Data Engineering Nanodegree

- Dealing with Blockers: Use your community
- Google well
- Be patient, and work through the problem

RELATIONAL DATABASES

ACID Properties of a DB

- **Atomicity:** The whole transaction is processed or nothing is processed. A commonly cited example of an atomic transaction is money transactions between two bank accounts. The transaction of transferring money from one account to the other is made up of two operations. First, you have to withdraw money in one account, and second you have to save the withdrawn money to the second account. An atomic transaction, i.e., when either all operations occur or nothing occurs, keeps the database in a consistent state. This ensures that if either of those two operations (withdrawing money from the 1st account or saving the money to the 2nd account) fail, the money is neither lost nor created. Source Wikipedia for a detailed description of this example.
- **Consistency:** Only transactions that abide by constraints and rules are written into the database, otherwise the database keeps the previous state. The data should be correct across all rows and tables. Check out additional information about consistency on Wikipedia.
- **Isolation:** Transactions are processed independently and securely, order does not matter. A low level of isolation enables many users to access the data simultaneously, however this also increases the possibilities of concurrency effects (e.g., dirty reads or lost updates). On the other hand, a high level of isolation reduces these chances of concurrency effects, but also uses more system resources and transactions blocking each other. Source: Wikipedia
- **Durability:** Completed transactions are saved to the database even in cases of system failure. A commonly cited example includes tracking flight seat bookings.

So once the flight booking records a confirmed seat booking, the seat remains booked even if a system failure occurs. Source: Wikipedia.

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.

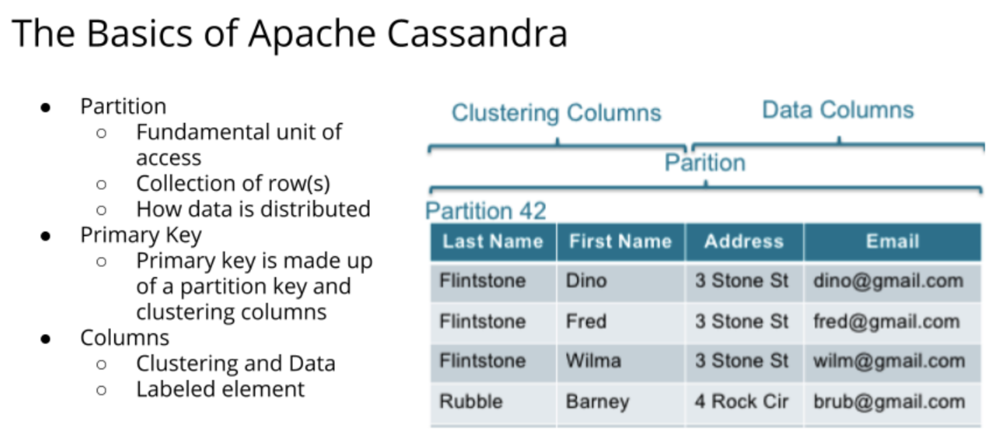
DATA MODELING: Suited to SQL databases

NOSQL DATABASES

NoSQL= Not Only SQL database or Non relational database. NoSQL databases were created to do some of the issues faced with Relational Databases. NoSQL databases have been around since the 1970's but they became more popular in use since the 2000's as data sizes have increased, and outages/downtime has decreased in acceptability.

NoSQL Database Implementations:

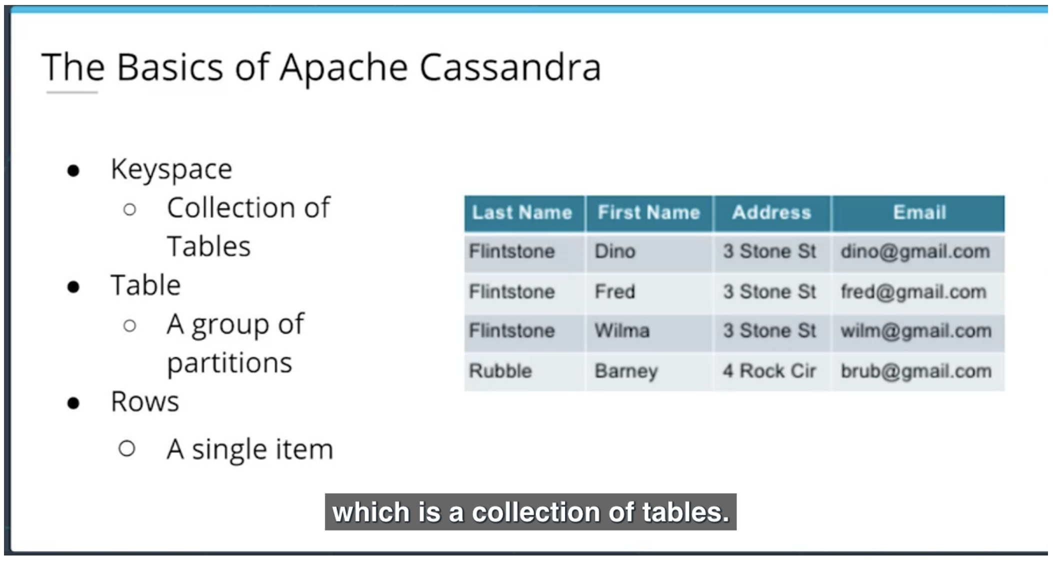
- Apache Cassandra (Partition Row store)
- MongoDB (Document store)
- DynamoDB (Key-Value store)
- Apache HBase (Wide Column Store)
- Neo4J (Graph Database



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



The Basics of Apache Cassandra

- Keyspace
  - Collection of Tables
- Table
  - A group of partitions
- Rows
  - A single item

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

which is a collection of tables.

What type of companies use Apache Cassandra?

All kinds of companies. For example, Uber uses Apache Cassandra for their entire backend. Netflix uses Apache Cassandra to serve all their videos to customers. Good use cases for NoSQL (and more specifically Apache Cassandra) are :

1. Transaction logging (retail, health care)

2. Internet of Things (IoT)
3. Time series data
4. Any workload that is heavy on writes to the database (since Apache Cassandra is optimized for writes).

Would Apache Cassandra be a hindrance for my analytics work? If yes, why?
Yes, if you are trying to do analysis, such as using `GROUP BY` statements. Since Apache Cassandra requires data modeling based on the query you want, you can't do ad-hoc queries. However you can add clustering columns into your data model and create new tables.

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

Caveats to NoSQL and ACID Transactions

There are some NoSQL databases that offer some form of ACID transaction. As of v4.0, MongoDB added multi-document ACID transactions within a single replica set. With their later version, v4.2, they have added multi-document ACID transactions in a sharded/partitioned deployment.

Importance of Relational Databases:

- **Standardization of data model:** Once your data is transformed into the rows and columns format, your data is standardized and you can query it with SQL
- **Flexibility in adding and altering tables:** Relational databases gives you flexibility to add tables, alter tables, add and remove data.
- **Data Integrity:** Data Integrity is the backbone of using a relational database

- **Structured Query Language (SQL):** A standard language can be used to access the data with a predefined language.
- **Simplicity :** Data is systematically stored and modeled in tabular format.
- **Intuitive Organization:** The spreadsheet format is intuitive but intuitive to data modeling in relational databases.

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 remembering 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).

Normalization and Denormalization.

- Normalization organizes the columns and tables in a database to ensure that their dependencies are properly enforced by database integrity constraints.
- We don’t want or need extra copies of our data, this is data redundancy. We want to be able to update data in one place and have that be the source of truth, that is data integrity.

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

How to reach First Normal Form (1NF):

a. Atomic values: each cell contains unique and single values
b. Be able to add data without altering tables
c. Separate different relations into different tables

d. Keep relationships between tables together with foreign keys

Second Normal Form (2NF):

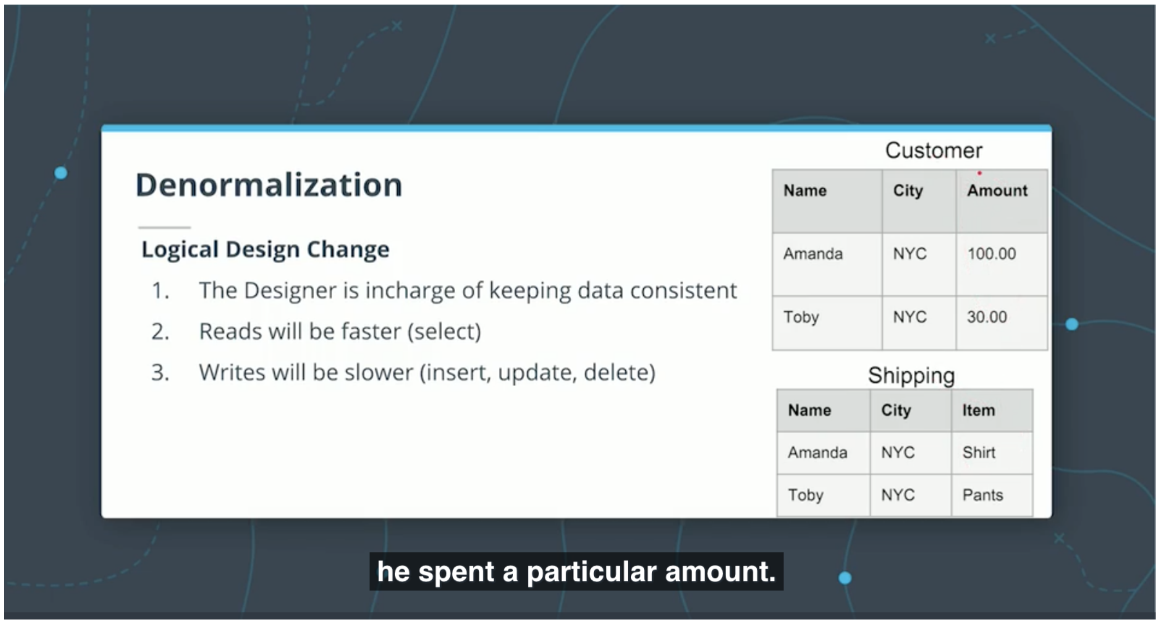
e. Have reached 1NF
f. All columns in the table must rely on the Primary Key

Third Normal Form (3NF):

g. Must be in 2nd Normal Form
h. No transitive dependencies
i. 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 it in just 1 place. We want to avoid updating the table in the Customers Detail table (in the example in the lecture slide).



Denormalization

Logical Design Change

1. The Designer is incharge of keeping data consistent
2. Reads will be faster (select)
3. Writes will be slower (insert, update, delete)

Customer

Name	City	Amount
Amanda	NYC	100.00
Toby	NYC	30.00

Shipping

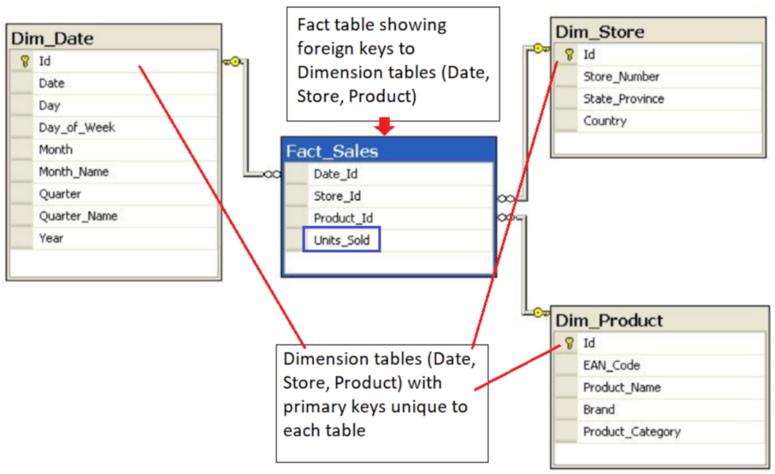
Name	City	Item
Amanda	NYC	Shirt
Toby	NYC	Pants

he spent a particular amount.

Facts Vs DImensions
The following image shows the relationship between the fact and dimension tables for the example shown in the video. As you can see in the image, the unique primary key for each Dimension table is included in the Fact table.In this example, it helps to think about the **Dimension tables** providing the following information:

- **Where** was the product bought? (Dim_Store table)
- **When** was the product bought? (Dim_Date table)
- **What** product was bought? (Dim_Product table)

The **Fact table** provides the **metric of the business process** (here Sales).

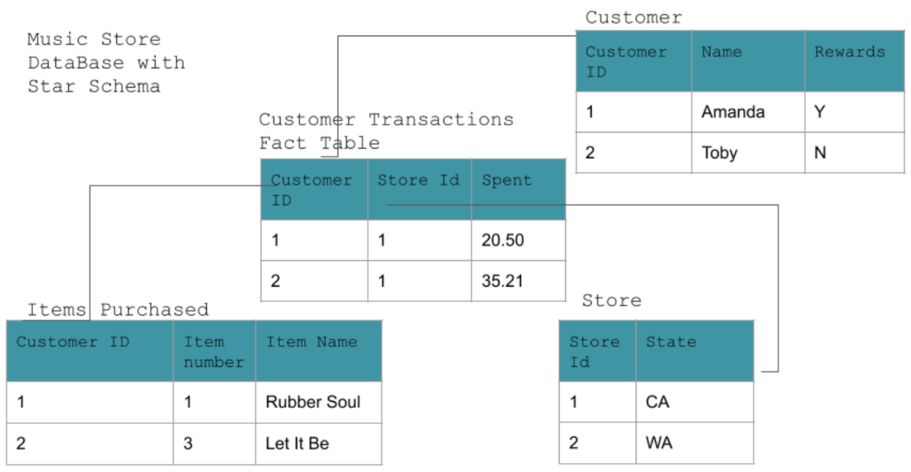
- **How many** units of products were bought? (Fact_Sales table)
- 

STAR SCHEMA:
A fact table surrounded by 1 or more dim tables.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



Customer

Customer ID	Name	Rewards
1	Amanda	Y
2	Toby	N

Customer Transactions Fact Table

Customer ID	Store Id	Spent
1	1	20.50
2	1	35.21

Items Purchased

Customer ID	Item number	Item Name
1	1	Rubber Soul
2	3	Let It Be

Store

Store Id	State
1	CA
2	WA

Music Store Database with Star Schema

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;
```

NO SQL DATA MODELLING

Non RELATIONAL DATABASES

NoSQL = Not Only SQL
Apache Cassandra -> created by Facebook in 2010

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**
- **Used by Uber, Netflix, Hulu, Twitter, facebook, etc**

A node: could be a server or entity on a system
Distributed Databases:
To have high data availability, it requires us to have numerous copies of our data. Our database will have to be distributed(scaled horizontally) i.e. have multiple systems running in parallel. It also requires our system to have little or no downtime. This requires us to have many copies of the same data on each node considering that nodes WILL happen.

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.

Commonly Asked Questions:

What does the network look like? Can you share any examples?

In Apache Cassandra every node is connected to every node -- it's peer to peer database architecture.

Is data deployment strategy an important element of data modeling in Apache Cassandra?

Deployment strategies are a great topic, but have very little to do with data modeling. Developing deployment strategies focuses on determining how many clusters to create or determining how many nodes are needed. These are topics generally covered under database architecture, database deployment and operations, which we will not cover in this lesson.

In general, the size of your data and your data model can affect your deployment strategies. You need to think about how to create a cluster, how many nodes should be in that cluster, how to do the actual installation. More information about deployment strategies can be found on this DataStax documentation page

Citation for above slides:

Here is the Wikipedia page cited in the slides.

Cassandra Architecture

We are not going into a lot of details about the Apache Cassandra Architecture. However, if you would like to learn more about it for your job, here are some links that you may find useful.

Apache Cassandra Data Architecture:

- Understanding the architecture
- Cassandra Architecture

The following link will go more in-depth about the Apache Cassandra Data Model, how Cassandra reads, writes, updates, and deletes data.

- Cassandra Documentation

Architecture in brief

Cassandra is designed to handle big data workloads across multiple nodes with no single point of failure. Its architecture is based on the understanding that system and hardware failures can and do occur. Cassandra addresses the problem of failures by employing a peer-to-peer distributed system across homogeneous nodes where data is distributed among all nodes in the cluster. Each node frequently exchanges state information about itself and other nodes across the cluster using peer-to-peer gossip communication protocol. A sequentially written commit log on each node captures write activity to ensure data durability. Data is then indexed and written to an in-memory structure, called a memtable, which resembles a write-back cache. Each time the memory structure is full, the data is written to disk in an SSTables data file. All writes are automatically partitioned and replicated throughout the cluster. Cassandra periodically consolidates SSTables using a process called compaction, discarding obsolete data marked for deletion with a tombstone. To ensure all data across the cluster stays consistent, various repair mechanisms are employed.

Cassandra is a partitioned row store database, where rows are organized into tables with a required primary key. Cassandra's architecture allows any authorized user to connect to any node in any datacenter and access data using the CQL language. For ease of use, CQL uses a similar syntax to SQL and works with table data. Developers can access CQL through cqlsh, DevCenter, and via drivers for application languages. Typically, a cluster has one keyspace per application composed of many different tables.

Client read or write requests can be sent to any node in the cluster. When a client connects to a node with a request, that node serves as the coordinator for that particular client operation. The coordinator acts as a proxy between the client application and the nodes that own the data being requested. The coordinator determines which nodes in the ring should get the request based on how the cluster is configured.

Key structures

- Node
  Where you store your data. It is the basic infrastructure component of Cassandra.
- datacenter
  A collection of related nodes. A datacenter can be a physical datacenter or virtual datacenter. Different workloads should use separate datacenters, either physical or virtual. Replication is set by datacenter. Using separate datacenters prevents Cassandra transactions from being impacted by other workloads and keeps requests close to each other for lower latency. Depending on the replication factor, data can be written to multiple datacenters. datacenters must never span physical locations.
- Cluster
  A cluster contains one or more data centers. It can span physical locations.
- Commit log
  All data is written first to the commit log for durability. After all its data has been flushed to SSTables, it can be archived, deleted, or recycled.

- SSTable

  A sorted string table (SSTable) is an immutable data file to which Cassandra writes memtables periodically. SSTables are append-only and stored on disk sequentially and maintained for each Cassandra table.
- CQL Table

  A collection of ordered columns fetched by table row. A table consists of columns and has a primary key.

Key components for configuring Cassandra

- Gossip

  A peer-to-peer communication protocol to discover and share location and state information about the other nodes in a Cassandra cluster. Gossip information is also persisted locally by each node to use immediately when a node restarts.
- Partitioner

  A partitioner determines which node will receive the first replica of a piece of data, and how to distribute other replicas across other nodes in the cluster. Each row of data is uniquely identified by a primary key, which may be the same as its partition key, but which may also include other clustering columns. A partitioner is a hash function that derives a token from the primary key of a row. The partitioner uses the token value to determine which nodes in the cluster receive the replicas of that row. The Murmur3Partitioner is the default partitioning strategy for new Cassandra clusters and the right choice for new clusters in almost all cases.

  You must set the partitioner and assign the node a num_tokens value for each node. The number of tokens you assign depends on the hardware capabilities of the system. If not using virtual nodes (vnodes), use the initial_token setting instead.

- Replication factor

  The total number of replicas across the cluster. A replication factor of 1 means that there is only one copy of each row on one node. A replication factor of 2 means two copies of each row, where each copy is on a different node. All replicas are equally important; there is no primary or master replica. You define the replication factor for each datacenter. Generally you should set the replication strategy greater than one, but no more than the number of nodes in the cluster.
- Replica placement strategy

  Cassandra stores copies (replicas) of data on multiple nodes to ensure reliability and fault tolerance. A replication strategy determines which nodes to place replicas on. The first replica of data is simply the first copy; it is not unique in any sense. The NetworkTopologyStrategy is highly recommended for most deployments because it is much easier to expand to multiple datacenters when required by future expansion.

  When creating a keyspace, you must define the replica placement strategy and the number of replicas you want.

- Snitch

  A snitch defines groups of machines into datacenters and racks (the topology) that the replication strategy uses to place replicas.

  You must configure a snitch when you create a cluster. All snitches use a dynamic snitch layer, which monitors performance and chooses the best replica for reading. It is enabled by default and recommended for use in most deployments. Configure dynamic snitch thresholds for each node in the cassandra.yaml configuration file.

  The default SimpleSnitch does not recognize datacenter or rack information. Use it for single-datacenter deployments or single-zone in public clouds. The GossipingPropertyFileSnitch is recommended for production. It defines a node's datacenter and rack and uses gossip for propagating this information to other nodes.
- The cassandra.yaml configuration file

  The main configuration file for setting the initialization properties for a cluster, caching parameters for tables, properties for tuning and resource utilization, timeout settings, client connections, backups, and security.

  By default, a node is configured to store the data it manages in a directory set in the cassandra.yaml file.

  In a production cluster deployment, you can change the commitlog-directory to a different disk drive from the data_file_directories.
- System keyspace table properties

  You set storage configuration attributes on a per-keyspace or per-table basis programmatically or using a client application, such as CQL.

The way SQL has ACID properties, NoSQL has CAP theorem:

This is a theorem in computer science which states that it is impossible for a distribributed data store to produce more than 2 of the 3. Apache Cassandra picks A and P over C when there is a failure/downtime

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

Commonly Asked Questions:

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 its 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**.

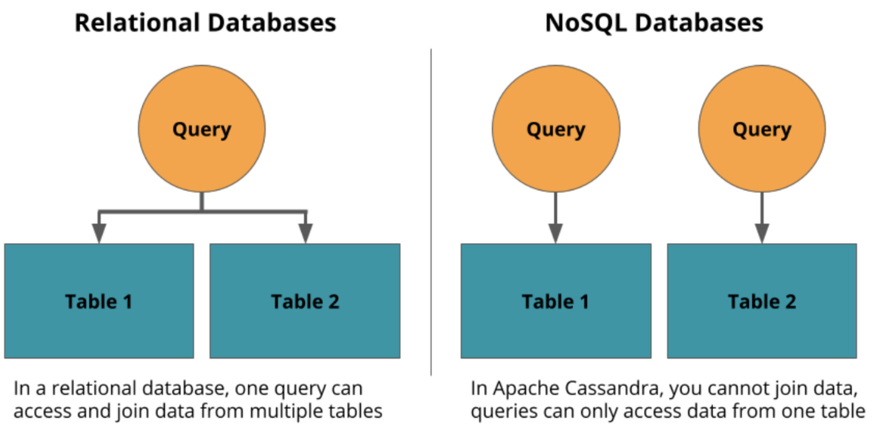
SECRET TO DENORMALISATION: THINK ABOUT YOUR QUERIES FIRST

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

Commonly Asked Questions:

- **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.



Relational Databases
NoSQL Databases
Query
Query
Query
Table 1
Table 2
Table 1
Table 2
In a relational database, one query can access and join data from multiple tables
In Apache Cassandra, you cannot join data, queries can only access data from one table

In Apache Cassandra, queries can only access data from one table

Cassandra Query Language

Cassandra query language is the way to interact with the database and is very similar to SQL.

The following are **not** supported by CQL

- JOINS
- GROUP BY
- Subqueries

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.

Additional Resources:

1. Here is the DataStax documentation on Composite Partition Keys.
2. This Stackoverflow page provides a nice description of the difference between Partition Keys and Clustering Keys.

WHERE clause

- Data Modeling in Apache Cassandra is query focused, and that focus needs to be on the WHERE clause
- This should be done based on the partition key first and if there are subsequent conditions, they should be on the clustering columns in the order
- Failure to include a WHERE clause will result in an error

Additional Resource

AVOID using "ALLOW FILTERING": Here is a reference in DataStax that explains ALLOW FILTERING and why you should not use it.

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.