## Introduction to Cassandra

Week-3 Part 1: Basics of Cassandra || Part 2: Working with Cassandra

Apache Cassandra is an open source database. It is best used by "always available" type of applications that require a database that is always available, that scales fast in situations of high traffic, and is the right choice when you need scalability and high availability without compromising performance. Apache Cassandra is best used by online services like Netflix, Uber, and Spotify. In this module, you will learn about the characteristics of Apache Cassandra. You will also expand your hands-on working knowledge of Cassandra performing various common tasks including using the CQL shell, keyspace operations, table operations, and CRUD operations.

**Learning Objectives**

•Describe Apache Cassandra and how it fits in the NoSQL space.

•Describe Cassandra's architecture and the components of a Cassandra node.

•Discuss how replication works in Cassandra and explain the scalability of Cassandra.

•Summarize the logical entities of the Cassandra data model.

•Define Clustering Keys and describe dynamic tables.

•Describe the Cassandra Query Language (CQL).

•Access the Cassandra server with cqlsh and perform basic cqlsh commands.

•Describe the main data types in CQL.

•Identify the role of keyspaces in Apache Cassandra.

•Create, use, and manipulate keyspaces in Cassandra.

•Discuss the role of Cassandra tables.

•Create, use, alter, and remove tables in Cassandra.

•Describe Cassandra's Write process and its INSERT and UPDATE operations.

•Explain the Read process in Cassandra clusters and how best to use SELECT statements.

•Insert, read, update, and delete table data in Cassandra.

# Overview of Cassandra

## Objectives

After watching this video, you will be able to:

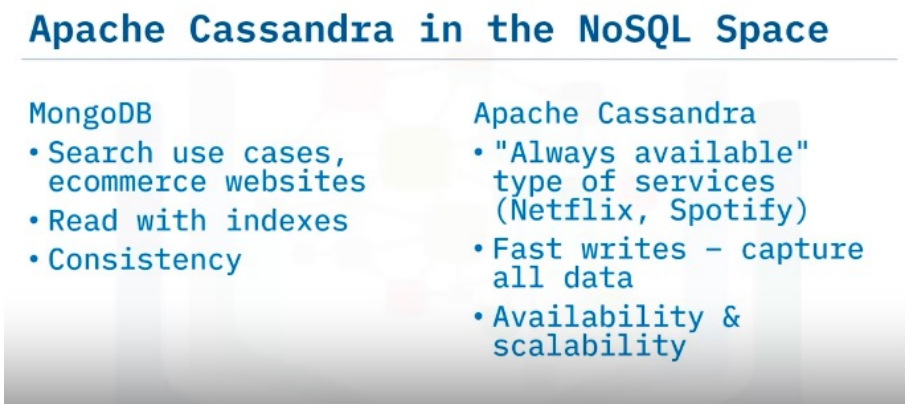
- Describe what Apache Cassandra is
- Explain the role of Apache Cassandra in the NoSQL space
- Explain the differences between Apache Cassandra and MongoDB
- Describe the main capabilities of Apache Cassandra
- Identify the best usage scenarios for Apache Cassandra
- Describe common use cases that are a best fit for Apache Cassandra

# What is Apache Cassandra?

*"Apache Cassandra is an open source, distributed, decentralized, elastically scalable, highly available, fault-tolerant, tunable, and consistent database that bases its distribution design on Amazon's Dynamo and its data model on Google's Bigtable. Created at Facebook, it is now used at some of the most popular sites on the Web."*

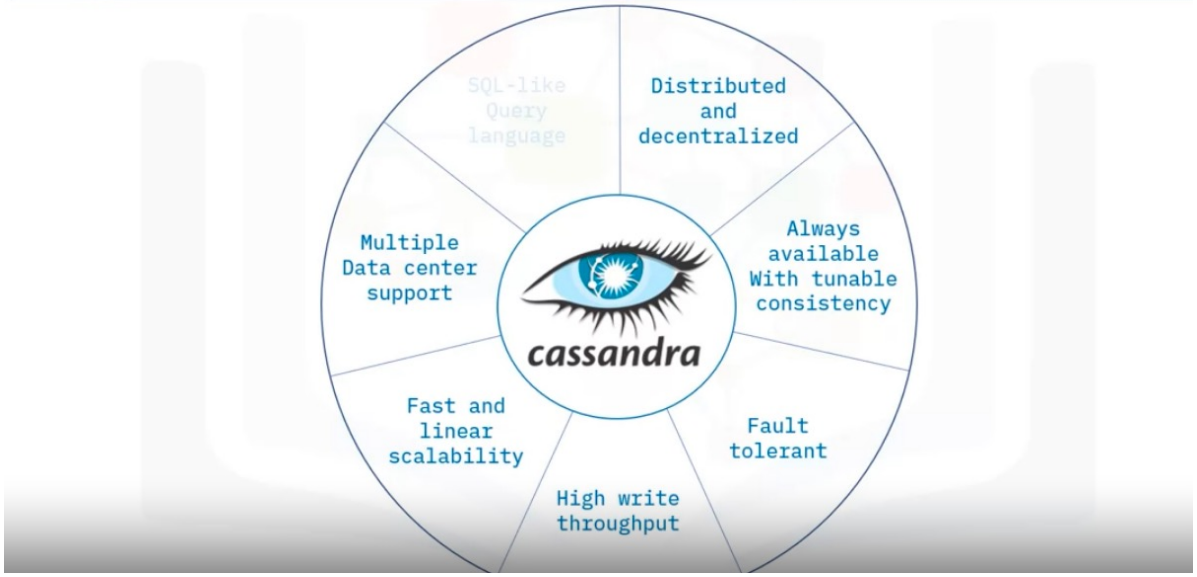

[Source: 'Cassandra: The definitive guide', Jeff Carpenter & Eben Hewitt]

- One of the best and most concise definitions of Cassandra has been given in the book 'Cassandra, the definitive guide'.
- Apache Cassandra is "an **open source, distributed, decentralized, elastically scalable, highly available, fault-tolerant, tunable and consistent database**" that bases its distribution design on Amazon's Dynamo and its data model on Google's Bigtable.
- **Created at Facebook,** it is now used at some of the most popular sites on the Web.
- Some of the services that use Cassandra are **Netflix, Spotify, and Uber.**




- Let's put Cassandra in a bit of **perspective v**ersus what you've learned so far about NoSQL databases and particularly about MongoDB – the document store database.
- As you have probably noticed, NoSQL databases tend to cater to very specific use cases: for example, **MongoDB** usually covers **search** related use cases, where the input data can be represented as **key : document type** of entries.
- But what about the use cases where you need to record some data extremely rapidly and make it available immediately for read operations, and all the while hundreds of thousands of requests are generated?
- Take for example, recording the transactions from an online shop or storing the user access info/profile for a service like Netflix.
- In this case a solution like Apache Cassandra could be of use. While MongoDB caters to read specific use cases and thus is very much focused on consistency of the data, **Cassandra c**aters to use cases that **require fast storage** of the data, **easy retrieval** of data **by key, availability at all times, fast scalability**, **and geographical distribution** of the servers.
- One other important difference lies in the architectural choice. While MongoDB has a Primary-Secondary architecture, Cassandra has a simpler, peer-to-peer one.

# Key Features of Apache Cassandra




- Cassandra has a set of features that sets it apart from other NoSQL solutions:
- distributed and decentralized - simple peer-to-peer architecture,
- making Cassandra one of the friendliest NoSQL database installations always available with tunable consistency
- favors availability over consistency fault tolerant
- extremely fast write throughput - while it maintains cluster performance for other operations
- like read capability to scale clusters extremely fast and in a linear fashion - without the need to restart or reconfigure the service
- multiple data centers deployment support - making it extremely useful for services that need to be accessed worldwide friendly SQL-like query language

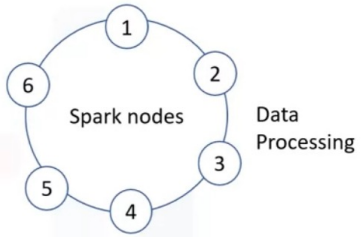
## What is Apache Cassandra?

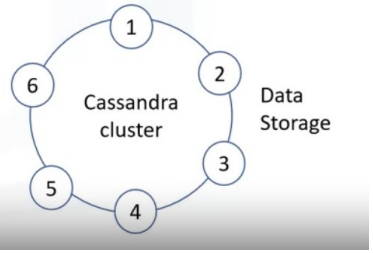
A reliable, performant, scalable database for data storage

Not a drop-in replacement for a relational database

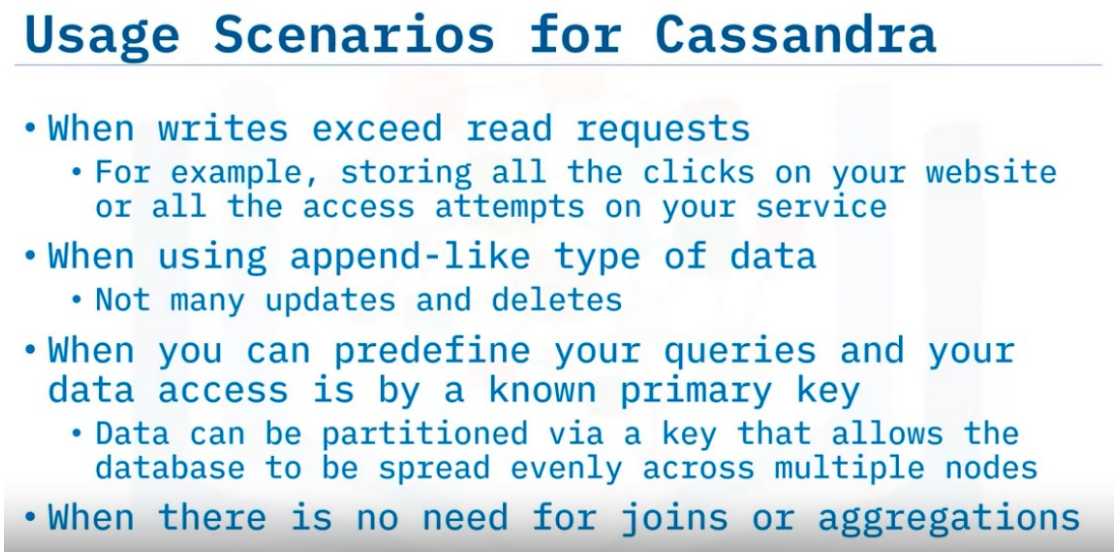
- does not support joins
- limited aggregations support
- limited support of transactions

For Joins and Aggregations: Cassandra + Spark

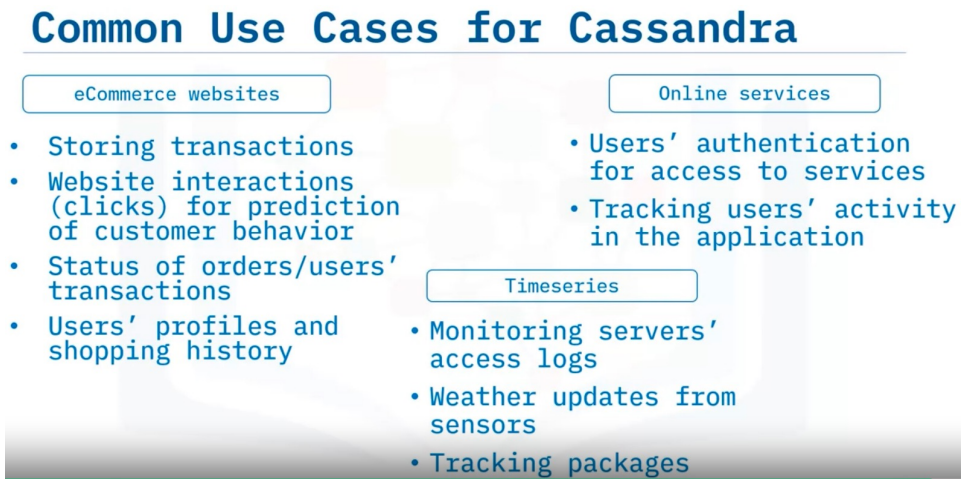


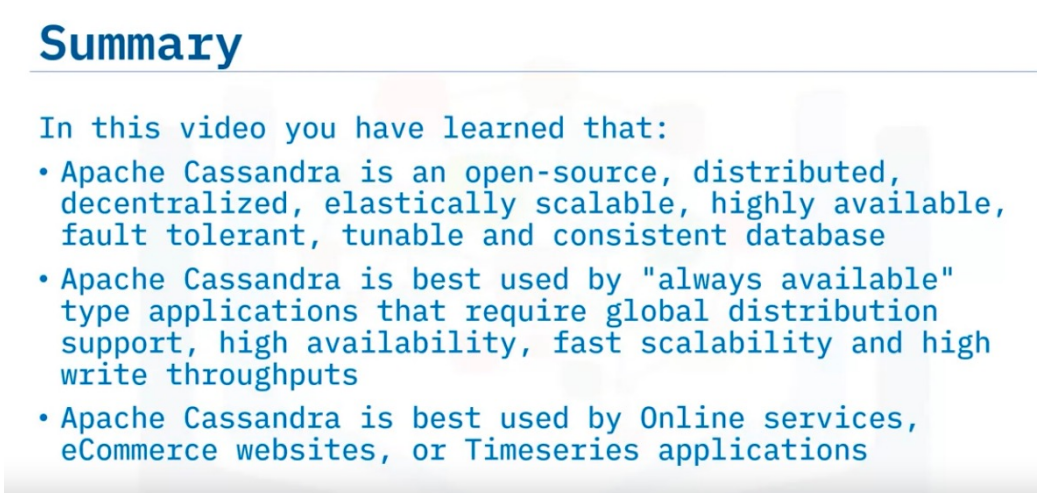




- Apache Cassandra is currently one of the **top ten most popular solutions** in the world; it's a reliable, super performant, and scalable database.
- Due to its popularity Cassandra is sometimes mistaken as being a **drop-in replacement for relational databases**, but Cassandra by design does not incorporate three major features of relational databases, and thus should not be seen as a drop-in replacement for a relational database:
- It does not support joins Has limited aggregations support and has limited support for transactions.
- While **writes t**o Cassandra are atomic, isolated, and durable in nature – the consistency part does not apply to Cassandra, as there is no concept of referential integrity or foreign keys. [**supports AID and not C from ACID]**
- So, in short, if you were thinking of using Cassandra to keep track of account balances at a bank, you probably should look at **alternatives**.
- If your application **has joins and aggregations requirements**, then it is best when Cassandra **is paired with** processing engines like **Apache Spark.**




- So, in which usage scenarios would Apache Cassandra be a good fit?
- When your application is write-intensive, and the number of writes exceeds the number of reads.
- For example, storing all the clicks on your website or all the access attempts on your service.
- When your data doesn't have that many updates or deletes, so it comes in an append-like manner.
- When data access is done via a known primary key, called a partition key.
- The key allows an even spread of the data inside the cluster.
- When there's no need for joins or complex aggregations as part of your queries.




- As mentioned previously, Cassandra is a best fit for globally "always available" types of online services and applications – such as **Netflix, Spotify, and Uber.**
- But of course, there are many other use cases that can take advantage of its capabilities.
- Storing transactions for analytical purposes for eCommerce websites or the users' interactions with the website in order to personalize their experience.
- Just storing users' profile info for services like session enrichment or granting personalized access to the service.
- Cassandra also thrives in timeseries use cases where data comes append-wise in a timely manner, like weather updates from sensors – where your query could be directed towards what happened to a certain sensor in a specific time interval.

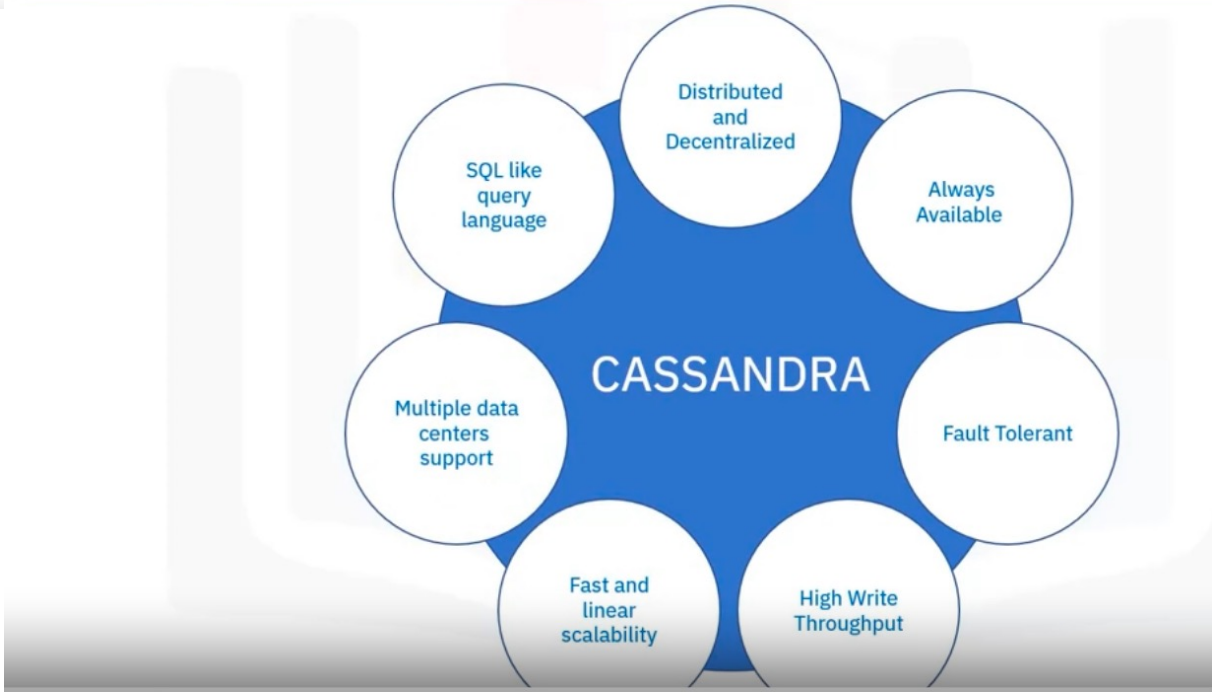
# Key Features of Cassandra

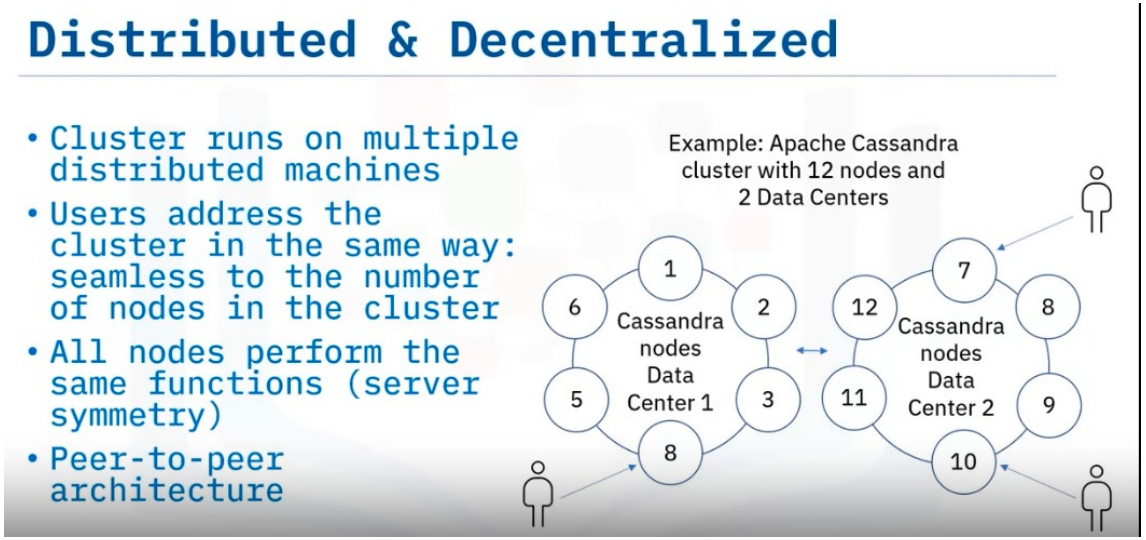
## Objectives

After watching this video, you will be able to:

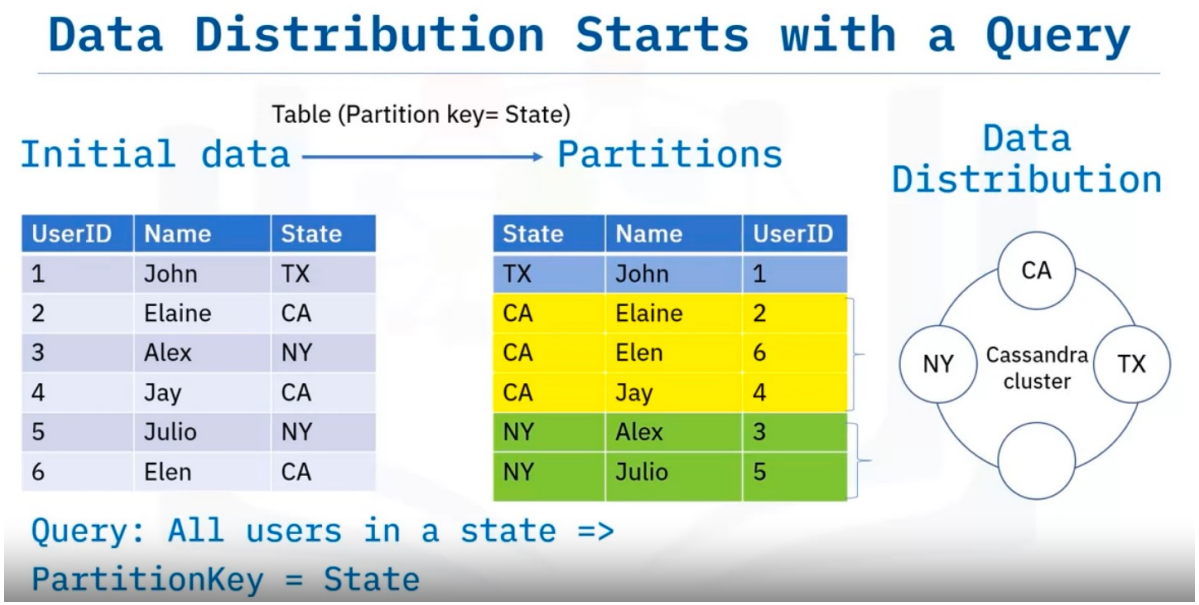
- Explain what is meant by distributed and decentralized
- Describe how replication works in Cassandra
- List the difference between availability and consistency
- Describe the scalability of Cassandra
- Explain how Cassandra provides high write throughp
- Explain what the CQL language is

## Introduction






- While all NoSQL databases are distributed, you will not find many NoSQL databases that are both **distributed and decentralized.**
- **Distributed** means that Cassandra clusters **can be run on multiple machines** while to the users and applications everything appears as a unified whole.
- The architecture is built in such a way that the combination of **Cassandra application client and server** will provide sufficient information to route the user request optimally in the cluster.
- So, as an end user, you can write data to any of the Cassandra nodes in the cluster and Cassandra will understand and serve your request.
- **Decentralized** means that **every node** in the Cassandra cluster is **identical** – that is, there are no primary or secondary nodes.
- Cassandra uses a peer-to-peer communication protocol and keeps all the nodes in-sync through a protocol called Gossip.



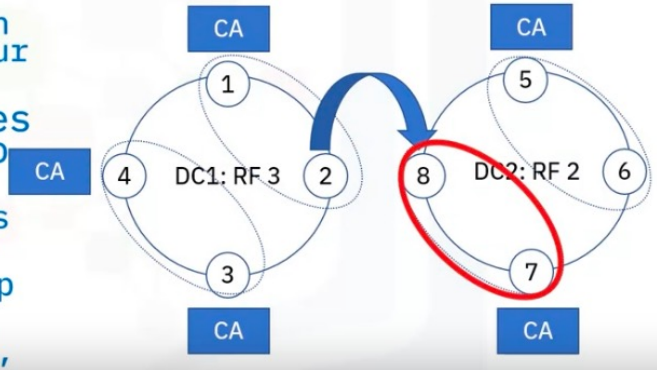

- How does data end up in this distributed architecture? It all starts with the queries that are planned to be performed.
- Just to take a very simple example: you have some initial data containing user info and the state.
- If your queries are like: "I would like to know about all users in a state..." In this case you need to group your data on the ‘state’ column, and this is done by declaring a table that has the ‘State’ column as the partition key.
- We will come back to how data needs to be stored in the cluster according to your queries in our next video, but until then, just remember that **Cassandra groups data based on your declared partition key** and **then distributes the data in the cluster** by hashing each partition key (called tokens).
- Each Cassandra node has a predefined list of supported token intervals and data is routed to the appropriate node based on the key value hash, and this predefined token allocation in the cluster.

## Data Replication and Multiple DC Support

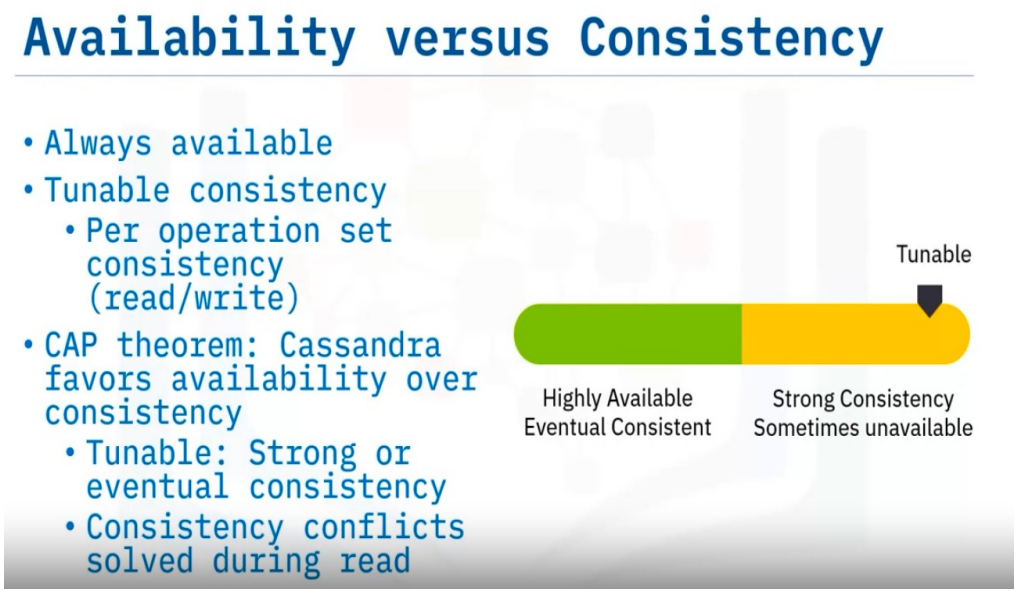
- Replicas
  - How many nodes contain a certain piece of your data (partition)
- Data Replication takes cluster topology into consideration
  - Racks and data centers distribution of nodes

8-node cluster, 2 DCs, Rep = 5 (DC1 RF3, DC2 RF2)

Nodes 1 & 2, 3 & 4, 5 & 6, 7 & 8 in the same rack



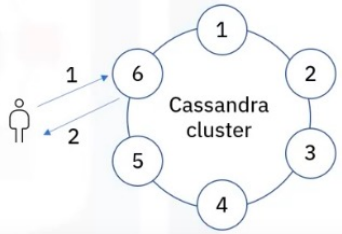

- After data is initially distributed in the cluster, Cassandra proceeds with replicating the data.
- The number of replicas refer to **how many nodes contain a certain piece of data at a particular time.**
- Data Replication is done clockwise in the cluster, taking into consideration the rack and the data center's placement of the nodes.
- Data Replication is done according to the **set Replication Factor** – which specifies the number of nodes that will hold the replicas for each partition.
- Let's take the data from the previous screen and try to distribute the California state data (partition CA) in an 8-node cluster, distributed in 2 Data Centers: DC1 with Replication Factor 3 and DC2 with Replication Factor 2.
- You can see in the diagram that some nodes are placed in the same rack. Cassandra will try to distribute data as much as possible between racks.




- One of the most important Cassandra features is its availability.
- Also, Cassandra is frequently referred to as "**eventual or tunable consistency**" in the sense that by default Cassandra trades consistency in order to achieve availability.
- But the good news is **that developers can control exactly how much consistency t**hey would like to have (strong or eventual). As you may recall from the NoSQL introduction video earlier in the course, distributed systems cannot – according to CAP theorem – be consistent and available at the same time.
- **Cassandra has been designed to be always available**, meaning that even if you lose a part of your cluster there will still be nodes available to answer the service request – though the returned data might be inconsistent.
- Consistency of the data can be controlled at the **operation level**, and it is tuned between **strong consistency and eventual consistency**.
- If data **inconsistencies** exist, these conflicts will be **resolved during read operations**. This is another unique Cassandra feature.

## High Availability and Fault Tolerance

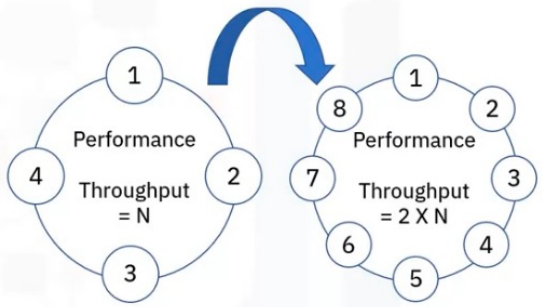
- Peer-to-Peer architecture
- Nodes' temporary/permanent failures are immediately recognized by the other nodes in the cluster
- Nodes reconfigure the data distribution once nodes are taken out of the cluster
- Failed requests can be re-transmitted to other nodes




- Fault tolerance is an inherent feature of the distributed and decentralized characteristic of Cassandra:
- the fact that all nodes have the same functions, communicate in a peer-to-peer manner, are distributed, and the data is replicated, makes Cassandra a very tolerant and adaptable solution when nodes fail.
- The user contacts one node of the cluster.
- If the node is not responding then the user will receive an error and contact another node.

## Fast and Linear Scalability

- Scales horizontally by adding new nodes in the cluster
- Performance increases linearly with the number of added nodes
- New nodes are automatically assigned tokens from existing nodes
- Adding and removing of nodes is done seamlessly



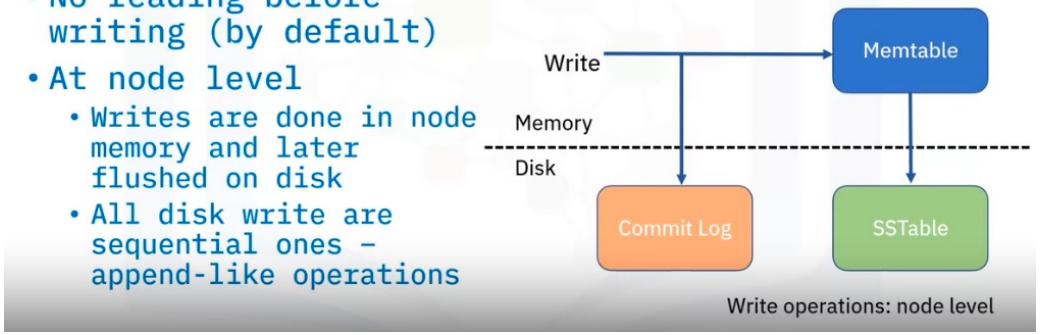

- The same architectural flexibility is visible in the way Cassandra scales the capability of the clusters.
- A cluster is scaled by simply adding nodes, and **performance increases linearly, with the number of added nodes**.
- New nodes that are added immediately start serving traffic, while existing nodes move some of their responsibilities towards the new added nodes.
- Adding and removing nodes is done **seamlessly** without interrupting cluster operations.

## High Write Throughput


- Cassandra gracefully handles large numbers of writes, first by **parallelizing writes to** all nodes holding a replica of your data.

- One important Cassandra fact: by default, there's **no read before write.**
- At the **node level, writes** are performed **in memory** (meaning no read before) and then flushed on disk.
- On disk, data is appended in a **sequential man**ner, with the data being reconciled later, through compaction.

## High Write Throughput



## Cassandra Query Language

Data Definition and Manipulation: CQL, an SQL-like syntax

```
CREATE TABLE test (
  groupid uuid,
  name text,
  occupation text,
  age int,
  PRIMARY KEY ((groupid), name));

INSERT INTO test (groupid, name, occupation, age)
  VALUES (1001, 'Thomas', 'engineer', 24), (1001, 'James', 'designer',
30,(1002, 'Lily', 'writer', 35));
SELECT * FROM test WHERE groupid = 1001;
```

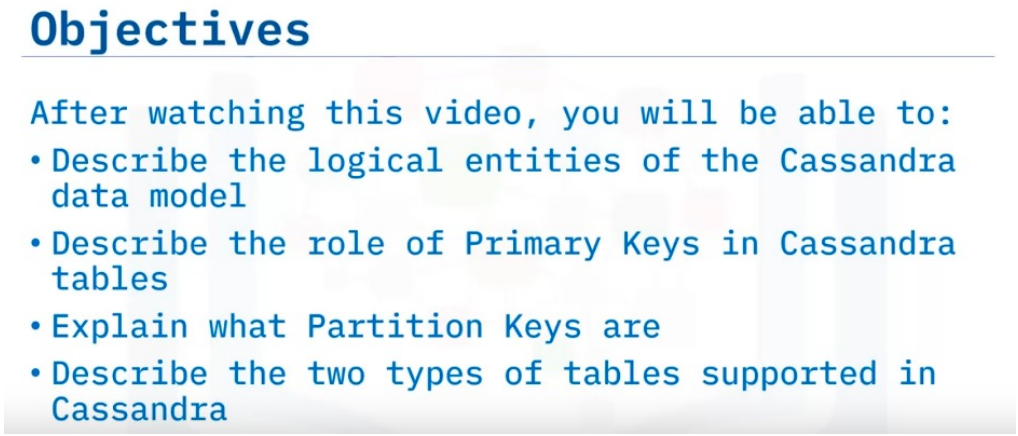
- Cassandra Query Language, or CQL for short, is the language used for data definition and manipulation.
- CQL syntax is similar to SQL syntax, thereby reducing the time a developer needs to get started with Cassandra.
- Operations like CREATE TABLE, INSERT, UPDATE, DELETE, ALTER, and more can be used in CQL.
- Be aware that while syntax-wise there are similarities between CQL and SQL, the resemblance stops here; the ways write and read operations are executed in Cassandra are different than how these are executed in relational databases.
- But we will come back to this point in later videos in the course

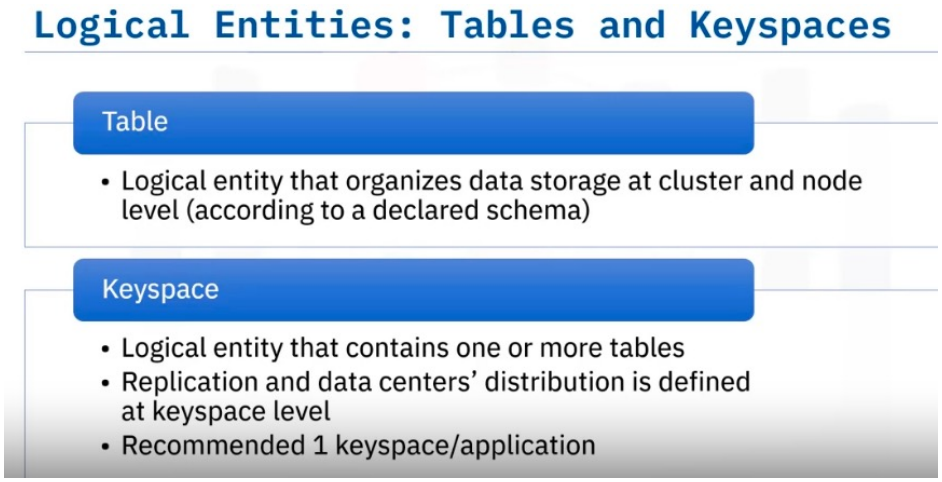
## Summary

In this video you have learned that:

- Its distributed and decentralized architecture helps Cassandra be available, scalable, and fault tolerant
- Data distribution and replication takes place in one or more data center clusters
- Cassandra provides high write throughput
- CQL is the language used to communicate with Cassandra

## Cassandra Data Model - Part 1




- **Cassandra stores data in tables**, whose schema defines the storage of the data at cluster and node level.
- **Tables are grouped in keyspaces.**
- A keyspace is a logical entity that **contains one or more tables**.
- A keyspace also defines a number of options that applies to all the tables it contains, most prominent of which is the replication strategy used by the keyspace.
- It is generally encouraged to **use one keyspace per application.**

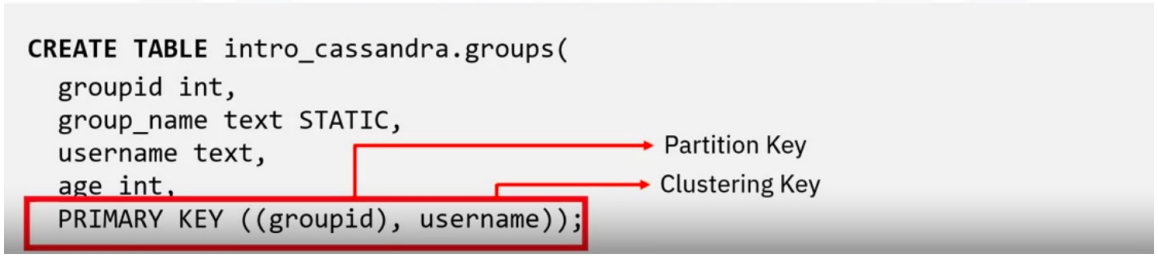
# Logical Entities: Tables and Keyspaces

```
CREATE KEYSPACE intro_cassandra  WITH REPLICATION = { 'class' :
'NetworkTopologyStrategy',  'datacenter1' : 2 ,  'datacenter2' : 3 };

USE intro_cassandra; //if you don't specify the keyspace you will need to
prefix your table with the keyspace name : intro_cassandra.groups

CREATE TABLE groups(
  groupid int,
  group_name text STATIC,
  username text,
  age int,
  PRIMARY KEY ((groupid), username));
```

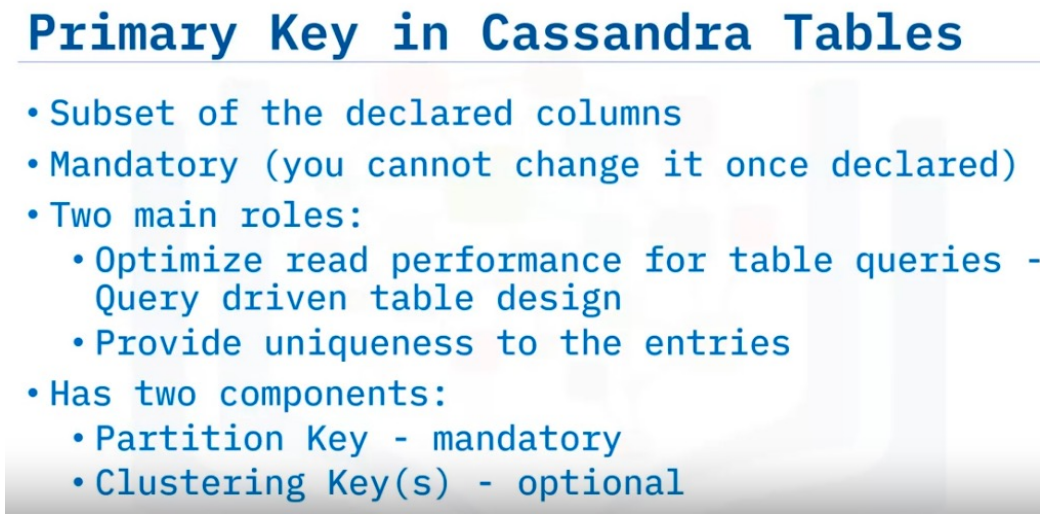
- Just take an example of a keyspace and a table definition.
- We have created a keyspace called ‘intro_cassandra’ that has a replication factor of 5, data being split between 2 data centers: with replication factor 2 in datacenter1 and replication factor 3 in datacenter2.
- The next step would be to create tables in this keyspace.
- To do this we can either declare the working keyspace, that is, ‘intro_cassandra;’ — or we just directly create the table — but in this case we need to prefix the table name with the intended keyspace name (intro_cassandra.groups).
- We create a table called ‘groups’ and the schema for the table.
- What you see in this example is just CQL syntax — which we introduced in the key features video earlier. We will come back to the table schema later in this video.

# Logical Entities: Tables

- Data is organized in tables containing rows of columns
- Tables can be created, dropped, and altered at runtime without blocking updates and queries
- To create a table, you must define a primary key and other data columns (regular columns)




- **tables** are the logical entities that **organize data storage** at **cluster and node level.** They contain rows of columns.
- You can **create, drop, and alter** your tables without impacting the running updates on your data or the running queries.
- In order to create a table we need to declare, using Cassandra Query Language **(CQL)**, a schema.
- A table schema comprises at least a definition of the table's primary key and the regular columns of the table. Going back to our previous example:
- Table groups store information regarding several groups, such as groupid, group name, and for each group, the username and age of their members.
- You can see that the **primary key is composed of two columns: 'groupid' and 'username'**.
- In Cassandra denomination, the 'groupid' column is called **the Partition Key** and the 'username' column is called the **Clustering Key**.
- Let's talk a bit more about the primary key

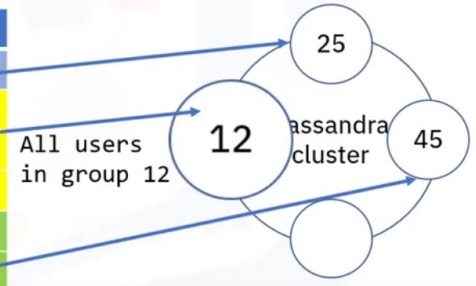



- the primary key is basically a subset of the table's declared columns.
- When creating a table, besides declaring the table's columns, it is mandatory to specify the primary key as well.
- The primary key – once defined – cannot be changed.
- In Cassandra, the primary key has two roles: The first role is to optimize the read performance of your queries.
- Do not forget that NoSQL systems are query driven data modeling systems – meaning that table definitions can start only after the queries you would like to answer are defined.
- You should build your primary key based on your queries.
- The second role is to provide uniqueness to the entries.
- A Primary Key has two components: The mandatory component is called the Partition Key and, optionally, you can have **one or more Clustering Keys**.

# Partition Keys

- When data is written to a table, it is grouped into partitions and distributed on cluster nodes – based on Partition Key
- Partition Key => Hash (token) => Node
- Partition key determines data (partition) locality in cluster

| GroupID | Group_name | Username | Age |
|---|---|---|---|
| 25 | Vegan cooking | Johns@gmail.com | 60 |
| 12 | Baking | Alaind@gmail.com | 32 |
| 12 | Baking | Elaine@yahoo.com | 60 |
| 12 | Baking | Peterd@gmail.com | 32 |
| 45 | Grilling | Moirad@yahoo.com | 35 |
| 45 | Grilling | Daveg@gmail.com | 43 |




```
PRIMARY KEY ((groupid), username))
```

- When data is inserted into the cluster in a table, **the data is grouped per partition key (into partitions)** and the first step is to apply a hash function to the partition key.
- The partition key hash is used to determine what node (and subsequent replicas) will get the data.

- Apache Cassandra utilizes '**Murmur3 consistent hashing**' which will take an arbitrary input and create a consistent token value.
- That token value will be inside the range of tokens owned by a single node.
- In simpler terms, **a partition key** determines the **data locality in the cluster**.
- You can see in the diagram and table, that data is grouped according to the partition key (groupid) and that each partition is distributed to one of the cluster nodes.
- **The partition is the atom of storage in Cassandra** – meaning that one partition's data will always be found on a node (and its replicas in the case of a replication factor greater than 1).
- So, if we want to answer the query "All users in group 12" then the query can address only the 4th node and will get the answer.
- In big clusters, (those consisting of hundreds or thousands of nodes), limiting the number of nodes required to be contacted in order to answer the query is crucial for query performance.

## Table Types

- Two types of tables: static and dynamic
- Static tables
  - PRIMARY KEY (username)
- Dynamic tables
  - PRIMARY KEY ((groupid), username)

- Before we move to our next video, to discuss clustering keys in more detail, first let's define two more concepts: **static and dynamic tables**.
- When the table has a primary key that contains only the partition key (single or multiple columns) and no clustering key, the table is called a static table.
- When the table has the primary key composed of both partition key(s) and clustering key(s), the table is called a dynamic table.

- This Users table is a static table. As you can see, the primary key consists only of the partition.
- This means that the number of distinct users we will have is the number of partitions we will have in our table.
- For the data distribution, **a hash is computed from each partition key** and **the partition's data is distributed according to predefined allocation of token intervals** to cluster nodes.
- In static tables, partitions have only one entry, and thus are called static partitions. We will continue with the dynamic tables concept in our next video, when we will also introduce the clustering key and its role in the Cassandra tables primary key.

## Static Tables

Users – static table
Partition key = username
No Clustering key
1 partition = 1 entry

| Username | Occupation | Age |
|---|---|---|
| Johns@gmail.com | Engineer | 60 |
| Alaind@gmail.com | Programmer | 32 |
| Elaine@yahoo.com | Engineer | 60 |
| Peterd@gmail.com | Marketer | 27 |
| Moirad@yahoo.com | None | 35 |
| Daveg@gmail.com | Pianist | 43 |
| JayZ@yahoo.com | Entertainer | 46 |

```
CREATE TABLE intro_cassandra.users(
  username text,
  occupation text,
  age int,
  PRIMARY KEY (username));
```

# Summary

In this video you learned that:

- Cassandra stores data in tables - grouped in keyspaces
- Recommendation - use one keyspace per application
- Create, drop, and alter without impacting running updates or queries
- Two roles of primary key - optimize read performance of table queries and provide uniqueness to entries
- Primary key components - mandatory partition key and optional clustering key(s)
- Cassandra has two types of tables: static and dynamic
- Static tables have a primary key that contains only the partition key, but no clustering key

# Cassandra Data Model - Part 2

## Objectives

After watching this video, you will be able to:

- Explain what Clustering Keys are
- Describe Dynamic tables
- Explain the basic guidelines for modeling your data

## Clustering Key

- Stores data in ascending or descending order within the partition for the fast retrieval of similar values
- Can have single or multiple columns
- Completes the primary key in dynamic tables
  - Gives uniqueness to primary key
  - Improves read query performance

```
CREATE TABLE intro_cassandra.groups(
  groupid int,
  group_name text STATIC,
  username text,
  age int,
  PRIMARY KEY ((groupid), username));
```

Groups – dynamic table
Partition key = groupid
Clustering key = username
1 partition = multiple entries

- Going back to the groups table we looked at in the previous video – in red you can see that the second component of the primary key is called the clustering key.
- While the partition key is important for data locality, the clustering column specifies the order that the data is arranged in inside the partition (that is, ascending or descending), and optimizes the retrieval of similar values column data inside a partition.
- The clustering key can be a **single or multiple column** key.
- In our case, our clustering key contains only one column – 'username' – which means that **data** inside the group partition is going to **be stored by username**, **by default in an ascending order.**
- So, when we **query** all users in a specific group, data will be ordered (by default in **ascending order**) by the users' username.
- In this example, the clustering key was used for one main reason: to add uniqueness to each entry. But actually, the clustering key is also very important for improving the read query performance. The next part of this video will illustrate this point.

## Clustering Key

- Query: "return all users in groupid 12 with age 32"
- Clustering key = age, username

```
CREATE TABLE intro_cassandra.groups_by_age(
  groupid int,
  group_name text STATIC,
  username text,
  age int,
  PRIMARY KEY ((groupid), age, username));
```

- Let's assume we would like an answer to the query: "Give me all users in groupid 12 that are aged 32".
- In this case, one of the possible data models for answering such a query is to have a table with '**groupid'** as the partition key and '**age' and 'username**' as clustering keys.
- Data inside each partition is first grouped and ordered by age and inside each age group it is then grouped by username.

- So, in this case all users in a certain group that have the same age will be stored together; thus, a query such as "Give me all users in a certain group (by groupid) that are aged 32", will just locate the node that contains the partition for groupid 12 and read from that node just the two sequential records.
- Reducing the amount of data to be read from a partition is crucial for query time, especially in the case of large partitions – in which Cassandra would have to read hundreds of megabytes of data in order to provide an answer from just a few kilobytes of data.

## Clustering Key

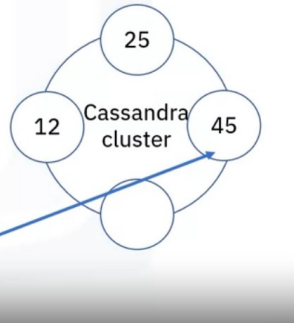
- Query: "return all users in groupid 12 with age 32"
- Clustering key = age, username

| GroupID | Group_name | Age | Username |
|---|---|---|---|
| 25 | Vegan cooking | 60 | Johns@gmail.com |
| 12 | Baking | 32 | Alaind@gmail.com |
| 12 | Baking | 32 | Peterd@gmail.com |
| 12 | Baking | 60 | Elaine@yahoo.com |
| 45 | Grilling | 35 | Moirad@yahoo.com |
| 45 | Grilling | 43 | Daveg@gmail.com |

# Dynamic Tables

```
INSERT INTO intro_cassandra.groups (groupid, group_name, username, age)
  VALUES (45, 'Grilling', 'JayZ@yahoo.com', 46));
```

| GroupID | Group_name | Username | Age |
|---|---|---|---|
| 25 | Vegan cooking | Johns@gmail.com | 60 |
| 12 | Baking | Alaind@gmail.com | 32 |
| 12 | Baking | Elaine@yahoo.com | 60 |
| 12 | Baking | Peterd@gmail.com | 32 |
| 45 | Grilling | Moirad@yahoo.com | 35 |
| 45 | Grilling | Daveg@gmail.com | 43 |
| 45 | Grilling | JayZ@yahoo.com | 46 |

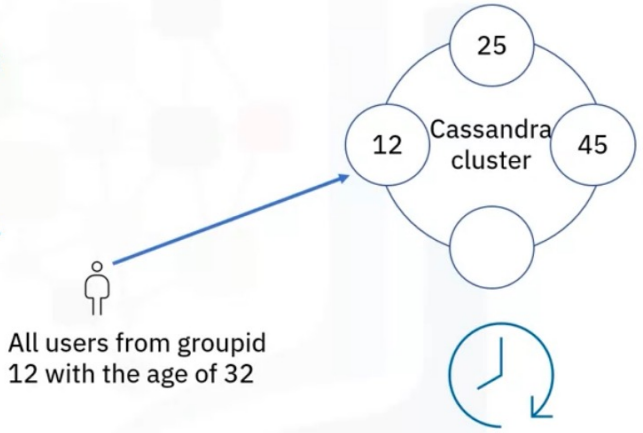



- Let's insert a new entry in a dynamic table like groups: Inserting new data in our table will just direct the write to the location of the partition and increase the partition size from 2 to 3 entries.
- In dynamic tables partitions grow dynamically with the number of distinct entries, due to the presence of the clustering key in the primary key.
- It's important to note that in this diagram, replication is not taken into account, so only one node holds the partition for group 45, for example.
- Every write or read for group 45 will be routed to the second node of the cluster.
- As long as the cluster is not changed, this node will hold the data for group 45.
- A change such as nodes added or leaving would trigger a new token allocation and subsequently data distribution.

# Basic Rules of Data Modeling

- Data Modeling – build a primary key that optimizes query execution time
- Choose a Partition Key – starts answering your query and spreads the data uniformly in the cluster
- Minimize the number of partitions read in order to answer the query

Note: To further optimize your query model, order clustering keys according to the query




- What we just did previously – building a primary key in order to answer the query in optimal time – is the beginning of the Data Modeling process.
- There's much more to Cassandra modeling than primary key definition, but to keep it simple we will refer back to this part, which is also the most important.

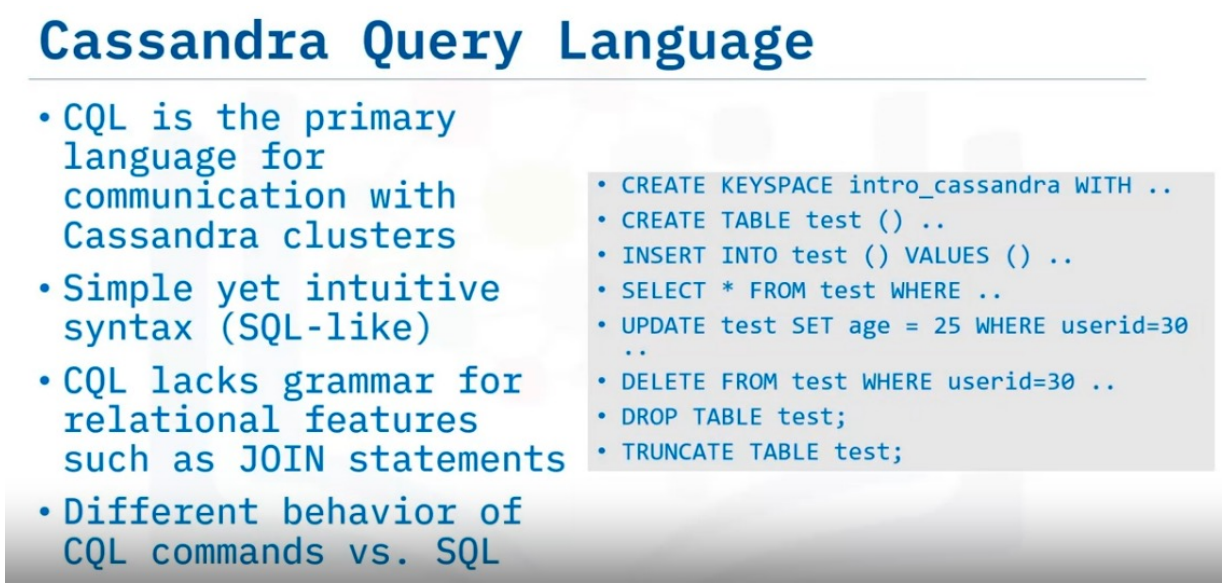
- When building a primary key for a table you should take into consideration the following simple rules: First, choose a partition key that starts answering your query, but that also spreads the data uniformly around the cluster. For example, 'groupid' might be a good partition key – if there any many groups (many distinct values for groupid) and group sizes are similar.
- And secondly, build a primary key that allows you to minimize the number of partitions read in order to answer a certain query. Remember that data is distributed throughout the cluster.
- If we would need to read more partitions to answer the query, then we would need to potentially access several nodes in order to answer our query.
- This would affect the query time, and even induce timeouts.
- Thus, the primary key needs to be designed so that optimally we read one partition in order to answer our query.
- And, one note before we summarize – besides the basic rules discussed here, make sure you build a clustering key that helps you reduce the amount of data that needs to be read even further, by ordering your clustering key columns according to your query.

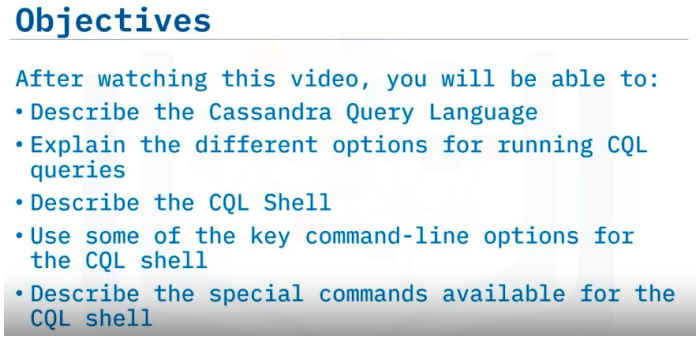
# Summary

In this video you learned that:

- Clustering keys specify the order of the data in partitions
- A clustering key can be a single or multiple column key
- Clustering keys provide uniqueness and improve query performance
- Reducing the amount of data read from partition is crucial
- Dynamic table partitions grow dynamically with the number of entries
- Building a primary key to answer queries is the first step of the Data Modeling process
- For good read performance start with queries you want to answer, then build your primary key accordingly

## Introduction to Cassandra Query Language Shell (cqlsh)




- Cassandra Query Language, referred to as CQL, is the primary language for communicating with Apache Cassandra clusters.
- CQL has a simple and intuitive, SQL-like syntax that allows the creation of **keyspaces, tables, inserts, updates, deletes, and select queries**.
- While CQL syntax has similarities to SQL, there are many d**ifference**s as well between what you can do in **CQL vs. SQL.**
- One of the most notable differences is that CQL does not support JOIN statements.
- In Cassandra, if you **need a join,** then you store the data already joined.
- Also, although, syntax-wise, some operations seem similar to SQL, their behavior in Cassandra is different; for example**, inserts, updates, and deletes are done in memory directly**, without prior reads to locate the data

## Cassandra Query Language

- CQL keywords are case-insensitive
- Identifiers in CQL are case-insensitive unless enclosed in double quotation marks
- Names for identifiers created using uppercase are stored in lowercase
- Commented text (//) is ignored by CQL

```
Select * from users;
SELECT * from users;
SELECT * FROM USERS;
//all commands are similar
```

```
CREATE TABLE USERS(..);
CREATE table users(..);
// Stored name of the table: users
```

- To run CQL queries, you can do one of the following: Run them programmatically using a licensed **Cassandra client driver**.
- There are many options available, including **Java, Python, and Scala.**
- The default driver is the open source Datastax Java Driver.
- Alternatively, you can run them on **the CQL Shell client** (referred to as 'cqlsh') that is provided with the Cassandra package.
- CQLSH is a python-based command line shell that uses CQL for communicating with a Cassandra cluster.
- The software is shipped with every Cassandra package.
- CQLSH connects to a single node – either the one that the command is run on (default), or the one specified as an option in the command line.
- There are also other CQL editors on the market that communicate with Cassandra using CQL.

- You will see, as you go through the lab exercises, that CQL keywords are case insensitive.
- Identifiers are also case insensitive, unless they are enclosed in double quotation marks.
- For example, 'Select' in sentence case, and 'SELECT' in uppercase are treated the same.
- If you enter names for identifiers, for example a table name, using uppercase letters, Cassandra will just store the names in lowercase anyway.
- As you can see in the examples, commented text, denoted by a double forward slash, will be ignored by CQL.

## Running CQL Queries

- Run using Cassandra client drivers
  - Java, Python, Ruby, Node.js, PHP, Scala, Clojure, …
  - Default = open source Datastax Java Driver
- Run using cqlsh client
  - Python-based command line shell for interacting with Cassandra through CQL
  - Shipped with every Cassandra package
  - Connects to a single node (default node or one specified on the command line)
- Other CQL client editors are available

## CQL Shell (cqlsh)

Using cqlsh, you can:

- Create, alter, drop keyspaces
- Create, alter, drop tables
- Insert, update, delete data
- Execute read queries (SELECT)

```
cqlsh> use intro_cassandra;
cqlsh:intro_cassandra> select * from users;

 username         | age | occupation
------------------+-----+---------------
   Daveg@gmail.com |  43 |       engineer
    JayZ@yahoo.com |  46 | entertainment
  Elaine@yahoo.com |  60 |       producer
   Johns@gmail.com |  60 |          actor
  Alaind@gmail.com |  32 |          actor
  Peterd@gmail.com |  32 |       producer
  Moirad@yahoo.com |  35 |             dj

(7 rows)
```

- Using CQL shell you can: Create, alter, and drop keyspaces Create, alter, and drop tables Insert, update, and delete data Execute queries using SELECT

- But before we look at some cqlsh examples, when launching cqlsh, there are a number of command line options you should know about.
- Let's just mention a few of them here:
- help – shows help about the options of CQLSH commands
- version – in order to check the cqlsh version being used
- user and password – for authentication keyspace – keyspace to authenticate to
- file – enables execution of commands from a given file
- request timeout – you can set a timeout for your queries; the default being 10 seconds

## Command-line Options for cqlsh

```
> cqlsh [options] [host [port]]

Options:
--help shows help about the cqlsh command options
--version shows the version of cqlsh being used
-u -user specifies the username to authenticate to Cassandra
with
-p -password specifies the password to authenticate to
Cassandra with
-k -keyspace specifies a keyspace to authenticate to
-f -file enables execution of commands from a given file
--request-timeout specifies the request timeout in seconds
(defaults to 10s)
```

# cqlsh – example

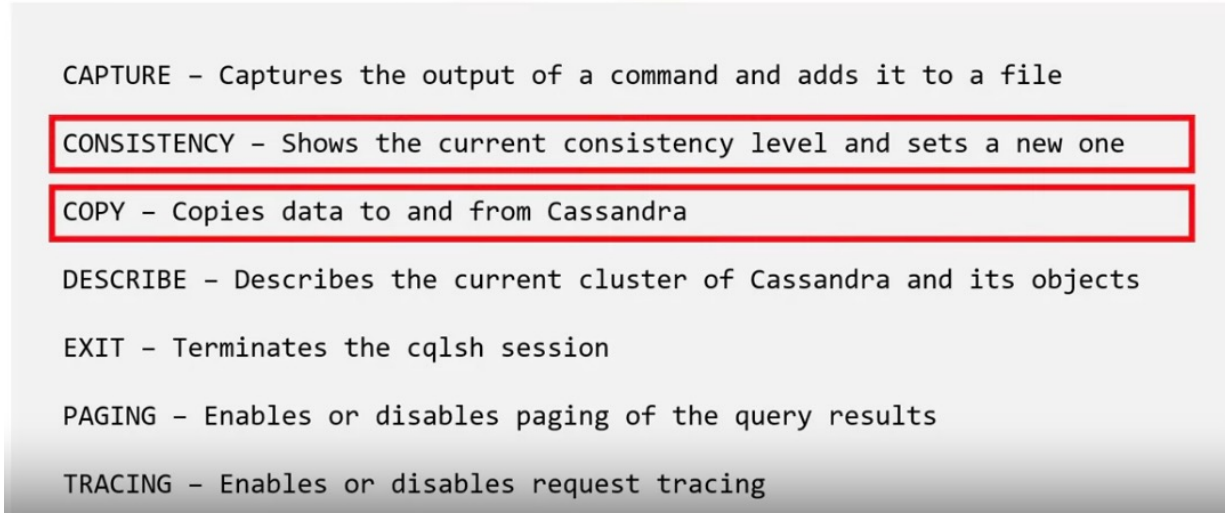
```
cqlsh> use intro_cassandra;
cqlsh:intro_cassandra> select * from groups where groupid=12;

 groupid | username         | group_name | age
---------+------------------+------------+-----
      12 | Elaine@yahoo.com |     Baking |  46
      12 | Peterd@gmail.com |     Baking |  32

(2 rows)
cqlsh:intro_cassandra> insert into groups(groupid,username,group_name,age) values(12,'Aland@gmail.com','Baking',32) ;
cqlsh:intro_cassandra> select * from groups where groupid=12;

 groupid | username         | group_name | age
---------+------------------+------------+-----
      12 |  Aland@gmail.com |     Baking |  32
      12 | Elaine@yahoo.com |     Baking |  46
      12 | Peterd@gmail.com |     Baking |  32

(3 rows)
```

**Commands used in cqlsh:**

```
USE intro_Cassandra;
SELECT * FROM groups where groupid=12;
INSERT INTO
groups(groupid,username,group_name,age)
VALUES (12,'Aland@gmail.com','baking',32);
SELECT * From groups where groupid=12;
```

- Let's see the cqlsh output for a simple example: Here we're just using the 'intro_cassandra' keyspace, and we select only the users from groupid 12.
- Then we insert a new record into the groups table for groupid 12, and finally select only the users from groupid 12, and also view the inserted data.
- And this would be the resulting output.

## cqlsh – special commands




- CQLSH also has some special commands, so let's take a quick look at some of them:
- CAPTURE – Captures the output of a command and adds it to a file.
- CONSISTENCY – Shows the current consistency level and sets a new one.
- COPY – Copies data to and from Cassandra.
- DESCRIBE – Describes the current cluster of Cassandra and its objects.
- EXIT – Terminates the cqlsh session.
- PAGING – Enables or disables the paging of query results.
- TRACING – Enables or disables request tracing. While in your hands-on sessions you will get to experience some of these.
- We will now look in a bit more detail at the CONSISTENCY and COPY commands.

# cqlsh CONSISTENCY

```
cqlsh: intro_cassandra_keyspace> CONSISTENCY

Sets the consistency level for the operations to follow. Consistency refers to
the number of nodes (out of the total replicas) that should respond to a query
(write/read) in order to consider the query successful.

ONE
TWO
THREE
QUORUM - majority of nodes from the entire cluster
ALL
LOCAL_QUORUM – majority of nodes from the local data center (according to the
specified data center set replication for the keyspace)
```

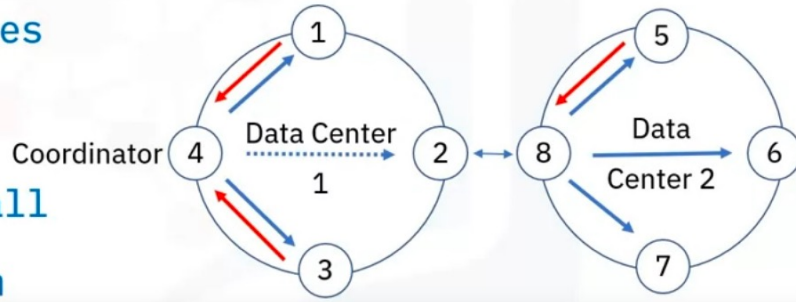
- Let's take a detailed look at the CONSISTENCY command.
- Cassandra **is tunable consistent.** But what does this mean? Well, it means you have the ability to set Cassandra consistency at an operations level.
- Consistency in Cassandra means the **number of nodes** (out of the total replicas) that **need to answer a request** (write/read) in order for the operation to be considered successful.
- Amongst the options available for the Consistency command are: ONE, TWO, or THREE nodes,
- QUORUM — which is a majority of nodes out of all replicas in the cluster, ALL replicas,
- LOCAL_QUORUM — which is a majority of nodes from the local data center (according to the specified data center set replication for the keyspace)

# Consistency example – QUORUM

```
cqlsh: intro_cassandra_keyspace> CONSISTENCY QUORUM
cqlsh: intro_cassandra_keyspace> INSERT INTO groups (groupid,
group_name,username,age) VALUES (45, 'Grilling', 'JayZ@yahoo.com', 46));
```

Multi DC cluster of 8 nodes
DC1 RF 2, DC2 RF 3

- Set consistency QUORUM (3)
- Write operations go to all replicas (5)
- Minimum of 3 nodes (from both DCs) need to answer for a successful operation




- So, for example, if we have a Cassandra cluster of 8 nodes and 2 Data Centers, one data center with replication factor 2 and the other one with **replication factor 3, then** overall, replication is 5.
- If we **set CONSISTENCY QUORUM** on a write operation: first of all, QUORUM means a majority of nodes out of the replicas, so a **majority here would be 3 Write operations** will go to all 5 replicas but a minimum 3 nodes (from both data centers) out of the 5 replicas need to answer in order for the operation to be successful

## cqlsh COPY (import/export data)

```
cqlsh: intro_cassandra_keyspace> COPY

CQL shell command that imports and exports CSV (comma-separated values or
delimited text files) - Not suitable for bulk loading

• COPY TO exports data from a table into a CSV file - Each row is written to
  a line in the target file with fields separated by the delimiter
• COPY FROM imports data from a CSV file into an existing table
   • Each line in the source file is imported as a row
   • All rows in the dataset must contain the same number of fields and have
     values in the PRIMARY KEY fields
   • The process verifies the PRIMARY KEY and imports data accordingly
```

- Another important command in cqlsh is the COPY command.
- While bulk copy in Cassandra should be done through special procedures, if you would like to just test your model or you're working with a smaller, text delimited dataset, then you can use **COPY FROM or COPY TO** operations to bring data in, and export data out, of Cassandra.
- COPY TO exports data from a table into a **CSV file.**
- Each row is written to a line in the target file with fields separated by the delimiter.
- **COPY FROM imports** data from a **CSV file into an existing table**.
- Each line in the source file is imported as a row.
- All rows in the dataset must contain the same number of fields and have values in the PRIMARY KEY fields.
- The process verifies the PRIMARY KEY and imports data accordingly.

# Summary

In this video you learned that:

- CQL is the primary language for communication
- CQL keywords and identifiers are case-insensitive
- CQL queries can be run using a Cassandra client driver, or they can be run using CQL Shell
- Using the CQL shell, you can create, alter, and drop keyspaces and tables, insert, update, and delete data, and execute SELECT queries
- CQL shell has several special commands
- CONSISTENCY can be used to tune data consistency
- COPY can be used to import/export data

**Summary and Highlights**

Congratulations! You have completed this lesson. At this point in the course, you know:

•Apache Cassandra is an open source, distributed, decentralized, elastically scalable, highly available, fault tolerant, and tunable and consistent database.

•Apache Cassandra is best used by "always available" type of applications that require a database that is always available.

•Data distribution and replication takes place in one or more data center clusters.

•Its distributed and decentralized architecture helps Cassandra be available, scalable, and fault tolerant.

•Cassandra stores data in tables.

•Tables are grouped in keyspaces.

•A clustering key specifies the order that the data is arranged inside the partition (ascending or descending).

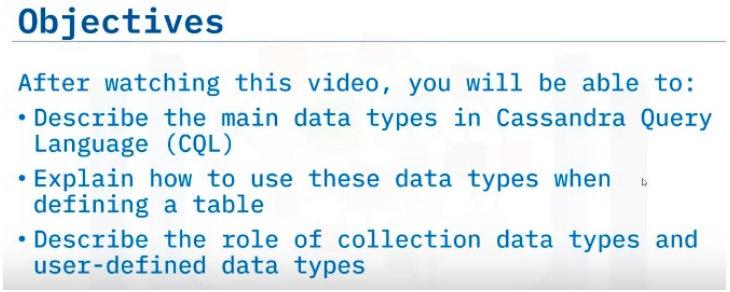
•Dynamic tables partitions grow dynamically with the number of entries.

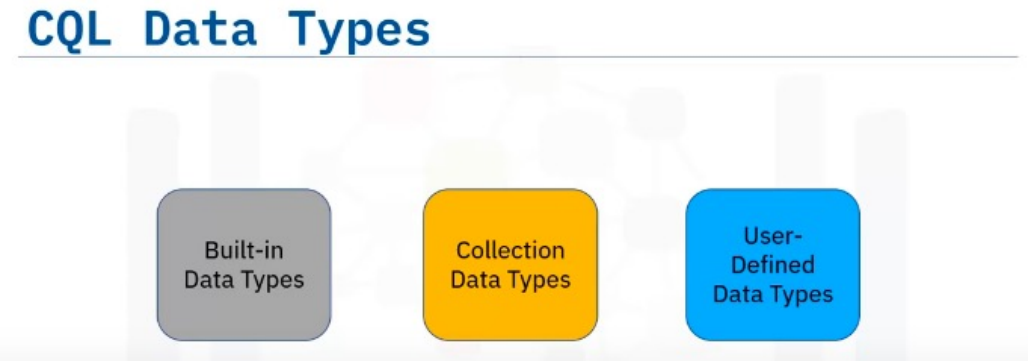
•CQL is the primary language for communicating with Apache Cassandra clusters.

•CQL queries can be run programmatically using a licensed Cassandra client driver, or they can be run on the Python-based CQL shell client provided with Cassandra.

# Part 2 : Working With Cassandra

## CQL Data Types




- Data types usually signify the type of variables, such as ‘int’, ‘char’, ‘float’, and others.
- In CQL there are many data types, but they can be grouped into three main categories:
- Built-in data types Collection data types and User-Defined data types.
- The user can choose any of them according to the requirements of the application and data model.

## Built-in Data Types

| Data Type | Data Type |
|---|---|
| Ascii | Int |
| Boolean | Text |
| Blob | Timestamp |
| Bigint | Timeuuid |
| Decimal | Tinyint |
| Double | Uuid |
| Float | Varchar |

Blob – arbitrary bytes. A blob type is suitable for storing a small image or short string (1MB)

Bigint – used for 64-bit signed long integer. This data type stores a higher range of integers as compared to int

Varchar – It is used for strings, and represents a UTF8 encoded string

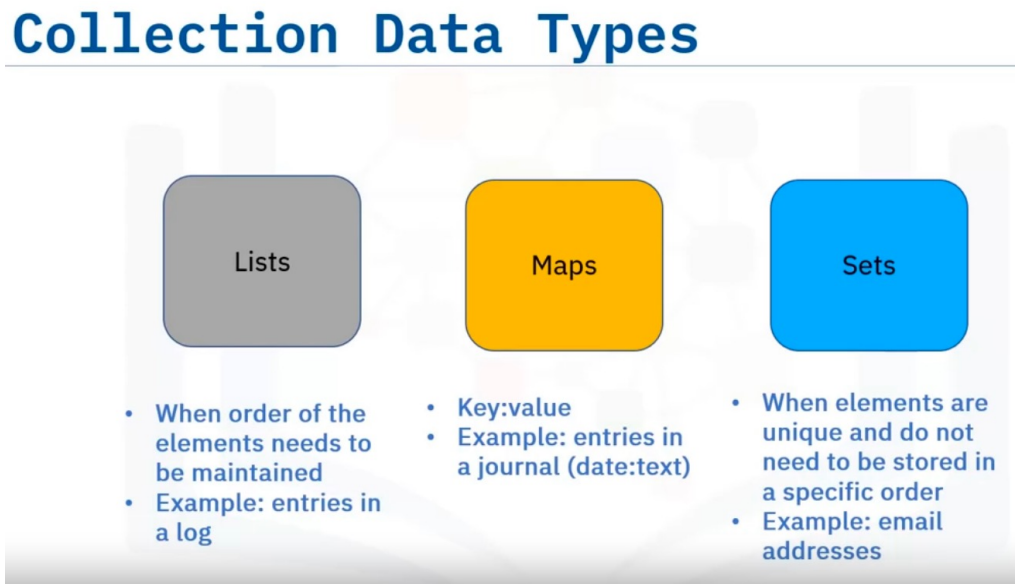
- The built-in data types are basically pre-defined in Cassandra. The user can refer the variables to any of them.

- Besides regular data types like Ascii, Boolean, Decimal, Double, Float, Int, and Text – which are fairly straightforward – there are some others that maybe require a bit of explanation.
- Let's take the 'Blob' data type. Although Cassandra mainly stores text-based information, there is also the possibility to store blobs, which stands for Binary Large Objects.
- Blobs are typically used to store images, audio, or other multimedia objects. While blobs represent a collection of binary data stored as a single entity, in Cassandra it is recommended that their size does not exceed 1MB.
- Thus, you could store a small image or string using a blob.
- The 'Bigint' data type can be used for a 64-bit signed long integer. This data type stores a higher range of integers when compared to the 'int' data type.
- The well-known 'Varchar' is also available in Cassandra as a data type. It represents UTF8 encoded strings

## Collection Data Types

- Collections = a way to group and store data together
- Example: user has multiple email addresses
  - In relational world: a many-to-one joined relationship between a ‘users’ table and an ’email’ table
  - In Cassandra: no joins => we store all the data in a collection column in the ‘users’ table
- Data for collection storage should be limited – no unbounded growth
  - Not suitable for storing sent messages or sensor events stored every second

- 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 ‘users’ table and an ‘email’ table.
- Cassandra avoids joins between two tables by storing the user's email addresses in a collection column in the ‘users’ table.
- Each collection specifies the data type of the data held.
- A collection is appropriate if the data for collection storage is limited.
- If the data has unbounded growth potential, like messages sent or sensor events registered every second, do not use the collection data type.
- **Instead, use a table with a compound primary key** where data is stored in the clustering columns.




- Within the Collection data types category there are three data types:
- **Lists**: This Cassandra data type represents a collection of one or more elements in a table.
- List is used in cases where **the order of the elements is to be maintained, and a value is to be stored multiple times,** such as entries in a log.
- **Maps**: This Cassandra data type represents a collection **of key-value** pairs.
- Map is a data type that is used to store a key-value pair of elements, such as entries in a journal entered using a date and then text.
- Sets: This Cassandra data type represents a **collection of one or more sorted elements** in a **table.**
- Set is a data type that is used to store a group of elements. The elements of a set will be returned in a sorted order.
- An example would be a list of email addresses.

# Collection Data Types - List

```
USE intro_cassandra;

ALTER TABLE users ADD jobs list<text>;

UPDATE users SET jobs = ['Walmart'] + jobs  where username
= 'Alaind@gmail.com'; // add the last job change to the list

UPDATE users SET jobs = jobs +
['Netflix']  where username = 'Alaind@gmail.com'; // add the last job
change to the list

UPDATE users SET jobs[0]='Reiss' where username
= 'Alaind@gmail.com'; //replaces Walmart with Reiss (lists start from 0)
```

Users table
PRIMARY KEY(userid)

- Let's go back to the ‘users’ table.
- Let's add a new column to the table, called jobs, which is basically just a list of jobs.
- We would like to store the jobs in the order of their occurrences.
- Remember that the ‘users’ table is a static table with its primary key consisting of the ‘userid’ column.
- In Cassandra we will store all the users’ jobs in a single column, since we cannot perform joins.
- In this case we will use the ‘list’ type of the collection data types because we want to preserve the order of the jobs.
- Another reason is that a person can work at a specific company more than once, so uniqueness is not required.
- We can add a job in the list: either at the beginning or end of the list or in a specific position.
- The entries in the list can be repetitive, as they are not unique.

# Collection Data Types - list

```
username         | age | jobs         | occupation
-----------------+-----+--------------+---------------
  Daveg@gmail.com |  43 |         null |      engineer
    JayZ@yahoo.com |  46 |         null | entertainment
 Elaine@yahoo.com |  60 |         null |      producer
  Johns@gmail.com |  60 |         null |         actor
    dan@gmail.com |  46 |         null |        banker
 Alaind@gmail.com |  32 | ['Netflix'] |         actor
 Peterd@gmail.com |  32 |         null |      producer
 Moirad@yahoo.com |  35 |         null |            dj
```

```
UPDATE users SET jobs = jobs - ['Reiss']  WHERE
username = 'Alaind@gmail.com';
SELECT * FROM users WHERE
username='Alaind@gmail.com';
```

- A select in cqlsh on our table shows that we currently have two jobs recorded for our user.
- We can also remove an entry from the list.
- A select on our table now shows that we only have one job listed for our user.
- Besides collection data types there are other data types that allow for the storing of data together.

- We have seen that collection data types can be used instead of a one-to-many join relationship.
- However, for one-to-one relations we can use Cassandra User-Defined Types (or UDTs).
- UDTs can attach multiple data types to a single column.
- Think of an address being a combination of an apartment number, building, street, and so on.
- CQL enables the user to create their own data type.
- The fields used in a created UDT can be any valid data type, including collections and other existing UDTs.
- After creating a data type and fields in it, the user can alter, verify, and even drop a field or the whole data type.
- Once created, UDTs can be used to define a column in a table.

## User-Defined Data Types (UDTs)

- Collection data types for one-to-many - UDTs for one-to-one
- 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, the user can alter, verify, and drop a field or the whole data type
- Once created, UDTs may be used to define a column in a table

# User-Defined Data Types (UDT)

```
CREATE TYPE address (
Street text,
Number int,
Flat text);

CREATE TABLE users_w_address(
Userid int,
Location address,
Primary key (userid));

INSERT INTO users_w_address(userid,location) VALUES (1, {street :
'Third', number : 34, flat : '34C'}); //insert data

DROP TYPE address; //we can drop a type
```

- In this example, we create a new data type called ‘address’.
- Once created, we can use it to define a column in our new table called ‘Location’.
- We can insert data using the new ‘address’ data type, and we can also drop the data type.

## Summary

In this video, you learned that:

- Cassandra supports built-in, collection, and user-defined data types
- Both collection and user-defined data types offer a way to group and store data together
- Collection data types can emulate one-to-many relationships
- There are three types of collection data types: lists, maps, and sets
- UDTs can emulate one-to-one relationships
- UDTs allow users to attach multiple data fields to a column

# Keyspace Operations

## Objectives

After completing this video, you will be able to:

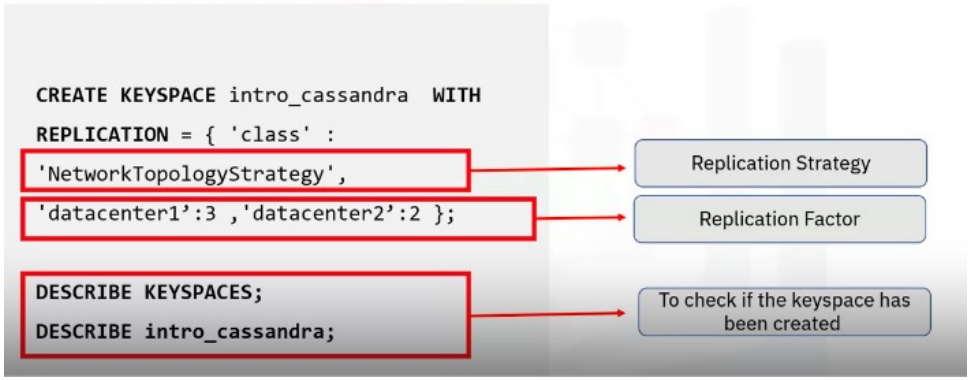
- Describe role of keyspaces in Apache Cassandra
- Describe replication factor and replication strategy
- Create, modify, and delete a keyspace

## Keyspaces

- Keyspace needs to be defined before creating tables
- Keyspace can contain any number of tables, and a table belongs to only one keyspace
- Replication is specified at the keyspace level
- You need to specify the replication factor during the creation of keyspace - can be modified later

- A keyspace needs to be defined before creating tables, as there is no default keyspace.
- A keyspace can contain any number of tables, and a table belongs to only one keyspace.
- Replication is specified at the keyspace level.
- You specify the replication factor during the creation of a keyspace, but the replication factor can be modified later.

## Create Keyspace




- Let's take a CQL 'create keyspace' example – you can see that besides the name we need to set two parameters:
- Class – which refers to the Replication Strategy and the Replication Factor, which is set at each data center level.
- In this example we are creating a keyspace called 'intro_cassandra' that replicates its data (aka tables partitions) 5 times between the cluster nodes; in 32 nodes from data center 1, and 23 nodes from data center 2.
- To check if the keyspace has been created you can use DESCRIBE KEYSPACES or DESCRIBE then the name of the keyspace.

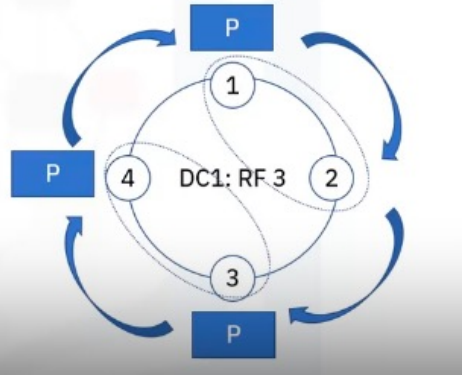
- Replication factor refers to the number of replicas of data placed on different nodes, while Replication Strategy determines on which cluster nodes the replicas are going to be located.
- After data is initially distributed (according to partition key hashed and token pre-allocation) then data is also replicated according to these two pieces of information: replication factor and replication strategy.
- Before we move on to some examples of replication factor and strategy, two important notes regarding the replicas: In Apache Cassandra all replicas are equally important; there are no primary or secondary replicas.
- As a general rule, the replication factor should not exceed the number of nodes in the cluster.

## Data Replication

- Replication Factor
  - Number of replicas placed on different nodes in the cluster
- Replication Strategy
  - Which nodes are going to house the replicas
- Replicas
  - All replicas are equally important - no primary or secondary replicas
  - Replication factor should not exceed the number of cluster nodes

## Single DC Cluster - Rep Factor 3

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

//Node 1 and 2 are in the same rack so
2nd replica is placed in node 3
```

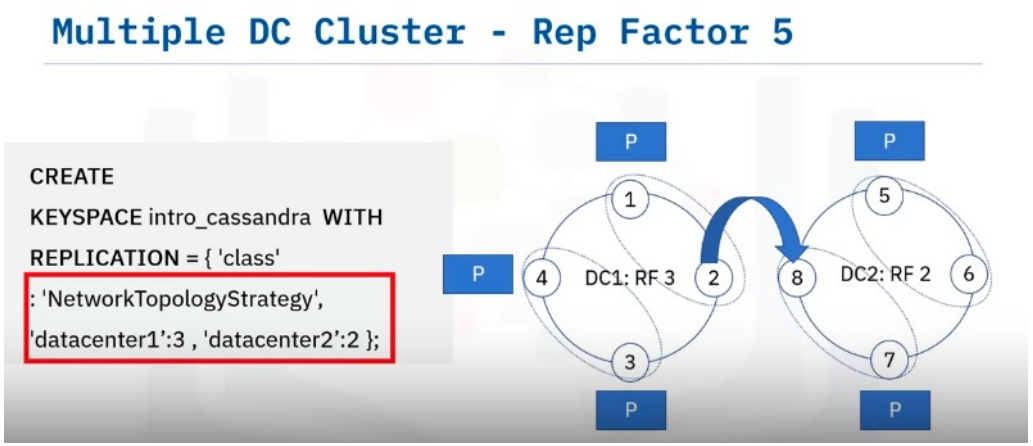



1 is partition Node and 1-2 are in same rack.

Since replication must be done on different node, we need to replicate on 3-4.

If there were many nodes in different racks in single cluster assigning can be done similarly in clockwise fashion.

- Let's take a simple example of a 4-node Cassandra cluster, with one data center and a keyspace called 'intro_cassandra' defined with Replication Factor 3.
- In CQL, when we create a keyspace, we need to specify the class as being Network Topology Strategy (the only option recommended for production systems) and specify the replication at datacenter level (in this case 3).
- Cluster topology indicates that nodes 1 and 2 are in the same rack, while 3 and 4 share another rack as well.
- We assume that **our partition (**named P in the diagram) - has been initally allocated to **Node 1.**
- We have a Replication Factor of 3 – meaning we need to replicate data into 2 more nodes in order to reach the Replication Factor of 3.
- Data Replication is done clockwise in the cluster, while taking into consideration the rack allocation of the servers.
- Since Nodes 1 and 2 are in the same rack, Cassandra will try to place the next replica on a node from a different rack – In this case, Node 3.
- And the last replica will be placed on Node 4. Since two replicas are placed in different racks, the fact that nodes 3 and 4 are in the same rack will not be an issue.




- Let's now take an example in which the cluster is deployed in a multi data center environment: here we have 2 data centers and our keyspace has a replication factor of 5.
- Just like in the previous example, we need to use CQL to create a keyspace.
- And again, we specify the class as being Network Topology Strategy, and we specify the replication factor at the datacenter level as being 32 for datacenter1 and 23 for datacenter2.
- We assume that our partition (named P in the diagram), has been initially allocated to Node 1 in datacenter1 and Node 5 in datacenter2. Just like in the previous example – in Datacenter1 two more replicas are going to be placed clockwise according to the rack placement of the servers.
- Thus nodes 3 and 4 will get the replicas of partition P.
- In our case we have 2 datacenters, so after completing the first datacenter replica placement, Cassandra will go to the next datacenter in order to place the other two replicas.
- Since the partition P has already been allocated initially to node 5 in datacenter2, we need to now place only one more replica in datacenter2. Taking into consideration the rack placement of the nodes – node 7 is chosen to hold the 5th replica of our data.
- This is how our 5 replicas ended up in the Cassandra cluster nodes

## Alter Keyspace

```
ALTER KEYSPACE intro_cassandra
   WITH REPLICATION = {'class' :
'NetworkTopologyStrategy',
'datacenter1' : 3, 'datacenter2' : 3};
```

Modifies the keyspace replication factor - the number of copies of the data Cassandra creates in each data center

## Drop Keyspace

```
DROP KEYSPACE intro_cassandra;
```

- Immediate removal of the keyspace, including objects such as tables, functions, and data it contains
- Cassandra takes a snapshot before dropping the keyspace

- we have already seen examples of using 'CREATE KEYSPACE', but now let's see how we can modify or delete a keyspace.
- You can modify an existing keyspace – for example, modify the replication factor of a keyspace – by using the CQL 'ALTER KEYSPACE' command.
- For example, you can increase the overall keyspace replication factor from 5 to 6, by increasing the replication factor in datacenter2 from 2 to 3.
- You can also **delete (or drop) a keyspace** – by using the 'DROP KEYSPACE' command in CQL.
- This will lead to the **removal of all the keyspace tables and the data those contain.**
- To protect the system against an unwanted DROP, Cassandra will, by default, take a snapshot of the keyspace – prior to executing the DROP.
- This means that you will be able to recover your data in case you need to.

# Summary

In this video you have learned that:

- Keyspaces are defined before creating tables, and a keyspace can contain any number of tables
- Replication is specified at the keyspace level
- Data replication depends on a replication factor and a replication strategy, both set at the keyspace level
- Replication Factor sets the number of replicas
- Replication Strategy determines on which cluster nodes the replicas are going to be located
- Common keyspace operations are CREATE KEYSPACE, ALTER KEYSPACE, and DROP KEYSPACE

## Table Operations

- ❑ Understand the role of Cassandra tables
- ❑ Describe some of the properties of Cassandra tables Create, alter, and delete a table

### CREATE TABLE

```
CREATE TABLE [IF NOT EXISTS] [keyspace_name.]table_name
( column_definition [, ...]
   PRIMARY KEY (column_name [, column_name ...])
[WITH table_options
   | CLUSTERING ORDER BY (clustering_column_name order])
```

```
column_name cql_type_definition [STATIC | PRIMARY KEY] [, ...]
```

- In Cassandra, prior to declaring a table, you first need to create a keyspace – a table will always be created in an existing keyspace.
- Cassandra stores data in tables whose schema defines the storage of the data at the cluster and node level.

- The generic create table syntax looks like this; 'CREATE TABLE' with the optional 'IF NOT EXISTS' parameter followed by the keyspace name (if it has not been already stated by USE keyspace), and then the table name.
- This command creates a new table under the specified keyspace.
- The IF NOT EXISTS keywords may be used in creating a table.
- Attempting to create an existing table returns an error unless the IF NOT EXISTS option is used.
- If the option is used and the table already exists, then the error message is suppressed, and no table is created.
- Columns are defined enclosed in parentheses after the table name, using a comma-separated list to define multiple columns.
- In the column definition you can also specify – optionally – the PRIMARY KEY parameter, which is used to indicate that the column is the only primary key for the table.
- In a case where you have a primary key formed of multiple columns then you declare it at the end of the table

## Static Table example

```
USE intro_cassandra;
CREATE TABLE users(
  username text PRIMARY KEY,
  occupation text,
  age int);
```

```
USE intro_cassandra;
CREATE TABLE users(
  username text,
  occupation text,
  age int,
PRIMARY KEY(username));
```

```
SELECT * FROM users;  //there are no entries in our table so the
SELECT will yield no data;
```

- Remember the users table from earlier in the course – let's now explain the CREATE TABLE statement based on what we just learned.
- Our **table is static** – meaning the **PRIMARY KEY contains only 1 column** partition key – 'username'.
- In this case we can signal the primary key in two ways:
- - just by adding it to the column definition.
- - but the same table would be created if we declared the primary key just like this.
- Once we've created the table, we can use a 'Select' statement, but since we have not inserted any data the select result will be empty.

- Now let's create **a dynamic table** as well; remember the groups table from earlier in the course?
- A dynamic table means that **our Primary Key** is composed of both a Partition Key and a Clustering Key.
- In our case the partition key is the column 'groupid' and the clustering key is the column 'username'.
- You probably have mentioned the static in the 'group_name' column definition.
- The STATIC clause is used for special single-value columns that are shared by all partition rows.
- A static column is also called a descriptive column of the partition key.
- Static columns cannot be used as a primary key.
- When using a Select statement, the same applies for 'groups' as it did for the 'users' table.
- A select on the 'groups' table would yield no result, since we have not yet inserted any data.

## Dynamic Table example

```
USE intro_cassandra;

CREATE TABLE groups(
groupid int,
group_name text STATIC,
username text,
age int,
PRIMARY KEY ((groupid), username));
```

```
SELECT * FROM groups;  //no data
```

# Table Properties and Options

```
DESCRIBE groups;
CREATE TABLE intro_cassandra.groups (
    groupid int,
    username text,
    age int,
    group_name text static,
    PRIMARY KEY (groupid, username)
) WITH CLUSTERING ORDER BY (username ASC)
    AND caching = {'keys': 'ALL', 'rows_per_partition': 'NONE'}
    AND compaction = {'class':
'org.apache.cassandra.db.compaction.SizeTieredCompactionStrategy',
'max_threshold': '32', 'min_threshold': '4'}
    AND default_time_to_live = 0
    AND gc_grace_seconds = 864000
    AND memtable_flush_period_in_ms = 0;
```

- If we describe the table, we can see the table's definition and properties.
- Let's explain some of the new things you can see on this screen: First, by default, data inside the partition is ordered by the clustering key – username – in ascending order.
- The WITH CLUSTERING ORDER BY parameter can be used to order the data inside the partition.
- The default_time_to_live means that you can set an expiration time for the data in the table.
- For example, an offer that is valid only for 5 minutes. After 5 minutes the data is deleted.
- Time To Live(TTL) can also be set at the operation level. For ex, with an INSERT, meaning that only that row will be expired after the TTL.
- We will come back to this in our next video when talking about inserting data into Cassandra tables.
- Write data (such as inserts, updates, or deletes) **is flushed from memory on disk in three situations**: when the Memtable is full, when the CommitLog is full, or at a specific interval.
- This setting refers to the per table setting – specific interval of flushing data to disk.
- By default, this is zero – meaning not activated.
- What you can see in this slide are only a few of the Cassandra table properties and options.
- Some of them: like ‘gc_grace_seconds’ and ‘compaction’ we will come back to in the next video after discussing how data is persisted on disk and how deletes are implemented in Cassandra.

## ALTER TABLE

- Add new columns
- Drop existing columns
- Renames columns
  - Regular columns
  - Clustering keys
- Changes table properties
- Restriction: Altering PRIMARY KEY columns is not supported
- Restriction: Changing data type of an existing column is not supported

```
ALTER TABLE groups ADD occupation TEXT;

ALTER TABLE groups DROP age;

ALTER TABLE groups RENAME username to
user_name;

ALTER TABLE groups WITH
default_time_to_live=10;
```

- after discussing how data is persisted on disk and how deletes are implemented in Cassandra.
- As in the case with keyspaces earlier, tables can also be altered.
- You can add new columns to the table schema drop columns from the table schema rename columns (be aware that it works for regular columns and clustering keys, but is not supported for partition keys) change table properties Please note that while we are mentioning adding and deleting columns, these operations can be applied only to regular columns, not to Primary Key columns.
- In Cassandra the primary key, once defined, cannot be changed – since it determines the way the data is stored at cluster and node level.
- If you want to change it, you will need to create a new table and re-import the data.
- Changing the data type for an existing column is not supported.

## TRUNCATE & DROP TABLE

- Truncate: removes all data from a table, but not the table's schema definition
- Truncate: needs Consistency ALL and all replicas available
- Drop: removes all the data, including schema
- Cassandra takes a snapshot before truncating or dropping the table

```
TRUNCATE TABLE groups;
TRUNCATE groups;
DROP TABLE groups;
```

- You can delete a table using either the TRUNCATE or DROP commands.
- Truncate removes all the data from the specified table, but not the table's defined schema.
- One thing to be aware of with the TRUNCATE command is that it requires an ALL consistency, thus all replicas need to be available.
- The DROP command removes all data and the table's defined schema.
- Before truncating or dropping the data, Cassandra takes a snapshot of the data as a backup.

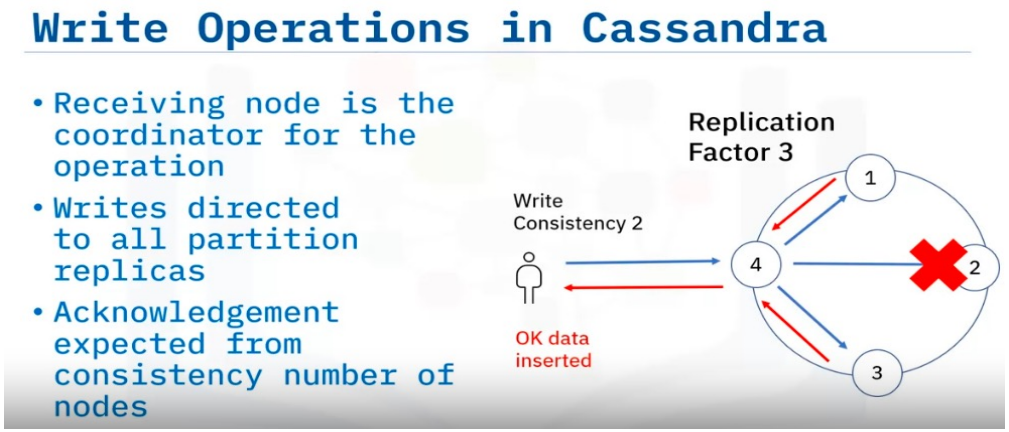
# Summary

In this video you have learned that:

- Data in Cassandra is organized logically in tables
- A table's metadata specifies the primary key – instructing Cassandra how to distribute the table data at cluster and node level
- You can add Time-To-Live at table level – meaning that you can expire (delete) all data that has surpassed the TTL
- You can modify the columns and column names but only for regular columns
- Primary Key, once defined at table creation, cannot be modified

## CRUD Operations - Part 1

- ❑ Part 1, focused on how we can insert and update data in Cassandra.
- ❑ Understand the Write process in Cassandra
- ❑ Explain the INSERT and UPDATE operations
- ❑ Explain the role of Lightweight transactions

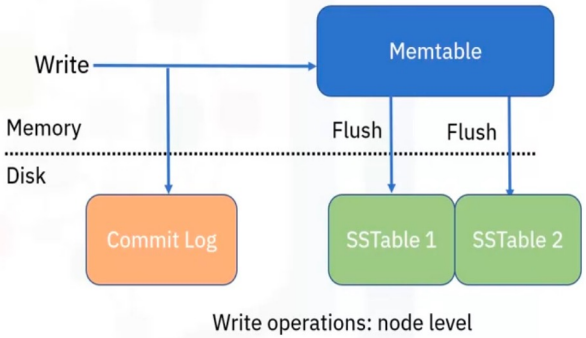



- let's see how a write is done at the cluster and node level.
- At cluster level when a write occurs, the node receiving the write becomes the coordinator of the operation.
- This means that it will make sure to complete the operation and send the result of the write back to the user.

- Write operations are directed towards all replicas of the partition into which they are writing, but for the operation to be successful, acknowledgement is expected from at least the minimum number of nodes specified for consistency.
- Let's take the example of a keyspace with replication factor 3 and a write operation with consistency set to 2.
- We assume our partition is located in nodes 1, 2, and 3.
- In this case the write operation arrives in node 4, and node 4 sends the write to nodes 1, 2, and 3 according to the replication factor.
- However, node 2 is not available, so the write acknowledgement is sent only by nodes 1 and 3.
- Coordinator node 4 checks the number of answers and compares it to the expected consistency value, and then returns an okay

# Write Operations in Cassandra

- No reading before writing (by default)
- At node level
  - Writes are done in memory and later flushed on disk (SSTables)
  - Every flush => new SSTable
  - All disk writes are sequential ones. Data will be reconciled through Compaction
- Cassandra attaches a Timestamp to every write



Write operations: node level

- At the node level there are a few important facts to remember: One important reminder regarding writes in Cassandra: by default, **there's no read before the write operation.**
- At the node level, writes are stored in memory and then flushed on disk in files called SSTables.
- The more writes we perform, the faster the Memtable gets filled and flushes the data on disk.
- **Every flush operation creates a new file** called SSTable.
- On disk data is appended in subsequent SSTables – later on the **data will be optimized on disk** through a process called compaction.
- Cassandra attaches a Timestamp to every write operation.
- Timestamps are used for data reconciliation. The most recent data wins.

## INSERT

- Insert operations require full primary key
- Cassandra doesn't perform read before write: an INSERT can behave as an UPSERT
  - If we INSERT data in an existing entry, data will be UPDATED
- Inserts require a value for all primary key columns, but not other columns
  - Only specified columns will be inserted/updated
- You can INSERT/UPDATE data with Time-To-Live

- Insert operations **require the full primary key to be specified.**
- This means inserts can only be done record by record.
- Since Cassandra, by default, doesn't perform a read before the write operation, an Insert operation **behaves both as an 'Insert' and an 'Upsert'** operation.
- If we insert data on an existing entry, then data will be updated.
- Insert operations require a value for all the primary key columns, but not for the other regular columns specified in the table definition.
- Only the specified columns will be provisioned with data.
- You can insert data with a specific Time-To-Live, just like we have done at the table level.
- Now we can do this at record level.
- This means that this data will be visible only for a defined time.

# INSERT

```
INSERT INTO groups(groupid,username,group_name,age)
VALUES (12,'aland@gmail.com','baking',32);

INSERT INTO groups(groupid,username,group_name,age)
VALUES (12,'Elaine@yahoo.com','baking',60);

INSERT INTO groups(groupid,username)
VALUES (45,'Moirad@yahoo.com');

INSERT INTO groups(groupid,username,group_name)
VALUES (25,'Johns@gmail.com','vegan cooking') USING
TTL 10;

INSERT INTO groups(groupid,username,group_name,age)
VALUES (12,'aland@gmail.com','baking',45);
```

- Let's see a few examples based on our data.
- In bold you can see the columns of the primary key: We can insert new data in our table.
- As you can see, we have specified all the table's columns.
- In this example we have inserted two users in group 12, the baking group.
- When inserting data, the Primary key is mandatory.
- You cannot insert data without fully specifying it.
- As you can see in this example where we insert a new user in group 45.
- We only added the mandatory information.
- Neither the group name nor the user's age had been added.
- We can also insert data with a Time-To-Live (or TTL).
- In this case we added a new user to group 25, 'vegan cooking' with a TTL of 10 seconds.
- This means that in 10 seconds (from insertion) the data will not be available for query anymore.
- We can use an INSERT as an update when the INSERT is done on existing data.
- In this case, the age of the user in group 12 will be updated to 45.

# UPDATE

```
groupid | username          | group_name | age
--------+-------------------+------------+------
     45 | Moirad@yahoo.com  |       null | null
     12 | Elaine@yahoo.com  |     baking |   60
     12 |  aland@gmail.com  |     baking |   32
```

```
UPDATE groups SET group_name = 'grilling' WHERE groupid = 45;

UPDATE groups SET AGE = 60 WHERE groupid = 12 and username
='aland@gmail.com';

UPDATE groups SET age = 55, group_name = 'coffee' WHERE groupid = 20
and username = 'bella@yahoo.com';
```

- If we do a 'Select' on our table this is what it is going to look like, after all the inserts.
- This is a cqlsh view of the data, so don't be misled by the 'null' value.
- In this case it means that these cells have no data.
- Let's do two updates to our data: Let's update the name of group 45.
- Because it is a static column we can update it using only the partition key.
- This is the only situation in which the UPDATE command does not require a full primary key to be mentioned.
- We can also update an existing record.
- In this case, we will update the age of a user in group 12.
- **We can call the UPDATE command on a non-existing entry and in this case, UPDATE will behave as an INSERT.**

## Lightweight transactions (LWT)

```
 groupid | username          | group_name | age
---------+-------------------+------------+------
      45 | Moirad@yahoo.com |   grilling | null
      20 |  bella@yahoo.com |     coffee |   55
      12 | Elaine@yahoo.com |     baking |   60
      12 |  aland@gmail.com |     baking |   60
```

```
UPDATE groups SET AGE = 62 WHERE groupid=12 and username =
'aland@gmail.com' IF EXISTS; // TRUE, age will be updated

UPDATE groups SET AGE = 63 WHERE groupid=12 and username =
'aland@gmail.com' IF age = 62; //TRUE, age will be updated

INSERT INTO groups(groupid,username,group_name,age) VALUES
(12,'Elaine@yahoo.com','baking',60) IF NOT EXISTS; //FALSE, no insert
```

## Lightweight transactions (LWT)

Lightweight transactions are at least four times slower than the normal INSERT/UPDATE in Cassandra. Use them sparingly in your application.

- Let's see what a Select on our table would yield at this moment.
- As you have seen, INSERT and UPDATE can behave in a fairly similar manner, given that by default, Cassandra doesn't locate and read data before executing a WRITE.
- However, it is possible to instruct Cassandra to look for the data, read it, and only then perform a given operation. This is possible through **(LWT)**.
- Syntax-wise, Lightweight Transactions are supported by introducing IF in INSERT and UPDATE statements.
- Let's see some examples: We can update the age of a user in a group only if that record exists.
- We can update the age of a user in a group only if the record exists and the age has a certain value.
- We can insert data into a Cassandra table only if that data does not exist.
- In our case the record exists, and in this case no INSERT will be performed.

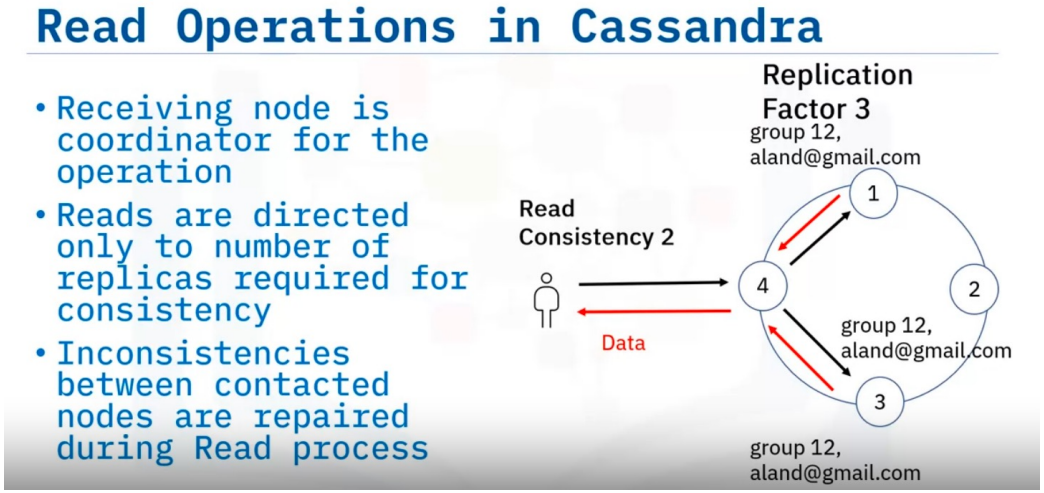
# Summary

In this video you learned that:

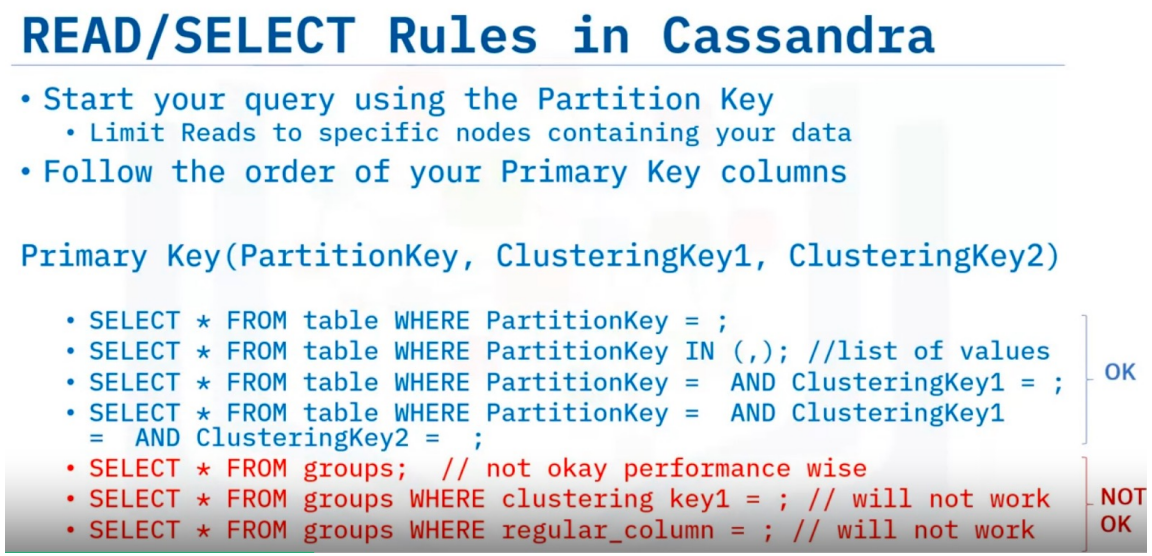
- Cassandra doesn't perform a read before writes by default, therefore INSERT and UPDATE operations behave similarly
- Lightweight Transactions can be used in order to enforce a read before write
- Cluster-level writes are sent to all partition replicas - Only number of nodes according to consistency are needed

## CRUD Operations - Part 2

- ❑ Understand the Read process in Cassandra clusters.
- ❑ Understand the DOs and DON'Ts of Select statements.
- ❑ Delete data in Cassandra.

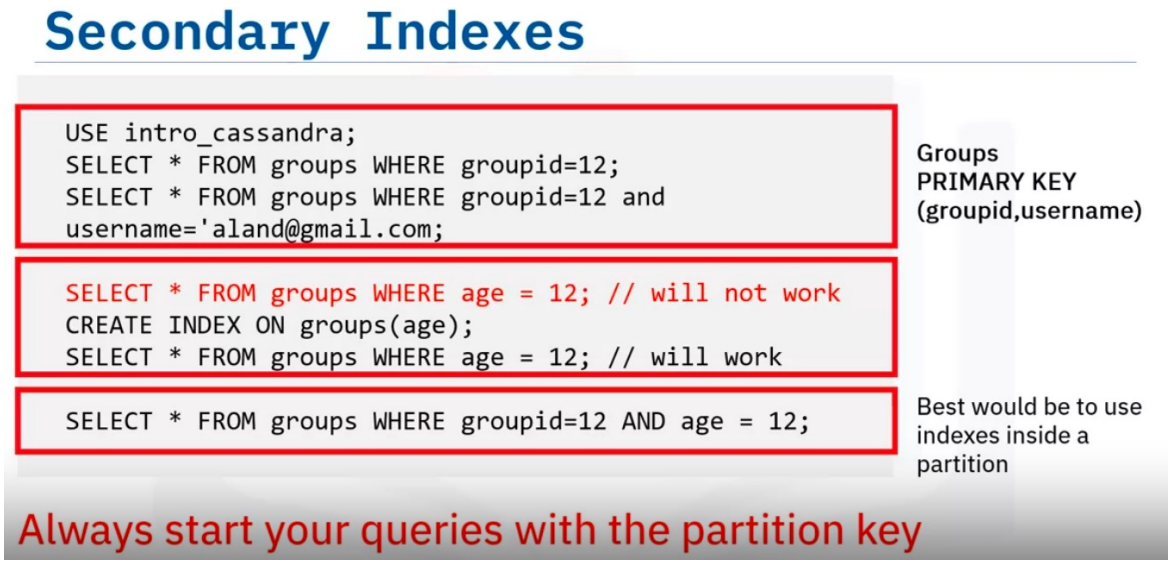



- When a Read operation is directed to a node in the cluster:
- That **receiving node becomes the coordinator** of the Read operation and is responsible for its completion.
- In our example, node 4 is the coordinator for this Read.
- Reads are sent only to the number of replicas specified by the consistency setting.
- For example, a **consistency of 2** means that only two nodes of the three replicas will be contacted.
- The coordinator will reconcile the responses received from the contacted nodes.
- If there are any **inconsistences,** they will be **resolved based** on the operation's timestamps.
- Then, the result will be sent back.



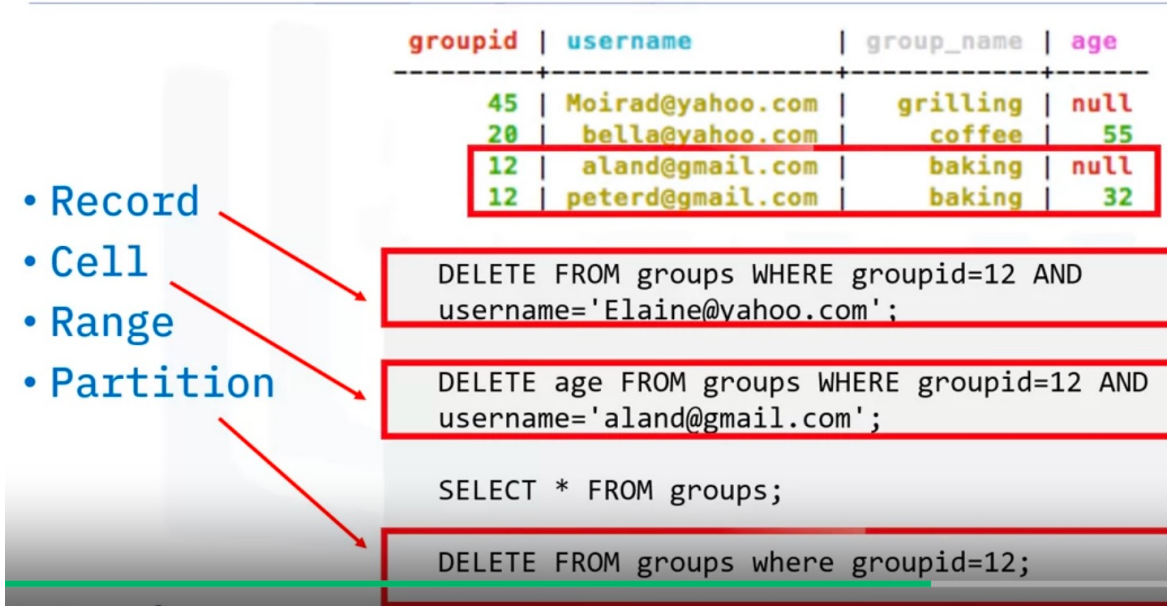

- Syntax-wise, in Cassandra, just like in many databases, **Reads** are performed using the **SELECT operation**.
- There are a few rules when it comes to Select operations in Cassandra: Always start your query with the partition key to limit your Read to only the replicas for your partition.
- Follow your primary key fields order in your query to get the best performance.

- For example, if we have a Primary Key formed of one Partition Key and two Clustering Keys: Filtering a specific value of the Partition Key is okay. (Note that Partition Key supports only '=' and 'IN' operations.)
- Filtering a list of possible values for the Partition Key is also okay.
- Filtering the Partition Key and the first Clustering Key is okay.
- Filtering the Partition Key and the first and second Clustering Keys will just read one record, which is okay.
- But selecting all data from your table is not okay in production systems, because this will send the request to all nodes in the cluster.
- Imagine you have a 1000-node cluster, would you be happy with the performance of your query? Your query will work, but the performance will be really bad, even if no timeout occurs.
- a regular column is not okay and will not work.
- And finally, filtering the clustering keys without the partition key is not okay and will not work.



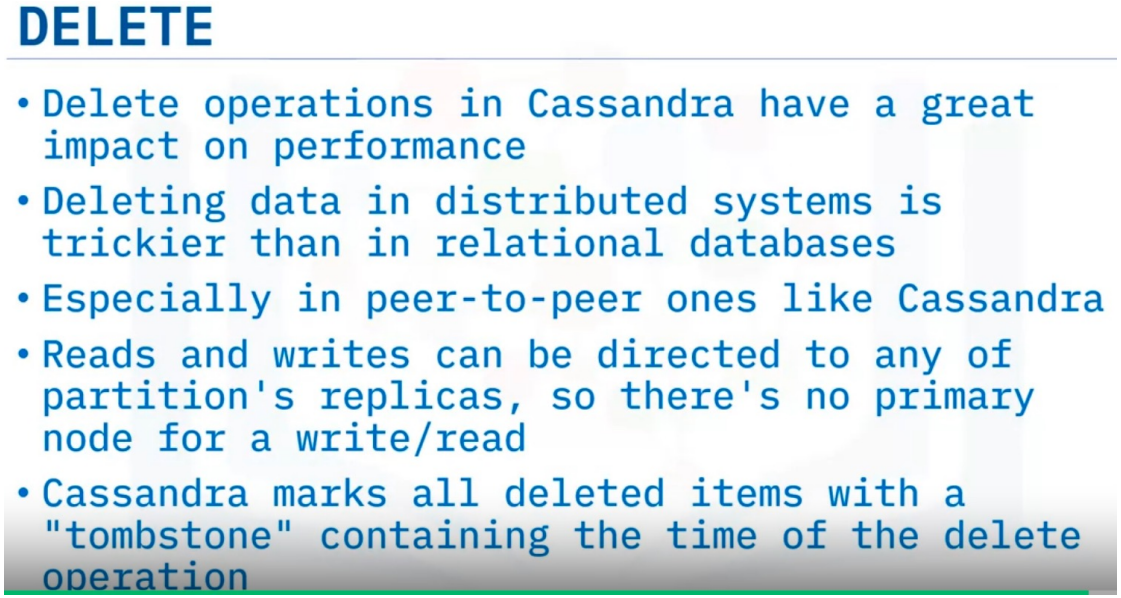

- Let's look at some examples based on our ‘groups’ table, where the primary key is composed of ‘groupid’ and ‘username’ columns.
- Filtering the ‘groupid’ or ‘groupid’ and ‘username’ will work and yield a very good performance.
- Filtering the age column, which is a regular one, will not work. Cassandra will return an error.
- What you could do in this case is either remodel your table or create an index on column age.
- And then the select query will work. But while the previous query will work, optimal performance will be obtained by filtering the age column together with a specific partition key: ‘groupid’.
- When you do this, you limit your query to only the nodes that host that particular partition key.
- And this brings us again to this very important Cassandra rule: Always start your queries with the partition key.

# DELETE

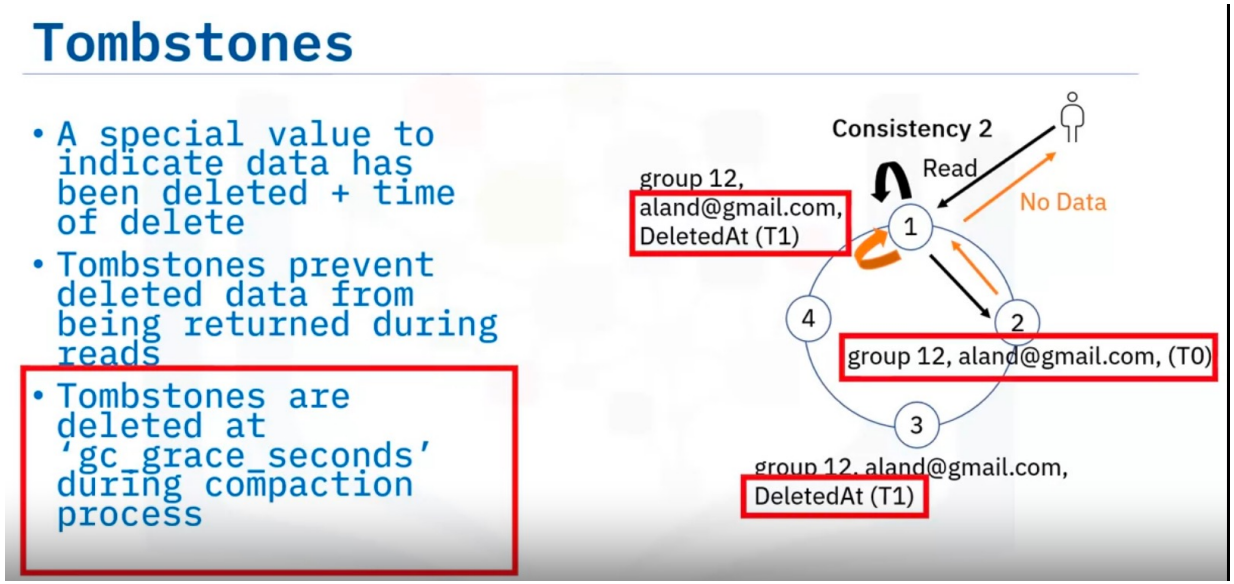



- We're starting off with some data to demonstrate some deletes.
- In Cassandra you can **delete: an entry identified by the full primary key**.
- a cell in an entry identified by the full primary key.

- If we perform a select from our table, you can see that we do not have the user called Elaine anymore, and the age cell has been deleted for the user called Alan.
- Deletes can also be done at the partition level. You can delete a continuous range in a partition.
- If your partition is 'sensorid' and your data is clustered by the time of the recording, you can, for example, delete all entries on a sensor between 1 PM and 3 PM or you can delete a whole partition, which is identified by a partition key




- We've seen the syntax of deleting data. But, in Cassandra you should use Delete operations sparingly, because their usage greatly affects system performance.
- Deleting distributed and replicated data from a system such as Apache Cassandra is far trickier than in a relational database, especially if we remember that Cassandra is a peer-to-peer architecture, where reads and writes can be directed to any node in the cluster, so there's no primary node for writes or reads.
- In such a system, to record the fact that a delete happened, a special value called **a "tombstone"** needs to be written as an indicator that previous values are to be considered deleted.
- From the moment an item has been marked with a tombstone its data is not visible any longer to queries, but the actual data will still reside on disk for a period of time.




- What are tombstones? In Cassandra, a Delete is just a Write operation that also has a special value appended to indicate that the **data has been deleted and the time of the delete**. This value is called a tombstone.
- As you can see in the diagram, our **Delete was acknowledged** by nodes 1 and 2, and now we have a Tombstone indicating that our specific Primary Key data was deleted at T1.

- Node 2 data is the initial one, the insert done at T0, since the delete operation did not succeed on Node2.
- Now, when a **Read operation** arrives and the data will be returned by nodes 1 and 2, node 1 will return no data with time T1, and node 2 will return requested data with time T0.
- T1 is more recent than T0, so the coordinator node (node 1), will decide that the newest data is the one from node 1, and return no data as part of the Read.
- One other thing to remember is that tombstones are deleted only when a configurable period has elapsed.
- This is called 'gc_grace_seconds', which is set to 10 days by default, and is set at the table level.

Delete operation at T0 and Read operation at T1 takes place.
In reading Node1 provides No Data with T1 stamp and Node2 Provides result with T0 stamp. Means only Node1 has acknoledged Delete operation.
But read operation at T1 is most recent one with the 'No Data' as per Node1. Means Delete has been successfully done at Node1.

**Summary and Highlights**

Congratulations! You have completed this lesson. At this point in the course, you know:

•Cassandra supports built-in, collection, and user-defined data types.

•Both collection and user-defined data types offer a way to group and store data together.

•Keyspaces are defined before creating tables, and a keyspace can contain any number of tables.

•Common keyspace operations are CREATE KEYSPACE, ALTER KEYSPACE, and DROP KEYSPACE.

•Cassandra organizes data logically in tables.

•A table’s metadata specifies the primary key – instructing Cassandra how to distribute the table data at the cluster and node level.

•Cluster level writes are sent to all the partition's replicas, irrespective of the consistency factor.

•By default, Cassandra doesn't perform a read before writes, therefore INSERT and UPDATE operations behave similarly.

•Reads at cluster level are sent only to the number of replicas according to the consistency setting.

•Reads should follow the Primary Key columns order for best performance.