

# Intro to MySQL

## 04: Joins

UReddit Class 292

February 7, 2012

### Brief Review: Indexes

In the previous document, we briefly spoke of using indexes to establish relationships between separate MySQL tables, again using the example of storing student class schedules. The example considered a “students” table that contained one row per student and was indexed by a unique integer ID, a “classes” table that contained one row per class and was indexed by a unique integer ID, and an “enrollment” table that associated students to classes by entering a row containing a student’s ID and a class ID in order to record that a particular student has enrolled in a particular class.

This is purely an organizational perspective, however. Given a student and his ID (say his ID is 5), how would one extract that student’s schedule from this database? Based on the little we’ve seen so far regarding SQL queries, one might:

1. `SELECT 'class_id' FROM 'enrollment' WHERE 'student_id' = '5';` which might return class IDs 3,7, and 10; and
2. `SELECT 'class_name' FROM 'classes' WHERE 'class_id' = '3';`
3. `SELECT 'class_name' FROM 'classes' WHERE 'class_id' = '7';`
4. `SELECT 'class_name' FROM 'classes' WHERE 'class_id' = '10';`

This is a total of four queries. More generally, to get the names of the  $n$  classes in which a student is enrolled,  $n + 1$  queries will be necessary using this approach. How can this be made faster?

### Joins

```
SELECT 'classes'.'class_subject', 'classes'.'class_number'  
FROM 'classes'  
INNER JOIN 'enrollment'  
ON 'enrollment'.'class_id' = 'classes'.'class_id'  
WHERE 'enrollment'.'student_id' = '5';
```

This query will return a list of the names of the classes in which the student with ID number 5 is enrolled using a single query by employing what is called a “join” of tables..

SQL joins allow you to succinctly and effectively extract relational data. In this case, we know that the “classes” ID field and the “enrollment” class\_id field encode the relationship we are after, so we use SQL’s INNER JOIN to extract the data we want, relative to this relationship. INNER JOIN will create a temporary table that is an amalgamation of the two tables that are being joined using and indexed by the relationship given in the ON clause. This temporary table contains only data that has an entry in both

tables (for example, it won't contain an entry for a class which has no students in it, nor an entry for a student that has enrolled in zero classes). Finally, this query instructs MySQL to apply the WHERE clause in order to filter for the particular data we want (in this case, student 5's classes).

There are other types of joins, such as cross join (rarely used), left outer join, right outer join. Furthermore, joins can be nested (though it is easy to write a slow query when nesting joins). Now that you have the basic idea of how a join works and where it is useful, I suggest reading up on relevant MySQL documentation and experimenting on your own. The sandbox database is at your disposal; I have given the sandbox user account permission to edit the data in the "students", "enrollment", and "classes" tables so that you may play with them as you like.

## Indexes, Joins, and Speed

We gave the "classes" and "students" tables primary keys because we were going to be looking up data based on ID. Each row in a table with a primary key field must have a unique value for that field; we cannot have two students with the same ID. However, the "enrollment" table is going to have duplicates. If we do not give it an index, every search through the "enrollment" table will be done sequentially through the entire table. As the table grows, this sort of search becomes very slow (its runtime is linear with respect to size).

In order to combat this problem, MySQL allows you to define a field as a regular index, and we should designate both the class\_id and student\_id fields of the "enrollment" table as indexes. Each index will create a B-tree using the values in its field in order to speed up search to be logarithmic with respect to size instead of linear, a drastic speedup. To give you a sense of how drastic this difference can be, given a table with 10,000 rows, a linear search for the last element in the table will take about 750 times longer than a logarithmic search (rough estimate). For more information on how to talk about runtimes with respect to data size, the Wikipedia article on computational complexity is a good start.