Using Subqueries In MySQL (part 2)

By RK Harigopal

2003−07−31

Printed from DevShed.com
URL: http://www.devshed.com/Server_Side/MySQL/MySQL_Subqueries2/

Doing More

In the first part of this article, I introduced you to subqueries, one of the most powerful and useful new features in MySQL 4.1, and showed you how you can use them to filter SQL result sets. I demonstrated how subqueries can be used in both the WHERE and HAVING clause of outer SELECT queries, in combination with MySQL's other comparison and logical operators, with MySQL's aggregate functions and with the GROUP BY clause.

However, that's just the tip of the iceberg. In the concluding segment of this tutorial, I'm going to delve into the more advanced aspects of subqueries, showing you how to use them within a SELECT's query's FROM clause, with the IN operator and the EXISTS test, and in UPDATE and DELETE queries. Let's get going!

Total Recall

Before we begin, a quick recap and re−introduction to the tables I'll be using throughout this tutorial seems to be in order. In order to explain subqueries, I'll be using a fictitious company's accounting database, which consists of the following four tables:

1. The "services" table: The fictitious system under discussion consists of a service company's accounting database. This company offers customers a number of outsourced services, each of which is associated with a fee and has a unique service ID. This information is stored in a "services" table, which looks like this:

```
mysql> SELECT * FROM services;

+-----+------------------+---------+

| sid | sname | sfee |

+-----+------------------+---------+

| 1 | Accounting | 1500.00 |

| 2 | Recruitment | 500.00 |

| 3 | Data Management | 300.00 |

| 4 | Administration | 500.00 |

| 5 | Customer Support | 2500.00 |

| 6 | Security | 600.00 |

+-----+------------------+---------+

6 rows in set (0.16 sec)
```

2. The "clients" table: The company also has a list of its current clients stored in a separate "clients" table. Each client is identified with a unique customer ID.

```
mysql> SELECT * FROM clients;

+-----+-----------------------------+

| cid | cname |

+-----+-----------------------------+

| 101 | JV Real Estate |

| 102 | ABC Talent Agency |

| 103 | DMW Trading |

| 104 | Rabbit Foods Inc |

| 110 | Sharp Eyes Detective Agency |

+-----+-----------------------------+

5 rows in set (0.00 sec)
```

3. The "branches" table: Each customer may have one or more branch offices. The "branches" table lists the branch offices per customer, together with each branch's location. Each branch has a description, a unique branch ID, and a foreign key reference to the customer ID.

```
mysql> SELECT * FROM branches;

+------+-----+--------------------------------+------+

| bid | cid | bdesc | bloc |

+------+-----+--------------------------------+------+

| 1011 | 101 | Corporate HQ | CA |

| 1012 | 101 | Accounting Department | NY |

| 1013 | 101 | Customer Grievances Department | KA |

| 1041 | 104 | Branch Office (East) | MA |

| 1042 | 104 | Branch Office (West) | CA |

| 1101 | 110 | Head Office | CA |

| 1031 | 103 | N Region HO | ME |

| 1032 | 103 | NE Region HO | CT |

| 1033 | 103 | NW Region HO | NY |

+------+-----+--------------------------------+------+

9 rows in set (0.01 sec)
```

4. The "branches_services" table: Services supplied to each branch office are listed in this table, which contains pairs of branch IDs and service IDs (foreign keys into the "branches" and "services" table respectively).

```
mysql> SELECT * FROM branches_services;

+------+-----+

| bid | sid |

+------+-----+

| 1011 | 1 |

| 1011 | 2 |

| 1011 | 3 |

| 1011 | 4 |

| 1012 | 1 |

| 1013 | 5 |

| 1041 | 1 |

| 1041 | 4 |

| 1042 | 1 |

| 1042 | 4 |

| 1101 | 1 |

| 1031 | 2 |

| 1031 | 3 |

| 1031 | 4 |

| 1032 | 3 |

| 1033 | 4 |

+------+-----+

16 rows in set (0.11 sec)
```

In order to use the examples in this tutorial, you'll need to recreate these tables in your MySQL 4.1 database server. Instructions and SQL code for doing so are available in the first part of this article. Once you're done reading that, come back and let's continue.

In And Out

As you saw in the previous segment of this article, MySQL permits you to use its numerous comparison operators to test the WHERE or HAVING clause of an outer query against the result set generated by an inner query. Now, comparison operators work great only so long as the subquery returns a result column consisting of a single value. However, if you have a good memory, you'll remember I said that a subquery can also return a single column of multiple values. How does MySQL handle this?

Very simply, with the IN operator. The IN operator makes it possible to test if a particular value exists in the result set, and perform the outer query if the test is successful. Let me show you how.

Let's suppose I need a list of all services being used by a particular branch office (say, branch ID 1031). Normally, I would need to get a list of all service IDs for this branch,

```
mysql> SELECT sid FROM branches_services WHERE bid = 1031;

+-----+

| sid |

+-----+

| 2 |

| 3 |

| 4 |

+-----+

3 rows in set (0.16 sec)
```

and then look up each service ID in the "services" table for the corresponding name.

```
mysql> SELECT sname FROM services WHERE sid = 2;

+-------------+

| sname |

+-------------+

| Recruitment |

+-------------+

1 row in set (0.28 sec)
```

```
mysql> SELECT sname FROM services WHERE sid = 3;
+------------------+
| sname |
+------------------+
| Data Management |
+------------------+
1 row in set (0.17 sec)

mysql> SELECT sname FROM services WHERE sid = 4;
+-----------------+
| sname |
+------------------+
| Administration |
+------------------+
1 row in set (0.11 sec)
```

With a subquery and the IN test, this becomes redundant.

```
mysql> SELECT sname FROM services WHERE sid IN (SELECT sid FROM

branches_services WHERE bid = 1031);

+------------------+

| sname |

+------------------+

| Recruitment |

| Data Management |

| Administration |

+------------------+

3 rows in set (0.27 sec)
```

In this case, MySQL will select only those records from the "services" table which match the service ID collection returned by the subquery.

A variant of this might be to obtain a list of all branches using the "Accounting" service (service ID 1).

```
mysql> SELECT bdesc FROM branches WHERE bid IN (SELECT bid FROM

branches_services WHERE sid = 1);

+------------------------+

| bdesc |

+------------------------+

| Corporate HQ |

| Accounting Department |

| Branch Office (East) |

| Branch Office (West) |

| Head Office |

+------------------------+

5 rows in set (0.17 sec)
```

Hmmm...not too useful. What might be nice here is the customer name for each branch as well − something easily accomplished by adding a quick join.

```
mysql> SELECT cname, bdesc FROM branches, clients WHERE branches.bid IN

(SELECT bid FROM branches_services WHERE sid = 1) AND clients.cid = branches.cid;

+-----------------------------+-----------------------+

| cname | bdesc |

+-----------------------------+-----------------------+

| JV Real Estate | Corporate HQ |

| JV Real Estate | Accounting Department |

| Rabbit Foods Inc | Branch Office (East) |

| Rabbit Foods Inc | Branch Office (West) |

| Sharp Eyes Detective Agency | Head Office |

+-----------------------------+-----------------------+

5 rows in set (0.16 sec)
```

Want just the customer list? Add the DISTINCT keyword,

```
mysql> SELECT DISTINCT cname FROM branches, clients WHERE branches.bid

mysql> IN

(SELECT bid FROM branches_services WHERE sid = 1) AND clients.cid = branches.cid;

+-----------------------------+

| cname |

+-----------------------------+

| JV Real Estate |

| Rabbit Foods Inc |

| Sharp Eyes Detective Agency |

+-----------------------------+

3 rows in set (0.17 sec)
```

or hey, just rewrite the query using the IN test again.

```
mysql> SELECT cname FROM clients WHERE cid IN (select cid from branches

where bid IN (SELECT bid FROM branches_services WHERE sid = 1));

+-----------------------------+
| cname |

+-----------------------------+

| JV Real Estate |

| Rabbit Foods Inc |

| Sharp Eyes Detective Agency |

+-----------------------------+

3 rows in set (0.22 sec)
```

Wanna mix things up a little? Let's say I want a list of high−value clients
− all those with individual branch offices having a monthly bill of 00 or more. I can get this information (among other ways) by using a subquery with a join, a GROUP BY clause and the IN operator...all at once, while simultaneously balancing a slice of pizza.

```
mysql> SELECT cname FROM clients WHERE cid IN (select cid from branches

where bid IN (SELECT bid FROM branches_services as bs, services as s where bs.sid
= s.sid group by bid having sum(sfee) >= 2000));

+-------------------+

| cname |

+-------------------+

| JV Real Estate |

| Rabbit Foods Inc |

+-------------------+

2 rows in set (1.32 sec)
```

You can use the NOT keyword to reverse the results of the IN operator − or, in other words, to return those records not matching the result collection generated by a subquery. If you had to rewrite the example above using the NOT operator, it would look something like this:

```
mysql> SELECT cname FROM clients WHERE cid IN (select cid from branches

where bid NOT IN (SELECT bid FROM branches_services as bs, services as s where
bs.sid = s.sid group by bid having sum(sfee) < 2000));

+-------------------+

| cname |

+-------------------+

| JV Real Estate |

| Rabbit Foods Inc |

+-------------------+
2 rows in set (1.54 sec)
```

A Solitary Existence

In addition to comparison operators and the IN membership test, MySQL also comes with a special EXISTS operator, designed specifically for use with subqueries. The EXISTS operator is used to test if a subquery generates any result rows, and executes the outer query only if it does.

Here's a simple example:

```
mysql> SELECT * FROM clients WHERE EXISTS (SELECT bid FROM

branches_services GROUP BY bid HAVING COUNT(sid) >= 5);

Empty set (0.17 sec)
```

In this case, since the subquery returns an empty result set − there are no branches using five or more services − the EXISTS test will return false and the outer query will not execute.

Look what happens when you doctor the subquery above to return something:

```
mysql> SELECT * FROM clients WHERE EXISTS (SELECT bid FROM

branches_services GROUP BY bid HAVING COUNT(sid) >= 4);

+-----+-----------------------------+

| cid | cname |

+-----+-----------------------------+

| 101 | JV Real Estate |

| 102 | ABC Talent Agency |

| 103 | DMW Trading |
| 104 | Rabbit Foods Inc |

| 110 | Sharp Eyes Detective Agency |

+-----+-----------------------------+

5 rows in set (0.27 sec)
```

In this case, since there are some branches using four or more services, the inner query will return a result set consisting of at least one row, the EXISTS test will return true, and the outer query will be executed.

It's important to note that the result set of the outer query above does
not list which customers have more than four services. Rather, it is simply a list of all customers, and it is returned only because the inner query generated a result set. In the example above, the result set generated by the inner query is itself immaterial; I could accomplish the same thing with the following query:

```
mysql> SELECT * FROM clients WHERE EXISTS (SELECT 1);

+-----+-----------------------------+

| cid | cname |

+-----+-----------------------------+

| 101 | JV Real Estate |

| 102 | ABC Talent Agency |

| 103 | DMW Trading |

| 104 | Rabbit Foods Inc |

| 110 | Sharp Eyes Detective Agency |

+-----+-----------------------------+

5 rows in set (0.16 sec)
```

With this in mind, you might be wondering what use the EXISTS operator really is. Good question.

You see, the EXISTS operator is most often used in the context of what SQL gurus like to call an "outer reference" − that is, a reference within the subquery to a field in one or more of the queries enclosing it. When such an outer reference is used, MySQL has to run the subquery once for every record generated by the outer query, and therefore test the subquery as many times as there are records in the outer query's result set, since the inner query contains a reference to a field (whose value is different for every record) in the outer query.

An example might make this clearer. Consider the following, which re−runs the previous example with an outer reference to make the result set more
useful:

```
mysql> SELECT bid, bdesc FROM branches WHERE EXISTS (SELECT bid FROM

branches_services WHERE branches.bid = branches_services.bid GROUP BY bid HAVING
COUNT(sid) >= 4);

+------+--------------+

| bid | bdesc |

+------+--------------+

| 1011 | Corporate HQ |

+------+--------------+

1 row in set (0.22 sec)
```

And if you look at the data, you'll see that there is, in fact, only one branch using four or more services (branch ID 1011).

In this case, since the inner query contains a reference to a field in the outer query, MySQL cannot run the inner query just once (as it usually does). Rather, it has to run it over and over, once for every row in the outer table, then substitute the value of the named field from that row in the subquery, and then decide whether or not to include that outer row in the final result set on the basis of whether or not the corresponding subquery returns a result set or not.

Needless to say, this is expensive in terms of performance, and outer references should hence be avoided unless absolutely necessary. Usually, you can accomplish the same thing faster and more optimally with a join, as
below:

```
mysql> SELECT branches.bid, branches.bdesc FROM branches_services,

mysql> branches

WHERE branches_services.bid = branches.bid GROUP BY bid HAVING COUNT(sid)

>= 4;

+------+--------------+

| bid | bdesc |

+------+--------------+

| 1011 | Corporate HQ |

+------+--------------+

1 row in set (0.28 sec)
```

Turning The Tables

You can also use the results generated by a subquery as a table in the FROM clause of an enclosing SELECT statement. In order to illustrate, let's say I want to know the average number of services each branch office has. Therefore, the first thing I need to do is group the branches together and count the total number of services each one has,

```
mysql> SELECT bid, COUNT(sid) AS stotal FROM branches_services GROUP BY
mysql> bid;
+------+--------+
| bid | stotal |
+------+--------+
| 1011 | 4 |
| 1012 | 1 |
| 1013 | 1 |
| 1031 | 3 |
| 1032 | 1 |
| 1033 | 1 |
| 1041 | 2 |
| 1042 | 2 |
| 1101 | 1 |
+------+--------+
9 rows in set (0.17 sec)
```

and then calculate the rounded−off average of the per−branch totals. Or, in a subquery,

```
mysql> SELECT AVG(z.stotal) FROM (SELECT bid, COUNT(sid) AS stotal FROM
branches_services GROUP BY bid) AS z;
+---------------+
| AVG(z.stotal) |
+---------------+
| 1.7778 |
+---------------+

1 row in set (0.21 sec)
```

Thus, the table of results generated by the inner query is used in the FROM clause of the outer query. You can convert the average number above into an integer with the CEILING() function if you like.

Note that when using subquery results in this manner, the result table generated by the inner query must be first aliased to a table name, or else MySQL will now know how to refer to columns within it. Look what happens when I re−run the query above without the table alias:

```
mysql> SELECT AVG(stotal) FROM (SELECT bid, COUNT(bid) AS stotal FROM

branches_services GROUP BY bid);

ERROR 1246: Every derived table must have it's own alias
```

Now, let's take it a step further. What if I need to list the branches where the number of services is above this average?

```
mysql> SELECT bid FROM branches_services GROUP BY bid HAVING COUNT(sid)

mysql> >

1.7778;

+------+

| bid |

+------+

| 1011 |

| 1031 |

| 1041 |

| 1042 |

+------+

4 rows in set (0.28 sec)
```

To make it truly dynamic, you should combine the two queries above into the following complex query:

```
mysql> SELECT bid FROM branches_services GROUP BY bid HAVING COUNT(sid)

mysql> >

(SELECT AVG(z.stotal) FROM (SELECT bid, COUNT(bid) AS stotal FROM branches_services
GROUP BY bid) AS z);

+------+
| bid |

+------+

| 1011 |

| 1031 |

| 1041 |

| 1042 |

+------+

4 rows in set (0.28 sec)
```

Wanna make it really complicated? Add another query (and a join) around it to get the customer names corresponding to those branches.

```
mysql> SELECT DISTINCT cname FROM clients, branches, (SELECT bid FROM

branches_services GROUP BY bid HAVING COUNT(sid) > (SELECT AVG(z.stotal)
FROM (SELECT bid, COUNT(bid) AS stotal FROM branches_services GROUP BY bid)
AS z)) AS x WHERE clients.cid = branches.cid AND branches.bid = x.bid;

+------------------+

| cname |

+------------------+

| JV Real Estate |

| DMW Trading |

| Rabbit Foods Inc |

+------------------+

3 rows in set (0.33 sec)
```

Whew!

Show Me The Money

Thus far, every single example you've seen in this tutorial has involved a subquery within an outer SELECT statement. However, subqueries can be used in other places too − most notably in UPDATE and DELETE statements, which can use the results of a subquery to constrain the records to which the UPDATE or DELETE is applied.

I'll explain this with a simple example. Let's suppose you want to delete all branches using the "Recruitment" service (service ID 2). Normally, you'd first look up the branches using that service in the "branches_services" table,

```
mysql> SELECT * FROM branches_services WHERE sid = 2;

+------+-----+

| bid | sid |

+------+-----+

| 1011 | 2 |

| 1031 | 2 |

+------+-----+

2 rows in set (0.16 sec)
```

and then delete the corresponding branch records from the "branches" table.

```
mysql> DELETE FROM branches WHERE bid = 1011;

Query OK, 1 row affected (0.00 sec)
```

mysql> DELETE FROM branches WHERE bid = 1031;
Query OK, 1 row affected (0.00 sec)

You can combine the two operations above into a single one with a subquery, as below:

```
mysql> DELETE FROM branches WHERE bid IN (SELECT bid FROM

mysql> branches_services

WHERE sid = 2);

Query OK, 2 rows affected (0.00 sec)
```

A check of the "branches" table confirms that the operation was successful.

```
mysql> SELECT * FROM branches;

+------+-----+--------------------------------+------+

| bid | cid | bdesc | bloc |
+------+-----+--------------------------------+------+

| 1012 | 101 | Accounting Department | NY |

| 1013 | 101 | Customer Grievances Department | KA |

| 1041 | 104 | Branch Office (East) | MA |

| 1042 | 104 | Branch Office (West) | CA |

| 1101 | 110 | Head Office | CA |

| 1032 | 103 | NE Region HO | CT |

| 1033 | 103 | NW Region HO | NY |

+------+-----+--------------------------------+------+

7 rows in set (0.00 sec)
```

How about deleting all those customers, any of whose branch offices generate service fee revenues of 0 or less?

```
mysql> DELETE FROM clients WHERE cid IN (SELECT DISTINCT b.cid FROM

branches AS b, branches_services AS bs, services AS s WHERE b.bid = bs.bid AND
bs.sid = s.sid GROUP BY bs.bid HAVING SUM(sfee) <= 500); Query OK, 1 row
affected (0.28 sec)
```

In this case, the inner query groups the various branches by branch ID, calculates the total service generated by each branch for all the services its using, and lists those records where the total is less than or equal to 0. The corresponding customer IDs are then used by the outer query to perform a DELETE operation on the "clients" table.

Adjusting For Inflation

You can use subqueries in an UPDATE statement in much the same manner. Let's suppose I wanted to find out which services are in use in 3 or more branch offices,

```
mysql> SELECT sid FROM branches_services GROUP BY sid HAVING COUNT(bid)

mysql> >= 3;

+-----+

| sid |

+-----+

| 1 |

| 3 |

| 4 |

+-----+

3 rows in set (0.11 sec)
```

and then increase the fee for those services by 25% (hey, those weekly yacht parties don't come cheap!).

```
mysql> UPDATE services SET sfee = sfee + (sfee * 0.25) WHERE sid = 1;

Query OK, 1 row affected (0.05 sec)

Rows matched: 1 Changed: 1 Warnings: 0
```

mysql> UPDATE services SET sfee = sfee + (sfee * 0.25) WHERE sid = 3;
Query OK, 1 row affected (0.05 sec)
Rows matched: 1 Changed: 1 Warnings: 0

mysql> UPDATE services SET sfee = sfee + (sfee * 0.25) WHERE sid = 4;
Query OK, 1 row affected (0.05 sec)
Rows matched: 1 Changed: 1 Warnings: 0

I could combine the operations above into the following subquery statement:

```
mysql> UPDATE services SET sfee = sfee + (sfee * 0.25) WHERE sid IN

mysql> (SELECT

sid FROM branches_services GROUP BY sid HAVING COUNT(bid) >= 3); Query OK,
3 rows affected (0.22 sec) Rows matched: 3 Changed: 3 Warnings: 0
```

Let's take another example. Let's suppose I wanted to have all branches located in California use the "Security" service instead of the "Administration" service. With a subquery, it's a piece of cake:

```
mysql> UPDATE branches_services SET sid = 6 WHERE sid = 4 AND bid IN

(SELECT bid FROM branches WHERE bloc = 'CA');

Query OK, 2 rows affected (0.00 sec)
```

```
Rows matched: 2 Changed: 2 Warnings: 0
```

In this case, the inner query takes care of isolating only those branch IDs in California, and provides this list to the outer query, which updates the corresponding records in the "branches_services" table. Notice how I've split the selection criteria for the rows to be UPDATEd: the inner query lists the records for California, the outer one further winnows it down to those using just the "Administration" service.

Wanna make it even more complicated? Add subqueries to the various SET clauses as well.

```
mysql> UPDATE branches_services SET sid = (SELECT sid FROM services

mysql> WHERE

sname = 'Security') WHERE sid = (SELECT sid FROM services WHERE sname =

'Administration') AND bid IN (SELECT bid FROM branches WHERE bloc = 'CA'); Query
OK, 2 rows affected (0.00 sec) Rows matched: 2 Changed: 2 Warnings: 0
```

A New Broom

Now for something a little off the beaten track. Let's suppose I'm on a "let's−clean−up−the−database" binge, and I need to check if the various tables are in sync with each other. For example, let's see if, for every customer, there exists at least one branch office in the "branches" table. The quickest way to do this is with a left join and an eyeball check for NULL values.

```
mysql> SELECT clients.cid, clients.cname, branches.bid, branches.bdesc

mysql> FROM

clients LEFT JOIN branches USING (cid);

+-----+-----------------------------+------+--------------------------------+

| cid | cname | bid | bdesc |

+-----+-----------------------------+------+--------------------------------+

| 101 | JV Real Estate | 1011 | Corporate HQ |
| 101 | JV Real Estate | 1012 | Accounting Department |

| 101 | JV Real Estate | 1013 | Customer Grievances Department |

| 102 | ABC Talent Agency | NULL | NULL |

| 103 | DMW Trading | 1031 | N Region HO |

| 103 | DMW Trading | 1032 | NE Region HO |

| 103 | DMW Trading | 1033 | NW Region HO |

| 104 | Rabbit Foods Inc | 1041 | Branch Office (East) |

| 104 | Rabbit Foods Inc | 1042 | Branch Office (West) |

| 110 | Sharp Eyes Detective Agency | 1101 | Head Office |

+-----+-----------------------------+------+--------------------------------+

10 rows in set (0.00 sec)
```

Hmmm. It's fairly obvious that we have a record for the customer "ABC Talent Agency", but no corresponding record for one or more branch offices. Since I've decided this is a major offense, I can nuke the client record with a fast DELETE using the customer ID,

```
mysql> DELETE FROM clients WHERE cid = 102;

Query OK, 1 row affected (0.01 sec)
```

or I could combine the two steps above into one using the following subquery:

```
mysql> DELETE FROM clients WHERE cid = (SELECT clients.cid FROM clients

LEFT JOIN branches USING (cid) WHERE bid IS NULL);

ERROR 1093: You can't specify target table 'clients' for update in FROM clause
```

Interesting. Do you know why?

It's actually pretty simple (and even makes sense when you think about it). MySQL won't let you delete or update a table's data if you're simultaneously reading that same data with a subquery, as doing so opens up a window where your subquery might reference rows which have already been deleted or altered. Therefore, the table named in an outer DELETE or UPDATE statement cannot appear in the FROM clause of an inner subquery (which is what MySQL more tersely said in its error message above).

Thus, as per the SQL standard, the query above is actually illegal, and should be rewritten using an EXISTS test as follows:

```
mysql> DELETE FROM clients WHERE NOT EXISTS (SELECT * FROM branches

mysql> WHERE
branches.cid = clients.cid);

Query OK, 1 row affected (0.11 sec)
```

A quick glance at the "clients" table will reveal that the offending entry has now been removed.

```
mysql> SELECT * FROM clients;

+-----+----------------------------+

| cid | cname |

+-----+----------------------------+

| 101 | JV Real Estate |

| 103 | DMW Trading |

| 104 | Rabbit Foods Inc |

| 110 | Sharp Eyes Detective Agency |

+-----+----------------------------+

4 rows in set (0.22 sec)
```

Cleaning Up

And that's about all we have time for. In this concluding segment of my MySQL subqueries tutorial, I explored some of the other ways in which subqueries can be used. These included membership tests with the IN operator and Boolean tests with the EXISTS operator; using subqueries to derive new tables which can then be used in the FROM clause of outer SELECT queries; and using subqueries with other SQL data manipulation commands like UPDATE and DELETE. In the context of the last item, I also focused on one of the more common things newbies trip up on − trying to write to a table in the outer query while it is being read by the inner one − and demonstrated an alternative approach to handling such a scenario.

As you have seen over the preceding pages, subqueries are a powerful tool, and open up all kinds of possibilities for the SQL developer. My intent, using a simple accounting system and some common (and no−so−common) tasks, was to demonstrate how subqueries can substantially simplify your life when working with complex MySQL databases, and why they're generally considered a Good Thing.

Remember, however, that power such as this comes at a price. Incorrectly−written subqueries can result in massive load increases on your RDBMS, and can reduce performance drastically, negating the overall gains provided by this feature. For this reason, I strongly advise you to always explore alternative methods of obtaining the data you need, including joins, unions and other SQL constructs (the first part of this tutorial demonstrates some of these constructs), so as to ensure optimal performance of your application and minimal resource wastage on the RDBMS.

Now go practice.

Note: 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!