MORE AND MORE NORMAL: TO 3NF AND BEYOND[1]

COMP 1140: Database and SQL | Brendan Shea, PhD (Brendan.Shea@rctc.edu)

In our last lesson, we'd discussed the process of database normalization, which is intended to reduce data redundancies. One way of thinking about what we are doing is to remember the slogan "THE KEY, THE WHOLE KEY, AND NOTHING BUT THE KEY, SO HELP ME CODD." This slogan breaks down as follows

1. "The key" : Tables may not contain repeating groups, which prevent a table from having a primary key. (1NF)
2. "the whole key": Every attribute must be functionally dependent on the entire primary key, and not merely partially dependent on a part of the key. (2NF)
3. "and nothing but the key," : The can be no transitive dependencies on a non-key field.(3NF)
4. " so help me, Codd." : The inventor of relational databases who gave us these rules.

In order to show this process, we were working through the creation of a music database. Here, we'll continue that process.

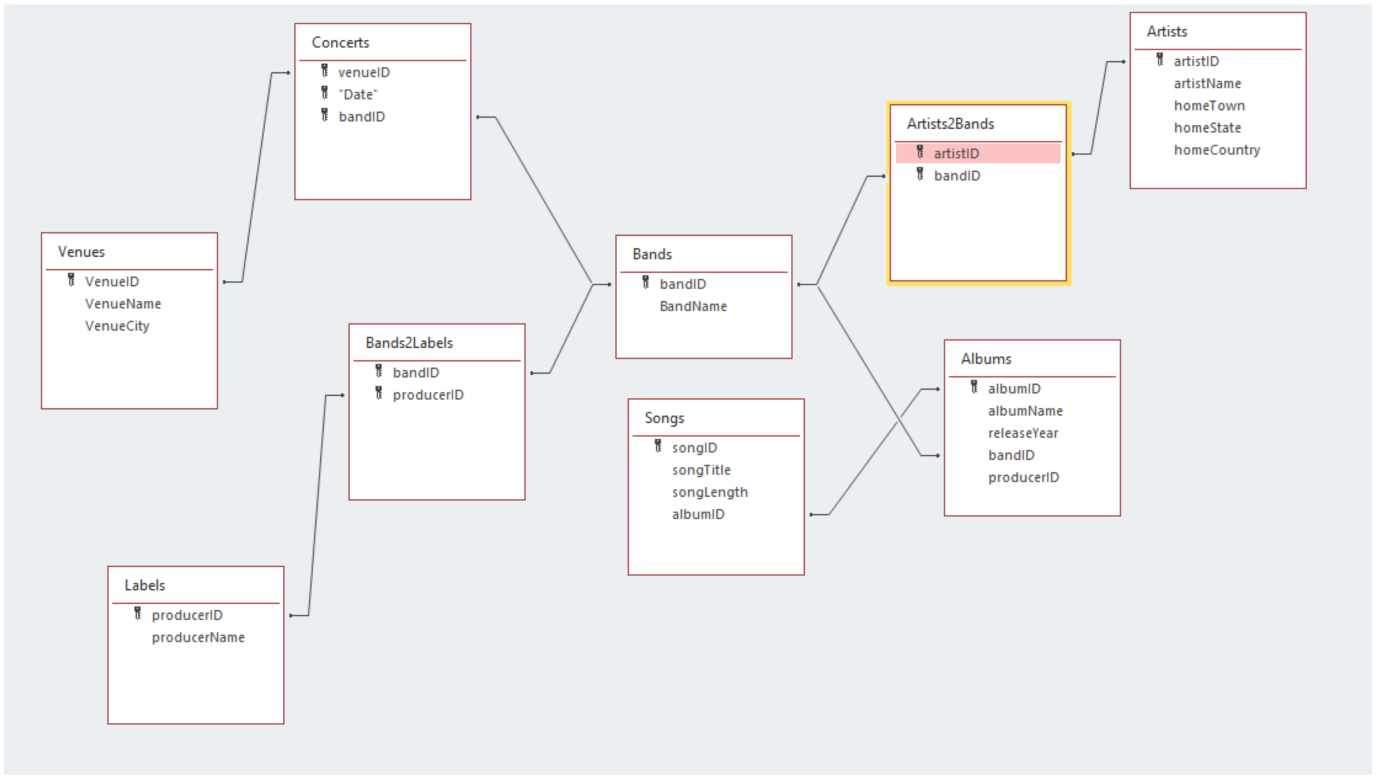
THINKING ABOUT RELATIONSHIPS

One of the most important things to review during normalization (which usually requires making lots of new tables!) is to make connections between the new tables that we have created should be facilitated by foreign keys, which will ensure we don't LOSE any of the information. **Database normalization should NEVER require that we lose data. It should ONLY involve "reformatting" this data to eliminate redundancies.** We have already started that process by earmarking a couple tables with notes where we knew we needed connections. Now that we have our primary keys, we have the unique values we will need to use. For this pass, we will look at how our tables relate to each other and see if we need connections. This is another step where remembering how to solve our data relationships will be important. Here's the steps we can take:

1. **Determine Tricky Relationships: Bands and Their Labels.** Looking at our "Labels" table, it could be argued that since a band belongs to a label that we should connect them. However, the relationship between a band and a label can change over time as contracts come and go, which would give us a many-to-many relationship. Another place we can associate this information is in the album. Once an album is published, the label that produced it will not change, and multiple labels do not publish the same album. To resolve these, we need album in two places.
    a. First, we need a many-to-many relationship table for labels and bands, and a one-to-many link between albums and labels. We already know how to link on-to-many, so we will add a foreign key to producerID in or albums table.
    b. Then we will add a table **Bands2Labels** that has an incrementing auto ID, a foreign key to labels, and foreign key to albums, and a timestamp. If we wanted to round out this information more, we could add start and end timestamps that represent contracts with the label. With the additional of these fields we could create even more timelines.
2. **Artists and Their Bands.** Continuing on, we have our "Artists" table. We know performers can be solo or in groups, and can belong to different bands over time, so we have another many-to-many relationship. We can create ANOTHER join table called **Artists2Bands** representing these relationships.

We now have foreign keys to link our tables together where needed, and do not have a situation where multiple records in a table would contain the same values. We can now be satisfied BOHT that (1) we have a table in 2NF AND that (2) we haven't lost any information. Here's what this might look like in MS Access (with primary keys and references shown).

[1] This is adapted, with significant modifications from: M. Mendez, 2021, *The Missing Link - An Introduction to Web Development.* Open Suny Publishing.

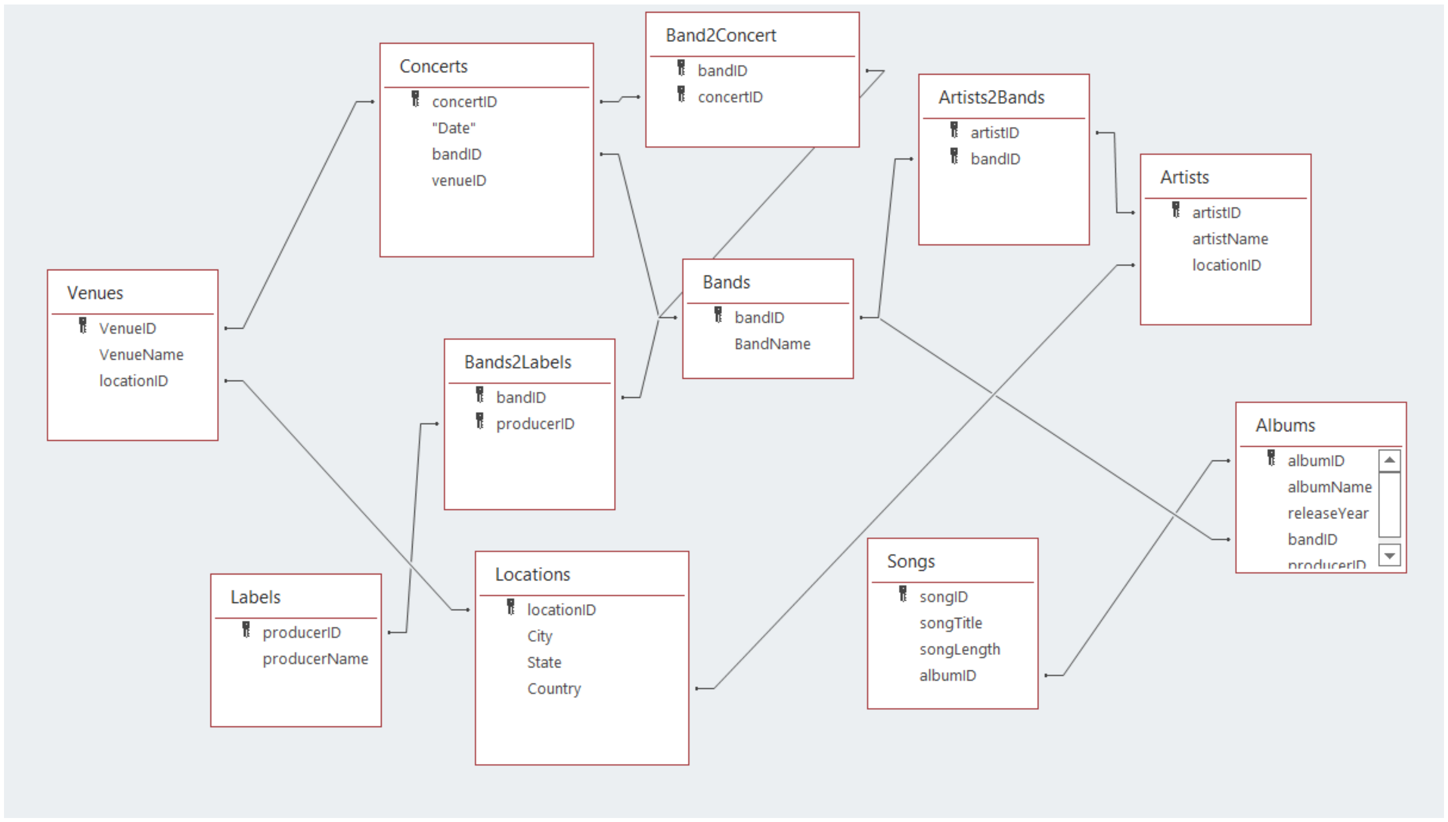


THIRD NORMAL FORM

Third Normal Form requires TWO things: (1) Our data be in second normal form. (2) We remove any **transitive dependencies,** where the value of some (non-key) attribute depends "indirectly" on the key by means of another attribute. Getting to 3N requires a few steps:

1. **Many of our tables are OK!** When we review "Albums" and "Songs" we only have a couple fields to consider from each table as the rest are primary and foreign keys. Album names and release years both refer to albums, and the same holds true for song titles and length in the songs table. Bands2Lables is also easy to review as all of the elements are keys—it is an all-reference table.
2. **A Problems With "Hometown", "HomeState," and "HomeCounty" in Artist, and Why We Need a Location Table.** Next, consider the "Artists" table. Artist name, obviously, fits with artist. What about hometown? Certainly they relate—a person usually identifies one location as home—but the actual information that would reside in the cell (likely a city) does not just relate to an artist. However, HomeState and HomeCountry are *determined* by HomeCity, which means that the "Location" really belong. This makes good, intuitive sense, as Location data is *distinct* from Artist date. For example, we might want to store the locations of "Concerts" as well (which this would allow us to do).
3. **Improving Venues.** Almost there! When we consider the "Venues" table, at first glance we appear to be in third normal form (because we are). While we are here though, we need to keep in mind that in each pass of normalization we need to consider the database as a whole and all of the other forms of normalization as we keep tweaking our tables. In particular, since we are currently storing a venue's *city,* it might make more sense to store it's full location Since we have created a location table, we can take advantage of it here as well.
4. **What if a Concert has Multiple Bands?** Does our dataset *really* capture all of the information we want to capture regarding concerts? Well, not quite yet. We have adjusted our concerts table to better meet third normal form, but that doesn't mean we've fully captured the *semantics* (or "meaning") of real-life concerts, or the type of information we might want to collect about them. In particular, our current design can't model multiple bands perform at the "same" concert. In order to capture this, we'll need to:
   a. Create a new *simple* primary key for Concerts, called concertID.
   b. Create a new table called "Bands2Concerts" (linking bandID to concertID) which allows us to represent a many-to-many relationship between bands and concerts.

Now, all of the tables we had at the beginning of third normal form are complete. We need to review the three we created to make sure they, too, meet all three forms. In this example, they do. Now that we have reached third normal form we can see how normalization helps us out. We could have a list of 2000 concerts in our system, and we can easily *refence* those concerts by means of a single numeric key. In doing this, we do not of repeat all of those details in every record.



FOURTH NORMAL FORM

While most practical design problems (and thus, many tutorials) usually stop at third normal form, we are going to explore a bit further, in order to illustrate the limitations of normalization, and we should NOT simply "normalize to the highest order that we can." There are good, practical reasons for stopping at 3NF in many cases. However, there are cases that will demand higher levels of normalization.

Fourth normal form (4NF) requires that we (1) meet all of the requirements of 3NF and (2) eliminate all **multivalued dependencies**. In particular, the idea is as follows:

- **NOT 4NF.** A single database table has TWO **multivalued** field A and B, where both A and B can take many different values, are not primary or foreign keys, and not dependent on each other (but are both "independently" dependent of on the primary key). So, for example, in a music database these multivalued fields might be anything from "artist name" to "state" to "song title" to "song length" to "producer name."
- **Why's Its (Sometimes) Bad.** This can lead to some data redundancy (for example, when we type in the same artist name again and again, as opposed to storing the artist name in its own table). It wastes storage space. This process can be helpful when you expect very high volumes of records and/or need to be very mindful of the size, or footprint, of your database (for example, running the system on a smartphone).
- **What Can We Do?** We can separate each multivalued field into its own table, and link them together through join tables (just as we did with 1NF, 2NF, and 3NF).

Getting to 4NF. The design process that we've followed so far actually take a fair bit of the way toward 4NF. At the time, not only did we split out tables where we found one-to-many relationships, we also split out all-reference tables to account for the many-to-many relationships we found. This was easy to do at the time as we were focused on looking for those relationship types. However, if we look closely, there are still a number of multi-valued dependencies:

- Songs contains fields for both songLength and songTitle. We could change one of these without necessarily changing the other.
- Albums contains fields for both albumName and releaseYear. Again, these are logically independent.
- Getting to 4NF would require separating these out into new tables.
- For many practical purposes, this wouldn't be worth it, and might even hinder database performance/flexibility (since it would require a JOIN every time we wanted to ask something like "How long is the song with title T?"

Congratulations, you now have a normalized database! Flip back to look at our original design, and you will see a number of trends. First, the process was a bit extensive, and turned into far more tables than you likely expected. However, it also helped us identify more pieces of information that we thought we would want, and helped us isolate the information into single pieces (like splitting location in city and sate) which allows us to search by any one of those items.