By The Disenchanted Developer

Table of Contents

Parlez−Vous SQL?

A few years ago, I decided to visit Paris for a short two−week holiday. I didn't know French, had never visited the country before, and only knew that it was famous for its wines and cheeses. However, I naively assumed that I'd have no trouble getting around − after all, I reasoned, Paris was a cosmopolitan city, and English was bound to be one of the more common languages there.

As you might imagine, my naïve optimism came crashing down to earth the moment I got off the plane. All the signs were in French, everyone seemed to be gesticulating frantically, and I couldn't understand a single word of what the people around me were saying. Sign language got me a cab, and a hotel booking slip got me to a bed − but I quickly realized that if I was going to survive in the city, I'd need to learn a little French...and fast!

With a little help from some friendly students, I quickly learnt the basics − "bon jour" when you meet someone, "au revoir" when you part, "merci" when you want to thank someone and − if all else fails − "parlez−vouz anglais?". By the end of my holiday, I understood a great deal more of what was happening around me, and was even able to conduct rudimentary conversations with the locals.

What does this have to do with anything? Quite a lot.

You see, learning a programming language is a lot like learning a spoken language. You start off knowing almost nothing and slowly work your way into a situation where you're comfortable with using it to accomplish basic tasks. You then build on this basic knowledge in order to acquire greater proficiency with the language, until your versatility and complete comfort with the language's most arcane nuances impress even your closest friends.

If you've been following this column over the past few months, you've probably picked up the basics of SQL, the standard language used to communicate with databases; I've used it frequently when discussing database−driven, dynamic Web applications. Even if you started out as a novice, you're probably now fairly comfortable using SQL to retrieve data from a database or insert new records into it. And so, it's time to begin the second phase of your education − learning some of the more advanced things you can do with SQL.

That's where this article comes in. Over the next few pages, I will be introducing you to joins, which have a reputation for being complex, difficult to understand and frightening. Most of this is just nasty rumour − they're actually simple, extremely friendly and nowhere near as frightening as some of the bills I got in Paris. Let me show you.

Meeting The Family

Before we begin, I'd like to introduce you to the three tables I'll be using in this tutorial. Say hello to table "a".

```
+----+------+
| a1 | a2 |
+----+------+
| 10 | u |
| 20 | v |
| 30 | w |
| 40 | x |
| 50 | y |
| 60 | z |
+----+------+
6 rows in set (0.05 sec)
```

Next up, drop by table "b",

```
+----+------+
| b1 | b2 |
+----+------+
| 10 | p |
| 20 | q |
+----+------+
2 rows in set (0.06 sec)
```

Finally, last but not least, table "c".

```
+-----+------+
| c1 | c2 |
+-----+------+
| 90 | m |
| 100 | n |
| 110 | o |
+-----+------+
3 rows in set (0.05 sec)
```

As you can see, I spent a lot of time naming them, so I expect you to be properly appreciative of my efforts.

These tables have been constructed in MySQL 4.x, a free (and very powerful) open−source RDBMS. Due to differences in capabilities, the techniques outlined in this article may not work on other RDBMS; you should refer to the documentation that comes with each system for accurate syntax.

I've deliberately kept these tables simple, so that you have as little difficulty as possible understanding the examples. At a later stage, once the basic joins are clear to you, I'll replace these simple tables with more complex, real−world representations, so that you understand some of the more arcane applications of joins.

Keeping It Simple

We'll start with something simple. The general format for a SELECT statement is:

```
SELECT <column list> FROM <table list> WHERE <condition list>;
```

Here's an example of how it can be used − the following SQL query retrieves all the records in table "a".

```
SELECT * FROM a;
```

Here's the output:

```
+----+------+
| a1 | a2 |
+----+------+
| 10 | u |
| 20 | v |
| 30 | w |
| 40 | x |
| 50 | y |
| 60 | z |
+----+------+
```

I can filter out some of these records by adding a WHERE clause:

```
SELECT * FROM a WHERE a1 > 20;
```

This returns

```
+----+------+
| a1 | a2 |
+----+------+
| 30 | w |
| 40 | x |
| 50 | y |
| 60 | z |
+----+------+
```

And I can even restrict the number of columns shown, by specifying a list of column names instead of the all−purpose asterisk:

```
SELECT a2 FROM a WHERE a1 > 20;
```

Which gives me:

```
+------+
| a2 |
+------+
| w |
| x |
| y |
| z |
+------+
```

Crossed Wires

The examples on the previous page dealt with a single table, so the question of a join never arose. But often, using a single table returns an incomplete picture; you need to combine data from two or more tables in order to see a more accurate result.

That's where joins come in − they allow you to combine two or more tables, and to massage the data within those tables, in a variety of different ways. For example,

```
SELECT * FROM a,b;
```

Here's the output:

```
+----+------+----+------+
| a1 | a2 | b1 | b2 |
+----+------+----+------+
| 10 | u | 10 | p |
| 20 | v | 10 | p |
| 30 | w | 10 | p |
| 40 | x | 10 | p |
| 50 | y | 10 | p |
| 60 | z | 10 | p |
| 10 | u | 20 | q |
| 20 | v | 20 | q |
| 30 | w | 20 | q |
| 40 | x | 20 | q |
| 50 | y | 20 | q |
| 60 | z | 20 | q |
+----+------+----+------+
12 rows in set (0.00 sec)
```

And that's your very first join!

In this case, columns from both tables are combined to produce a resultset containing all possible combinations. This kind of join is referred to as a "cross join", and the number of rows in the joined table will be equal to the product of the number of rows in each of the tables used in the join. You can see this from the example above − table "a" has 6 rows, table "b" has 2 rows, and so the joined table has 6x2 = 12 rows.

There are two basic types of SQL joins − the "inner join" and the "outer join". Inner joins are the most common − the one you just saw was an example of an inner join − and also the most symmetrical, since they require a match in each table which forms a part of the join. Rows which do not match are excluded from the final resultset.

As you might imagine, a cross join like the one above can have huge implications for the performance of your

database server. In order to illustrate, look what happens when I add a third table to the join above:

```
SELECT * FROM a,b,c;
```

Here's the output:

```
+----+------+----+------+-----+------+
| a1 | a2 | b1 | b2 | c1 | c2 |
+----+------+----+------+-----+------+
| 10 | u | 10 | p | 90 | m |
| 20 | v | 10 | p | 90 | m |
| 30 | w | 10 | p | 90 | m |
| 40 | x | 10 | p | 90 | m |
| 50 | y | 10 | p | 90 | m |
| 60 | z | 10 | p | 90 | m |
| 10 | u | 20 | q | 90 | m |
| 20 | v | 20 | q | 90 | m |
| 30 | w | 20 | q | 90 | m |
| 40 | x | 20 | q | 90 | m |
| 50 | y | 20 | q | 90 | m |
| 60 | z | 20 | q | 90 | m |
| 10 | u | 10 | p | 100 | n |
| 20 | v | 10 | p | 100 | n |
| 30 | w | 10 | p | 100 | n |
| 40 | x | 10 | p | 100 | n |
| 50 | y | 10 | p | 100 | n |
| 60 | z | 10 | p | 100 | n |
| 10 | u | 20 | q | 100 | n |
| 20 | v | 20 | q | 100 | n |
| 30 | w | 20 | q | 100 | n |
| 40 | x | 20 | q | 100 | n |
| 50 | y | 20 | q | 100 | n |
| 60 | z | 20 | q | 100 | n |
| 10 | u | 10 | p | 110 | o |
| 20 | v | 10 | p | 110 | o |
| 30 | w | 10 | p | 110 | o |
| 40 | x | 10 | p | 110 | o |
| 50 | y | 10 | p | 110 | o |
| 60 | z | 10 | p | 110 | o |
| 10 | u | 20 | q | 110 | o |
| 20 | v | 20 | q | 110 | o |
| 30 | w | 20 | q | 110 | o |
| 40 | x | 20 | q | 110 | o |
| 50 | y | 20 | q | 110 | o |
| 60 | z | 20 | q | 110 | o |
+----+------+----+------+-----+------+
```

```
36 rows in set (0.05 sec)
```

Though each of the tables used in the join contains less than ten records each, the final joined result contains 6x2x3=36 records. This might not seem like a big deal when all you're dealing with are three tables containing a total of 11 records, but imagine what would happen if you had three tables, each containing 100 records, and you decided to cross join them...

Finding Common Ground

Since a cross join produces a huge resultset, it is considered a good idea to attach a WHERE clause to the join to filter out some of the records. Here's an example:

```
SELECT * FROM a,b WHERE a1 > 30;
```

This returns

```
+----+------+----+------+
| a1 | a2 | b1 | b2 |
+----+------+----+------+
| 40 | x | 10 | p |
| 50 | y | 10 | p |
| 60 | z | 10 | p |
| 40 | x | 20 | q |
| 50 | y | 20 | q |
| 60 | z | 20 | q |
+----+------+----+------+
6 rows in set (0.06 sec)
```

A variant of the cross join is the "equi−join", where certain fields in the joined tables are equated to each other. In this case, the final resultset will only include those rows from the joined tables which have matches in the specified fields.

```
SELECT * FROM a,b WHERE a1 = b1;
```

This returns

```
+----+------+----+------+
| a1 | a2 | b1 | b2 |
+----+------+----+------+
| 10 | u | 10 | p |
| 20 | v | 20 | q |
+----+------+----+------+
2 rows in set (0.00 sec)
```

Here, too, you can use a WHERE clause to further narrow the resultset:

```
SELECT * FROM a,b WHERE a1 = b1 AND a1 = 20;
```

This returns

```
+----+------+----+------+
| a1 | a2 | b1 | b2 |
+----+------+----+------+
| 20 | v | 20 | q |
+----+------+----+------+
1 row in set (0.00 sec)
```

One Step Left...

Let's move on to outer joins. You'll remember, from a few pages back, that inner joins are symmetrical − in order to be included in the final resultset, rows must match in all joined tables. Outer joins, on the other hand, are asymmetrical − all rows from one side of the join are included in the final resultset, regardless of whether or not they match rows on the other side of the join.

This might seem a little complicated, but an example should soon clear that up. Consider the following SQL query:

```
SELECT * FROM a LEFT JOIN b ON a1 = b1;
```

In English, this translates to "select all the rows from the left side of the join (table a) and, for each row selected, either display the matching value from the right side (table b) or display an empty row". This kind of join is known as a "left join" or, sometimes, a "left outer join".

Here's the result:

```
+----+------+------+------+
| a1 | a2 | b1 | b2 |
+----+------+------+------+
| 10 | u | 10 | p |
| 20 | v | 20 | q |
| 30 | w | NULL | NULL |
| 40 | x | NULL | NULL |
| 50 | y | NULL | NULL |
| 60 | z | NULL | NULL |
+----+------+------+------+
6 rows in set (0.06 sec)
```

As you can see, all the rows from the left side of the join − table "a" − appear in the final resultset. Those which have a corresponding value on the right side − table "b" − as per the match criteria a1 = b1 have that value displayed; the rest have a null row displayed.

This kind of join comes in very handy when you need to see which values from one table are missing in another table − all you need to do is look for the null rows. Just from a quick glance at the example above, it's fairly easy to see that rows 30−60 are present in table "a", but absent in table "b". This technique also comes in handy when you're looking for corrupted, or "dirty", data.

In fact, you can even save your eyeballs the trouble of scanning the output − just let SQL do it for you, with an additional WHERE clause:

```
SELECT a1 FROM a LEFT JOIN b ON a1 = b1 WHERE b1 IS NULL;
```

Here's the output:

```
+----+
| a1 |
+----+
| 30 |
| 40 |
| 50 |
| 60 |
+----+
4 rows in set (0.05 sec)
```

...Two Steps Right

Just as there's a left join, there's also a "right join", which does exactly what a left join does, but in reverse. Consider the following query,

```
SELECT * FROM a RIGHT JOIN c ON a1=c1;
```

which translates to "select all the rows from the right side of the join (table c) and, for each row selected, either display the matching value from the left side (table a) or display an empty row" and displays

```
+------+------+-----+------+
| a1 | a2 | c1 | c2 |
+------+------+-----+------+
| NULL | NULL | 90 | m |
| NULL | NULL | 100 | n |
| NULL | NULL | 110 | o |
+------+------+-----+------+
3 rows in set (0.06 sec)
```

All the rows from the right side of the join − table "c" − appear in the final resultset. Those which have a corresponding value on the left side − table "a" − as per the match criteria a1 = c1 have that value displayed; the rest have a null row displayed.

The Bookworm Turns

So that's the theory − now let's see how it plays out in real life. Consider the following tables, which contain data for a small lending library:

```
+----+-------------------------------------+
| id | title |
+----+-------------------------------------+
| 1 | Mystic River |
| 2 | Catch-22 |
| 3 | Harry Potter and the Goblet of Fire |
| 4 | The Last Detective |
| 5 | Murder on the Orient Express |
+----+-------------------------------------+
```

This table, named "books", contains a complete list of books available for lending. Each book has a unique ID.

```
+-----+-------+
| id | name |
+-----+-------+
| 100 | Joe |
| 101 | John |
| 103 | Dave |
| 109 | Harry |
| 110 | Mark |
| 113 | Jack |
+-----+-------+
```

This table, named "members", contains a list of all the members of the library. Like the books, each member also has a unique ID.

```
+-----------+---------+
| member_id | book_id |
+-----------+---------+
| 101 | 1 |
| 103 | 3 |
| 109 | 4 |
+-----------+---------+
```

As members borrow books from the library, the "data" table, above, is updated with information on which book has been lent to which member.

Using these tables and whatever you've just learned about joins, it's possible to write queries that answer the most common questions asked of the system by its operators. For example, let's suppose I wanted a list of all the current members:

```
SELECT * FROM members;

+-----+-------+
| id | name |
+-----+-------+
| 100 | Joe |
| 101 | John |
| 103 | Dave |
| 109 | Harry |
| 110 | Mark |
| 113 | Jack |
+-----+-------+
```

How about a list of all the books currently in the library catalog?

```
SELECT * FROM books;

+----+-------------------------------------+
| id | title |
+----+-------------------------------------+
| 1 | Mystic River |
| 2 | Catch-22 |
| 3 | Harry Potter and the Goblet of Fire |
| 4 | The Last Detective |
| 5 | Murder on the Orient Express |
+----+-------------------------------------+
```

How about an enquiry on Dave's account, to see which books he's currently checked out?

```
SELECT books.title FROM data,books WHERE data.book_id =
books.id AND
data.member_id = 103;

+-------------------------------------+
| title |
+-------------------------------------+
| Harry Potter and the Goblet of Fire |
+-------------------------------------+
```

Can we find out which members *don't* have any books checked out to them (maybe these are inactive accounts)?

```
SELECT members.id, members.name FROM members LEFT JOIN data ON
data.member_id = members.id WHERE data.book_id IS NULL;

+-----+------+
| id | name |
+-----+------+
| 100 | Joe |
| 110 | Mark |
| 113 | Jack |
+-----+------+
```

How about a list of all the books that are currently in stock (not checked out to anyone)?

```
SELECT books.title FROM books LEFT JOIN data ON data.book_id =
books.id
WHERE data.book_id IS NULL

+-----------------------------+
| title |
+-----------------------------+
| Catch-22 |
| Murder on the Orient Express |
+-----------------------------+
```

Now, how about a list of all the books currently checked out, together with the name of the member they've been checked out to?

```
SELECT members.name, books.title FROM data, members, books
WHERE
data.member_id = members.id AND data.book_id = books.id;

+-------+-----------------------------------+
| name | title |
+-------+-----------------------------------+
| John | Mystic River |
| Dave | Harry Potter and the Goblet of Fire |
| Harry | The Last Detective |
+-------+-----------------------------------+
```

You could also accomplish the above with the following query (an outer join instead of an inner join):

```
SELECT members.name, books.title FROM books,members LEFT JOIN
data ON
data.book_id = books.id WHERE data.member_id = members.id AND
data.book_id IS NOT NULL;
```

As you can see, joins allow you to extract all kinds of different information from a set of tables. Go ahead and try it for yourself − think of a question you would like to ask this system, and then write a query to answer it.

Up A Tree

In addition to inner and outer joins, SQL also allows a third type of join, known as a "self join". Typically, this type of join is used to extract data from a table whose records contain internal links to each other. Consider the following table, which illustrates what I mean:

```
+----+----------------------+--------+
| id | label | parent |
+----+----------------------+--------+
| 1 | Services | 0 |
| 2 | Company | 0 |
| 3 | Media Center | 0 |
| 4 | Your Account | 0 |
| 5 | Community | 0 |
| 6 | For Content Publishers | 1 |
| 7 | For Small Businesses | 1 |
| 8 | Background | 2 |
| 9 | Clients | 2 |
| 10 | Addresses | 2 |
| 11 | Jobs | 2 |
| 12 | News | 2 |
| 13 | Press Releases | 3 |
| 14 | Media Kit | 3 |
| 15 | Log In | 4 |
| 16 | Columns | 5 |
| 17 | Colophon | 16 |
| 18 | Cut | 16 |
| 19 | Boombox | 16 |
+----+----------------------+--------+
```

This is a simple menu structure, with each record identifying a unique node in the menu tree. Each record has a unique record ID, and also contains a parent ID − these two IDs are used to define the parent−child relationships between the branches of the menu tree. So, if I were to represent the data above hierarchically, it would look like this:

```
<root>
|−Services
|−For Content Publishers
|−For Small Businesses
|−Company
|−Background
|−Clients
|−Addresses
|−Jobs
|−News
```

```
|-Media Center
|-Press Releases
|-Media Kit
|-Your Account
|-Log In
|-Community
|-Columns
|-Colophon
|-Cut
|-Boombox
```

Now, let's suppose I need to display a list of all the nodes in the tree, together with the names of their parents. What I'm looking for is a resultset resembling this:

```
+--------------+-----------------------+
| parent_label | child_label |
+--------------+-----------------------+
| Services | For Content Publishers |
| Company | Background |
| Your Account | Log In |
+--------------+-----------------------+
```

If you think about it, you'll see that there's no easy way to obtain this resultset − since all the data is in a single table, a simple SELECT won't work, and neither will one of those complicated outer joins. What I really need here is something called a self join, which allows me to create a second, virtual copy of the first table, and then use a regular inner join to map the two together and get the output I need.

Here's the query I would use:

```
SELECT a.label AS parent_label, b.label AS child_label FROM
menu AS a,
menu AS b WHERE a.id = b.parent;
```

Here's the output:

```
+--------------+-----------------------+
| parent_label | child_label |
+--------------+-----------------------+
| Services | For Content Publishers |
| Services | For Small Businesses |
| Company | Background |
| Company | Clients |
| Company | Addresses |
```

```
| Company | Jobs |
| Company | News |
| Media Center | Press Releases |
| Media Center | Media Kit |
| Your Account | Log In |
| Community | Columns |
| Columns | Colophon |
| Columns | Cut |
| Columns | Boombox |
+--------------+----------------------+
```

Exactly what I need!

Most of the magic here lies in the table aliasing − I've created two copies of the "menu" table, and aliased them as "a" and "b" respectively. This will result in the following two "virtual" tables.

```
+----+----------------------+--------+
| TABLE a |
+----+----------------------+--------+
| id | label | parent |
+----+----------------------+--------+
| 1 | Services | 0 |
| 2 | Company | 0 |
| 3 | Media Center | 0 |
| 4 | Your Account | 0 |
| 5 | Community | 0 |
| 6 | For Content Publishers | 1 |
| 7 | For Small Businesses | 1 |
| 8 | Background | 2 |
| 9 | Clients | 2 |
| 10 | Addresses | 2 |
| 11 | Jobs | 2 |
| 12 | News | 2 |
| 13 | Press Releases | 3 |
| 14 | Media Kit | 3 |
| 15 | Log In | 4 |
| 16 | Columns | 5 |
| 17 | Colophon | 16 |
| 18 | Cut | 16 |
| 19 | Boombox | 16 |
+----+----------------------+--------+

+----+----------------------+--------+
| TABLE b |
+----+----------------------+--------+
| id | label | parent |
+----+----------------------+--------+
```

```
| 1 | Services | 0 |
| 2 | Company | 0 |
| 3 | Media Center | 0 |
| 4 | Your Account | 0 |
| 5 | Community | 0 |
| 6 | For Content Publishers | 1 |
| 7 | For Small Businesses | 1 |
| 8 | Background | 2 |
| 9 | Clients | 2 |
| 10 | Addresses | 2 |
| 11 | Jobs | 2 |
| 12 | News | 2 |
| 13 | Press Releases | 3 |
| 14 | Media Kit | 3 |
| 15 | Log In | 4 |
| 16 | Columns | 5 |
| 17 | Colophon | 16 |
| 18 | Cut | 16 |
| 19 | Boombox | 16 |
+----+------------------------+--------+
```

Once these two tables have been created, it's a simple matter to join them together, using the node IDs as the common column, and to obtain a list of child and parent labels in the desired format.

A Long Goodbye

And that's about it for the moment. In this article, I helped you dip your toes into the deeper waters of SQL, showing you how you could use SQL to massage your resultsets so that you only see the data you want to see. I showed you a basic inner join, the cross join, which will have your database crying for Mommy if you use it too often, and the equi−join, which you can use to filter our unwanted results from your inner join.

Next, I moved on to outer joins, demonstrating how the left and right outer joins can be used to compare tables and isolate the missing or common elements. Finally, I wrapped things up with the self join, a very neat little concept that is sure to come in useful when you're dealing with a single table containing linked records.

In order to avoid having the theory overwhelm you, I also showed you how the various types of joins may be used in real−world situations, to answer basic questions that arise when dealing with linked tables in an RDBMS. Hopefully, the two real−world examples in this article fully demonstrated the value of SQL joins, and also destroyed some of the myths associated with their ease of use (or lack thereof).

This isn't all, though − SQL is very popular on the Web, and there are reams of information out there which will teach you how to manipulate inner and outer joins to do ever more complicated acrobatics. Here are a few good links to get you started:

Speaking SQL, at http://www.devshed.com/Server_Side/MySQL/Speak/

The MySQL manual, at http://www.mysql.com/doc

The W3School's tutorial on joins, at http://www.w3schools.com/sql/sql_join.asp

CNET's tutorial on joins, at http://asia.cnet.com/itmanager/netadmin/0,39006400,39042301,00.htm

Until next time...au revoir!

Note: All examples in this article have been tested on Linux/i586 with MySQL 4. Examples are illustrative only, and are not meant for a production environment. Melonfire provides no warranties or support for the source code described in this article. YMMV!