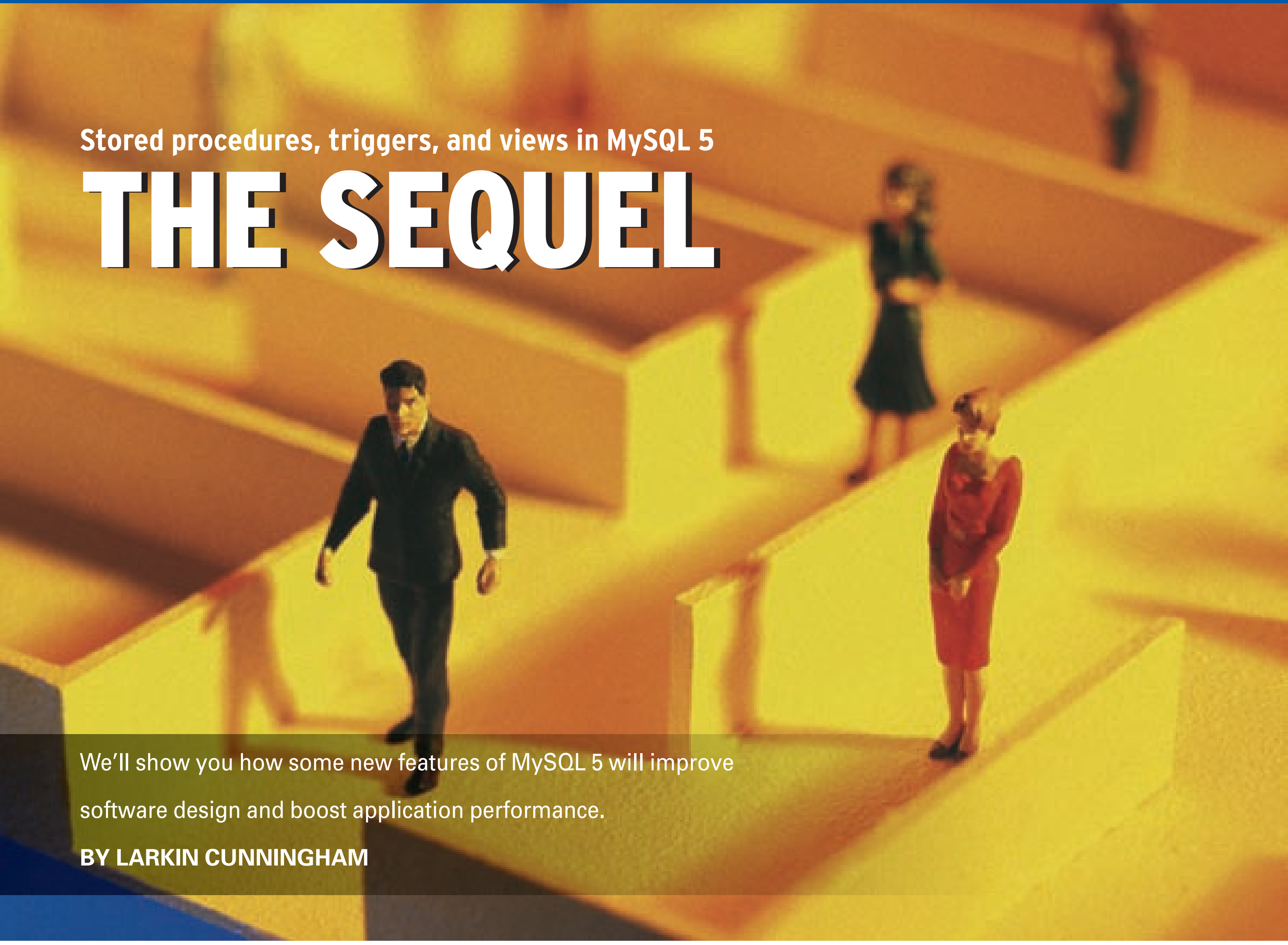


Stored procedures, triggers, and views in MySQL 5

THE SEQUEL

We'll show you how some new features of MySQL 5 will improve software design and boost application performance.

BY LARKIN CUNNINGHAM

The open source MySQL database system serves many of the world's leading enterprises, such as Yahoo and Ticketmaster. It also powers a legion of high volume open source websites like Wikipedia. Many enterprise organizations, however, have traditionally avoided MySQL in favor of feature-rich commercial database systems, such as Oracle and DB2. Starting with MySQL 5.0 [1], the MySQL developers have begun introducing a range of enterprise features that will ultimately make MySQL more competitive with commercial database systems. This article examines some of the enterprise features that have been making their way into MySQL. Many of these features were introduced in version 5.0, and some may be enhanced in version 5.1, which is in beta at this time of writing but may be official by the time you read this article. I used version 5.1.9-beta when testing the listings in this article.

Three of the most appealing new features in MySQL 5.x are stored procedures, triggers, and views. These features are hardly new for the industry. Oracle, for example, first introduced PL/SQL [2], its implementation of a procedural language for SQL, in 1991. Sybase, PostgreSQL, and DB2 are among the other database management systems with a procedural language for SQL. However, triggers, views, and stored procedures are nevertheless a welcome addition to MySQL.

I should point out that some of these MySQL enterprise features are in early stages of development. Many of these features are either incomplete or not performing at optimum levels. Version 5.1 has addressed some of the issues related to these new features, and the situation will no doubt continue to improve with later versions of MySQL.

THE AUTHOR

Larkin Cunningham just loves open source software. Larkin currently works with Oracle PL/SQL and Java, but he still finds time to dabble in all things Linux. You can contact Larkin at *larkin.cunningham@gmail.com.*

The Ordering Scenario

Throughout this article I will refer to a *products* table, an *order_headers table*, an *order_lines* table, a *stock_quantities* table and a *customers* table for the purposes of illustration. Listing 1 shows the SQL *create table* statements that create the tables. When giving examples of stored procedures, triggers, and views, I will refer to the tables in the listing.

Stored Procedures

Before explaining what stored procedures are, I should explain that when I use the term stored procedure, I am usually referring to both stored procedures and stored functions. A stored procedure accepts multiple input and output parameters. A stored function also accepts multiple input parameters but returns a single value to the caller. This restriction allows stored functions to be used within SQL statements, which effectively allows you to extend the capability of SQL.

Stored procedures are a powerful tool to have in a developer's arsenal. They can be of great benefit in terms of performance and application design. In terms of performance, it is possible to reduce a lot of network traffic by performing more data processing within the confines of the database. By reducing network traffic, you can eliminate the latency associated with the application server communicating with the database server, particularly when they are on separate servers, as is the case with most large scale applications.

With stored procedures you can take a black box approach to application design and development. A developer programming in Java, PHP, Ruby, or any other language with MySQL driver support does not need to have extensive knowledge of SQL or PL/SQL. On a multi-member development team, you can have stored procedure developers concentrating on stored procedure development and Java, PHP or Ruby developers concentrating on their particular programming language. As long as each developer is aware of the inputs and expected outputs, both developers can work in parallel. This can be a way of best leveraging the expertise of your developers, if the project is large enough to warrant dedicated development resources.

Portability is also aided by developing more of your logic within the database. It would be possible, for example, to develop a batch application using C, a web application using Ruby on Rails, and a web service developed using Java, and have them all using the same set of stored procedures.

The approach of many to developing applications that use a relational database is to either embed all of the SQL within their code or to embed all of the SQL in stored procedures and only call stored procedures from their code. Many developers rely on object-relational mappers such as Hibernate [3] (for Java) and ActiveRecord [4] (for Ruby on Rails), where stored procedures are largely irrelevant. Deciding on your approach to how to handle data processing in your applications will depend on factors such as performance and portability. If performance is not a concern, then you would be in a position to consider an object-relational mapper that generates your SQL on the fly. But if you care about performance, and you have service level agreements that demand a certain number of transactions per second or a response time in a certain number of milliseconds, you will want to investigate the merits of using stored procedures. If you operate a heterogeneous environment with many development platforms, then using stored procedures may be a way for you to develop your data processing logic once in a central location. After all, stored procedures do not care what programming language makes the call.

Triggers

Triggers have many uses, including house-keeping jobs like auditing and archival. They can have many other uses too. One common scenario is where a trigger is fired (more on trigger timing and firing later) after a row is created, for example an order line being added to the *order_lines* table. A trigger could be fired after the row is inserted to update the stock quantity of the product in the *stock_quantities* table.

Where archiving is required, you can have an additional archive table for each table where you want to store archive information. For example, the *products* table may have an associated *products_archive* table with all of the same columns as the products table. To automatically archive, you would create triggers on the products table to insert a row into the *products_archive* table after every update or delete. You would not create a trigger that is fired after an insert because to query the entire history for a product, you would retrieve the union of

Listing 1: The database schema for these examples

```
CREATE TABLE products (
    id          MEDIUMINT NOT NULL AUTO_INCREMENT,
    name        CHAR(40)  NOT NULL,
    cost        DOUBLE(9,2) UNSIGNED DEFAULT 0.0,
    PRIMARY KEY (id)
);

CREATE TABLE stock_quantities (
    id          MEDIUMINT NOT NULL AUTO_INCREMENT,
    product_id  MEDIUMINT NOT NULL,
    quantity    MEDIUMINT NOT NULL DEFAULT 0,
    PRIMARY KEY (id)
);

CREATE TABLE order_headers (
    id           MEDIUMINT NOT NULL AUTO_INCREMENT,
    customer_id  MEDIUMINT NOT NULL,
    order_date   DATETIME  NOT NULL,
    order_status CHAR(1) DEFAULT '0',
    PRIMARY KEY (id)
);

CREATE TABLE order_lines (
    id          MEDIUMINT NOT NULL AUTO_INCREMENT,
    order_id    MEDIUMINT NOT NULL,
    product_id  MEDIUMINT NOT NULL,
    quantity    MEDIUMINT NOT NULL DEFAULT 0,
    PRIMARY KEY (id)
);

CREATE TABLE  customers (
    id          MEDIUMINT NOT NULL AUTO_INCREMENT,
    name        VARCHAR(70) NOT NULL,
    address     VARCHAR(200) NOT NULL,
    phone       VARCHAR(20) NOT NULL,
    email       VARCHAR(40) NOT NULL,
    PRIMARY KEY  (id)
);
```

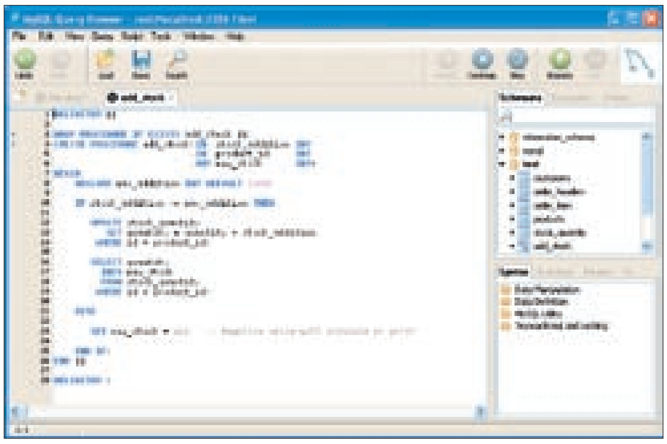


Figure 1: A simple stored procedure in MySQL Query Browser.

the row in the *products* table and the associated rows in the *products_archive* table.

The approach is similar when auditing is required. Instead of the approach of having an associated archive table where archiving is concerned, you might have a single audit table. For the tables you wanted to retain an audit trail of activity for, you might have triggers fired after any add, update, or delete. These triggers would insert a row into the audit table containing the nature of the action, the table affected, the user performing the action, the time stamp, and any key data or non-key data deemed appropriate. The approach of using triggers in the database for auditing and not in your application code can reduce the coding burden on your developers and encourage consistency in an environment where many applications access the same database. There are many good approaches to auditing that can be employed within your application code, so each case will need to be examined in context.

About Views

A view is a virtual table generated from a stored query. The stored query is often a multi-join query taking data from many tables with certain conditions attached. At a simpler, level it might be just a subset of a large table. A trivial example, again using the *products* table, is to create a view called *products_out_of_stock*, which joins the *products* table with the *stock_quantities* table, where the stock level is zero.

Views help you cut down on writing SQL code for commonly accessed data sets. Views also help efficiency because the underlying view query will be cached and will load faster than several versions of the same query running from different locations.

About the MySQL Procedural Language

MySQL 5 provides a procedural language you can use to create your stored procedures and triggers. Instead of going for a procedural language based on C or Python, the developers of MySQL created a procedural language compliant with the ANSI SQL:2003 standard [5]. The ANSI standard is used by the developers of other relational database management systems to varying degrees, so by following the standard, the skillset acquired in developing stored procedures and triggers for MySQL is transferable to other databases such as Oracle, DB2, and PostgreSQL, which have similar procedural language implementations.

Like the programming languages you might be familiar with, such as PHP and Java, MySQL's procedural language has the constructs you need to develop useful code. This includes conditional statements (IF-THEN-ELSE and CASE-WHEN) and iterative statements (REPEAT-UNTIL and WHILE-DO).

The length of this article does not allow for an exhaustive reference of all MySQL procedural language features. Instead, I will explain how MySQL stored procedures and triggers are structured and provide some simple examples that offer a flavor of what stored procedures, triggers and views really are. If you are a seasoned programmer in any modern language, MySQL's procedural language will seem rather simplistic. MySQL's procedural language is designed as a means for providing input into SQL statements and manipulating the results, not as a language to compete with the likes of PHP and Java.

The Structure of a Stored Procedure

Stored procedures are written in a way that allows them to be created by any tool that executes SQL. Some of my listings are displayed in MySQL Query Browser [6], a very useful and free tool from MySQL. They are written as SQL scripts that basically tell MySQL what the name of the stored procedure is and what the contents are. If the stored procedure contains any errors, MySQL will inform you when you attempt to create the stored procedure.

Figure 1 shows a stored procedure that accepts an integer value for an amount to be added to stock. Because the default delimiter in MySQL is a semi-colon, and MySQL's procedural language uses semi-colons to terminate each program statement, we need to instruct MySQL to change the delimiter while we attempt to create our procedure. The usual convention is to change the delimiter to double dollar signs with the *DELIMITER $$* statement (Line 1). The next statement (Line 3) instructs MySQL to drop (destroy) the existing stored procedure of the same name if it exists. If it does not exist, then this statement will be ignored and the MySQL parser will move on. Line 4 instructs MySQL to create a new stored procedure with the name and parameters provided. All stored procedure logic begins with the *BEGIN* statement

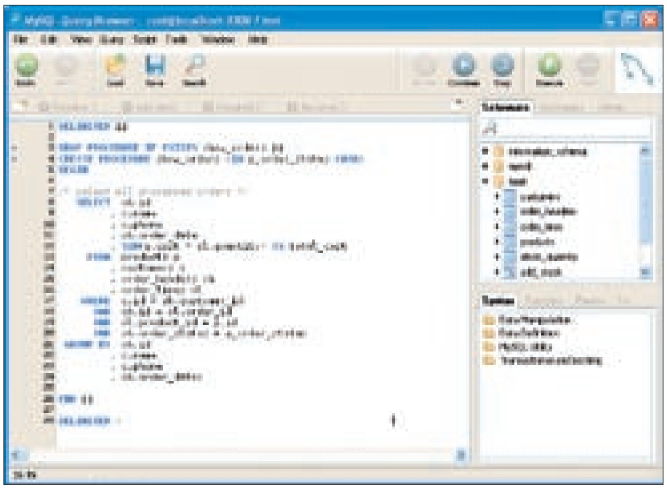


Figure 2: A procedure designed to return a result set to the caller.

(Line 7). A number of declaration, sequence, conditional, and iterative statements can follow before the stored procedure logic finishes with an *END* statement (Line 26). Note how the END statement is followed by our temporary delimiter, the double dollars. This is because we have now left the stored procedure and have returned to normal MySQL SQL parsing. At this point, we switch back to the default semi-colon delimiter (Line 28).

Variables, Parameters, and Data Types

In Figure 1, I have declared one variable *max_addition* (Line 8) and three parameters *stock_addition*, *product_id* and *new_stock* (Lines 4 to 6). The *IN* and *OUT* keywords tell MySQL that the parameter can receive a value in, pass a value back to the caller, or both (by declaring a parameter to be IN OUT). Parameters can be used like normal variables, but only OUT parameters should have their values changed in the procedure.

Variables must be explicitly declared and assigned a type as well as an optional default value. The types you can choose from are the standard SQL data types for table columns. All of the data types are scalar, that is, they can only store a single discrete value. This rules out data types such as arrays, which can be frustrating for developers from languages like PHP and Java, but there are workarounds, such as temporary tables using a memory storage engine. Some of the typical data types include *CHAR* and *VARCHAR* (for characters and strings), *DATE*, *DATETIME*, *INT* (including *TINYINT*, *SMALLINT*, *MEDIUMINT* and *BIGINT*), *DECIMAL*, *FLOAT*, *DOUBLE*, and others. Large amounts of data can be stored using other data types, such as *TEXT* (up to 64 kilobytes) and *BLOB* (binary large object-- in theory you can store up to 4 Terabytes in a *LONGBLOB*).

Using SQL in Stored Procedures

Unlike programming languages such as PHP and Java, there are no drivers to worry about and no special function or method calls to execute your SQL. Instead, SQL statements can be run on the fly and results read directly into variables. *UPDATE* and *INSERT* statements can read values directly from your variables and parameters.

In Figure 1, an *UPDATE* statement (Line 12) intermingles table names, column names, and parameters. In the following *SELECT* statement (Line 16), a value is selected directly INTO an *OUT* parameter. As I said earlier, MySQL's procedural language is ultimately a means for inputting data to SQL and processing the results.

In the *SELECT* statement (Line 16) in Figure 1, a value was selected into an *OUT* parameter. Assuming the id column guarantees uniqueness, this is fine. But, what if there were multiple values returned by the SQL statement? You should only select into a variable if you are 100% certain of a single value being returned. This will be the case where the discriminator (the clauses after the *WHERE* keyword) uses a unique key,

Listing 2: A stored procedure using a cursor

```
DELIMITER $$

DROP PROCEDURE IF EXISTS show_orders_processed $$
CREATE PROCEDURE show_orders_processed ()
BEGIN

DECLARE v_o_id      MEDIUMINT;
DECLARE v_c_name    VARCHAR(70);
DECLARE v_c_phone   VARCHAR(20);
DECLARE v_o_date    DATETIME;
DECLARE v_o_total   DOUBLE(9,2);
DECLARE not_found   TINYINT;

/* Select all processed orders */
DECLARE order_summary_cur CURSOR FOR
   SELECT  oh.id
         , c.name
         , c.phone
         , oh.order_date
         , SUM(p.cost * ol.quantity) AS total_cost
      FROM products p
         , customers c
         , order_headers oh
         , order_lines ol
     WHERE c.id = oh.customer_id
       AND oh.id = ol.order_id
       AND ol.product_id = p.id
       AND oh.order_status = 'P'
  GROUP BY oh.id
         , c.name
         , c.phone
         , oh.order_date;

DECLARE CONTINUE HANDLER FOR
   NOT FOUND
     SET not_found = 1;

SET not_found = 0;

OPEN order_summary_cur;

order_summary_loop:REPEAT

  FETCH  order_summary_cur
   INTO  v_o_id
       , v_c_name
       , v_c_phone
       , v_o_date
       , v_o_total;

  IF not_found THEN
     LEAVE order_summary_loop;
  END IF;

  SELECT CONCAT('Order ID: ', v_o_id, ', Name: ', v_c_name,
               ', Phone: ', v_c_phone, ', Order Date: ', v_o_date,
               ', Order Total: ', v_o_total);

UNTIL not_found
END REPEAT order_summary_loop;

CLOSE order_summary_cur;

END $$

DELIMITER ;
```

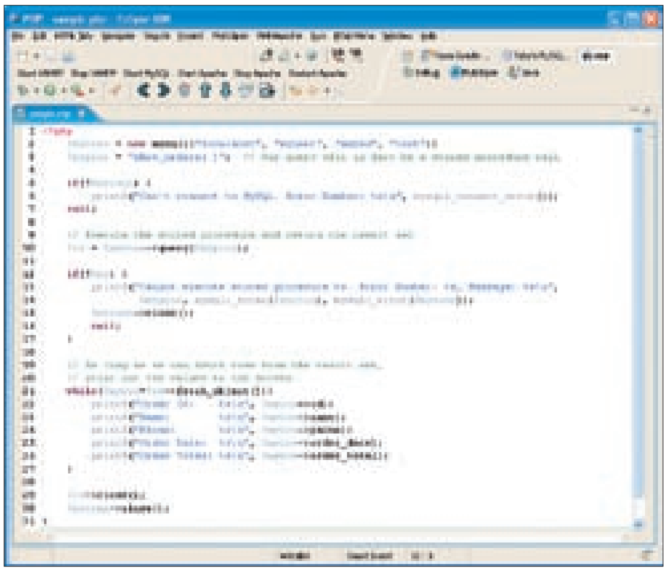


Figure 3: Sample PHP call to a stored procedure using mysqli.

such as an id column using an *auto_increment*, or where you select a function value into the variable, such as with *SUM()* or *MAX()*. I mentioned earlier that variables were scalar and could only hold single values. This rules out the possibility of returning a list of values directly into a variable. This is where the concept of cursors come to the rescue.

Using Cursors

A cursor is a pointer into a result set returned by a *SELECT* query. Though used primarily for *SELECT* queries that return more than one row, you can still use a cursor where only one row is returned. Even if there are no rows returned, an error will not be generated. However, if we try to fetch a row either from a null result set or if we try and fetch beyond the last row in the result set, a MySQL error will be thrown.

Listing 2 shows my preferred way of handling cursors using *REPEAT-UNTIL*. We begin the procedure by declaring some standard variables, including one called *not_found*. The *not_found* variable is used in conjunction with a HANDLER for the *NOT FOUND* condition. That is, when a *NOT FOUND* condition is encountered, such as with our cursor going beyond its bounds, the value of *not_found* will be set to *1* or *TRUE*.

Our cursor *order_summary_cur* is nothing more than a value assigned to a variable of that name until we actually OPEN it. Once opened, we can begin fetching from our cursor into variables in the same order as the columns in our cursor's select statement. To fetch all of the rows returned by our select query, we must use an iterative statement. There are a number of ways to do this, but my preferred way is the *REPEAT-UNTIL*. Though our *REPEAT* statement continues UNTIL a specific condition is found to be true (the *not_found* variable, in this case), we have the option to LEAVE the iteration before the UNTIL condition is reached. To do so, we use a label to name the iteration, *order_summary_loop* in this example. This allows us to leave the iteration before using any of the fetched variables, which, in the case of a fetch beyond the last row, will result in an error.

The *SELECT CONCAT* statement may look odd, but it is how we display the values returned by our cursor's query.

Returning Result Sets to Your Calling Code

If you are a hardened programmer using something like PHP, Java, Python or Ruby, you may wonder about the purpose of a stored procedure like the one in Listing 2, since it only displays the result to the console or in MySQL Query Browser. It's not much use if you would like to manipulate the data in the result set of that cursor. It is, however, possible to return a result set to your calling program without the cursor and handler code.

A plain SQL statement can be placed in a stored procedure without declaring a cursor and without performing a *SELECT* into any variables. It is written just as you would write it if you were executing your SQL in MySQL Query Browser or phpMyAdmin [7].

In the example in Listing 2, you can simply abandon all statements between the BEGIN and END other than the SQL query. Figure 2 shows this rather sleek stored procedure.

Notice how I have now changed the stored procedure to receive a parameter to select orders of a particular order status. This stored procedure can now potentially return the result set of the query to a calling program, assuming your calling programming language can support retrieving these unbounded result sets.

It is possible to have multiple SQL queries like in Figure 2. This may be useful for related sets of data, however, I prefer to stay clear of that approach and have single queries returning single result sets.

Example of a Stored Procedure Call

Many of you will have been waiting for me to show some sample code for your preferred programming language. I am going to show a sample using PHP. The approach is similar for other languages with MySQL driver support, such as Java, Ruby, and Python. My sample code will call the stored procedure in Figure 2.

For MySQL 5, you must have the object-oriented mysqli extension [8] loaded or compiled into PHP. Figure 3 shows the method calls using mysqli. Line 10 shows the stored procedure being called. You will notice that it does not appear to be different from a normal SQL call. The *while* statement on Line 21 loops through the rows in the result set

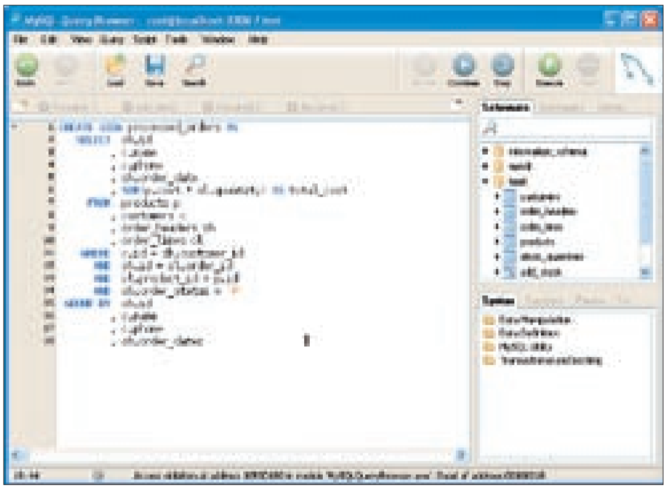
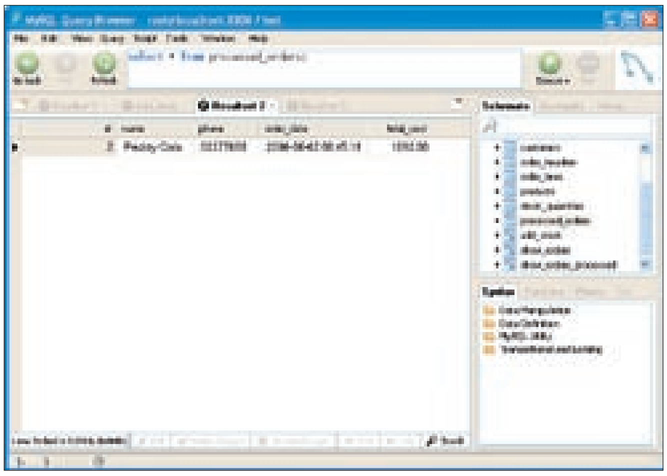


Figure 4: The SQL to create a view.



```
select * from processed_orders;
```

Figure 5: Using a view just like a regular query.

returned, just like a result set returned from executing a normal SQL query.

Trigger Example

While stored procedures are initiated by direct calls as their execution is required, triggers, on the other hand, are initiated by events that cause the triggers to be fired. The events in question are inserts, updates, and deletes.

Returning to the example tables in Listing 1, we can imagine a scenario where customer A orders product B. One approach to recording this in our database is to execute a stored procedure that inserts an order header and an order line and then updates the stock quantity for the product. However, another approach is to say that any time an order line is created, the corresponding stock quantity for the product ordered will always be reduced by the quantity ordered. We can consider this one of our business rules. Rather than having to write the update query every time we order a product, we can create a trigger that is executed every time an order line is inserted. This allows us to have concrete business rules enforced in the database.

Listing 3: A simple after update trigger

```
DELIMITER $$
CREATE TRIGGER order_lines_ins_trg
  AFTER UPDATE ON order_lines
  FOR EACH ROW
BEGIN
  UPDATE stock_quantities
     SET quantity = quantity - NEW.quantity
   WHERE product_id = NEW.product_id;
END $$
DELIMITER ;
```

Listing 3 shows a trigger that is fired after an update occurs. The *NEW* keyword refers to the new row values, as they are after the update has completed. The keyword *OLD* is also available and will contain the values of the row as it was before an update or delete. For archiving, for example, you might want to insert the old row values into an archive table for access to historical data. For auditing, you might insert both old and new values into an audit trail table.

You can also have a trigger fired before an update, insert, or delete occurs. This can be useful where you want to modify the *NEW* row values, for example, for validation purposes.

A View Example

In Listing 2, we had an attractive looking piece of SQL that retrieved processed order summaries. For many applications, this may be a useful result set to have around and re-use in several places within your application. A view is a mechanism for us to store and re-use such useful queries and access them as if they were just a regular table. The underlying complexity of the query is hidden from us, and we can simply select columns from this virtual table.

Views cannot accept parameters. If you need your query to accept parameters, you must create a stored procedure. Figure 4 shows the query from Listing 3 created in the form of a view. Figure 5 shows the results of running a query using the view. Notice how it looks like a query against any regular table.

Next Steps

In this article I have attempted to introduce you to some of the capabilities of MySQL 5 stored procedures, triggers, and views. These examples will give you an idea of whether these new features will be useful to your software development efforts. Don't forget that stored procedures are all about SQL queries. If you were writing inefficient SQL in your program code, you will probably still be writing inefficient SQL queries in your stored procedures.

The feature set is completely new to MySQL, but those of you who have worked with stored procedures in other databases, such as Oracle, DB2, and PostgreSQL, will probably be more interested in the differences between MySQL's implementation and what you are used to. MySQL's procedural language is not yet finished. Subsequent releases of MySQL 5 should improve the feature set considerably and address the areas where MySQL's implementation falls short of its competitors.

The documentation [9] of the new feature set on the MySQL website is adequate at best, though I am being kind when I write that. However, books are being published by MySQL Press and other publishers that give a more detailed overview of MySQL features. ■

INFO

[1] MySQL 5.0 Community Edition: *http://www.mysql.com/products/database/mysql/community_edition.html*

[2] Oracle's PL/SQL Technology Center: *http://www.oracle.com/technology/tech/pl_sql/index.html*

[3] Hibernate Object Relational Mapper for Java and .NET: *http://www.hibernate.org/*

[4] ActiveRecord Object Relational Mapper for Ruby: *http://rubyforge.org/projects/activerecord/*

[5] A Publication on ANSI SQL:2003 by Eisenberg et al: *http://www.sigmod.org/sigmod/record/issues/0403/E.JimAndrew-standard.pdf*

[6] MySQL Query Browser: *http://www.mysql.com/products/tools/query-browser/*

[7] phpMyAdmin: *http://sourceforge.net/projects/phpmyadmin*

[8] PHP's mysqli extension: *http://www.php.net/mysqli*

[9] MySQL's rather sparse online documentation for stored procedures: *http://dev.mysql.com/doc/refman/5.0/en/stored-procedures.html*