) operator.

cqlsh> **UPDATE** cycling.cyclist\_teams **SET** teams = teams + {2009 : 'DSB Bank - Nederland bloeit'} **WHERE** **id** = 5b6962dd-3f90-4c93-8f61-eabfa4a803e2;

5.Inserting Tuple :Tuples are used to group small amounts of data together that are then stored in a column.

* Insert data into the table cycling.route which has tuple data. The tuple is enclosed in parentheses. This tuple has a tuple nested inside; nested parentheses are required for the inner tuple, then the outer tuple.

**INSERT** **INTO** cycling.route (race\_id, race\_name, point\_id, lat\_long) **VALUES** (500, '47th Tour du Pays de Vaud', 2, ('Champagne', (46.833, 6.65)));

* Insert data into the table cycling.nation\_rank which has tuple data. The tuple is enclosed in parentheses. The tuple called info stores the rank, name, and point total of each cyclist.

cqlsh> **INSERT** **INTO** cycling.nation\_rank (nation, info) **VALUES** ('Spain', (1,'Alejandro VALVERDE' , 9054));

6.If a table specifies a user-defined type (UDT) to hold data, then either INSERT or UPDATE is used to enter data.

* Set or replace user-defined type data, using the INSERT or UPDATE command, and enclosing the user-defined type with curly brackets, separating each key-value pair in the user-defined type by a colon.

cqlsh> **INSERT** **INTO** cycling.cyclist\_stats (**id**, lastname, basics) **VALUES** (

e7ae5cf3-d358-4d99-b900-85902fda9bb0,

'FRAME',

{ birthday : '1993-06-18', nationality : 'New Zealand', weight : null, height : null }

);

**Note:** Note the inclusion of null values for UDT elements that have no value. A value, whether null or otherwise, must be included for each element of the UDT.

* Data can be inserted into a UDT that is nested in another column type. For example, a list of races, where the race name, date, and time are defined in a UDT has elements enclosed in curly brackets that are in turn enclosed in square brackets.

cqlsh> **INSERT** **INTO** cycling.cyclist\_races (**id**, lastname, firstname, races) **VALUES** (

5b6962dd-3f90-4c93-8f61-eabfa4a803e2,

'VOS',

'Marianne',

[{ race\_title : 'Rabobank 7-Dorpenomloop Aalburg',race\_date : '2015-05-09',race\_time : '02:58:33' },

{ race\_title : 'Ronde van Gelderland',race\_date : '2015-04-19',race\_time : '03:22:23' }]

);

**Note:** The UDT nested in the list is frozen, so the entire list will be read when querying the table.

7.To insert JSON data, add JSON to the INSERT command.. Note the absence of the keyword VALUES and the list of columns that is present in other INSERT commands.

cqlsh> INSERT INTO cycling.cyclist\_category JSON '{

"category" : "GC",

"points" : 780,

"id" : "829aa84a-4bba-411f-a4fb-38167a987cda",

"lastname" : "SUTHERLAND" }';

* A null value will be entered if a defined column like lastname, is not inserted into a table using JSON format.

cqlsh> INSERT INTO cycling.cyclist\_category JSON '{

"category" : "Sprint",

"points" : 700,

"id" : "829aa84a-4bba-411f-a4fb-38167a987cda"

}';

A picture containing table, knife

Description automatically generated

| CQL Data Types | | |
| --- | --- | --- |
| **CQL Type** | **Constants supported** | **Description** |
| ascii | strings | US-ASCII character string |
| bigint | integers | 64-bit signed long |
| blob | blobs | Arbitrary bytes (no validation), expressed as hexadecimal |
| boolean | booleans | true or false |
| counter | integers | Distributed counter value (64-bit long) |
| date | strings | Value is a date with no corresponding time value; Cassandra encodes date as a 32-bit integer representing days since epoch (January 1, 1970). Dates can be represented in queries and inserts as a string, such as 2015-05-03 (yyyy-mm-dd) |
| decimal | integers, floats | Variable-precision decimal  Java type  **Note:** When dealing with currency, it is a best practice to have a currency class that serializes to and from an int or use the Decimal form. |
| double | integers, floats | 64-bit IEEE-754 floating point  Java type |
| float | integers, floats | 32-bit IEEE-754 floating point  Java type |
| frozen | user-defined types, collections, tuples | A frozen value serializes multiple components into a single value. Non-frozen types allow updates to individual fields. Cassandra treats the value of a frozen type as a blob. The entire value must be overwritten.  **Note:** Cassandra no longer requires the use of frozen for tuples:  frozen <tuple <**int**, tuple<text, **double**>>> |
| inet | strings | IP address string in IPv4 or IPv6 format, used by the python-cql driver and CQL native protocols |
| int | integers | 32-bit signed integer |
| list | n/a | A collection of one or more ordered elements: [literal, literal, literal].  **CAUTION:**  Lists have limitations and specific performance considerations. Use a frozen list to decrease impact. In general, use a [set](http://cassandra.apache.org/doc/latest/cql/types.html#sets) instead of list. |
| map | n/a | A JSON-style array of literals: { literal : literal, literal : literal ... } |
| set | n/a | A collection of one or more elements: { literal, literal, literal } |
| smallint | integers | 2 byte integer |
| text | strings | UTF-8 encoded string |
| time | strings | Value is encoded as a 64-bit signed integer representing the number of nanoseconds since midnight. Values can be represented as strings, such as 13:30:54.234. |
| timestamp | integers, strings | Date and time with millisecond precision, encoded as 8 bytes since epoch. Can be represented as a string, such as 2015-05-03 13:30:54.234. |
| timeuuid | uuids | Version 1 UUID only |
| tinyint | integers | 1 byte integer |
| tuple | n/a | Cassandra 2.1 and later. A group of 2-3 fields. |
| uuid | uuids | A UUID in [standard UUID](http://en.wikipedia.org/wiki/Universally_unique_identifier) format |
| varchar | strings | UTF-8 encoded string |
| varint | integers | Arbitrary-precision integer  Java type |

## Java types

The Java types, from which most CQL types are derived, are obvious to Java programmers. The derivation of the following types, however, might not be obvious:

| Derivation of selective CQL types | |
| --- | --- |
| **CQL type** | **Java type** |
| decimal | [java.math.BigDecimal](http://docs.oracle.com/javase/7/docs/api/java/math/BigDecimal.html) |
| float | [java.lang.Float](http://docs.oracle.com/javase/7/docs/api/java/lang/Float.html) |
| double | [java.lang.Double](http://docs.oracle.com/javase/7/docs/api/java/lang/Double.html) |
| varint | [java.math.BigInteger](http://docs.oracle.com/javase/7/docs/api/java/math/BigInteger.html) |

# **Cassandra Security**

There are two types of security in Apache Cassandra.

* **Authentication**
* **Authorization**

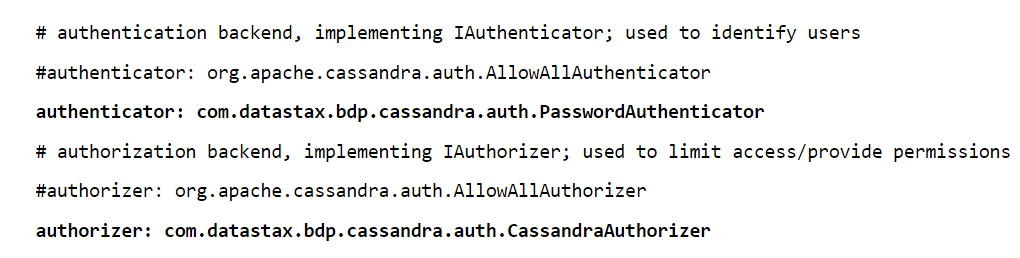
Authentication is basically validating user connection. The user is authenticated with login and password. All the user accounts are managed in Cassandra internally.

Authorization deals with user's permission. It deals with what actions user can be performed. For example, we can give user's permission such as which user has only data read permission, which user has data write permission and which user has data delete permission.

## Configure Authentication and Authorization

In Cassandra, by default authentication and authorization options are disabled. You have to configure Cassandra.yaml file for enabling authentication and authorization.

Open Cassandra.yaml file and uncomment lines that deals with internal authentication and authorization.

[](https://www.guru99.com/images/cassandra/021216_0545_CassandraSe1.png)

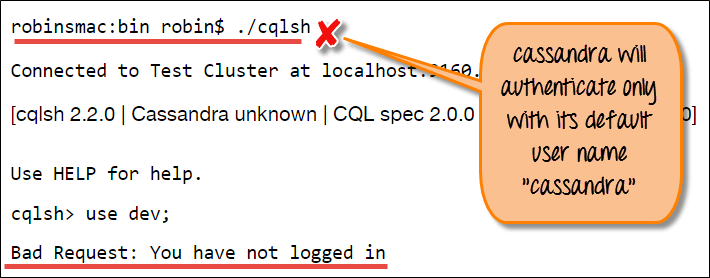
* In Cassandra.yaml file, by default, authenticator value is 'AllowAllAuthenticator'. Change this authenticator value from 'AllowAllAuthenticator' to 'com.datastax.bdp.cassandra.auth.PasswordAuthenticator'.
* Similarly, in Cassandra.yaml file, by default, authorizer value will be 'AllowAllAuthorizor'. Change this authorizer value from 'AllowAllAuthorizor' to 'com.datastax.bdp.cassandra.auth.CassandraAuthorizor'.

## Logging in

Now authentication is enabled, if you try to access any keyspace, Cassandra will return an error.

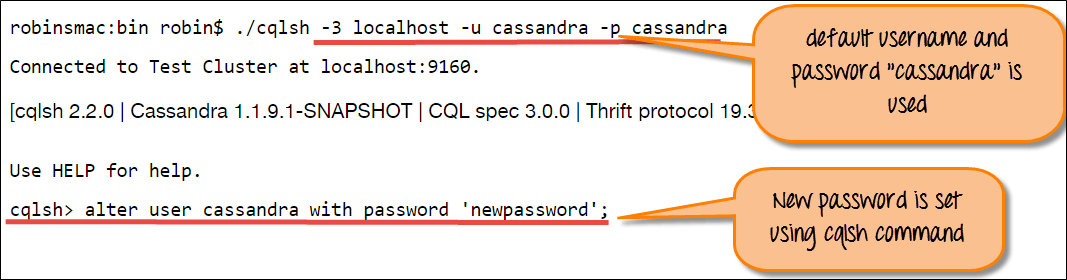
By default, Cassandra provides the super account with user name 'cassandra' and password 'cassandra'. By logging in to 'Cassandra' account, you can do whatever you want.

Let's see the below screenshot for this, where it will not allow you to login if you are not using the default Cassandra "username" and "password".

[](https://www.guru99.com/images/cassandra/021216_0545_CassandraSe2.png)

Now, in the second screenshot, you can see after using Cassandra default login credential, you are able to login.

You can also create another user with this account. It is recommended to change the password from the default. Here is the example of login Cassandra user and change default password.

[](https://www.guru99.com/images/cassandra/021216_0545_CassandraSe3.png)

alter user cassandra with password 'newpassword';

## Create New User

New accounts can be created with the 'Cassandra' account.

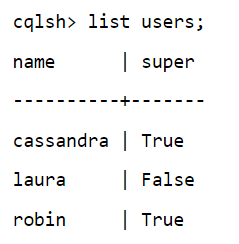
For creating a new user, login, the password is specified along with whether the user is super user or not. Only Super user can create new users.

create user robin with password 'manager' superuser;

create user robin with password 'newhire';

You can get a list of all users by the following syntax.

list users;

[](https://www.guru99.com/images/cassandra/021216_0545_CassandraSe5.png)

Users can be dropped by the following syntax.

drop user laura;

**Authorization**

Authorization is the assigning permission to users that what action a particular user can perform.

Here is the generic syntax for assigning permission to users.

GRANT permission ON resource TO user

There are following types of permission that can be granted to the user.

1. ALL
2. ALTER
3. AUTHORIZIZE
4. CREATE
5. DROP
6. MODIFY
7. SELECT

Here are examples of assigning permission to the user.

Create user laura with password 'newhire';

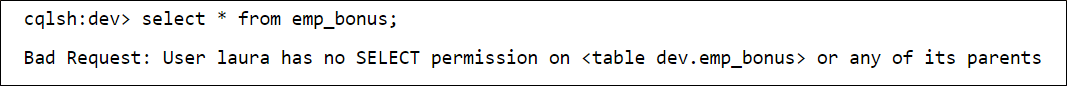
grant all on dev.emp to laura;

revoke all on dev.emp to laura;

grant select on dev.emp to laura;

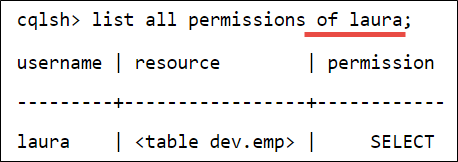
A new user 'laura' is created with password 'newhire'.

Here is the example where user 'laura' try to access emp\_bonus table. Laura has only permission to access dev.emp and no permission to this table dev.emp\_bonus that's why an error was returned.

[](https://www.guru99.com/images/cassandra/021216_0545_CassandraSe9.png)

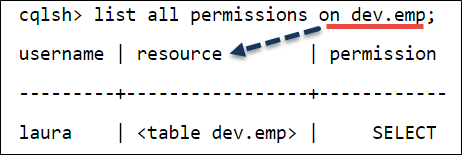
select\* form emp\_bonus;

You can get a list of all permissions that is assigned to the user. Here is the example of getting permission information.

[](https://www.guru99.com/images/cassandra/021216_0545_CassandraSe10.png)

list all permissions of laura;

You can also list all the permission on the resource. Here is the example of getting permission from a table.

[](https://www.guru99.com/images/cassandra/021216_0545_CassandraSe11.png)

list all permissions on dev.emp;

## Configuring Firewall

If the firewall is running, following ports must be opened for communication between nodes including some Cassandra ports. If Cassandra ports will not be opened, Cassandra nodes will act as standalone database server rather than joining the database cluster.

**Cassandra Client Ports**

|  |  |
| --- | --- |
| **Port Number** | **Description** |
| 9042 | Cassandra Client Port |
| 9160 | Cassandra Client Port Thrift |

**Cassandra Internode ports**

|  |  |
| --- | --- |
| **Port Number** | **Description** |
| 7000 | Cassandra internode cluster communication |
| 7001 | Cassandra SSL internode cluster communication |
| 7199 | Cassandra JMX monitoring port |

**Public Ports**

|  |  |
| --- | --- |
| **Port number** | **Description** |
| 22 | SSH port |
| 8888 | OpsCenter Website. Browser http request. |

**Cassandra OpsCenter ports**

|  |  |
| --- | --- |
| **Port Number** | **Description** |
| 61620 | OpsCenter monitoring port. |
| 61621 | Opscenter agent port |