**Codd's 12 rules:**

**Rule 0: The *foundation rule*:**

For any system that is advertised as, or claimed to be, a relational data base management system, that system must be able to manage data bases entirely through its relational capabilities.

**Rule 1: The *information rule*:**

All information in a relational data base is represented explicitly at the logical level and in exactly one way – by values in tables.

**Rule 2: The *guaranteed access rule*:**

Each and every datum (atomic value) in a relational data base is guaranteed to be logically accessible by resorting to a combination of table name, primary key value and column name.

**Rule 3: *Systematic treatment of null values*:**

Null values (distinct from the empty character string or a string of blank characters and distinct from zero or any other number) are supported in fully relational DBMS for representing missing information and inapplicable information in a systematic way, independent of data type.

**Rule 4: *Dynamic***[***online***](https://en.wikipedia.org/wiki/Online)[***catalog***](https://en.wikipedia.org/wiki/Database_catalog)***based on the relational model*:**

The data base description is represented at the logical level in the same way as ordinary data, so that authorized users can apply the same relational language to its interrogation as they apply to the regular data.

**Rule 5: The *comprehensive data sublanguage rule*:**

A relational system may support several languages and various modes of terminal use (for example, the fill-in-the-blanks mode). However, there must beat least one language whose statements are expressible, per some well-defined syntax, as character strings and that is comprehensive in supporting all of the following items:

1. Data definition.
2. View definition.
3. Data manipulation (interactive and by program).
4. Integrity constraints.
5. Authorization.
6. Transaction boundaries (begin, commit and rollback).

**Rule 6: The**[***view***](https://en.wikipedia.org/wiki/View_(database))***updating rule*:**

All views that are theoretically updatable are also updatable by the system.

**Rule 7: *Possible for high-level insert, update, and delete*:**

The capability of handling a base relation or a derived relation as a single operand applies not only to the retrieval of data but also to the insertion, update and deletion of data.

**Rule 8: *Physical data independence*:**

Application programs and terminal activities remain logically unimpaired whenever any changes are made in either storage representations or access methods.

**Rule 9: *Logical data independence*:**

Application programs and terminal activities remain logically unimpaired when information-preserving changes of any kind that theoretically permit unimpairment are made to the base tables.

**Rule 10: *Integrity independence*:**

Integrity constraints specific to a particular relational data base must be definable in the relational data sublanguage and storable in the catalog, not in the application programs.

**Rule 11: *Distribution independence*:**

The end-user must not be able to see that the data is distributed over various locations. Users should always get the impression that the data is located at one site only.

**Rule 12: The *nonsubversion rule*:**

If a relational system has a low-level (single-record-at-a-time) language, that low level cannot be used to subvert or bypass the integrity rules and constraints expressed in the higher level relational language (multiple-records-at-a-time).

**More information on Codd's 12 Rules can be found here:**

[Wikipedia link](https://en.wikipedia.org/wiki/Codd%27s_12_rules)

#### 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.

**OLAP vs OLTP**

* **"OLTP queries will have little aggregations really, if any, while OLAP will heavily focus on aggregations."**

**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).

#### Additional Resource on the difference between OLTP and OLAP:

This [Stackoverflow post](https://stackoverflow.com/questions/21900185/what-are-oltp-and-olap-what-is-the-difference-between-them" \t "_blank) describes it well.

**Note:** We will also be going more in depth with OLAP vs OLTP in **Course 3. Cloud Data Warehouses**

**Normal Forms:**

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).

**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.

**Citation for slides:** [**https://en.wikipedia.org/wiki/Denormalization**](https://en.wikipedia.org/wiki/Denormalization)

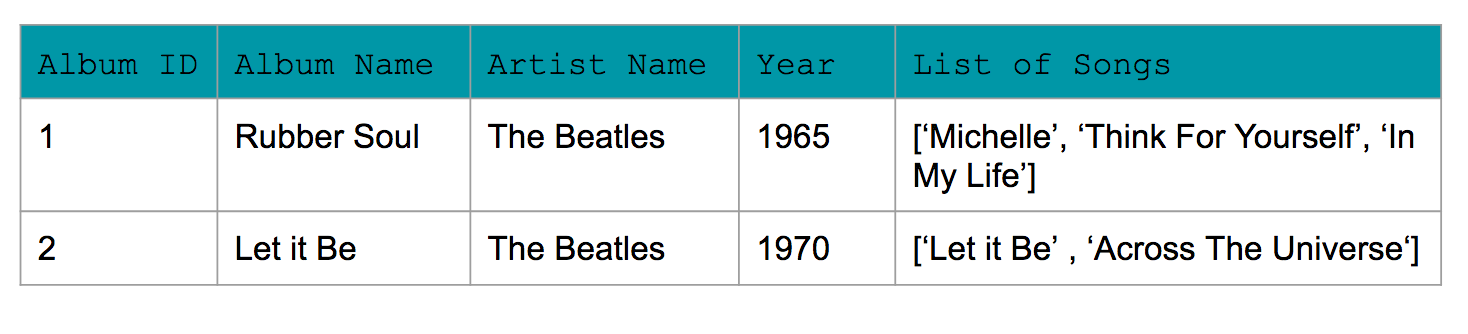
Let's take a moment to make sure you understand what was in the demo regarding denormalized vs. normalized data. These are important concepts, so make sure to spend some time reflecting on these.

**Normalization** is about trying to increase data integrity by reducing the number of copies of the data. Data that needs to be added or updated will be done in as few places as possible.

**Denormalization** is trying to increase performance by reducing the number of joins between tables (as joins can be slow). Data integrity will take a bit of a potential hit, as there will be more copies of the data (to reduce JOINS).

## Example of Denormalized Data:

As you saw in the earlier demo, this denormalized table contains a column with the Artist name that includes duplicated rows, and another column with a list of songs.

[[](https://classroom.udacity.com/nanodegrees/nd027/parts/f7dbb125-87a2-4369-bb64-dc5c21bb668a/modules/c0e48224-f2d0-4bf5-ac02-3e1493e530fc/lessons/dd3dfaa9-ba17-48eb-a6e6-8edda0b3c7df/concepts/757d847f-b220-4196-b82b-7e4cda2ae590)](https://classroom.udacity.com/nanodegrees/nd027/parts/f7dbb125-87a2-4369-bb64-dc5c21bb668a/modules/c0e48224-f2d0-4bf5-ac02-3e1493e530fc/lessons/dd3dfaa9-ba17-48eb-a6e6-8edda0b3c7df/concepts/757d847f-b220-4196-b82b-7e4cda2ae590)

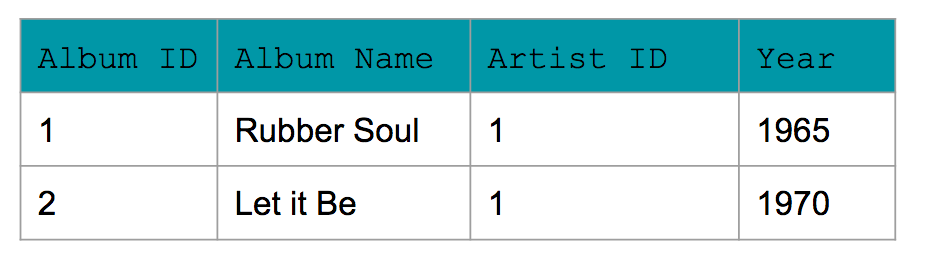
## Example of Normalized Data:

Now for normalized data, Amanda used 3NF. You see a few changes:  
1) No row contains a list of items. For e.g., the list of song has been replaced with each song having its own row in the Song table.  
2) Transitive dependencies have been removed. For e.g., album ID is the PRIMARY KEY for the album year in Album Table. Similarly, each of the other tables have a unique primary key that can identify the other values in the table (e.g., song id and song name within Song table).

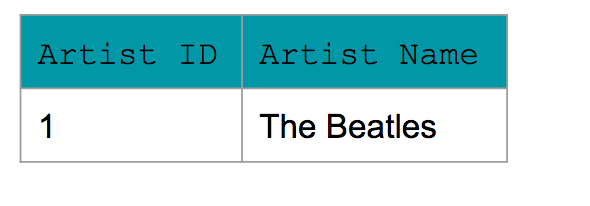
#### Song\_Table

[[](https://classroom.udacity.com/nanodegrees/nd027/parts/f7dbb125-87a2-4369-bb64-dc5c21bb668a/modules/c0e48224-f2d0-4bf5-ac02-3e1493e530fc/lessons/dd3dfaa9-ba17-48eb-a6e6-8edda0b3c7df/concepts/757d847f-b220-4196-b82b-7e4cda2ae590)](https://classroom.udacity.com/nanodegrees/nd027/parts/f7dbb125-87a2-4369-bb64-dc5c21bb668a/modules/c0e48224-f2d0-4bf5-ac02-3e1493e530fc/lessons/dd3dfaa9-ba17-48eb-a6e6-8edda0b3c7df/concepts/757d847f-b220-4196-b82b-7e4cda2ae590)

#### Album\_Table

[[](https://classroom.udacity.com/nanodegrees/nd027/parts/f7dbb125-87a2-4369-bb64-dc5c21bb668a/modules/c0e48224-f2d0-4bf5-ac02-3e1493e530fc/lessons/dd3dfaa9-ba17-48eb-a6e6-8edda0b3c7df/concepts/757d847f-b220-4196-b82b-7e4cda2ae590)](https://classroom.udacity.com/nanodegrees/nd027/parts/f7dbb125-87a2-4369-bb64-dc5c21bb668a/modules/c0e48224-f2d0-4bf5-ac02-3e1493e530fc/lessons/dd3dfaa9-ba17-48eb-a6e6-8edda0b3c7df/concepts/757d847f-b220-4196-b82b-7e4cda2ae590)

#### Artist\_Table

[[](https://classroom.udacity.com/nanodegrees/nd027/parts/f7dbb125-87a2-4369-bb64-dc5c21bb668a/modules/c0e48224-f2d0-4bf5-ac02-3e1493e530fc/lessons/dd3dfaa9-ba17-48eb-a6e6-8edda0b3c7df/concepts/757d847f-b220-4196-b82b-7e4cda2ae590)](https://classroom.udacity.com/nanodegrees/nd027/parts/f7dbb125-87a2-4369-bb64-dc5c21bb668a/modules/c0e48224-f2d0-4bf5-ac02-3e1493e530fc/lessons/dd3dfaa9-ba17-48eb-a6e6-8edda0b3c7df/concepts/757d847f-b220-4196-b82b-7e4cda2ae590)

**Citations for slides:**

* [**https://en.wikipedia.org/wiki/Dimension\_(data\_warehouse)**](https://en.wikipedia.org/wiki/Dimension_(data_warehouse))
* [**https://en.wikipedia.org/wiki/Fact\_table**](https://en.wikipedia.org/wiki/Fact_table)

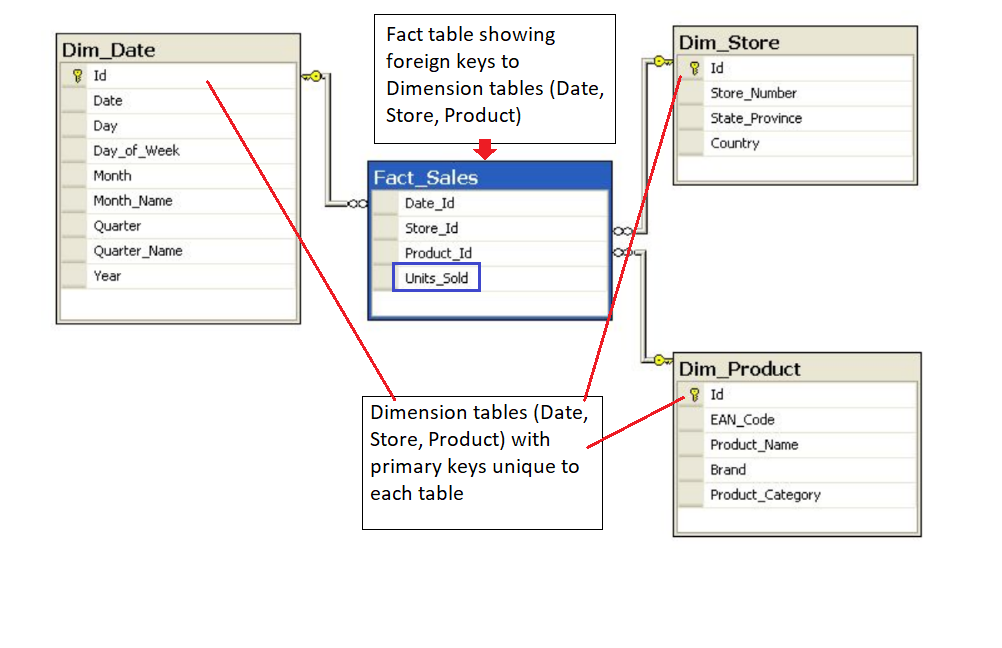
The following image shows the relationship between the fact and dimension tables for the example shown in the video. As you can see in the image, the unique primary key for each Dimension table is included in the Fact table.

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)

**[[](https://classroom.udacity.com/nanodegrees/nd027/parts/f7dbb125-87a2-4369-bb64-dc5c21bb668a/modules/c0e48224-f2d0-4bf5-ac02-3e1493e530fc/lessons/dd3dfaa9-ba17-48eb-a6e6-8edda0b3c7df/concepts/1523f798-d9c1-4ce8-bb67-e8e1c8fd7ef4)](https://classroom.udacity.com/nanodegrees/nd027/parts/f7dbb125-87a2-4369-bb64-dc5c21bb668a/modules/c0e48224-f2d0-4bf5-ac02-3e1493e530fc/lessons/dd3dfaa9-ba17-48eb-a6e6-8edda0b3c7df/concepts/1523f798-d9c1-4ce8-bb67-e8e1c8fd7ef4)**

If you are familiar with **Entity Relationship Diagrams** (ERD), you will find the depiction of STAR and SNOWFLAKE schemas in the demo familiar. The ERDs show the data model in a concise way that is also easy to interpret. ERDs can be used for any data model, and are not confined to STAR or SNOWFLAKE schemas. Commonly available tools can be used to generate ERDs. However, more important than creating an ERD is to learn more about the data through conversations with the data team so as a data engineer you have a strong understanding of the data you are working with.

More information about ER diagrams can be found at this [**Wikipedia**](https://en.wikipedia.org/wiki/Entity%E2%80%93relationship_model) page.

## 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)

);

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

## 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)

#### What we learned:

* What makes a database a relational database and Codd’s 12 rules of relational database design
* The difference between different types of workloads for databases OLAP and OLTP
* The process of database normalization and the normal forms.
* Denormalization and when it should be used.
* Fact vs dimension tables as a concept and how to apply that to our data modeling
* How the star and snowflake schemas use the concepts of fact and dimension tables to make getting value out of the data easier.

NEXT