**Videos URL:**

**Intro** -----------------------------------------------------------------------------------------------------

Introduction to the course:

https://youtu.be/c7G8sMId7KM

What is data modeling:

https://www.youtube.com/watch?v=CAf5CzKJHmc&feature=youtu.be

Why is Data Modeling Important ?

https://youtu.be/hq0Q86fEd2E

Who does this type of work ?

https://youtu.be/LBqMfr1PniQ

Intro to Relational Databases:

https://youtu.be/TAXJtSgGzqA

When to use a relational Database?

https://youtu.be/cHErsgQZNlw

ACID Transactions:

https://youtu.be/rr35DQljFnc

When Not to Use a Relational Database:

https://youtu.be/yzB2G33pxbQ

What is PostgreSQL?

https://youtu.be/5L9O9QK7nLY

Apache Cassandra:

<https://youtu.be/jjwYgUEPSUc>

Create table Cassandra:

<https://youtu.be/u-GlcpAndmY>

<https://youtu.be/QFYO0SqZMuw>

Install Apache Cassandra:

[http://cassandra.apache.org/](http://cassandra.apache.org/doc/latest/getting_started/installing.html)

**Modelink** -----------------------------------------------------------------------------------

Data Modeling:

https://youtu.be/S1G7BurtYJ0

Databases:

https://youtu.be/EvjmD7mTqQM

Importance of Relational Databases:

https://youtu.be/QgBHz0bL1Sw

OLAP vs OLTP:

https://youtu.be/ocoyWgYllFE

Structuring the Database: Normalization:

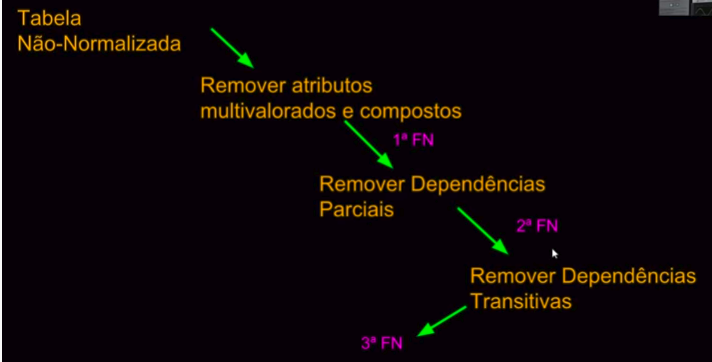
https://youtu.be/92dREpe9SLg

Objectives of Normal Form:

https://youtu.be/B\_JtLJbhszM

Normal Forms:

https://youtu.be/ZUnI99efjJQ



Boyce Codd (FNBC):

https://www.youtube.com/watch?v=o6mSiTO-vak

Creating Normalized Tables:

https://youtu.be/zdonkmTttaI

Denormalization:

https://youtu.be/jJezt6YGweA

Fact and Dimension Table:

https://youtu.be/3ala0SDBCyY

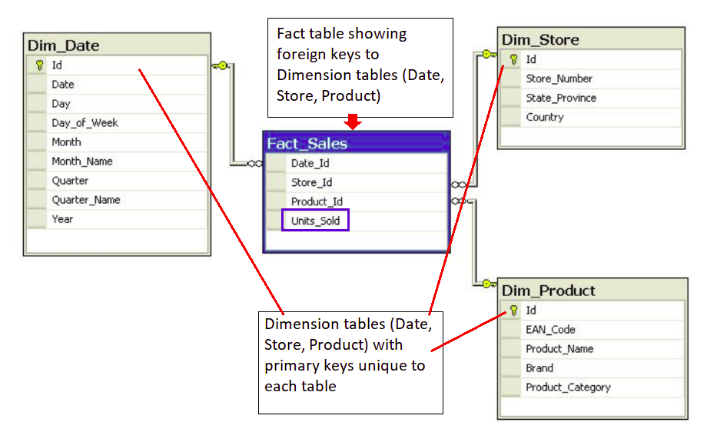
The following image shows the relationship between the fact and dimension tables for the example shown in the video. As you can see in the image, the unique primary key for each Dimension table is included in the Fact table.

In this example, it helps to think about the **Dimension tables** providing the following information:

* **Where** the product was bought? (Dim\_Store table)
* **When** the product was 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 Schemas:

https://youtu.be/i\_0hq6KsjMo

Benefits of Star Schemas:

https://youtu.be/sfsnFmE74yY

Snowflake Schema:

https://youtu.be/UAceZsZSyUs

Creating Fact and Dimension Tables:

https://youtu.be/GVRAWaESxfk

## **a 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)

);

To read more about these constraints, check out the [**PostgreSQL documentation**](https://www.postgresql.org/docs/9.4/ddl-constraints.html).

PRÓXIMO

**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;

We recommend checking out these two links to learn other ways to insert data into the tables.

* [**PostgreSQL tutorial**](http://www.postgresqltutorial.com/postgresql-upsert/)
* [**PostgreSQL documentation**](https://www.postgresql.org/docs/9.5/sql-insert.html)