

## 4.6: Advanced Queries

To use relational databases to their fullest extent, we need to be able to connect our tables using our foreign keys in order to extract our full records. This is done by combining statements and joining tables together.

### Joining

We can join tables a number of ways. The primary portion of the join is specified in our where clause, which is where we will specify which fields between the two tables are to be connected. The simplest join will return only the records from the two tables where the value is found in both tables. We will begin by getting all the values from our albums and bands tables so we can finally see the band name in our results:

1. Select \* from bands, albums where album.bandID=bands.bandID;

You now probably have a messy looking table, but all our fields are there. Now, we can cut it back to just a simple list:

1. Select bands.bandName as "Band", albums.albumName as "Album", releaseDate as "Released" from bands, albums where albums.bandID=bands.bandID;

Band	Album	Released
The Who	Tommy	1969-05-23
Moxy Fruvous	Bargainville	1993-07-20
The Doors	Full Circle	1972-07-17
The Doors	Strange Days	1967-10-16

You will notice we started to append the table name and a period before each field name. This helps clarify which field we are referring to when the same field name is used in more than one table. As a best practice, you may wish to always use this dot notation when selecting fields as it will help “future-proof” your queries if you expand or alter your database in the future, even if the field in question is unique to the database now.

You may also be noticing that we still have not seen Maroon 5 come up in any of these examples, even though they were created when we first set up our bands table. That is because the basic join (also called “inner join”), as stated above, only returns results where records exist in both joined tables. Since we never added an album for Maroon 5, they did not come back as a result. We can capture all results of either table, and still pair them when records are available, by using different approaches to our join called left join and right join. Each of these performs roughly how it sounds—a left join will include *all* records from the left table, plus additional values from the right table when they exist, and a right join will do the opposite, including all the records from the right table and adding data from the left when it exists. Next we will look at all “Bands,” and any records they have, by using a left join. All we need to do is replace the comma between the tables in our “from” clause with the join method we want, and change the word “where” to “on”:

1. Select bands.bandName as "Band", albums.albumName as "Album", releaseDate as "Released" from bands left join albums on albums.bandID=bands.bandID;

Band	Album	Released
The Who	Tommy	1969-05-23
Moxy Fruvous	Bargainville	1993-07-20
The Doors	Full Circle	1972-07-17
The Doors	Strange Days	1967-10-16
Maroon 5	NULL	NULL

There are more complex forms of join than just left, right, and inner. However, these three cover most use cases and a well-designed database will usually reduce or eliminate the need for overly complex queries.

We can continue to add tables, and joins, to our queries to get more and more comprehensive results. We can even nest queries within one another inside sets of parenthesis. The query is then executed from the inside out just like it would in an equation, where the resulting data from the nested query is available to the query it sits inside of. First, we can look at a more complicated query that tells us everything about a particular song. We will specify a song title, and build a query that would connect all the related tables. Since our database is limited, we will start by looking at our table structure. If you want to fully test this example, you will need to spend some time populating your tables further. Since we will have a song title, and the question is what else we can glean, we will use all the keys that make sense in combination with a song title. Within the song table, we have albumID. That is relevant, as it tells us the album(s) the song has been released on. Now that we have at least one albumID, we can get from our “Albums” table to the band table and producer table as well. Tracing to these does not reveal any additional keys we can use, so without extra nested queries this is our reach:

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

We are able to connect data from half of our database (ignoring our reference tables) just from having a song title. This query could look like the following:

1. Select bandName, albumName, releaseDate, title, length, producer from bands, albums, songs, labels where songs.albumID=albums.albumID and albums.bandID=bands.bandID and albums.producerID=labels.producerID;

Each pairing of fields in our where clause creates another join between tables. As our queries become more complex, you may find they take longer to run. This is because more data has to be reviewed, and more connections found, to create the resulting table.

This is also the point where optimization techniques like indexing (automatically building trees in the database) and other more advanced MySQL tools will come into play.

## Nested Queries

We can take the results of one query into consideration in another by nesting queries within one another using parenthesis. This will frequently come into play when we do not have a starting value for a question we want to ask. For example if we wanted to find the artist with the largest album in terms of tracks, we would break the goal down into its elements. First, we need to find which album has the highest track count, since we do not have a known value to search for:

1. Select max(length) from songs;

This query looks at each record in the songs table and finds the one with the largest value. We could also have done this by sorting the table as descending on the tracks column, but since we are going to nest it we only want one value returned to keep things simpler. Our next step is to join the bandID value from albums to the id field in our bands table, in order to get our name:

1. Select bandName from bands, albums, songs where songs.length= (select max(length) from songs) and songs.albumID=albums.albumID and albums.bandID=bands.bandID;

Nested queries are also a great place to use a few more methods to search with, namely ANY, IN, SOME, ALL and EXISTS. ANY and SOME are equivalent, and IN works the same as ANY when your comparison is strict equality (just =). What this means is that when we interact with the results of a nested query, we can look at each record returned as a match against our where clause. Let us look at some mock examples:

Get the name of every artist who has an album with the word “free” anywhere in the title:

1. Select artistName from artists where artist.id = ANY (select artist.id from albums where album.title like "% Free %");
2. Select artistName from artists where artist.id IN (select artist.id from albums where album.title like "% Free %");
3. Select artistName from artists where artist.id = SOME (select artist.id from albums where album.title like "% Free %");

To understand where these verbs become different, we could ask the question of which album(s) contain a certain set of songs. In this case, our nested select would include the songs we are interested in. Here, using the verb ANY would return all albums that have one or more of the songs listed on their albums. If we changed to ALL, then we would only get albums where all of the values returned by our nested query existed on the album.

## Learn more

Keywords, search terms: Nested queries, sql joins, indexing, mysql optimization

Jeff Atwood's Joins Examples: <http://www.codinghorror.com/blog/2007/10/a-visual-explanation-of-sql-joins.html>

MySQL's Nested Examples: [http://dev.mysql.com/tech-resources/articles/subqueries\\_part\\_1.html](http://dev.mysql.com/tech-resources/articles/subqueries_part_1.html)

---

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

- 4.6: Advanced Queries by [Michael Mendez](#) has no license indicated.