# Starting from a raw data set

In this example, we start from a raw data set and design and populate a database from it.

We could use Python or some other easy-to-script language for this. But we could also just use Excel. That is what we will use here.

We will start with this data set:

<https://www.dropbox.com/scl/fi/1jtncvg354j2r5rsebjy7/music_data.txt?rlkey=g11q43kanlevdmux3ungwijwu&st=7zzxf5cw&dl=0>

Download it. Do not try copying its contents into a text file.

This is a tab-delimited file about songs consisting of several columns. Here are the columns.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| title | artist | album | track | duration |
| id | idtype | playcount | playcount | rating |
| year | notes | playlist |  |  |

There is also an extra column at the end that contains nothing but the number 27. We don't care about that column, and we also probably don't care about the idtype column because we don't know what it means.

We will pursue two strategies for bringing this into a relational database structure. The first will be based just on normalization. The second will rely on a critical analysis of the data instead.

## Normalization approach

Let's look for duplicates. It seems like id might be a good primary key. Let's see if there are duplicates in that column. Use Excel's duplicate detection. Select the entries in the id column and Choose Home >> Conditional Formatting >> Highlight Cell Rules >> Duplicate Values. You will see that there are some duplicates.

A screenshot of a computer

AI-generated content may be incorrect.

So that id alone doesn't work. Keep looking at the data. How can we break the ties?

It seems that if we add in the playlist, then we have found our unique identifier – our primary identifier or primary key. Let's draw the dependencies now.

id, playlist 🡪 title, artist, album, track, duration, playcount, rating, year, genre, notes

Let's rename id "song\_id" for clarity, because it seems to identify a song exclusively as opposed to a combination of song and playlist.

There are no repeating groups in this data set, so it's already in first normal form.

How about 2nd normal form? Are there partial dependencies? To answer this, we have to think critically about our data.

The song id itself – what does it determine? Well, the title clearly depends on the song\_id and not on the playlist, as does the artist, album, track, playcount, rating, year, genre, and notes. So, practically everything depnds on it – except for the playlist. Split it into two tables

song\_id, playllist 🡪 nothing

song\_id 🡪 title, artist, album, track, duration, playcount, rating, year, genre, notes

The database is now in second normal form.

How about third normal form? Are there any transitive dependencies? It doesn't seem like it. No columns other than song\_id determine the value of any other column. So we are already in third normal form.

With no more changes to recommend from normalization, it seems our design would consist of two tables. Almost all the columns will be in the table whose primary key is song\_id. The other table will consist of just the song\_id and the play\_list. This other table will give us room to store things that are germane to the combination of song and playlist, such as the date the song was added to the playlist perhaps.

There are some problems with this design, however. Consider the names of albums. It would be so easy to misspell an album title and thus end up with inconsistent data. Remember that such mistakes are called *anomalies* and they are caused by having too much uncontrolled redundancy. So, even after coming up with a normalized design, we have work to do. A lot of that work can be automatically taken care of by following a more analytical approach from the start that focuses on understanding the data itself. Let's call this the data-centric approach.

## Data-Centric Approach

The Data-Centric Approach aims to understand the data by identifying the different things – the different entity sets – for which it contains data. We list those, and then we assign attributes to those entity sets.

Look at the data and then identify the different things we are storing data about. As we do this, we want to focus on those kinds of things that are "rich" – they are the kinds of things that aren't just single values but are or at least can be described in terms of several quantities. We also focus on values that frequently repeat that we want to avoid typing over and over again because that would result in typos.

What are the "rich" things in our data? Title is not one of those "rich" things: it's just a value. What's it a value for? A title exists because it goes into describing a song. So clearly Song is one of those "rich" things that we will want to identify as an entity set.

What about album? Right now, it would seem that album is also just a piece of data, something used to describe a song. But really, it could be a "rich' thing, because an album can be described in terms of more than just its name but also perhaps the year it was made and its record label. An album is a rich thing.

In terms of frequently repeating things that we might mistype every so often, consider genre. It would be good to standardize genres into a finite set where each genre is identified by a code or abbreviation. Plus, you could argue that genre is a rich thing as well, because you might want to say something about a genre's country of origin or date it first surfaced in music. Genre is potentially a rich thing, and it is a frequently repeated free-form text that we should standardize. So it should be treated as an entity set.

We go through the data doing this kind of analysis. When we do, we end up with these entity sets:

* song
* album
* artist
* genre
* playlist

Now that we've identified the entity sets, let's identify the relationships, thus coming up with a conceptual model.

The relationship between song and playlist is a many-to-many relationship (M:N)

* A song can appear on several playlists.
* A playlist can have many songs.

The relationship between song and artist is one-to-many

* A song is made by one artist.
* An artist makes many songs.

The relationship between song and genre is one-to-many.

* A song has one genre.
* A genre encompasses many songs.

The relationship between album and song is one-to-many

* a song appears on an album
* an album contains many songs

Let's draw the conceptual diagram in Vertabelo.

A diagram of a music system

AI-generated content may be incorrect.

Now let's move to the conceptual model by adding attributes from the data set. Remember that some of the attributes will serve as primary identifiers and primary keys. We recommended using surrogate keys as unique identifiers to avoid using multi-column ones or ones that would be easy to accidentally misspell.

List the attributes that are *native* to each entity set.

Song: song\_id, title, track, duration, playcount, year, notes, rating

artist: id, name

album: Id, title

genre: id, name

playlist: id, name

Add the attributes to complete the logical model.

A diagram of a computer program

AI-generated content may be incorrect.

Now let's generate the physical model. Click the three-dot link next to the name of the file in the top margin in Vertabelo and choose Generate Physical Model. Choose mysql 8.x as the target.

A screenshot of a computer

AI-generated content may be incorrect.

After a little cleanup, you end up with this:

A diagram of a computer

AI-generated content may be incorrect.

Let's make some adjustments. Make the primary keys of album, genre, artist, and playlist auto-incrementing.

Next, add an int auto-incrementing PK to Song\_Playlist called id. Then, for each of Playlist, Artist, Genre, and Album, adjust the PK field id to be auto=incrementing as well. To do that, click the … button and click "Set" next to the auto-increment setting.

A screenshot of a computer

AI-generated content may be incorrect.

Also, for rating, allow nulls by checking the "N" checkbox next to rating.

Now generate the DDL commands to build this. Select the SQ icon in the top pane of Vertabelo.

A screenshot of a computer

AI-generated content may be incorrect.

Then click the Generate button and click Download. The script for building the database will be saved to your machine.

A screenshot of a computer

AI-generated content may be incorrect.

Now open XAMPP to begin importing the new structure into your mysql server. Click the Shell button and then change directory to where you saved the downloaded DDL script from Vertabelo.

A screenshot of a computer

AI-generated content may be incorrect.

In the console window, type

mysql -h localhost -u root -p

Enter the password (if you set one up). Then type these commands:

CREATE DATABASE MUSIC;

USE MUSIC;

Then import the SQL commands Vertabelo produced with this command:

source music\_database.sql

Replae "music\_database.sql" with whatever you saved the Vertabelo DDL script as.

After you run the sql script using the source command, you'll be able to issue this command:

show tables;

and it will list the tables in your database.

A screen shot of a computer program

AI-generated content may be incorrect.

The database is now ready to receive your data.

# Using Excel to generate the required SQL commands

We have various columns that have expanded into tables that need primary keys. These include artist, album, and playlist. (Song already has values for its primary key).

Create separate sheets in the Excel workbook for Artist, Album, Genre, and PlayList. Do this by pressing the Plus sign and renaming each seet as it is created.

A screenshot of a computer

AI-generated content may be incorrect.

Switch to the Artist tab. In Cell A2, enter this formula

= UNIQUE(music\_data!B2:B114)

Note that "music\_data" must match the name of the main data sheet.

This will list the unique artists found in the music data set. With those now listed, number them in column A on the Artist sheet.

Then, in column C of Artist, type this formula:

=CONCATENATE("insert into artist values (null, '",B1,"');")

Expand that formula to the other rows of column B.

Copy the resulting commands in column B into a new text document. Save the document as "artists.sql".

Follow the same process for albums, genres, and playlists. This way, you'll generate commands to populate all the tables that song connects to through foreign keys.

Next, return to the main spreadsheet. We need to replace the string values for artist, album, genre, and playlist with their corresponding id's from the artist, album, genre, and playlist spreadsheets. Let's work on artists first. In row 2 of an unused column to the right, enter this formula:

=XLOOKUP(B2,Artist!B:B, Artist!A:A, "null")

This will look for the value of cell B2 on the main sheet in the Artist spreadsheet in column B. It will then return the value found in the same row in column A. If it doesn't find a match, it will return "null".

Do the same for album, genre, and playlist. By the time you are done with this, you'll have all the foreign key values populated.

Next, we are able to create the insert queries for the songs. On the main sheet, enter this formula into row 2 in an unused column to the right:

=CONCATENATE("insert into Song values ('",F2,"','",A2,"',",D2,",",E2,",",H2,",",J2,",'",L2,"',",I2,",",R2,",",Q2,",",P2,");")

Drag this formula down to fill the rest of the songs' insert queries. Copy the queries to a new .sql file called songs.sql.

Next, use the source command in mysql to execute these files in this order:

artist.sql

album.sql

genre.sql

playlist.sql

songs.sql

song\_playlist.sql

Finally, create a new spreadsheet – songplaylist. Use formulas to set the values in column B to match the song id's from the main sheet and the values of column C to match the play list id's in the main sheet. Then number the records in column A. Finally, in cell B1, enter this formula:

=CONCATENATE("insert into song\_playlist values (null,'",B1,"',",C1,");")

Copy that formula to the rest of the cells in the song\_playlist spreadsheet.

Now all the data is populated into the music database, and we are ready to start using it to generate insights – i.e. to generate information.

# Queries

DML is the part of SQL that focuses on extracting information from the data. It stands for Data Manipulation Language. The most popular DML command is Select, but others include Insert and Update and Delete. Let's write some queries to start learning SQL.

1. Write a query to return all the data from the Artist table.
2. Write a query to return all the data from the Artist table ordered by name in ascending order.
3. Write a query to return the title and duration of all songs in the Song table.
4. Write a query to return the title and duration of all songs in the Song table whose year is 2015.
5. Write a query to return the title, duration, year, playcount of all songs in the Song table whose year is 2015 or whose playcount is 3. Order the results by year.
6. Write a query to return the title, duration, and year of all Songs in the Song table that were made after 2000.
7. Write a query to return the title, duration, and year of all Songs made in 2015, 2009, or 2017.
8. Write a query to return the title and rating of all songs where the rating is specified.
9. Update the value of the notes field for all songs made in 2016 to "song made in 2016".
10. Update the value of the notes field for all songs to "song made in " followed by the year.
11. Update the value of the notes field for all songs to an empty string.
12. Insert a new Artist into the Artist table.
13. Remove the Artist you just added to the Artist table.
14. Show the title, album title, and track of all songs.
15. Show the title, album title, and track of all songs made in 2015 or later.
16. Show the title, album. title, genre name, and artist name of all songs made in 2015 or later.
17. Update the notes field of all Pop songs to "Pop song made in " followed by the year.
18. Show the names of all songs whose name begins with "A"
19. Show the names of all songs whose name contains the word "the".
20. Show the unique album ids in the Song table.
21. Show the album ids and number of songs from each album.
22. Show the album ids and number of songs from each album where the number of songs on the album is greater than 1.
23. Show the album titles and number of songs from each album where the number of songs on the album is greater than 1.
24. Show the sum of durations of all songs written each year.
25. Show the year and average duration of all songs written each year where that average is greater than 160000 milliseconds.