WHY CAN’T YOU BE NORMAL? DATABASE NORMAL FORMS[1]

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

Database normalization is the process of structuring and refining the data we want to store in such a way that we eliminate repeated information and represent as much connection between records as possible. It has its roots in the work of Edgar Codd (the creator of relational databases). When a database meets particular rules or features of normalization, it is usually referred to as being in a particular normal form. The collection of rules progress from the least restrictive (first normal, or 1NF) through the most restrictive (fifth normal, or 5NF) and beyond. Databases can still be useful and efficient at any level depending on their use, and anything beyond third normal form is more of a rarity in real-world practice.

Bear in mind, that normalization is a theory on data organization, not law. Your database can operate just fine without adhering to the following steps, but following the process of normalizing will make your life easier and improve the efficiency of your website. Not every set of circumstances will require all of these rules to be followed. This is especially true if they will make accessing your data more difficult for your particular application. These rules are designed to help you eliminate repeated data, are able to keep your overall database size as small as possible and create integrity in your records.

AN OVERVIEW OF THE PROCESS

The process of normalization involves thinking carefully about the **candidate keys** (or **prime attributes,** as they are called in relational algebra) for each table, and the *relationships* that non-key attributes (or **non-prime attributes)** bear to them. Here’s the process. To simplify, I’ve assumed there is only ONE candidate key (which may be simple or composite)

1. **Zero-Normal Form (0NF)** is our “starting” point of unnormalized data. There is often *no* candidate key, and some “cells” may contain many values. This means the data can’t be entered into a relational database.
2. **First Normal Form (1NF)** requires that we (1) create tables for related information so that we have no **repeating groups** (which occur when we have *more than one item in a single table cell*) and (2) assign an appropriate primary key for each table.
    a. Data in 1NF *can* be entered into a relational database (unlike 0NF).
    b. We still might have lots of data redundancy, though, because many of our attributes might be “determined” by attributes *besides* those we’ve chosen as the primary key.
3. **Second Normal Form (2NF)** requires that we (1) satisfy 1NF and (2) eliminate all **partial dependencies** (where attributes depend on only *part of* a primary key). We do this by moving the partially dependent data (and the attribute it depends on) to a new table, which we link to existing tables by means of a foreign key.
    a. This reduces redundancy, since we no longer need to enter data of the form “A, B, and C” when we *know* that the values of B and C depend on A. (We can simply enter “A” instead!).
4. **Third Normal Form (3NF)** requires that we (1) meet second normal form and (2) move/eliminate attributes that depend on any attributes *besides* the primary key (called **transitive dependencies).**
    a. Similar to 2NF, this reduces redundancy. For most business/scientific purposes, we can stop here.
5. **Fourth Normal Form (4NF)** requires that we (1) Meet third normal form and (2) Has no **multi-valued dependencies.** A multi-valued dependency occurs when attribute X fully determines independent attributes Y and Z, and both Y and Z can take any many values.
    a. The main goal of 4NF is to reduce the number of “NULLS” we have to enter for Y and Z (as opposed to worries about data redundancy). However, there is a practical tradeoff, since doing so will (a) require we create more tables, more primary keys, and more foreign keys and this, in turn, means that (b) some database operations requiring JOINS will be slower.

Codd’s Slogan. The idea behind normalization sometimes expressed in 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 theorist who gave us these rules.

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

GETTING STARTED: ZERO NORMAL FORM

What sort of data do we need store? To begin, we need some data to normalize. To provide a concrete example, let's suppose would like to create a database to keep track of a music collection. First, we need a list of what we want to track. We will follow what is generally useful for a collection of music, like albums, artists, and songs. These categories give us a list of things we want to store, so let us come up with what a list of attributes that we might need to store:

Music Database: Data to Record

Band Name	Album Title	Song Title	Song Length	Producer Name
Release Year	Artist Hometown	Concert Venue	Concert Date	Artist Name

To get a visual of what this table BEFORE we normalize it (for example, when it is still in spreadsheet form) might look like consider the following excerpt from a spreadsheet (which might contains As an example of non-normalized or "zero normal form" data, you can look to the data above where you see long, repeated fields. While this table is easy to read without a query or software, it quickly becomes unmanageable even in its readable format as 25 records turn into just a few hundred. Let us take a summary look at our forms that will help us tackle this problem: When working with an existing data set like our example above, you can quickly move through normalization once you are familiar with the rules. By adjusting our table(s) until they meet each set of rules, our data becomes normalized. Since we are learning, we will formulate our database from scratch.

Band (Hometown)	Album (Year)	Tracks : Time	Tour Locations
Pink Floyd (London, UK)	Darkside of the Moon (1973)	Speak to Me : 01:08, Breathe: 02:49 , On the Run: 03:51	U of London (London), Wrigley Field (Chicago), et
Pink Floyd (London, UK)	The Wall (1980)	More tracks	More tour locations
Prince (Minneapolis, MN)	Purple Rain (1984)	"Let's Go Crazy", 4:39 "Take Me with U", 3:54 "The Beautiful Ones", 5:13	First Avenue (Minneapolis), Metro (Chicago)
Prince (Minneapolis, MN)	1999 (1982)	More tracks	More tour locations
Many more artists and albums			

FIRST NORMAL FORM: GETTING TO A PRIMARY KEY

When we put a table in First Normal Form (1NF) we are basically ensuring that *it is the sort of thing that can be entered into a relational database at all*. To do this, we need to ensure TWO different things:

1. Each row/column intersection (each "cell" of the table) has exactly ONE value. So, for example, we don't want a single cell with the values for *different* song titles, or which contains information on both artists and their hometown. We'd want each character to have their OWN row.
2. Closely related to this, there is a primary key for each table that fully determines all other attributes.

To get to 1NF, we need to start thinking about which tables will store which attributes.

Why "Band" is good choice for a table. "Band Name" refers to the official name of the group or artist we are talking about. A good question to ask is does the column sounds like a concept, object, or idea that represents something, or would have properties, in our database. The concept of a band for us is important, and it will have properties made up of other columns we plan to have, like songs and albums. Since we can see this is a concept, it is a good candidate for a table, so we will start by creating a table for our bands.

With databases, it can be helpful to treat your tables as the plural form of what its rows will contain. In this case, we will name our table **Bands.** Under this name, we will list "Band Name" to the list of information in that table. A key element to think about every time we consider adding a field to a table is to make sure it represents only one piece of information. A band name meets our criteria of being only one piece of information, so we are on track. Now, we can add the following to our design notes

Table Name:	**Bands**
Attribute List	Band Name

Next Table: Albums: Our next element, Album Title, seems like it would relate to a band. At first, we might be tempted to put it in the band table because the album record we add belongs to the band. However, when we consider our data relationships, it becomes clear that we have a one to many situation; one band will (one-hit wonders aside) release more than one album. Since we always program to meet our highest possible level of relationship, the one-hit-wonders will exist perfectly fine even with only one album in our database. If you recall, we address one to many relationships by placing the two sides in separate tables and identifying in the "many" table which "one" the record is paired with. To do this, we will make an albums table just like we did for bands, and add a placeholder in our note to refer to the band table:

Table Name:	**Bands**	**Albums**
Attribute List	Band Name	Album Name (Reference to Bands)

Table 3: Songs. Now we are on to "Song Titles." The songs are organized into albums, so we will add it in. Apply our tests and see if this works. Does this field represent one piece of information? Titles are plural as we have it written, but we cannot put more than one piece of information in a single cell. We will need to break that up into individual titles to resolve. To make that change in this table though, we would have to create a column for each track. In order to do that, we would need to know ahead of time the max number of tracks any album we enter will have. This is not only impractical but violates our first normal form of not repeating columns. Through this, we can see that we again have a one to many relationships on our hands as one album will have multiple tracks. To resolve this, once again we will break our multiple fields out to its own table, where we can create a link back to our album table:

Table Name:	**Bands**	**Albums**	**Songs**
Attribute List	Band Name	Album Name (reference to Bands)	Song Title (reference to albums)

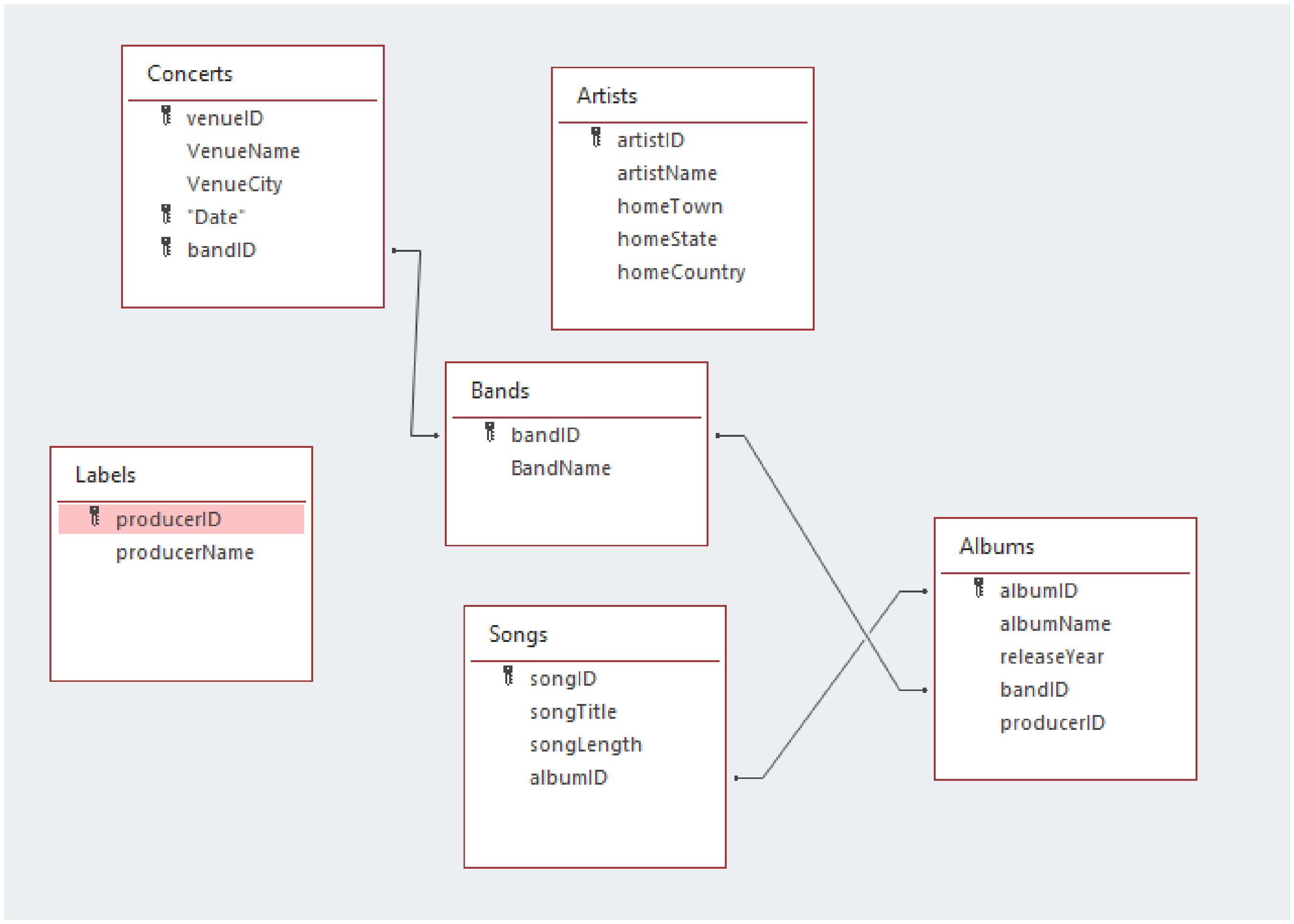
We can already see a thread weaving its way through our tables. Even though these fields are no longer all in one record together, you can see how we can trace our way through by looking for the band we want in the albums table, and when we know the albums, we can find all the tracks the band has published. To continue with our design we will move to song length. This field sounds fitting in our songs table and is only one piece of information, so we are off to a good start! We can also see that we would only have one song length per record as each record here is a song, so we comply with column count, too. We can put it there for now and will see if it meets the rest of our tests as we move on.

Table Name:	**Bands**	**Albums**	**Songs**
Attribute List	Band Name	Album Name (reference to Bands)	Song Title (reference to albums) Song Length

More Tables: Labels and Concerts. Now that we have an idea of first normal form, we will get the rest of our initial columns out of the way (I've consolidated the format a bit here):

Bands Band Name	**Albums** Album Name Release Year (reference to Band)	**Songs** Song Title Song Length (reference to Album)
Labels Producer Name	**Artists** Artist Name Hometown	**Concerts** Venue Location Venue Name Date (reference to Band)

Now that we have exhausted our initial list, we will consider the last element of 1NF, which is ensuring that each table has a primary key. Many of our tables are not presenting us with good candidates, as band names, venues, albums, tracks, and even artists could share the same names as time goes on. To make things consistent, we will create auto incrementing IDs for each table. To follow best practices, we will use the singular version of the noun with ID after it to denote our primary keys. This identifies the row as a singular version of the concept our table name is a plural of.



Have We Met the Requirements of 1NF? Let's quickly double-check that we are on the right track:

1. Does each table have a primary key? YES!
2. Are there any attributes that look like they will contain "repeating groups" (so, for example, that store multiple artists, songs, etc. in a single cell)? NO!

A QUICK MOVE TO 2NF

Once we've gotten to 1F, we are only a hop, skip, a jump away from second normal form (2F), which requires that we ensure that there are no *partial dependencies* in our table. Technically speaking, a partial dependency occurs when

(1) there is a composite candidate key and
(2) some other (non-key) attribute depends on only part of this key.

For example, the following table would SATISFY 1NF but VIOLATE 2NF:

Made Up Muscians		
Artist	Album	State
Kayne East	Greatest Hits	NY
Kayne East	Songs	NY
Drink Boyd	Album #7	MN
Grimey	The Weeds	MN
Blue	Hello!	IA
Taylor Slow	2022	CA
Taylor Slow	Greatest Hits	CA

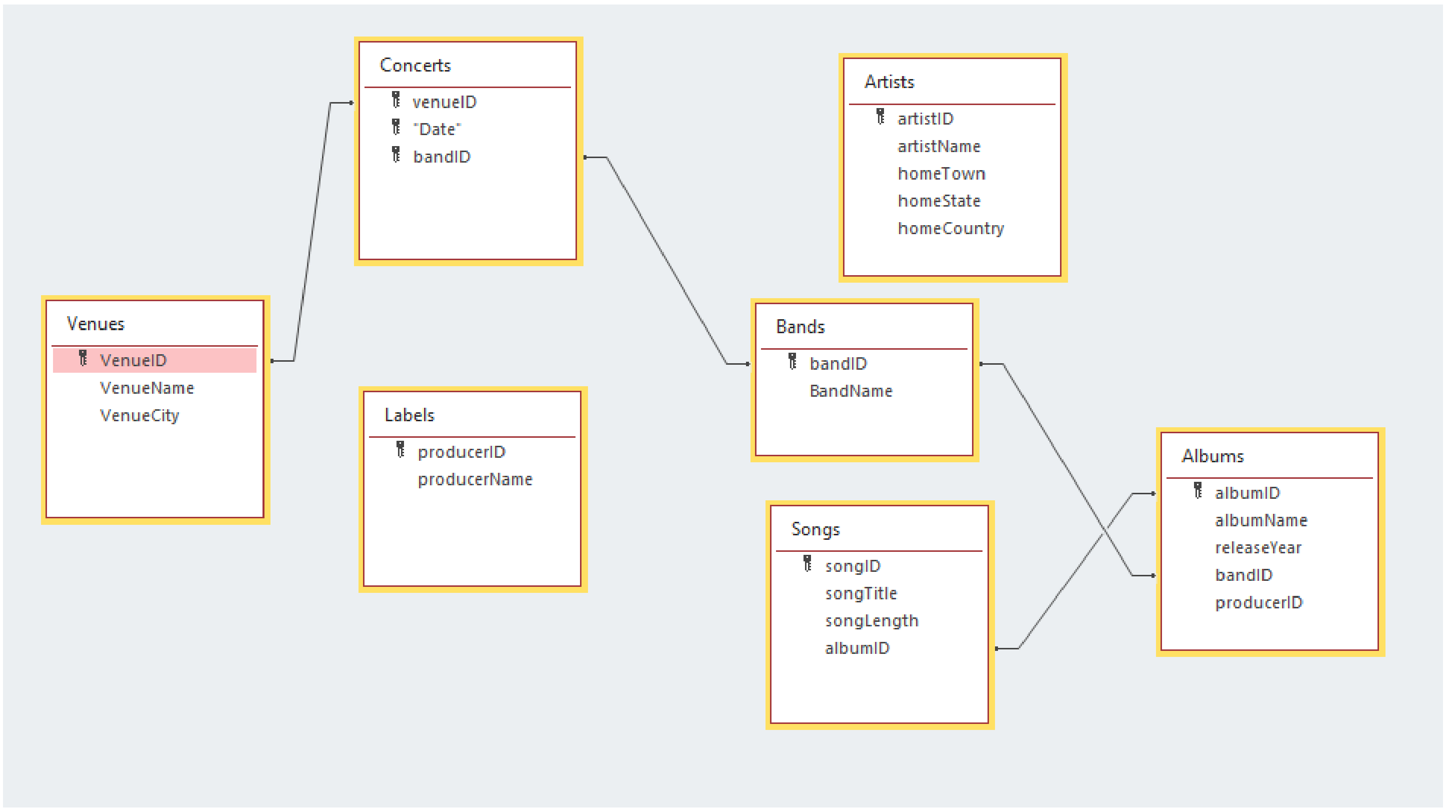
We are already most of the way toward 2NF, which requires that we eliminate any *partial dependencies* on components of composite keys. However, we do have ONE partial dependency, in the Table Concerts:

A row from the version of the Concert table given above might look something the following:

venueID (PK)	venueName	venueCity	Date (PK)	bandID (PK)
73	Civic Center	Rochester, MN	5/21/2022	5134
73	Civic Center	Rochester, MN	4/20/2013	2678
4	First Avenue	Minneapolis, MN	4/20/2013	5134
4	First Avenue	Minneapolis, MN	11/11/2019	3466

So:

1. Concerts has a composite key made up of Venue ID, bandID, AND date.
2. However, venueID *by itself* determines things such as the venue name and location. **This violates 2NF, and leads to data redundancy. Oh no!**
3. We can solve this by creating a new "Venue" table that contains information on venues. Our new "Concerts" table can then contain a reference to this table. Our REVISED relational schema now looks like



And we are in 2NF!

Where to Next? For many business/scientific applications, we'll want to go beyond 2NF (to 3NF), as this can help us further reduce data redundancy. However, for some applications (such as **data warehousing,** which involves storing a ton of semistructured, which involves a ton of semi-structured data, but involves comparatively few updates/complex queries), 2NF may be a perfectly fine stopping point.