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 First, you have to withdraw money in one account, and second you have to save the withdrawn money to the second account.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.

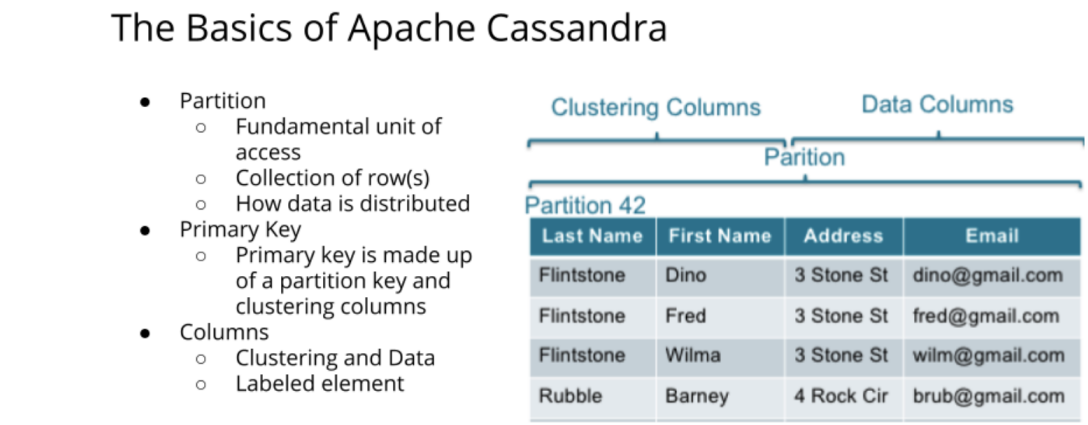
DATA MODELING: Suited to SQL databases

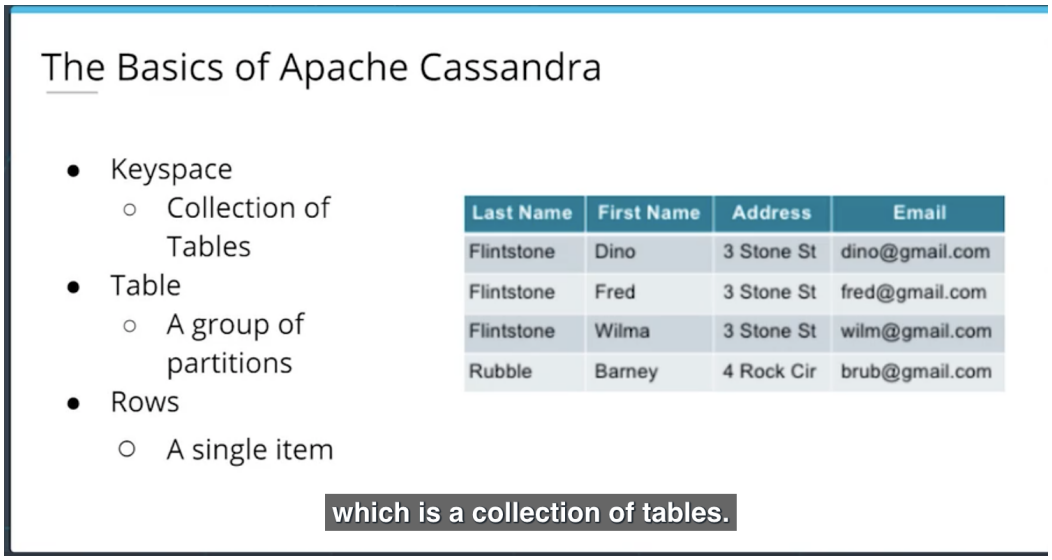
NOSQL DATABASES

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





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):

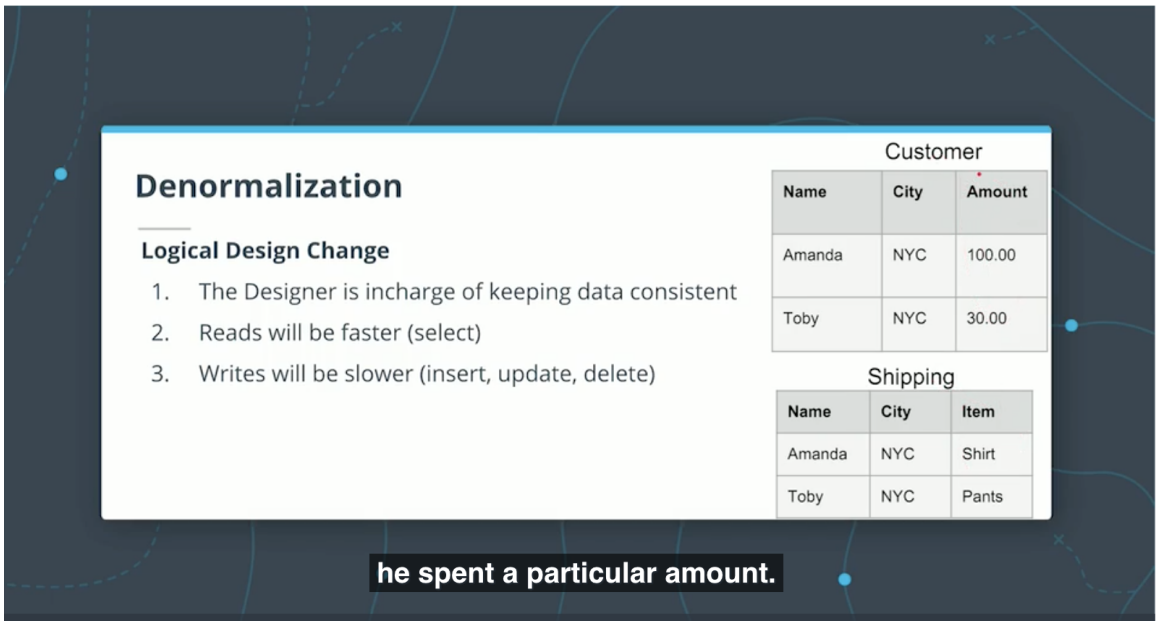
e. Have reached 1NF
f. All columns in the table must rely on the Primary Key

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).



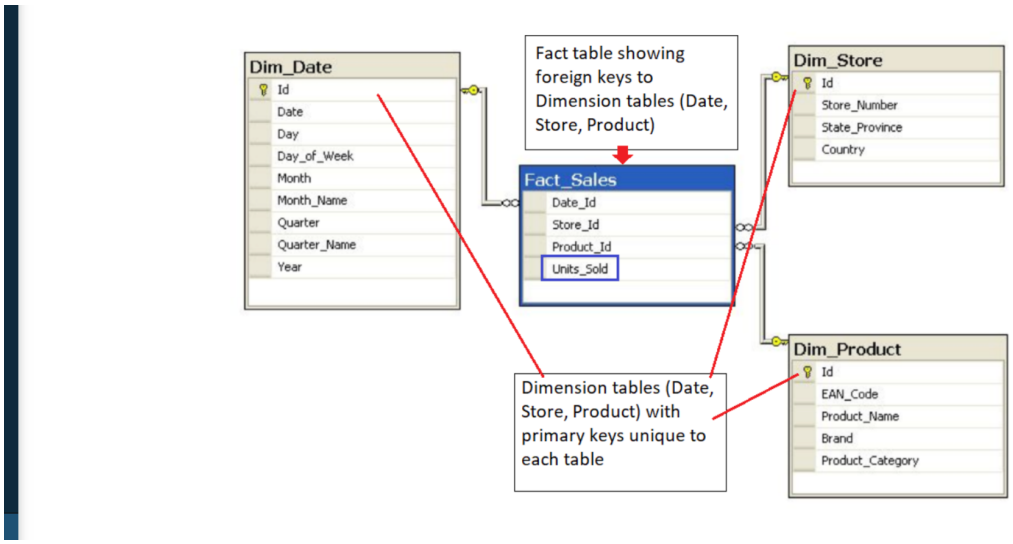
Facts Vs DImensions

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)



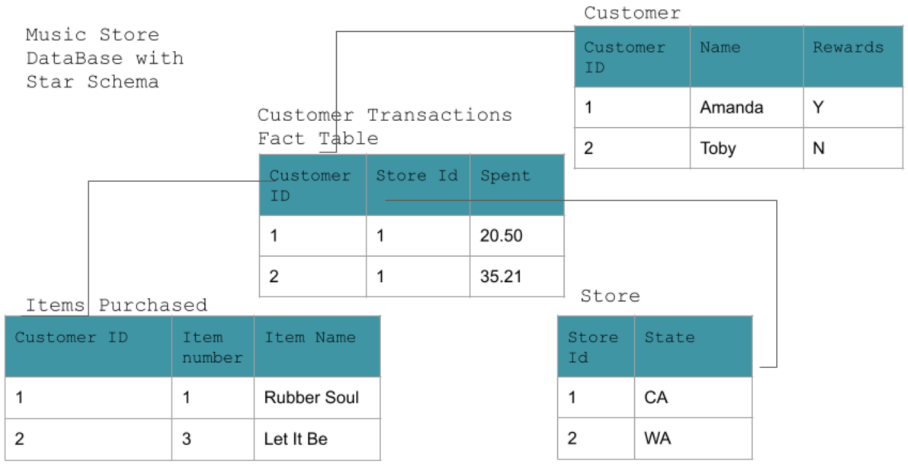
STAR SCHEMA:

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



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;
```