

# Joins

## Solomon Kevin Chang

### Hello World – The Full Join

*“What do you want to do tomorrow night, Brain?”*

*“The same thing we do every night, Pinky...”*

*--Animaniacs*

Okay, so we're not going to plan to take over the World. We're not going to even try to change the world, at least not yet. We're just going to just look at a database called World. We've seen from earlier lessons that you can specify multiple tables in your FROM clause:

```
select * from City, Country;
```

However, this doesn't provide much useful information, since it returns (a la Dr. Evil) “every frickin' row from table `City` in every frickin' combination possible with every frickin' row from table `Country`”. In other words, if there are 4079 records in `City`, and 239 records in the `Country`, the above query will return 974,881 records ( $4079 * 239 = 974,881$ ). That's a lot of sharks with frickin' lasers strapped to their heads. You can try it for yourself, but if you plan on wading through the tedium of your computer scrolling through all those records, I suggest watching an equally exciting alternative activity, such as Paint Drying, Grass Growing, or Tournament Poker.

Congratulations – you've just done the simplest kind of Join in MySQL Land: the Full Join, derived from the archaic term “Fullmonty”, which translates literally to, “Show me everything”.

Hmm, tough crowd. Is this thing turned on?

### Adventures Through Inner Join

*“How often have not the daemons of \*Nix... drawn 'Aints to become 'Inners?”*

*--Luther*

...But chances are, you want to filter for something that's actually useful, such as the Continent that every `City` appears on. Okay, that's a problem, because names of cities are stored in the `City` table, and Continents are stored in the `Country` table. If all we want is simply a Continent and City Name, the first thing that comes to mind is to filter our fields:

```
select City.Name, Country.Continent from City, Country;
```

Whoops, that *still* returns almost a million rows, not to mention the fact that most of the Continents we see don't have any cities by *those* names.! We filtered for fields but we didn't filter for records themselves. Let's add a where clause:

```
select City.Name, Country.Continent from City, Country
where City.CountryCode = Country.Code;
```

What this does is: keep only the records in which `CountryCode` in the `City` table equals the `Code` field in the `Country` table. Our end result is that we've retained every record in the `City` table, paired up with any

corresponding record in the `Country` table. These records correspond by Keys: the `CountryCode` field in `City` correspond to the `Code` field in the `Country` table. The `Code` field in the `Country` table also happens to be our Primary Key (which we have already covered in a previous class). The `CountryCode` field in the `City` table is a Foreign Key – it looks up values against another field (often the Primary key) of another table. We have, in essence, relegated the `Country` table into a “lookup” chart to fill in some of those values we were wondering about. Just as values of `CountryCode` may repeat, values drawn in from `Country` may thus repeat as well.

So you've learned to juggle two, are you ready to juggle three?

```
select City.Name, Language
from City, Country, CountryLanguage
where City.CountryCode = Country.Code
and CountryLanguage.CountryCode = Country.Code;
```

We now get our previous list of cities, plus any languages they might speak... but wait! There's now 30670 records! This is far more than the 4079 Cities in our database, and far more than the 984 Language-to-Country mappings listed in our `CountryLanguage` table. Where did these extra records come from?

Let's look at some of the output from this last query:

```
+-----+-----+
| Name      | Language      |
+-----+-----+
| Kabul     | Balochi       |
| Kabul     | Dari          |
| Kabul     | Pashto        |
| Kabul     | Turkmenian    |
| Kabul     | Uzbek         |
| Qandahar  | Balochi       |
| Qandahar  | Dari          |
| Qandahar  | Pashto        |
| Qandahar  | Turkmenian    |
| Qandahar  | Uzbek         |
+-----+-----+
```

It appears that we now have repetition amongst our cities. That's because their countries have multiple language listings in the `CountryLanguage` table. When we joined against that table, countries with multiple languages received that many more records. In short, countries have multiple cities and multiple languages; our Inner Join Example was to show all possible combinations of their correlations. It becomes difficult to predict exactly how many records you will get back as you increase the number of tables you join together.

Note that you do not have to show fields from every table in your Join clauses. In the above example, we are using `Country` as an intermediary table to link `City` and `CountryLanguage`.

## Weaning Yourself Off Of Comma Notation

*“Comma Comma Comma Comma Chameleon... You come and go. You come and go...”*  
*--Culture Club*

Up until now, our joins have been defined strictly by naming them in the `from` clause, joined only by fields named in the `where` clause. This is known as Comma Notation. Even though the study guide portrays almost all joins in comma notation, you will be expected to also know Inner Join notation. In our earlier example:

```
select City.Name, Country.Continent from City, Country
where City.CountryCode = Country.Code;
```

...can be rewritten as:

```
select City.Name, Country.Continent
from City
inner join Country
    on City.CountryCode = Country.Code;
```

Instead of using commas to separate our table names, we are explicitly naming the type of Join to be used with our other table(s). (Other types of Joins will be covered later.) The ON clause is used here to show which keys you are joining. Although you can use ON as a replacement for the `where` clause:

```
select City.Name, Country.Continent
from City
inner join Country
    on City.CountryCode = Country.Code
    and Country.Code = 'AFG';
```

...it is considered good form to only use ON for matching keys specified in join clauses.

Obviously, Joins can consist of as many tables as you would like. Our three-table join example from earlier:

```
select City.Name, Continent, Language
from City, Country, CountryLanguage
where City.CountryCode = Country.Code
and CountryLanguage.CountryCode = Country.Code;
```

...can be re-written as:

```
select City.Name, Continent, Language
from City
inner join Country
    on City.CountryCode = Country.Code
inner join CountryLanguage
    on CountryLanguage.CountryCode = Country.Code;
```

Speed tips: Make sure your keys are indexed. Comma Radiation is what turned Bruce Banner into the Hulk, and created the Fantastic Four.

## Left/Right Joins

*“Think left and think right and think low and think high.  
Oh, the things you can think up if only you try!”  
--Dr. Theodore “Seuss” Geisel*

Comma Notation is generally suitable for Full Joins and Inner Joins, but it cannot be used in many databases for Left Joins\*. A Left Join is a kind of join that looks at the first table named in a join, and includes *all* records within that table, and any Foreign Keys that happens to match causes corresponding records to display. For example:

```
select City.Name, Country.Continent
from Country
inner join City
    on Country.Code = City.CountryCode;
```

...normally returns 4079 records. If, however, we made it a Left Join:

```
select City.Name, Country.Continent
from Country
left join City
    on Country.Code = City.CountryCode;
```

...we get 4086 records. Where did the extra 7 records come from? In the first example, our query ignored seven countries because there were no matches in the `Code` field in the `Country` table against the `CountryCode` field in the `City` table, so the query engine naturally ignored these records. Our second example, the Left Join, says “Show me all records from `Country`, regardless of whether there are corresponding records in `City` or not, and all records with matching keys from the `City` table. Records in `City` that have no matching keys in `Country` will be omitted entirely.”

Just because you get to see all records from the left side of the Join statement, does not mean that they will be unique, just as we saw in the Inner Join example output with Kabul and Qandahar: multiple cities with the same `CountryCode` will cause information from the `Country` table to repeat.

Right Joins are almost exactly the same, except instead of including all records from the fields declared on the left side of the join statement, you're including all records from the right side of the join statement.

---

\* All you Certified Sybase ASE nitpickers out there are going to tell me that Comma Notation is indeed usable within Sybase/MS TransactSQL for Left and Right Joins, through “`:=`” and “`=:`” respectively. Well, this is a class about MySQL SQL and not T-SQL, PL/SQL, nor Beat/A-Dead\_HorSQL; and frankly the SQL in MySQL doesn't let you use “`:=`” for Comma Notation Left Joins. So there. Phhhbbbt!

## Outer Joins

*“There is nothing wrong with your television set. Do not attempt to adjust the picture. We are controlling the transmission...”*

*--Introduction to the Outer Joins Limits*

One of the most common uses of Left and Right Joins is the creation of Outer Joins. Whereas Inner Joins represented a coupling of two or more tables based on matching key values, Outer Joins explicitly display records with missing key values in the table you joined against. For another pointless example, suppose you would like to look for all countries that have no cities. You recall from an earlier example that the following Left Join statement:

```
select City.Name, Country.Continent
from Country
left join City on Country.Code = City.CountryCode;
```

...will display all records from the Country, plus any matching records from the City table, if any. However, if we were to specify only those Countries that have no matching CountryCode in the City table:

```
select Country.Name, Country.Continent
from Country
left join City on Country.Code = City.CountryCode
where City.CountryCode is null;
```

...We get filtered down to a mere 7 countries:

```
+-----+-----+
| Name                                     | Continent |
+-----+-----+
| Antarctica                               | Antarctica |
| Bouvet Island                           | Antarctica |
| British Indian Ocean Territory          | Africa     |
| South Georgia and the South Sandwich Islands | Antarctica |
| Heard Island and McDonald Islands       | Antarctica |
| French Southern territories              | Antarctica |
| United States Minor Outlying Islands    | Oceania    |
+-----+-----+
```

A useful application of Outer Joins is quickly finding orphaned records.

At my work, I often use Outer Joins to track new customers from QuickBooks: by comparing the customer tables in QuickBooks to my own customer tables<sup>±</sup>, I can quickly find out which Customers were added recently. Then I synchronize QuickBooks' records with my MySQL database, and any new customers added henceforth from our Finance/Collections department will appear in a new Outer Join Query.

---

<sup>±</sup> Intuit creates an SDK for the QuickBooks API. Many third-party applications exist which can provide seamless integration between QuickBooks and MySQL, treating QuickBooks as a ODBC/JDBC database.

## Updates/Deletes

It is also possible to update records in join statements. If we were to suddenly receive a visit from Demonic Netherworld Invaders who demanded that had all countries without any cities must immediately append “Nether” to their names. After being conquered in High Resolution 24-bit Color Dolby Surround humiliation, we could comply with the following query:

```
update Country
left join City
  on Country.Code = City.CountryCode
set Country.Name = concat("Nether",Country.Name)
where City.CountryCode is null;
```

Note that this resembles our last Outer Join query, with a couple of exceptions. Gone is the `Select` clause, only to be replaced with the `Update...Set` clause. Our Countries now look like the following:

```
select Country.Name from Country
left join City on Country.Code = City.CountryCode
where City.CountryCode is null;
```

```
+-----+
| Name          |
+-----+
| NetherAntarctica          |
| NetherBouvet Island      |
| NetherBritish Indian Ocean Territory |
| NetherSouth Georgia and the South Sandwich Islands |
| NetherHeard Island and McDonald Islands |
| NetherFrench Southern territories |
| NetherUnited States Minor Outlying Islands |
+-----+
```

You can even change fields from multiple tables in your joins<sup>¥</sup>. Deletion in multi-table Join statements is just as easy. Assuming that we launch a counter offensive with a crapload of rocket-launcher-laden marines<sup>¶</sup>, and actually wiped out all seven offending nonpopulated countries, we could try this with:

```
delete from Country
left join City
  on Country.Code = City.CountryCode
where City.CountryCode is null
```

---

<sup>¥</sup> ...Although you should be careful about changing fields in tables whose primary keys you look up. These tables have values that may repeat in your multi-table Join, and if you try to change one of these lookup values without affecting the others, MySQL will complain if it is forced to isolate those same lookup values elsewhere.

<sup>¶</sup> Everything I know I learned from **Doom 3**, including the factoid that a marine can carry a 35 lb. Rocket launcher and still have enough room in his backpack to hold fifty more 10 lb. rockets, while slinging a double-barrelled shotgun stuffed with 250 twelve-gauge shells, for a total carrying capacity somewhere between a pickup truck and a construction crane. I also know that in any theater of engagement, ammunition is liberally sprinkled all over the ground just waiting to be picked up by the next opportunistic combatant, and food that you eat off the floor is equivalent to any form of medical care.

Whoops, it appear that MySQL doesn't like that. While this is legal in many a traditional RDBMS (specifically Access and SQL Server), MySQL wants you to be explicit about which table you're planning to delete records from. For MySQL, our Delete statement requires a ~~BFG-9000~~ USING clause:

```
delete from Country
using Country
left join City
  on Country.Code = City.CountryCode
where City.CountryCode is null;
Query OK, 7 Unpopulated Countries Nuked to Kingdom Come (1.14 sec.)
```

Don't you just love Bush Administration Foreign Policy?

It may also have been simpler to type:

```
delete from Country where left(Name, 6) = 'Nether';
Query OK, 7 Unpopulated Countries and one hapless country caught in the
crossfire Nuked to Kingdom Come (1.28 sec.)
```

...if you want to be absolutely sure, and you think people who wear wooden clogs and make delicious meatballs are evil anyway. In the former example, however, we are specifically, targetting countries that have no cities.

Just as with multi-table Updates, you can delete from multiple tables:

```
DELETE FROM t1, t2 USING t1, t2, t3 WHERE t1.id=t2.id AND t2.id=t3.id;
```

In this case, only matching records in t1 and t2 will be deleted. Matching records in t3 will be left alone.