Data Modeling

Course 1 | Data Modeling

What is a Data Model?

"...an abstraction that **organizes elements of data** and **how they** will **relate** to each other"

Lesson 1

(Relational [Postgres] & NoSQL [Apache Cassandra] DBs)

▼ **Relational Databases**

ACID transactions are properties of database transactions intended to guarantee validity even in the event of errors or power failures.

- **Atomicity:** The whole transaction is processed or nothing is processed
- **Consistency:** Only transactions that abide by constraints and rules are written into the database, otherwise the database keeps the previous state
- **Isolation:** Transactions are processed independently and securely, order does not matter
- **Durability:** Completed transactions are saved to database even in cases of system failure

When Not to Use a Relational Database

- **Have large amounts of data:** Relational Databases are not distributed databases and because of this they can only scale vertically by adding more storage in the machine itself. You are limited by how much you can scale and how much data you can store on one machine. You cannot add more machines like you can in NoSQL databases.
- **Need to be able to store different data type formats:** Relational databases are not designed to handle unstructured data.
- **Need high throughput -- fast reads:** While ACID transactions bring benefits, they also slow down the process of reading and writing data. If you need very fast reads and writes, using a relational database may not suit your needs.
- **Need a flexible schema:** Flexible schema can allow for columns to be added that do not have to be used by every row, saving disk space.
- **Need high availability:** The fact that relational databases are not distributed (and even when they are, they have a coordinator/worker architecture), they have a single point of failure. When that database goes down, a fail-over to a backup system occurs and takes time.
- **Need horizontal scalability:** Horizontal scalability is the ability to add more machines or nodes to a system to increase performance and space for data.

PostgreSQL (Link) is an open-source object-relational database system.

- PostgreSQL uses and builds upon SQL database language by providing various features that reliably store and scale complicated data workloads.
- PostgreSQL SQL syntax is different than other relational databases SQL syntax.

DEMOs

PostgreSQL and AutoCommits (Link)

```
## DEMO 0
## import postgreSQL adapter for the Python
import psycopg2
## Create a connection to the database
conn = psycopg2.connect("host=127.0.0.1 dbname=studentdb user=student password=student")
## Use the connection to get a cursor that will be used to execute queries
cur = conn.cursor()
## IMP: autocommitting transactions to avoid getting blocked and restarting connection
conn.set_session(autocommit=True)
##creating table
cur.execute("CREATE TABLE test (col1 int, col2 int, col3 int);")
## parsing table
cur.execute("select * from test")
##table is empty, checking dimensions
cur.execute("select count(*) from test")
print(cur.fetchall())

## DEMO 1
## Adding the try except will make sure errors are caught and understood
try:
    conn = psycopg2.connect("host=127.0.0.1 dbname=studentdb user=student password=student")
except psycopg2.Error as e:
    print("Error: Could not make connection to the Postgres database")
    print(e)

##Use the connection to get a cursor that can be used to execute queries
try:
    cur = conn.cursor()
except psycopg2.Error as e:
    print("Error: Could not get curser to the Database")
    print(e)

##enable autocommit
conn.set_session(autocommit=True)

##create database to work on
try:
    cur.execute("create database udacity")
except psycopg2.Error as e:
    print(e)

## create table for music records, 3 fields
try:
    cur.execute("CREATE TABLE IF NOT EXISTS music_library (album_name varchar, artist_name varchar, year int);")
except psycopg2.Error as e:
    print("Error: Issue creating table")
    print (e)

##check if table has been created and print dimensions - will be emtpy as we didn't add
try:
    cur.execute("select count(*) from music_library")
except psycopg2.Error as e:
    print("Error: Issue creating table")
    print (e)

print(cur.fetchall())

##insert two rows
## postgres allows for duplicate in case you run this twice
cur.execute("INSERT INTO music_library (album_name, artist_name, year) \
                VALUES (%s, %s, %s)", \
                ("Let It Be", "The Beatles", 1970))

cur.execute("INSERT INTO music_library (album_name, artist_name, year) \
                VALUES (%s, %s, %s)", \
                ("Rubber Soul", "The Beatles", 1965))

##validate data was inserted in the table
cur.execute("SELECT * FROM music_library;")

row = cur.fetchone()
while row:
   print(row)
   row = cur.fetchone()

##drop the table to avoid duplicates and clean up
cur.execute("DROP table music_library")

##close cursor and connection
```

```
cur.close()
conn.close()
```

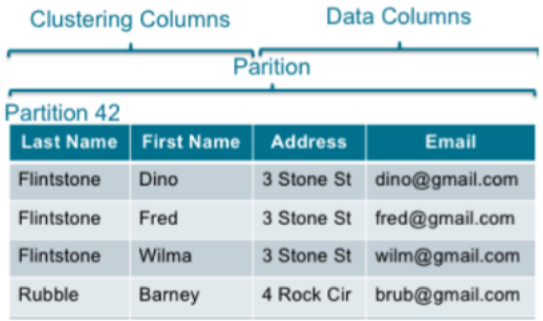
NoSQL Databases

NoSQL Database Implementations:

- Apache Cassandra (Partition Row store)
- MongoDB (Document store)
- DynamoDB (Key-Value store)
- Apache HBase (Wide Column Store)
- Neo4J (Graph Database)

The Basics of Apache Cassandra

- Partition
  - Fundamental unit of access
  - Collection of row(s)
  - How data is distributed
- Primary Key
  - Primary key is made up of a partition key and clustering columns
- Columns
  - Clustering and Data
  - Labeled element



Last Name	First Name	Address	Email
Flintstone	Dino	3 Stone St	dino@gmail.com
Flintstone	Fred	3 Stone St	fred@gmail.com
Flintstone	Wilma	3 Stone St	wilm@gmail.com
Rubble	Barney	4 Rock Cir	brub@gmail.com

When to use a NoSQL Database

- **Need to be able to store different data type formats**: NoSQL was also created to handle different data configurations: structured, semi-structured, and unstructured data. JSON, XML documents can all be handled easily with NoSQL.
- **Large amounts of data**: Relational Databases are not distributed databases and because of this they can only scale vertically by adding more storage in the machine itself. NoSQL databases were created to be able to be horizontally scalable. The more servers/systems you add to the database the more data that can be hosted with high availability and low latency (fast reads and writes).
- **Need horizontal scalability**: Horizontal scalability is the ability to add more machines or nodes to a system to increase performance and space for data
- **Need high throughput**: While ACID transactions bring benefits they also slow down the process of reading and writing data. If you need very fast reads and writes using a relational database may not suit your needs.
- **Need a flexible schema**: Flexible schema can allow for columns to be added that do not have to be used by every row, saving disk space.
- **Need high availability**: Relational databases have a single point of failure. When that database goes down, a failover to a backup system must happen and takes time.

When NOT to use a NoSQL Database?

- **When you have a small dataset**: NoSQL databases were made for big datasets not small datasets and while it works it wasn't created for that.
- **When you need ACID Transactions**: If you need a consistent database with ACID transactions, then most NoSQL databases will not be able to serve this need. NoSQL database are eventually consistent and do not provide ACID

transactions. However, there are exceptions to it. Some non-relational databases like MongoDB can support ACID transactions.

- **When you need the ability to do JOINS across tables**: NoSQL does not allow the ability to do JOINS. This is not allowed as this will result in full table scans.
- **If you want to be able to do aggregations and analytics**
- **If you have changing business requirements** : Ad-hoc queries are possible but difficult as the data model was done to fix particular queries
- **If your queries are not available and you need the flexibility** : You need your queries in advance. If those are not available or you will need to be able to have flexibility on how you query your data you might need to stick with a relational database

DEMO 2 - Creating Table with Cassandra (NoSQL DBs) (Link)

```
! pip install cassandra-driver
import cassandra

# Create a connection to the database:
#Connect to the local instance of Apache Cassandra *['127.0.0.1']*.
#The connection reaches out to the database (studentdb) and uses the correct privileges to connect to the database (user and password =
#Once we get back the cluster object, we need to connect and that will create our session that we will use to execute queries.

from cassandra.cluster import Cluster
cluster = Cluster(['127.0.0.1']) #If you have a locally installed Apache Cassandra instance
session = cluster.connect()

#Test the connection and error handling code
session.execute("""select * from music_libary""")
#Error from server: code=2200 [Invalid query] message="No keyspace has been specified. USE a keyspace, or explicitly specify keyspace.t

#Create a keyspace to work in (Schema in postgres/DB)
session.execute("""
    CREATE KEYSPACE IF NOT EXISTS udacity
    WITH REPLICATION =
    { 'class' : 'SimpleStrategy', 'replication_factor' : 1 }"""
  )

#connect to keyspace
session.set_keyspace('udacity')

#before modeling data now, we need to know how the DB will be used (what queries will be done?)
#i.e. We want to be able to get every album that was released in a particular year.
# select * from music_library WHERE YEAR=1970
#To do that:
  #We need to be able to do a WHERE on YEAR.
  #YEAR will become my partition key,
  #artist name will be my clustering column to make each Primary Key unique.
  #Remember there are no duplicates in Apache Cassandra.

query = "CREATE TABLE IF NOT EXISTS music_library (year int, artist_name text, album_name text, PRIMARY KEY (year, artist_name)) "
session.execute(query)

#query will run smoothly, let's insert rows
query = "INSERT INTO music_library (year, artist_name, album_name)"
query = query + " VALUES (%s, %s, %s)"

session.execute(query, (1970, "The Beatles", "Let it Be"))
session.execute(query, (1965, "The Beatles", "Rubber Soul"))

#validate that data was inserted correcting (for loop not needed in cql - cassandra query language)
query = 'SELECT * FROM music_library'
try:
    rows = session.execute(query)
except Exception as e:
    print(e)

for row in rows:
    print (row.year, row.album_name, row.artist_name)

#run the aggregate query to retrieve the desired record
query = "select * from music_library WHERE YEAR=1970"
```

```
#drop table and close connection
query = "drop table music_library"
rows = session.execute(query)

session.shutdown()
cluster.shutdown()
```

Lesson 2

(Postgres: Normalization, Denormalization, Fact/dimension tables, Different schema models)

▼ **OLAP/OLTP**

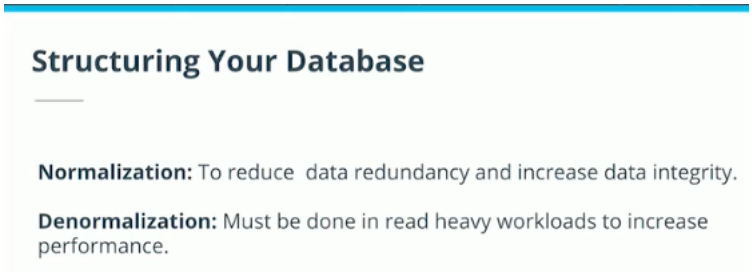
Online Analytical Processing (OLAP):

Databases optimized for these workloads allow for complex analytical and ad hoc queries, including aggregations. These type of databases are optimized for reads.

Online Transactional Processing (OLTP):

Databases optimized for these workloads allow for less complex queries in large volume. The types of queries for these databases are read, insert, update, and delete.

The key to remember the difference between OLAP and OLTP is analytics (A) vs transactions (T). If you want to get the price of a shoe then you are using OLTP (this has very little or no aggregations). If you want to know the total stock of shoes a particular store sold, then this requires using OLAP (since this will require aggregations). (Link)



▼ **Normalization**

data redundancy: repetitiveness in data, want to have this in only one column in one table to create one source of truth
data integrity: assurance that querying the data will retrieve the truth/reality (denormalised DBs may have the issue of not having all the source updated therefore potentially compromising integrity)
A series of **normal forms** to abide by ensures reduction of data redundancy and increase of data integrity in relational dbs.

Objectives of Normal Form:

1. To free the database from unwanted insertions, updates, & deletion dependencies
2. To reduce the need for refactoring the database as new types of data are introduced
3. To make the relational model more informative to users
4. To make the database neutral to the query statistics

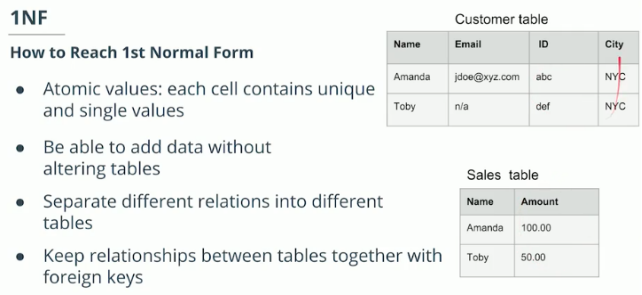
Steps on Normalization:

Normal Forms

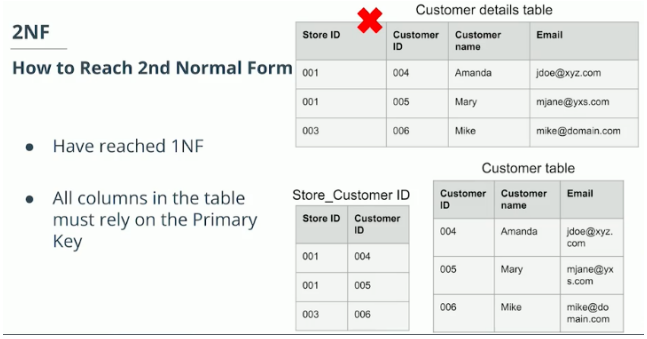
The process of normalization is a step by step process:

- First Normal Form (1NF)
- Second Normal Form(2NF)
- Third Normal Form (3NF)

1. **How to reach First Normal Form (1NF):**
    - Atomic values: each cell contains unique and single values
    - Be able to add data without altering tables
    - Separate different relations into different tables
    - Keep relationships between tables together with foreign keys



2. **Second Normal Form (2NF):**
    - Have reached 1NF
    - All columns in the table must rely on the Primary Key



3. **Third Normal Form (3NF):**
    - Must be in 2nd Normal Form
    - No transitive dependencies
    - Remember, transitive dependencies you are trying to maintain is that to get from A-> C, you want to avoid going through B.

When to use 3NF:

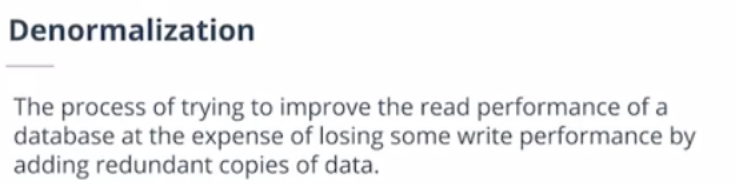
- When you want to update data, we want to be able to do in just 1 place. We want to avoid updating the table in the Customers Detail table (in the example in the lecture slide).

DEMO - Creating Normalized Tables

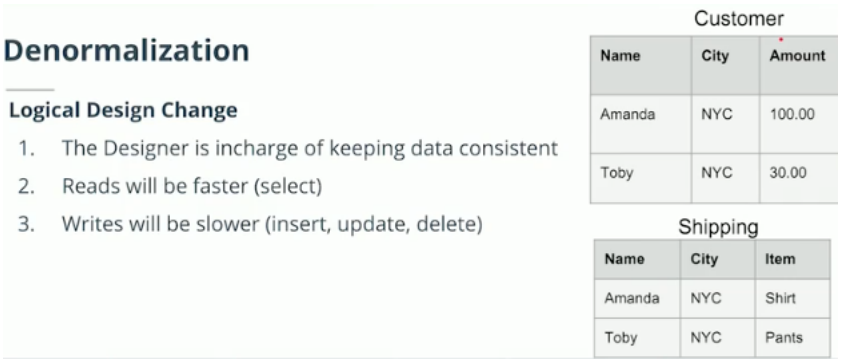
```
import psycopg2
#!echo "alter user student createdb;" | sudo -u postgres psql
conn = psycopg2.connect("host=127.0.0.1 dbname=studentdb user=student password=student")
cur = conn.cursor()
conn.set_session(autocommit=True)

#Let's imagine we have a table called Music Library.
#Table Name: music_library
#column 0: Album Id
#column 1: Album Name
#column 2: Artist Name
#column 3: Year
#column 4: List of songs <- denormalise here for 1NF

#create table
cur.execute("CREATE TABLE IF NOT EXISTS music_library (album_id int, \
                                                       album_name varchar, artist_name varchar, \
                                                       year int, songs text[]);")
#insert row, repeat for all the rows
cur.execute("INSERT INTO music_library (album_id, album_name, artist_name, year, songs) \
                VALUES (%s, %s, %s, %s, %s)", \
                (1, "Rubber Soul", "The Beatles", 1965, ["Michelle", "Think For Yourself", "In My Life"]))
cur.execute("SELECT * FROM music_library;")
row = cur.fetchone()
while row:
   print(row)
   row = cur.fetchone()

#now we need to move to 1NF by breaking the song list
#Table Name: music_library2
#column 0: Album Id       <-duplicated
#column 1: Album Name     <-duplicated
#column 2: Artist Name.   <-duplicated
#column 3: Year           <-duplicated
#column 4: Song Name.    <- now individual song

cur.execute("CREATE TABLE IF NOT EXISTS music_library2 (album_id int, \
                                                       album_name varchar, artist_name varchar, \
                                                       year int, song_name varchar);")
#repeat this for every SONG now
cur.execute("INSERT INTO music_library2 (album_id, album_name, artist_name, year, song_name) \
                VALUES (%s, %s, %s, %s, %s)", \
                (1, "Rubber Soul", "The Beatles", 1965, "Michelle"))
cur.execute("SELECT * FROM music_library2;")
row = cur.fetchone()
while row:
   print(row)
   row = cur.fetchone()

#moving to 2NF, album_id is not unique yet
#Table Name: album_library
```

```
#column 0: Album Id
#column 1: Album Name
#column 2: Artist Name
#column 3: Year
#Table Name: song_library
#column 0: Song Id
#column 1: Song Name
#column 3: Album Id

#create 2 tables
cur.execute("CREATE TABLE IF NOT EXISTS album_library (album_id int, \
                                                       album_name varchar, artist_name varchar, \
                                                       year int);")
cur.execute("CREATE TABLE IF NOT EXISTS song_library (song_id int, album_id int, \
                                                      song_name varchar);")
#insert all albums in the first, repeat for all albums
cur.execute("INSERT INTO album_library (album_id, album_name, artist_name, year) \
                VALUES (%s, %s, %s, %s)", \
                (1, "Rubber Soul", "The Beatles", 1965))
#insert all songs into the second, repeat for all songs
cur.execute("INSERT INTO song_library (song_id, album_id, song_name) \
                VALUES (%s, %s, %s)", \
                (1, 1, "Michelle"))

#to get the all the info back now we can JOIN these 2 tables
cur.execute("SELECT * FROM album_library JOIN\
                song_library ON album_library.album_id = song_library.album_id ;")

#moving to 3NF -> check for transitive dependencies
# Album_library can move Artist_name to its own table, called Artists, which will leave us with 3 tables

#after creating the tables like code above, we can join to get it back
cur.execute("SELECT * FROM (artist_library JOIN album_library2 ON \
                            artist_library.artist_id = album_library2.artist_id) JOIN \
                            song_library ON album_library2.album_id=song_library.album_id;")
#we reached 3NF!
#drop tables and close cur and connection
cur.execute("DROP table music_library") #repeat
cur.close()
conn.close()
```

▼ **Denormalization**



JOINS on the database allow for outstanding flexibility but are extremely slow. If you are dealing with heavy reads on your database, you may want to think about denormalizing your tables. You get your data into normalized form, and then you proceed with denormalization. So, denormalization comes after normalization.



▼ **Facts & Dimensions - Schemas**

Fact and Dimension Tables

- Work together to create an organized data model
- While fact and dimension are not created differently in the DDL, they are conceptual and extremely important for organization.

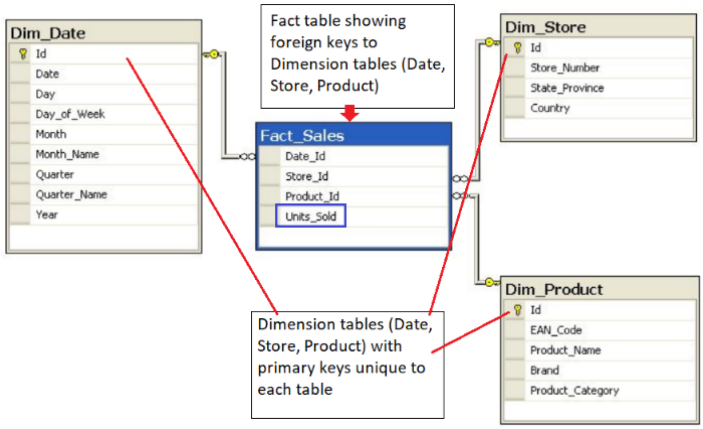
Fact Tables

Fact table consists of the measurements, metrics or facts of a business process.

Dimension

A structure that categorizes facts and measures in order to enable users to answer business questions. Dimensions are people, products, place and time.

Star Schema



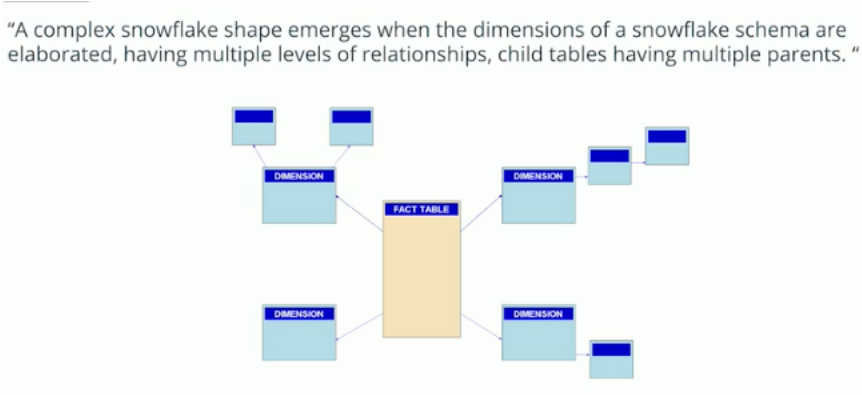
Star Schema is the simplest style of data mart schema. The star schema consists of one of more fact tables referencing any number of dimension tables.

- Gets its name from the physical model resembling a star shape
- A fact table is at its center
- Dimension table surrounds the fact table representing the star's points.

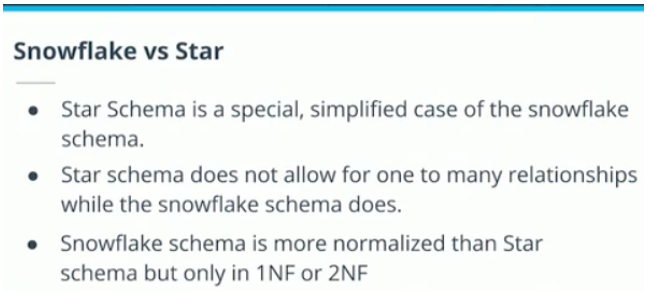
Benefits of Star Schema

- Getting a table into 3NF is a lot of hard work, JOINs can be complex even on simple data
- Star schema allows for the relaxation of these rules and makes queries easier with simple JOINS
- Aggregations perform calculations and clustering of our data so that we do not have to do that work in our application. Examples : COUNT, GROUP BY etc

Snowflake Schema



- Star Schema is a special, simplified case of the snowflake schema.
- Star schema does not allow for one to many relationships while the snowflake schema does.
- Snowflake schema is more normalized than Star schema but only in 1NF or 2NF



▼ **Data definition and Constraints**

The CREATE statement in SQL has a few important constraints that are highlighted below.

NOT NULL

The **NOT NULL** constraint indicates that the column cannot contain a null value.

Here is the syntax for adding a NOT NULL constraint to the CREATE statement:

```
CREATE TABLE IF NOT EXISTS customer_transactions (
    customer_id int NOT NULL,
    store_id int,
    spent numeric
);
```

You can add **NOT NULL** constraints to more than one column. Usually this occurs when you have a **COMPOSITE KEY**, which will be discussed further below.

Here is the syntax for it:

```
CREATE TABLE IF NOT EXISTS customer_transactions (
    customer_id int NOT NULL,
    store_id int NOT NULL,
    spent numeric
);
```

UNIQUE

The **UNIQUE** constraint is used to specify that the data across all the rows in one column are unique within the table.
The **UNIQUE** constraint can also be used for multiple columns, so that the combination of the values across those columns will be unique within the table. In this latter case, the values within 1 column do not need to be unique.Let's look at an example.

```
CREATE TABLE IF NOT EXISTS customer_transactions (
    customer_id int NOT NULL UNIQUE,
    store_id int NOT NULL UNIQUE,
    spent numeric
);
```

Another way to write a **UNIQUE** constraint is to add a table constraint using commas to separate the columns.

```
CREATE TABLE IF NOT EXISTS customer_transactions (
    customer_id int NOT NULL,
    store_id int NOT NULL,
    spent numeric,
    UNIQUE (customer_id, store_id, spent)
);
```

PRIMARY KEY

The **PRIMARY KEY** constraint is defined on a single column, and every table should contain a primary key. The values in this column uniquely identify the rows in the table. If a group of columns are defined as a primary key, they are called a **composite key**. That means the combination of values in these columns will uniquely identify the rows in the table. By default, the **PRIMARY KEY** constraint has the unique and not null constraint built into it.Let's look at the following example:

```
CREATE TABLE IF NOT EXISTS store (
    store_id int PRIMARY KEY,
    store_location_city text,
    store_location_state text
);
```

Here is an example for a group of columns serving as **composite key**.

```
CREATE TABLE IF NOT EXISTS customer_transactions (
    customer_id int,
    store_id int,
    spent numeric,
    PRIMARY KEY (customer_id, store_id)
);
```

==========

Upsert

In RDBMS language, the term *upsert* refers to the idea of inserting a new row in an existing table, or updating the row if it already exists in the table. The action of updating or inserting has been described as "upsert".

The way this is handled in PostgreSQL is by using the `INSERT` statement in combination with the `ON CONFLICT` clause.

INSERT

The **INSERT** statement adds in new rows within the table. The values associated with specific target columns can be added in any order.

Let's look at a simple example. We will use a customer address table as an example, which is defined with the following **CREATE** statement:

```
CREATE TABLE IF NOT EXISTS customer_address (
    customer_id int PRIMARY KEY,
    customer_street varchar NOT NULL,
```

```
    customer_city text NOT NULL,
    customer_state text NOT NULL
);
```

Let's try to insert data into it by adding a new row:

```
INSERT into customer_address (
VALUES
    (432, '758 Main Street', 'Chicago', 'IL'
);
```

Now let's assume that the customer moved and we need to update the customer's address. However we do not want to add a new customer id. In other words, if there is any conflict on the `customer_id`, we do not want that to change.

This would be a good candidate for using the **ON CONFLICT DO NOTHING** clause.

```
INSERT INTO customer_address (customer_id, customer_street, customer_city, customer_state)
VALUES
 (
 432, '923 Knox Street', 'Albany', 'NY'
 )
ON CONFLICT (customer_id)
DO NOTHING;
```

Now, let's imagine we want to add more details in the existing address for an existing customer. This would be a good candidate for using the **ON CONFLICT DO UPDATE** clause.

```
INSERT INTO customer_address (customer_id, customer_street)
VALUES
    (
    432, '923 Knox Street, Suite 1'
)
ON CONFLICT (customer_id)
DO UPDATE
    SET customer_street  = EXCLUDED.customer_street;
```

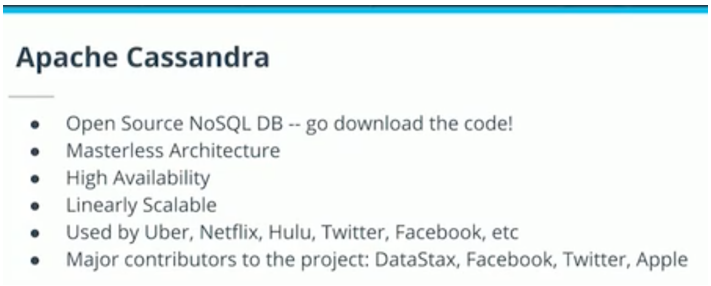
Lesson 3

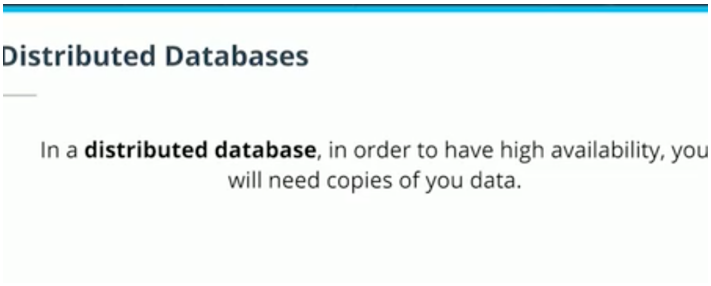
(NoSQL: Denormalization, Primary keys, Clustering columns, The WHERE clause)

▼ **Not Relational Databases**

When to Use NoSQL:

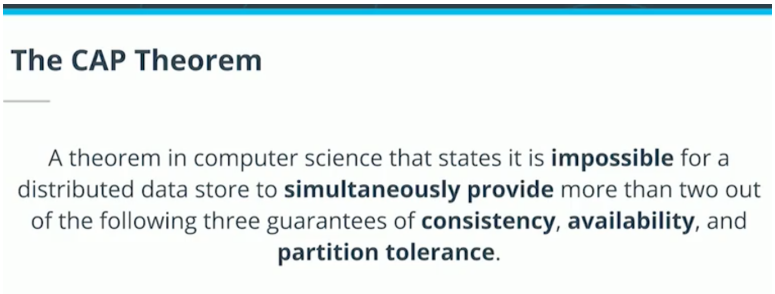
- **Need high Availability in the data**: Indicates the system is always up and there is no downtime
- **Have Large Amounts of Data**
- **Need Linear Scalability**: The need to add more nodes to the system so performance will increase linearly
- **Low Latency**: Shorter delay before the data is transferred once the instruction for the transfer has been received.
- **Need fast reads and write**





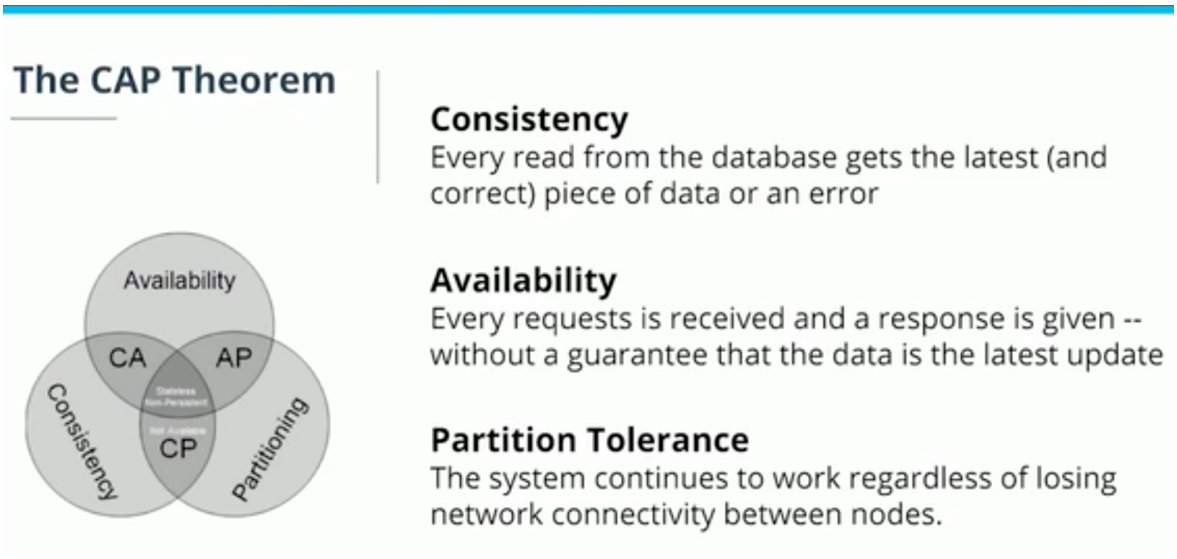
Eventual Consistency:

Over time (if no new changes are made) each copy of the data will be the same, but if there are new changes, the data may be different in different locations. The data may be inconsistent for only milliseconds. There are workarounds in place to prevent getting stale data.



CAP Theorem:

- **Consistency**: Every read from the database gets the latest (and correct) piece of data or an error
- **Availability**: Every request is received and a response is given -- without a guarantee that the data is the latest update
- **Partition Tolerance**: The system continues to work regardless of losing network connectivity between nodes



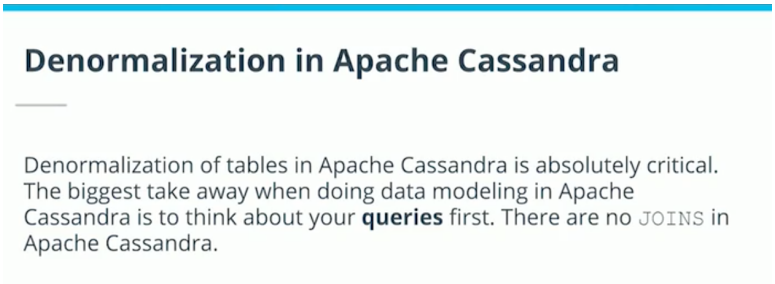
Is Eventual Consistency the opposite of what is promised by SQL database per the ACID principle?Much has been written about how *Consistency* is interpreted in the ACID principle and the CAP theorem. Consistency in the ACID principle refers to the requirement that only transactions that abide by constraints and database rules are written into the database, otherwise the database keeps previous state. In other words, the data should be correct across all rows and tables. However, consistency in the CAP theorem refers to every read from the database getting the latest piece of data or an error.To learn more, you may find this discussion useful:

- Discussion about ACID vs. CAP

Which of these combinations is desirable for a production system - Consistency and Availability, Consistency and Partition Tolerance, or Availability and Partition Tolerance?As the CAP Theorem Wikipedia entry says, "The CAP theorem implies that in the presence of a network partition, one has to choose between consistency and availability." So there is no such thing as Consistency and Availability in a distributed database since it must always tolerate network issues. You can only have Consistency and Partition Tolerance (CP) or Availability and Partition Tolerance (AP). Remember, relational and non-relational databases do different things, and that's why most companies have both types of database systems.

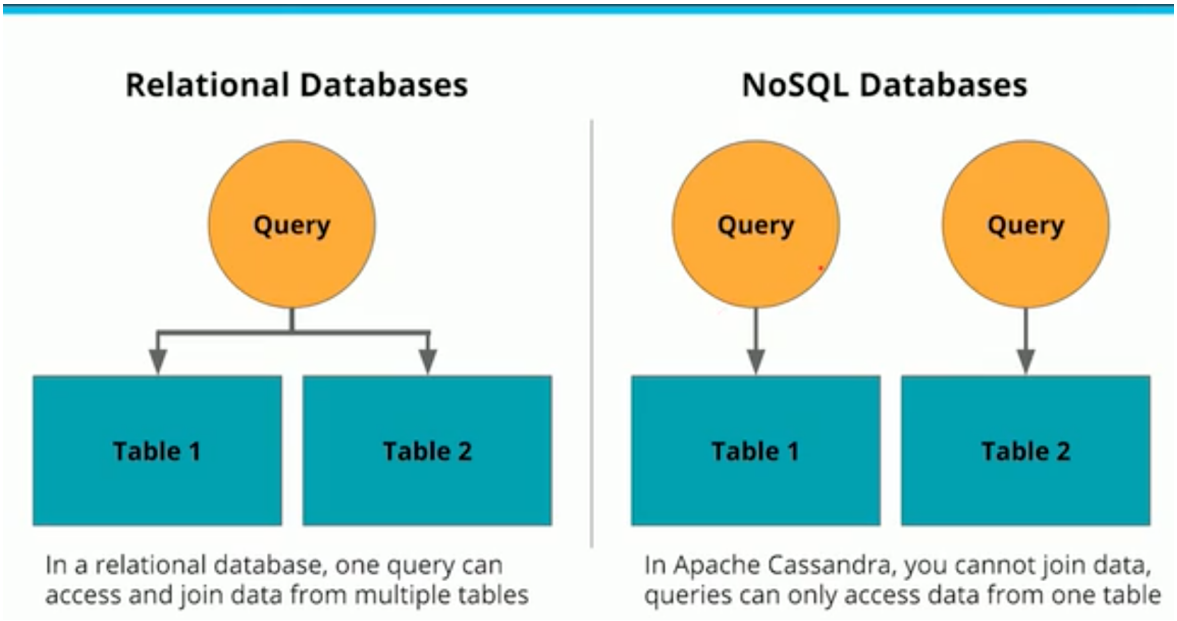
Does Cassandra meet just Availability and Partition Tolerance in the CAP theorem?According to the CAP theorem, a database can actually only guarantee two out of the three in CAP. So supporting Availability and Partition Tolerance makes sense, since Availability and Partition Tolerance are the biggest requirements.

If Apache Cassandra is not built for consistency, won't the analytics pipeline break?If I am trying to do analysis, such as determining a trend over time, e.g., how many friends does John have on Twitter, and if you have one less person counted because of "eventual consistency" (the data may not be up-to-date in all locations), that's OK. In theory, that can be an issue but only if you are not constantly updating. If the pipeline pulls data from one node and it has not been updated, then you won't get it. Remember, in Apache Cassandra it is about **Eventual Consistency**.

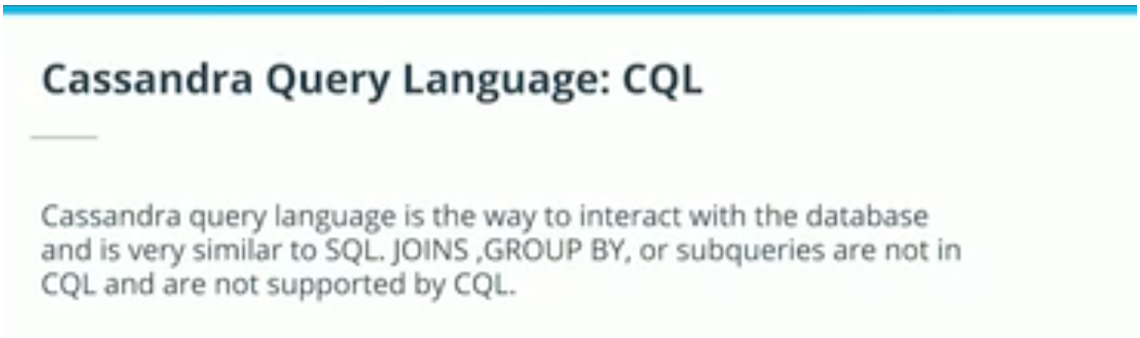


Data Modeling in Apache Cassandra:

- Denormalization is not just okay -- it's a must
- Denormalization must be done for fast reads
- Apache Cassandra has been optimized for fast writes
- ALWAYS think Queries first
- One table per query is a great strategy
- Apache Cassandra does **not** allow for JOINs between tables



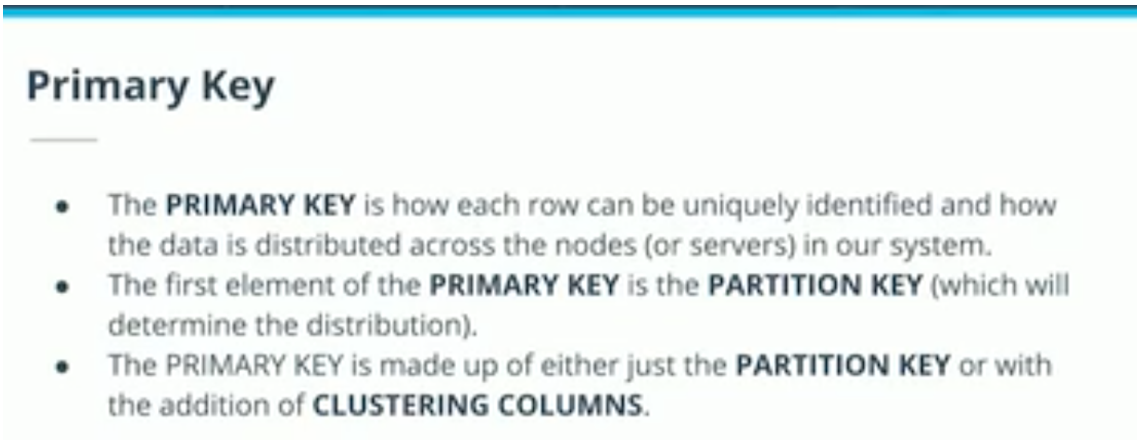
  - **I see certain downsides of this approach, since in a production application, requirements change quickly and I may need to improve my queries later. Isn't that a downside of Apache Cassandra?**In Apache Cassandra, you want to model your data to your queries, and if your business need calls for quickly changing requirements, you need to create a new table to process the data. That is a requirement of Apache Cassandra. If your business needs calls for ad-hoc queries, these are not a strength of Apache Cassandra. However keep in mind that it is easy to create a new table that will fit your new query.



DEMO

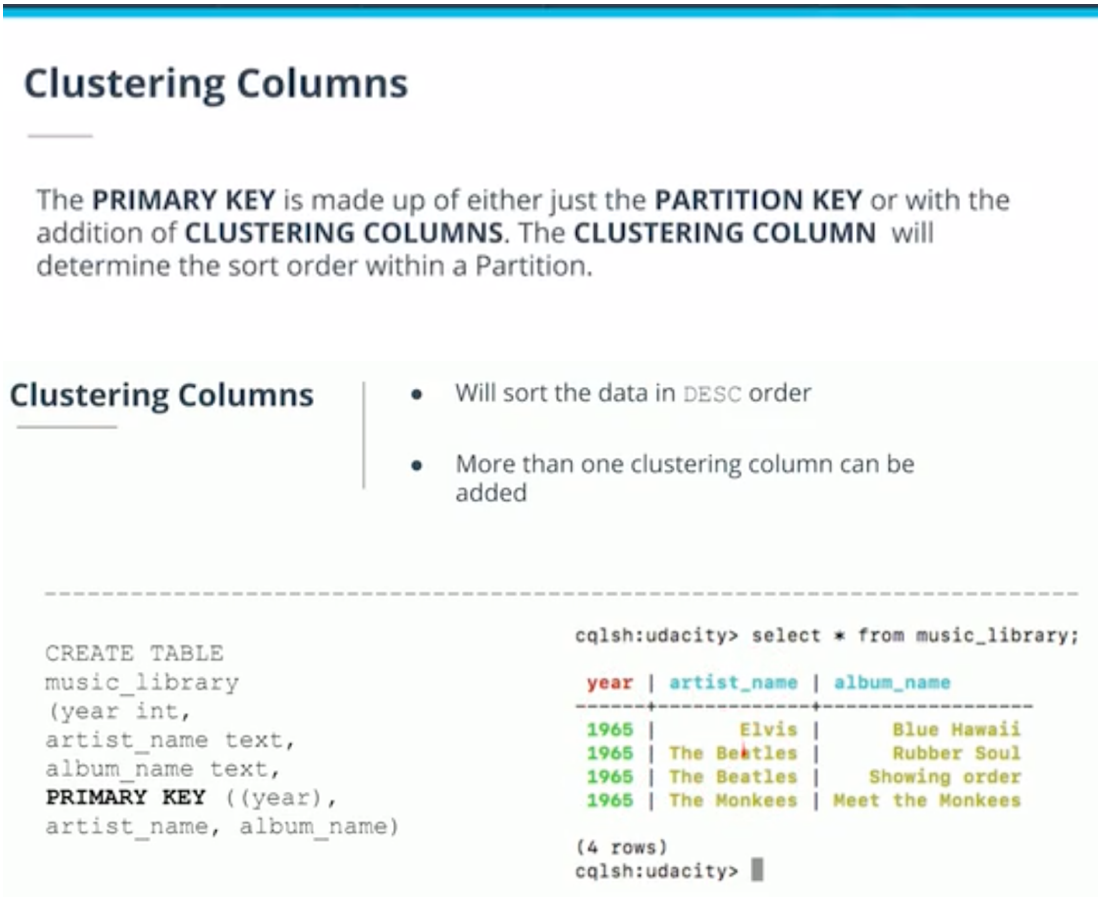
```
import cassandra
from cassandra.cluster import Cluster

#create a connection to the database
cluster = Cluster(['127.0.0.1']) #If you have a locally installed Apache Cassandra instance
session = cluster.connect()
```

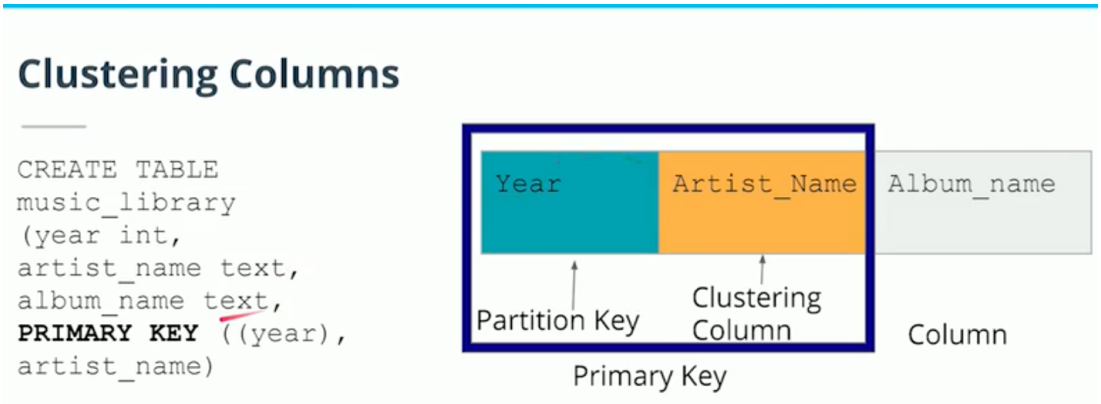
```
#create a keyspace to work in
session.execute("""
    CREATE KEYSPACE IF NOT EXISTS udacity
    WITH REPLICATION =
    { 'class' : 'SimpleStrategy', 'replication_factor' : 1 }""")
)

#connect to the keyspace
session.set_keyspace('udacity')

## define your queries (1: all albums in 1970, 2: all albums from beatles, 3: all info on a specific album)
## after this you can design your tables (1 per table - no joins)

query = "CREATE TABLE IF NOT EXISTS music_1970 \
    (album_name varchar, artist_name varchar, year int, PRIMARY KEY (year, album_name))"

query1 = "CREATE TABLE IF NOT EXISTS beatles \
    (album_name varchar, artist_name varchar, year int, PRIMARY KEY (artist_name, year)) "

query2 = "CREATE TABLE IF NOT EXISTS specific_album_info \
    (album_name varchar, artist_name varchar, year int, PRIMARY KEY (album_name, year)) "

session.execute(query)
session.execute(query1)
session.execute(query2)

## Insert REPETITIVE DATA into your tables

query = "INSERT INTO music_1970 (year,artist_name,album_name)"
query = query + " VALUES (%s, %s, %s)"

query1 = "INSERT INTO beatles (artist_name, year,album_name)"
query1 = query1 + " VALUES (%s, %s, %s)"

query2 = "INSERT INTO specific_album_info (album_name, artist_name, year)"
query2 = query2 + " VALUES (%s, %s, %s)"

session.execute(query, (1970, "The Beatles", "Let it Be"))
session.execute(query, (1965, "The Beatles", "Rubber Soul"))
session.execute(query, (1965, "The Who", "My Generation"))
session.execute(query, (1966, "The Monkees", "The Monkees"))
session.execute(query, (1970, "The Carpenters", "Close To You"))

session.execute(query1, ("The Beatles", 1970, "Let it Be"))
session.execute(query1, ("The Beatles", 1965, "Rubber Soul"))
session.execute(query1, ("The Who", 1965, "My Generation"))
session.execute(query1, ("The Monkees", 1966, "The Monkees"))
session.execute(query1, ("The Carpenters", 1970, "Close To You"))

session.execute(query2, ("Let it Be", "The Beatles", 1970))
session.execute(query2, ("Rubber Soul", "The Beatles", 1965))
session.execute(query2, ("My Generation", "The Who", 1965))
session.execute(query2, ("The Monkees", "The Monkees", 1966))
session.execute(query2, ("Close To You", "The Carpenters", 1970))

## VAlidate your data models

query = "select * from music_1970 WHERE year = 1970"
rows = session.execute(query)

for row in rows:
    print (row.year, row.artist_name, row.album_name)
#==========
query = "select * from beatles WHERE artist_name = 'The Beatles'"
rows = session.execute(query)

for row in rows:
    print (row.year, row.artist_name, row.album_name)
#=========
query = "select * from specific_album_info WHERE album_name = 'Close To You'"
rows = session.execute(query)

for row in rows:
    print (row.year, row.artist_name, row.album_name)

#Close session and Cluster
session.shutdown()
cluster.shutdown()
```



Primary Key

- Must be unique
- The PRIMARY KEY is made up of either just the PARTITION KEY or may also include additional CLUSTERING COLUMNS
- A Simple PRIMARY KEY is just one column that is also the PARTITION KEY. A Composite PRIMARY KEY is made up of more than one column and will assist in creating a unique value and in your retrieval queries
- The PARTITION KEY will determine the distribution of data across the system





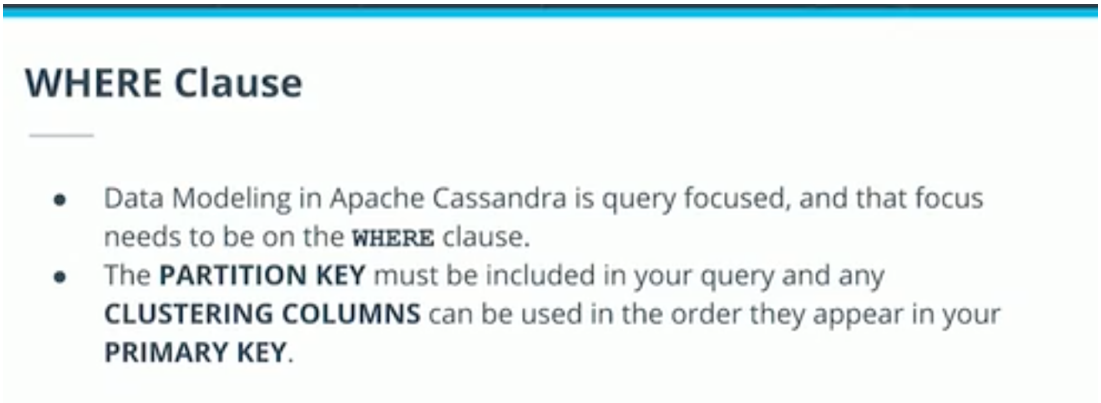
Clustering Columns:

- The clustering column will sort the data in sorted **ascending** order, e.g., alphabetical order.*
- More than one clustering column can be added (or none!)
- From there the clustering columns will sort in order of how they were added to the primary key

Commonly Asked Questions:

How many clustering columns can we add?

You can use as many clustering columns as you would like. You cannot use the clustering columns out of order in the SELECT statement. You may choose to omit using a clustering column in your SELECT statement. That's OK. Just remember to use them in order when you are using the SELECT statement.



WHERE clause

- Data Modeling in Apache Cassandra is query focused, and that focus needs to be on the WHERE clause
- Failure to include a WHERE clause will result in an error

Commonly Asked Questions:

Why do we need to use a `WHERE` statement since we are not concerned about analytics? Is it only for debugging purposes?The `WHERE` statement is allowing us to do the fast reads. With Apache Cassandra, we are talking about big data -- think terabytes of data -- so we are making it fast for read purposes. Data is spread across all the nodes. By using the `WHERE` statement, we know which node to go to, from which node to get that data and serve it back. For example, imagine we have 10 years of data on 10 nodes or servers. So 1 year's data is on a separate node. By using the `WHERE year = 1` statement we know which node to visit fast to pull the data from.