

4.4: Normalization

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. 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.

Zero Normal Form

To begin, we need something to normalize. Through this section we will 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 full record might contain:

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 might look like, let us take a look at some sample data with many of these fields:

SONG TITLE	ARTIST	GENRE	SUB-GENRE	YEAR
Shannon	Henry Gross	Rock	Light Rock	1976
Lover's Will	Bonnie Raitt	Rock	Light Rock	1998
I Don't Wanna Live Without Your Love	Chapercago	Rock	Light Rock	1988
Heart Attack	Olivia Newton-John	Pop	Adult Contemporary	1982
In A Dream	Badlands	Rock	Hard Rock	1991
With A Little Luck	Paul McCartney	Rock	Classic Rock	1978
It's A Miracle	Barry Manilow	Pop	Adult Contemporary	1975
It's Only Love	Bryan Adams / Tina Turner	Pop	Adult Contemporary	1984
Jazzman	Carole King	Pop	Adult Contemporary	1974
Jesse	Carly Simon	Pop	Adult Contemporary	1980
Just Like Jesse James	Chapterr	Pop	Adult Contemporary	1989
Little Miss Cannot Be Wrong	Spin Doctors	Pop	Adult Contemporary	1992
Lost In Love	Air Supply	Pop	Adult Contemporary	1980
Good Times	Sam Cooke	Hip-Hop	Soul	1964
Make It With You	Bread	Pop	Adult Contemporary	1970
Mandy	Barry Manilow	Pop	Adult Contemporary	1974

Miss Chaptertelaine	K.D. Lang	Pop	Adult Contemporary	1992
Never Gonna Fall In Love Again	Eric Carmen	Pop	Adult Contemporary	1976
People Get Ready	Rod Stewart	Pop	Adult Contemporary	1985
Try Honesty (Radio Version)	Billy Talent	Rock	Modern Rock	2007
Silver Threads And Golden Needles	Linda Ronstadt	Pop	Adult Contemporary	1974
So Far Away	Carole King	Pop	Adult Contemporary	1971
Fat Lip	Sum 41	Rock	Modern Rock	2001
Thank You For Being a Friend	Andrew Gold	Pop	Adult Contemporary	1978

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 turns into just a few hundred.

Let us take a summary look at our forms that will help us tackle this problem:

First Normal Form

- Create separate tables for related information
- Eliminate duplicated columns within tables
- Create primary keys for each table

Second Normal Form

- Meet first normal form
- Move data that will repeat often into a reference table
- Connect reference tables with foreign keys

Third Normal Form

- Meet second normal form
- Eliminate columns that do not relate to their primary key

Fourth Normal Form

- Meet third normal form
- Has no multi-valued dependencies

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.

Additional notes

As we draw up our design, each bolded word represents a table, and the names underneath represent our columns. We can simulate a record, or row, in our database by writing out sample values for each thing in our list.

First Normal Form

To get started, we will go through our draft list piece by piece. “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. Just like we studied in the PHP section, it is good practice to keep a set of conventions when naming

elements. 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 our design notes might look more like this:

Band Name	Album Title	Song Titles	Song length	Producer Name
Release Year	Artist hometown	Concert Venue	Concert Date	Artist Names

Bands Band Name		
---------------------------	--	--

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:

Band Name	Album Title	Song Titles	Song length	Producer Name
Release Year	Artist hometown	Concert Venue	Concert Date	Artist Names

Bands Band Name	Albums Album Name (reference to Band)	
---------------------------	--	--

Now we are on to “Song Titles.” The songs are organized into albums, so we will add it in.

Albums

Album Name

Song Titles

Apply our tests and see if this works. Does this field represent one piece of information? Titles is 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 relationship on our hands as one album will have multiple tracks. To resolve this, once again we will break our multiple field out to its own table, where we can create a link back to our album table:

Band Name	Album Title	Song Titles	Song length	Producer Name
Release Year	Artist hometown	Concert Venue	Concert Date	Artist Names

Bands Band Name	Albums Album Name (reference to Band)	Songs Song Title (reference to Album)
---------------------------	--	--

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.

Band Name	Album Title	Song Titles	Song length	Producer Name
Release Year	Artist hometown	Concert Venue	Concert Date	Artist Names

Bands Band Name	Albums Album Name (reference to Band)	Songs Song Title Song Length (reference to Album)
---------------------------	--	---

Now that we have an idea of first normal form, we will get the rest of our initial columns out of the way:

Band Name	Album Title	Song Titles	Song length	Producer Name
Release Year	Artist hometown	Concert Venue	Concert Date	Artist Names

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 Date

Now that we have exhausted our initial list, we will consider the last element of 1NF, which is primary keys for each table. 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:

Bands bandID Band Name	Albums albumID Album Name Release Year (reference to Band)	Songs songID Song Title Song Length (reference to Album)
Labels producerID Producer Name	Artists artistID Artist Name Hometown	Concerts venueID Venue Date

Second Normal Form

We have now reached first normal form. Now I must admit, I have been a bit sneaky. By introducing data relationships, and showing you how to apply the relationship when considering where to put columns, we have already addressed part of second normal form, so, technically, we are already beyond first normal form. The first piece of second normal form is creating tables anywhere where a value of a cell could apply to multiple records of that table. When we moved song title out of albums, we were fulfilling this requirement. Looking over our tables again, we can see that, as we have things now, this has been met.

The other element of second normal form is that connections between tables should be facilitated by foreign keys. 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. To start with the tables we earmarked, we will look at “Albums.” Our reference calls for connecting it to “Bands,” so we will add a foreign key in “Albums” that points to “Bands.” To make things easy on us, we can use the same name in both tables so we know what it is for.

Bands bandID Band Name	Albums albumID Album Name Release Year bandID	Songs songID Song Title Song Length (reference to Album)
Labels producerID Producer Name	Artists artistID Artist Name Hometown	Concerts venueID Venue Date

We can do the same with our “Songs” table as well to reference our “Albums” table. 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. 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 our albums table. Then we will add a table that has an incrementing auto ID, a foreign key to labels, and a timestamp:

Bands bandID Band Name	Albums albumID Album Name Release Year bandID producerID	Songs songID Song Title Song Length (reference to Album)	
Labels producerID Producer Name	Artists artistID Artist Name Hometown	Concerts venueID Venue Date	Bands2Labels id producerID bandID timestamp

By adding the timestamp column in our many-to-many table, we can sort by the date the records were added, assuming they were added chronologically. This means the newest record would represent who the band is signed with now, and we can look at all the records with a particular band to see who a band has worked with, and we can look at all the records for a label to see who they have signed.

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.

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. You will notice the name given to the table bridging our bands and labels relationship is labelled Bands2Labels. This of course is only one possible name we could use, but is an example of how to clearly identify the purpose of the table, as we are linking “bands to labels.” Our last table to look at is “Concerts.” We need a way to associate a particular concert with the band that performed. Since each row of this table is a particular concert we will add a foreign key in.

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 have now reached second normal form.

Third Normal Form

Our next normal form requires that all of the fields in a table relate to the key (or are a key), or in other words the concept of that table. In our smaller tables this is immediately apparent to us—a band name relates directly to a band, and a producer name relates directly to a label. It can be remembered in a popular rewording to the well-known court room oath that references Edgar Codd, who created the concept of third normal form. My favorite variation is the following: “All columns in the table must relate to the key, the whole key, and nothing but the key, so help me Codd”(source unknown).

To see third normal form in action we will review our current design. We already considered bands and labels while describing this form, so we will mark them green as OK.

Bands bandID Band Name	Albums albumID Album Name Release Year bandID producerID	Songs songID Song Title Song Length (reference to Album)	
Labels producerID Producer Name	Artists artistID Artist Name Hometown	Concerts venueID Venue Date	Bands2Labels id producerID bandID timestamp

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. Bands2Labels is also easy to review as all of the elements are keys—it is an all-reference table.

Bands bandID Band Name	Albums albumID Album Name Release Year bandID producerID	Songs songID Song Title Song Length (reference to Album)	
Labels producerID Producer Name	Artists artistID Artist Name Hometown	Concerts venueID Venue Date	Bands2Labels id producerID bandID timestamp

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. Looking at our concert table for example, a venue would have a physical location in which it resides as well. This tells us that hometown needs to be moved somewhere else. Since we do not have a place in our database that speaks specifically to locations, we will have to add one. When we do this, we should also consider that in reality the hometown city name by itself is not sufficiently unique without a state and zip code reference as well, and we will need to change our existing hometown column to reference the new table:

Bands bandID Band Name		Albums albumID Album Name Release Year bandID producerID	Songs songID Song Title Song Length (reference to Album)	
Labels producerID Producer Name	Locations locID city state zip	Artists artistID Artist Name locationID	Concerts venueID Venue Date	Bands2Labels id producerID bandID timestamp

Almost there! When we consider the “Concerts” 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. Here, while venue makes sense as a column, using venue as the primary key seems confusing, as we are identifying a particular concert, not a particular place. When we consider this, it may also become apparent that just knowing the name of a venue may not be enough to uniquely identify it either. Since we have created a location table, we can take advantage of it here as well:

Bands bandID Band Name		Albums albumID Album Name Release Year bandID producerID	Songs songID Song Title Song Length (reference to Album)	
Labels producerID Producer Name	Locations locID city state zip	Artists artistID Artist Name locationID	Concerts concertID Venue locID Date	Bands2Labels id producerID bandID timestamp

Have we satisfied all forms? Well, not quite yet. We have adjusted our concerts table to better meet third normal form, but is it fully compliant or did we miss something? Imagine this table populated and you will notice that the venue field—the name of our location—would be repeated each time the venue was used. This violates second normal form. To solve this, we know we need to split the data out to its own table, so we need to see if anything else should go with it. The location ID we just created relates to the venue, not the event, so that should go too. The date is correct where it is, as it identifies a particular piece of information about the concert. Does this cover everything? We do not seem to have a means to identify who actually performed the concert at that venue on that date, do we? This is a key piece of information about a concert, and any given concert usually involves more than one performer. Not only do we need to add the field but we need to remember our data relationships and see that this is a many-to-many between artists and concerts. Multiple artists can perform at the same event, and with any luck a given artist will perform more than one concert. We can address all of these changes by creating a Venues table, a many-to-many reference table for concerts and performers, and adjusting our concerts table to meet these changes. Try writing it out yourself before looking at the next table!

Bands bandID Band Name		Albums albumID Album Name Release Year bandID producerID	Songs songID Song Title Song Length (reference to Album)	Concerts2Artists Id artistID concertID
-------------------------------------	--	--	---	--

Labels	Locations	Artists	Venues	Concerts	Bands2Labels
producerID	locID	artistID	venueID	concertID	id
Producer Name	city	Artist Name	Name	venueID	producerID
	state	locationID	locID	Date	bandID
	zip				timestamp

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 each of those records would just be numerical references to one record for each artist and concert. In doing this, we do not of repeat all of those details in every record.

Fourth Normal Form

While most systems (and most tutorials) usually stop at third normal form, we are going to explore a bit further. Fourth normal form is meant to address the fact that independent records should not be repeated in the same table. We began running into this when we looked for problems in complying with second normal form as we began to consider the data relationships between our fields. 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. That also means, however, that we have already met fourth normal form, by preventing many-to-many records from creating repeated values within our tables.

To keep fourth normal compliance in other systems, you will need to be mindful of places where user-submitted data could be repeated. For example, you may allow users to add links to their favorite sites. Certainly at some point more than one user will have entered the same value, and this would end up repeated in the table. To prevent this, you would store all links in a table and create a many-to-many reference with your user records. Each time a link is saved, you would check your links table for a match, update records if it exists, or add it to the table if it has never been used.

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).

Before we move on, we will make one more pass to clean up our design. Since we started with words as concepts, but want to honor best practices when we create our database, we will revise all of our tables to follow a consistent capitalization and pluralization pattern:

Bands bandID bandName		Albums albumID albumName releaseDate bandID producerID		Songs songID title length albumID	Concerts2Artists id artistID concertID
Labels producerID producer	Locations locID city state zip	Artists artistID artistName locationID	Venues venueID venueName locationID	Concerts concertID venueID date	Bands2Labels id producerID bandID timestamp

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, state, zip) which allows us to search by any one of those items.

[Learn more](#)

Keywords, search terms: Normalization, boyce-cobb normal form

MySQL's Guide: <http://ftp.nchu.edu.tw/MySQL/tech-resources/articles/intro-to-normalization.html>

High Performance MySQL: <http://my.safaribooksonline.com/book/-/9781449332471>

4.4: Normalization is shared under a [not declared](#) license and was authored, remixed, and/or curated by LibreTexts.

- 4.4: Normalization by [Michael Mendez](#) has no license indicated.