

Lecture 11

- Stored procedures
- User-defined functions
- Prepared statements

Stored procedures, user-defined functions, and prepared statements

- A stored procedure is a collection of SQL statements that can be called via a CALL statement.
- A user-defined function is also a collection of SQL statements, but it can be called and used like any built-in function
- A prepared statement is a query that is stored on the server and that can be executed in the future

Stored procedures

- Stored procedures must be declared before they can be called.
- The declaration can include parameters.
- If parameters are changed inside the procedure, their modified values are accessible after the call. De facto, this provides a way of returning values from the procedure.

Stored procedures

- Stored procedures are defined like this:

```
delimiter //  
  
CREATE PROCEDURE insert_pic(image BLOB,  
                             caption varchar(255),  
                             owner_id INT,  
                             is_public BOOL)  
  
    BEGIN  
        INSERT INTO Image (`image_data`, `caption`, `is_public`)  
            VALUES (image, caption, is_public);  
        SET @image_id = LAST_INSERT_ID();  
        INSERT INTO Image_owner (`image_id`, `owner_id`)  
            VALUES (@image_id, owner_id);  
    END  
  
//  
  
delimiter ;
```

- Note that the delimiter needs to be changed temporarily from a semicolon to a // or somesuch so we can use semicolon delimiters inside the BEGIN...END block of the procedure

Calling stored procedures

- We can now call our stored procedure like this:

```
CALL insert_pic('FFD8...', 'test pic', 5, 1);
```

- This inserts a new image into the image table with caption “test pic”, makes it public, and assigns it to user 5

- Get rid of a stored procedure with:

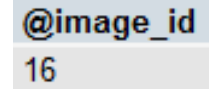
```
DROP PROCEDURE insert_pic;
```

Accessing values from stored procedures

- We might like to get the autoincrement insert id from the **Image_owner** table:

```
CALL insert_pic('FFD8...', 'test pic', 5, 1);  
SELECT @image_id;
```

- This is a little naughty, of course: why can we see a local variable outside a procedure?



@image_id
16

Accessing values from stored procedures via OUT parameters

- The alternative to this is to use an OUT parameter.

- For this, extend the declaration by another parameter:

```
CREATE PROCEDURE insert_pic(image BLOB,  
                             caption varchar(255),  
                             owner_id INT,  
                             is_public BOOL,  
                             OUT insert_id INT)...
```

- Now call the procedure as follows:

```
CALL insert_pic('FFD8...', 'test pic', 5, 1, @insert_id);  
SELECT @insert_id;
```

Note

- The following slides on user-defined functions describe a procedure that is not enabled on the university's student DB server
- You can try this at home though, but please refer to the MySQL manual for the required configuration of your server

User-defined functions

- A function is declared in a way similar to a procedure, with two differences:
- The declaration must specify which data type the function **RETURNS**, e.g., **RETURNS INT**
- The function must terminate with a **RETURN** statement, passing back a value of the declared type

User-defined functions

- A function for the same purpose as our procedure example might be created like this:

```
delimiter //  
  
CREATE FUNCTION insert_pic(image BLOB,  
                           caption varchar(255),  
                           owner_id INT,  
                           is_public BOOL)  
  
  RETURNS INT  
  BEGIN  
    INSERT INTO Image (`image_data`, `caption`, `is_public`)  
      VALUES (image, caption, is_public);  
    SET @image_id = LAST_INSERT_ID();  
    INSERT INTO Image_owner (`image_id`, `owner_id`)  
      VALUES (@image_id, owner_id);  
    RETURN @image_id;  
  END  
//  
delimiter ;
```

Using user-defined functions

- We can now call our function like this:

```
SELECT insert_pic('FFD8...', 'test pic', 5, 1);
```

- Get rid of a user-defined function with:

```
DROP PROCEDURE insert_pic;
```

Prepared statements

- A prepared statement is a SQL statement that generally contains one or more parameters in the form of placeholders
- The advantage of a prepared statement is that the RDBMS can work out those parts of the statement that do not depend on the parameters, so execution is faster at the time the statement is run
- A prepared statement is prepared with the PREPARE statement, which also gives the statement a name. The statement itself is given as a string.
- At a later point in time, the prepared statement can be executed via the EXECUTE statement, giving the statement's name and values for the parameters.

Preparing a statement

- Statements are prepared with the PREPARE statement, with the actual statement to be executed later contained in a string:

```
PREPARE stm_user_messages
FROM
    'SELECT subject
     FROM Message
     WHERE owner_id = ?';
```

- The question mark is a placeholder for an owner ID value to be supplied later.

Executing a prepared statement

- Prepared statements are only available to us within the same database session – once we log out, they are *deallocated*.
- To execute the statement, we need to set a variable for each question mark parameter, like so:

```
SET @a = 1;
EXECUTE stm_user_messages USING @a;
```

- In the case of several parameters, we separate them by comma:
... USING @a, @b, @c;

Notes on prepared statements

- If a placeholder is supposed to supply a string, then the ? is not put inside quotes.
- We cannot supply literals to the USING clause:
~~... USING 3, 'foo';~~
- PHPMyAdmin logs in and back out again every time you hit the "Go" button – so your prepared statements get deallocated. This means that in order to try this out, you must have all code in one SQL window before you hit "Go".

Today's lab sheet

- ...is on the web
- Today: we define procedures, functions, and a prepared statement