SQL Procedures, Triggers

CSC 321/621 – 3/27/2012

SQL Procedures, Triggers

- Keep your eyes open for some “meta-ideas” that appear in lots of different contexts
  - Iterators
    - Traversing over datasets
  - Event driven programming & exception handling
    - Responding to specific events on the fly with specific code
  - Encapsulation/information hiding
    - Only exposing what is needed to the end user
    - Localized ‘sanity checking’

SQL: Stored Procedures

- Why Stored Procedures?
  - Support more complex management of data
  - Offload work to DBMS instead of client
  - Generic parameterizable interfaces to queries
  - Employ same logic across multiple clients
  - Write & debug the code on the server side, make clients just call procedures
  - Support complex processing on inserts/updates/deletes (triggers)

SQL: MySQL Specific

- My examples are going to be MySQL specific
  - Other database implementations are slightly different
  - Chapters 12 and 18 of the MySQL documentation

- To cover:
  - Blocks, Variables, Assignments, Conditionals, Loops

Stored Procedures/Functions

- Function: Has arbitrary number of input parameters, returns one value

- Procedure: Arbitrary number of input and output parameters, returns no values (except via setting of output parameters)

SQL Syntax: Simple Stored Function

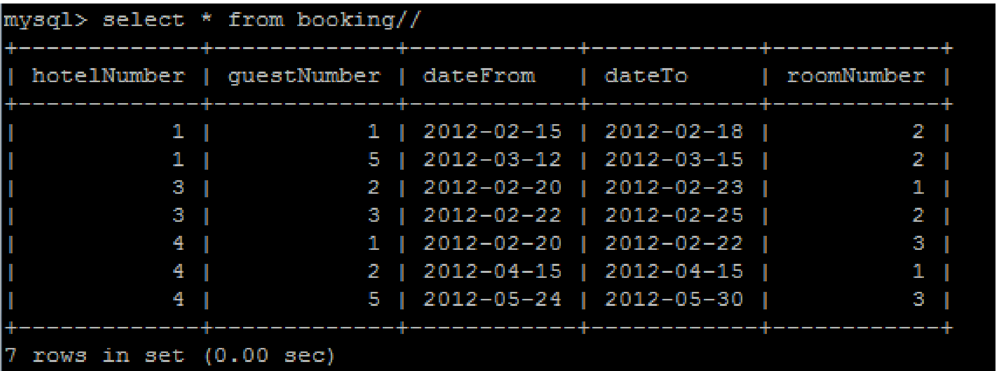
```
CREATE FUNCTION
   reservationsMadeAtHotel(hn INT)
   RETURNS INT
          BEGIN
                 DECLARE x INT;
                 SELECT COUNT(*) INTO x FROM
                   booking WHERE hotelNumber=hn;
                 RETURN(x);
          END;
```

New ideas:
* Variable declarations
* SELECT INTO
* Parameters

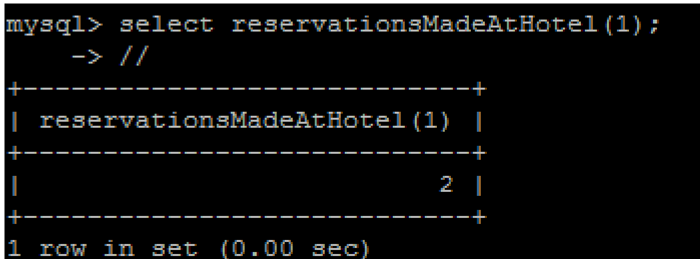
Functions can only return a singular value, not a result set – can't directly return SELECT result (use INTO *x* to push into a declared variable named *x)*

SQL Syntax: Simple Stored Function

```
CREATE FUNCTION
    reservationsMadeAtHotel(hn INT)
    RETURNS INT
        BEGIN
            DECLARE x INT;
            SELECT COUNT(*) INTO x FROM
         booking WHERE hotelNumber=hn;
            RETURN(x);
        END;
```



```
mysql> select * from booking//
+-------------+-------------+------------+------------+------------+
| hotelNumber | guestNumber | dateFrom   | dateTo     | roomNumber |
+-------------+-------------+------------+------------+------------+
|           1 |           1 | 2012-02-15 | 2012-02-18 |          2 |
|           1 |           5 | 2012-03-12 | 2012-03-15 |          2 |
|           3 |           2 | 2012-02-20 | 2012-02-23 |          1 |
|           3 |           3 | 2012-02-22 | 2012-02-25 |          2 |
|           4 |           1 | 2012-02-20 | 2012-02-22 |          3 |
|           4 |           2 | 2012-04-15 | 2012-04-15 |          1 |
|           4 |           5 | 2012-05-24 | 2012-05-30 |          3 |
+-------------+-------------+------------+------------+------------+
7 rows in set (0.00 sec)
```



```
mysql> select reservationsMadeAtHotel(1);
    -> //
+----------------------------+
| reservationsMadeAtHotel(1) |
+----------------------------+
|                          2 |
+----------------------------+
1 row in set (0.00 sec)
```

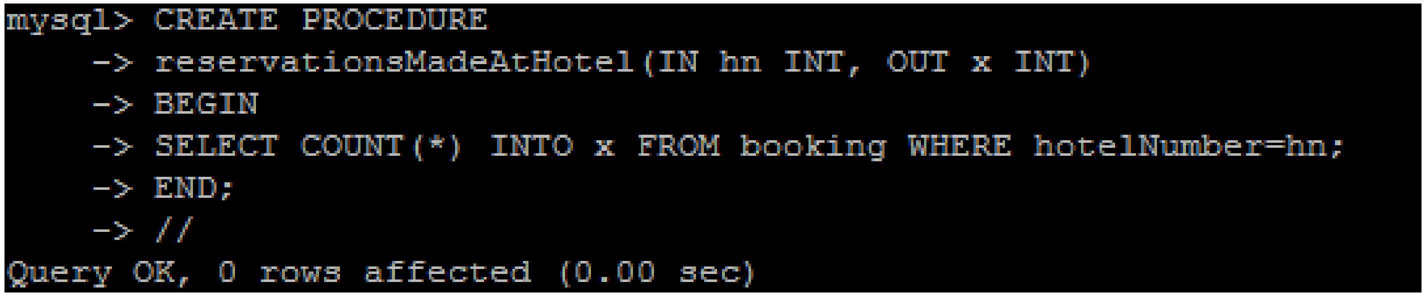
SQL Syntax: Stored Procedure

```
CREATE PROCEDURE
        reservationsMadeAtHotel
                (IN hn INT, OUT x INT)
        BEGIN
        SELECT COUNT(*) INTO x FROM
            booking WHERE hotelNumber=hn;
        END;

    CALL reservationsMadeAtHotel(1,@theCount);
    SELECT @theCount;
```

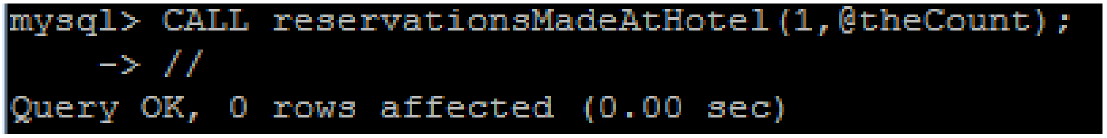
New ideas:

- IN and OUT parameters
- Variables at prompt

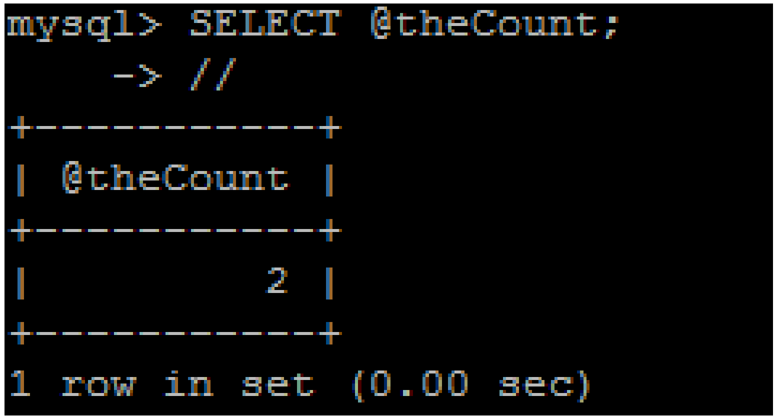
SQL Syntax: Stored Procedure



```
mysql> CREATE PROCEDURE
    -> reservationsMadeAtHotel(IN hn INT, OUT x INT)
    -> BEGIN
    -> SELECT COUNT(*) INTO x FROM booking WHERE hotelNumber=hn;
    -> END;
    -> //
Query OK, 0 rows affected (0.00 sec)
```



```
mysql> CALL reservationsMadeAtHotel(1,@theCount);
    -> //
Query OK, 0 rows affected (0.00 sec)
```



```
mysql> SELECT @theCount;
    -> //
+-----------+
| @theCount |
+-----------+
|         2 |
+-----------+
1 row in set (0.00 sec)
```

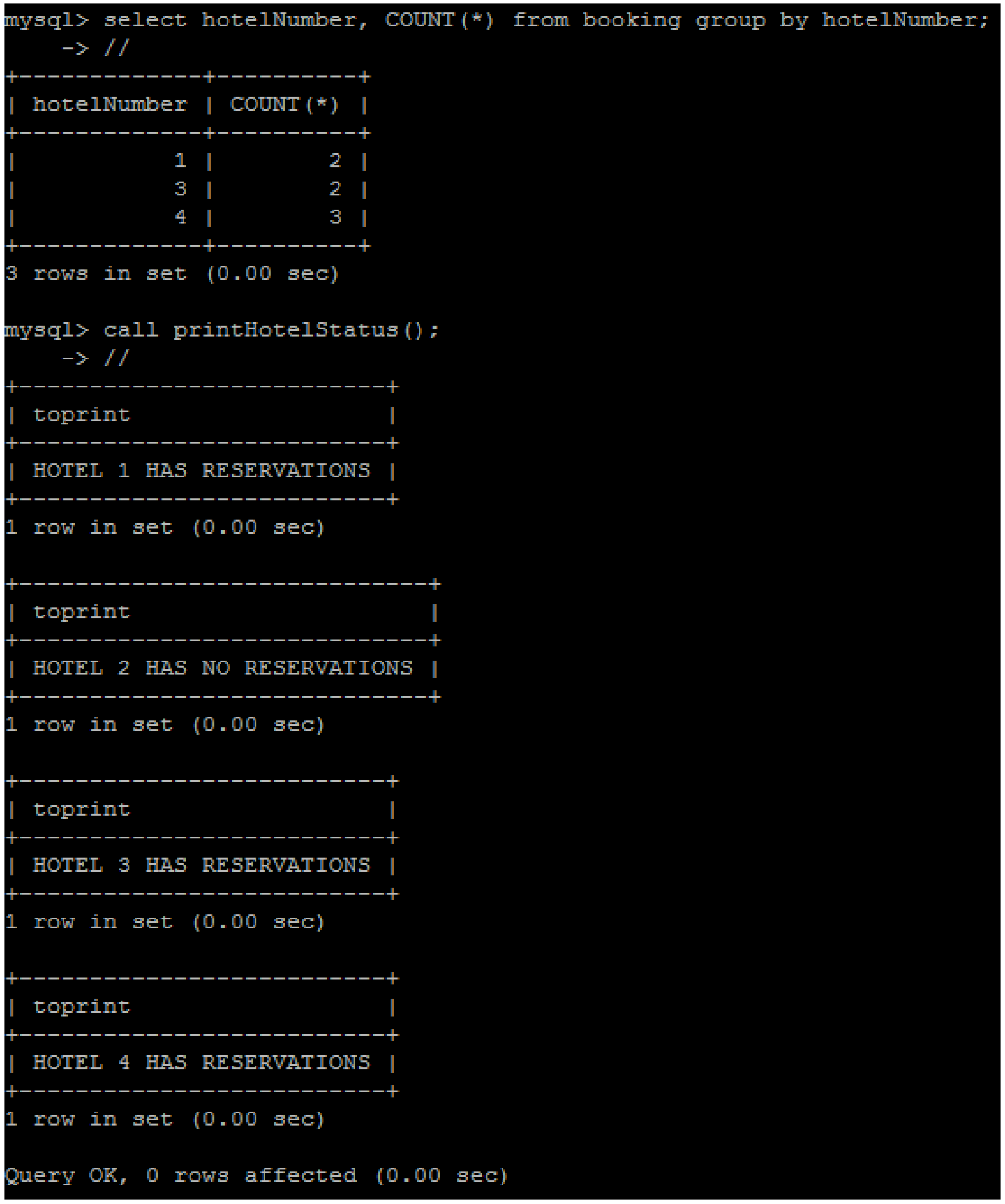
SQL Syntax: Stored Procedure

```
CREATE PROCEDURE      printHotelStatus()
  BEGIN
     DECLARE mh INT;
     DECLARE h INT;
     DECLARE x INT;
     DECLARE toprint TEXT;
     SELECT MAX(hotelNumber) INTO  mh FROM hotel;
     SET h = 1;
     WHILE (h <= mh) DO
          SELECT COUNT(*) INTO x FROM  booking WHERE hotelNumber=h;
          IF x = 0 THEN SET toprint = CONCAT("HOTEL ", h, " HAS NO RESERVATIONS");
          ELSE SET toprint = CONCAT("HOTEL ", h, " HAS RESERVATIONS");
          END IF;
          SELECT toprint; -- this will immediately print to the screen our string of interest
          SET h = h + 1;
     END WHILE;
   END;
```

New ideas:
- SET, = for assignment
- WHILE Loop
- IF Statement

What do you think this will do?

SQL Syntax: Stored Procedure



```
mysql> select hotelNumber, COUNT(*) from booking group by hotelNumber;
    -> //
+-------------+----------+
| hotelNumber | COUNT(*) |
+-------------+----------+
|           1 |        2 |
|           3 |        2 |
|           4 |        3 |
+-------------+----------+
3 rows in set (0.00 sec)

mysql> call printHotelStatus();
    -> //
+--------------------------+
| toprint                  |
+--------------------------+
| HOTEL 1 HAS RESERVATIONS |
+--------------------------+
1 row in set (0.00 sec)

+-----------------------------+
| toprint                     |
+-----------------------------+
| HOTEL 2 HAS NO RESERVATIONS |
+-----------------------------+
1 row in set (0.00 sec)

+--------------------------+
| toprint                  |
+--------------------------+
| HOTEL 3 HAS RESERVATIONS |
+--------------------------+
1 row in set (0.00 sec)

+--------------------------+
| toprint                  |
+--------------------------+
| HOTEL 4 HAS RESERVATIONS |
+--------------------------+
1 row in set (0.00 sec)

Query OK, 0 rows affected (0.00 sec)
```

SQL Syntax: Conditions

- Similar to Java exceptions, it is possible to catch when certain conditions have occurred and respond to (“handle”) those conditions
- Conditions may include:
  - Errors (very similar to Java Exceptions model)
  - Special states of the system
  - Programmer defined states of the system

(Unfortunately, the ability to have programmer defined conditions is not allowed until MySQL 5.5)

SQL Syntax: Conditions

- Conditions we can listen for:
  - SQLEXCEPTION
  - SQLWARNING
  - NOT FOUND (important for cursors – coming up next!)
  - Any MySQL error code

- We can define *handlers* to respond to the conditions
- The handler can be set to
  - Execute and then return control back to the calling compound statement (CONTINUE)
  - Execute and leave the calling compound statement (EXIT)

Which of these options is closer to how Java try/catch works?

SQL Syntax: Conditions

Syntax: DECLARE ***handler_action*** HANDLER FOR ***condition_value*** [, ***condition_value***] ... ***Statement***

Statement can be any SQL

Example: DECLARE *CONTINUE* HANDLER FOR *NOT FOUND* **SET done = TRUE**;

Explanation: This is a handler for the condition raised for when data is *NOT FOUND*.

Handling it, we will set a variable called *done* to the value *true* (we can refer to that variable elsewhere).

The function which triggered the NOT FOUND condition will then be returned to after the condition is handled (and will likely deal with the *done* variable).

SQL Syntax: Conditions

- If you want to recognize and handle specific SQL errors/warnings, you can define conditions for the appropriate MySQL error/warning number, and then handle that condition.
- An example from MySQL documentation:

```
DECLARE no_such_table CONDITION FOR 1051;
DECLARE CONTINUE HANDLER FOR no_such_table
  BEGIN
    -- body of handler
  END;
```

There is a long list of error codes here:
http://dev.mysql.com/doc/refman/5.1/en/error-messages-server.html

SQL Syntax: Conditions

A few examples of error conditions/SQL states:

- Error: 1004 SQLSTATE: HY000 (ER_CANT_CREATE_FILE)

  Message: Can't create file '%s' (errno: %d)
- Error: 1005 SQLSTATE: HY000 (ER_CANT_CREATE_TABLE)

  Message: Can't create table '%s' (errno: %d)
- Error: 1006 SQLSTATE: HY000 (ER_CANT_CREATE_DB)

  Message: Can't create database '%s' (errno: %d)
- Error: 1007 SQLSTATE: HY000 (ER_DB_CREATE_EXISTS)

  Message: Can't create database '%s'; database exists
- Error: 1008 SQLSTATE: HY000 (ER_DB_DROP_EXISTS)

  Message: Can't drop database '%s'; database doesn't exist
- Error: 1009 SQLSTATE: HY000 (ER_DB_DROP_DELETE)

  Message: Error dropping database (can't delete '%s', errno: %d)
- Error: 1010 SQLSTATE: HY000 (ER_DB_DROP_RMDIR)

  Message: Error dropping database (can't rmdir '%s', errno: %d)
- Error: 1011 SQLSTATE: HY000 (ER_CANT_DELETE_FILE)

  Message: Error on delete of '%s' (errno: %d)
- Error: 1012 SQLSTATE: HY000 (ER_CANT_FIND_SYSTEM_REC)

  Message: Can't read record in system table
- Error: 1042 SQLSTATE: 08S01 (ER_BAD_HOST_ERROR)

  Message: Can't get hostname for your address
- Error: 1043 SQLSTATE: 08S01 (ER_HANDSHAKE_ERROR)

  Message: Bad handshake
- Error: 1044 SQLSTATE: 42000 (ER_DBACCESS_DENIED_ERROR)

  Message: Access denied for user '%s'@'%s' to database '%s'
- Error: 1045 SQLSTATE: 28000 (ER_ACCESS_DENIED_ERROR)

  Message: Access denied for user '%s'@'%s' (using password: %s)
- Error: 1046 SQLSTATE: 3D000 (ER_NO_DB_ERROR)

  Message: No database selected
- Error: 1047 SQLSTATE: 08S01 (ER_UNKNOWN_COM_ERROR)

  Message: Unknown command
- Error: 1048 SQLSTATE: 23000 (ER_BAD_NULL_ERROR)

  Message: Column '%s' cannot be null
- Error: 1049 SQLSTATE: 42000 (ER_BAD_DB_ERROR)

  Message: Unknown database '%s'
- Error: 1050 SQLSTATE: 42S01 (ER_TABLE_EXISTS_ERROR)

  Message: Table '%s' already exists
- Error: 1051 SQLSTATE: 42S02 (ER_BAD_TABLE_ERROR)

  Message: Unknown table '%s'

SQL Syntax: Conditions

- SQLWARNING is an abstraction of lots of various issues (all SQLSTATE values beginning with 01)
- NOT FOUND is an abstraction of various states (all SQLSTATE VALUES beginning with 02)
- SQLEXCEPTION is an abstraction of various states (all other SQLSTATE values than those starting with 00,01, and 02)

Does Java have a similar hierarchy of Exceptions?

SQL Syntax: Conditions

- Why introduce conditions?
  - Used when dealing with cursors in a stored procedure (needing to access more than one row of data)
  - Similar to triggers (in concept) – handling special events

- Next set of slides deal with cursors and triggers

SQL Syntax: Cursors

- The SELECT statement in a stored procedure is a singular call – we can extract one value out of it.
  - We can use INTO to push that value into a variable
  - We can also assign that value directly to a variable using the SET and assignment operator

- What if the SELECT returns a table of > 1 tuple and we want to visit every tuple in the stored procedure?

- SQL supports cursors, which are very similar to iterators
  - A way to visit each tuple returned

SQL Syntax: Cursors

- A cursor has to be declared
  - Like a variable
  - Includes setting SQL SELECT which cursor represents
- A cursor has to be opened
  - Makes the actual query
- Each tuple is retrieved using a FETCH command
- The end of a cursor (out of data) is indicated by a NOT FOUND condition being raised (CONDITIONS!)
- A cursor has to be closed
  - Cleans everything up/indicates done

SQL Syntax: Cursors

- Declare a cursor *cur* by:

```
DECLARE cur CURSOR FOR <query>;
```

- To use cursor *cur*, we must issue the command:

```
OPEN cur;
```

  - The query of *cur* is evaluated, and *cur* is set to point to the first tuple of the result.

- When finished with *cur*, issue command:

```
CLOSE cur;
```

SQL Syntax: Cursors

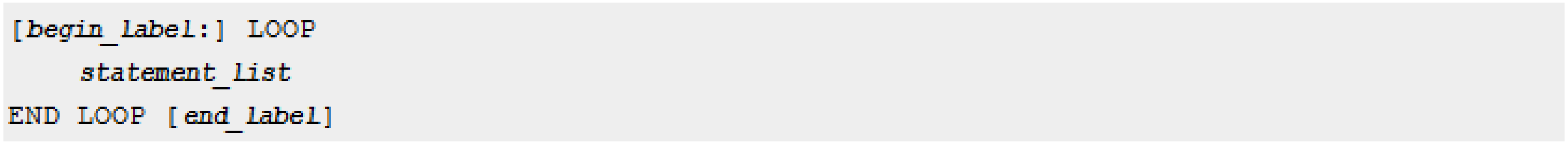
- To get the next tuple from cursor c, issue command:

  FETCH cur INTO x1, x2,…,x*n* ;
- The *x* ’s are a list of variables, one for each component of the tuples referred to by *cur*.
- *cur* is moved automatically to the next tuple.

- Usually, put FETCH statements in a LOOP (SQL syntax)
- A tricky point is how we get out of the loop when the cursor has no more tuples to deliver

SQL Syntax: Loop

- The LOOP structure is as follows:



```
[begin_label:] LOOP
    statement_list
END LOOP [end_label]
```

- Inside the loop statement list, a LEAVE command says to exit the loop
- For some structuring of loops, an ITERATE command says to restart the loop (not necessarily needed)
- ITERATE and LEAVE both take as their argument the LOOP label to repeat/exit

SQL Syntax: Cursors

- A tricky point is how we get out of the loop when the cursor has no more tuples to deliver

In what other programming situations do you run across this case?

SQL Syntax: Cursors

- A tricky point is how we get out of the loop when the cursor has no more tuples to deliver

In these cases, we typically need to see a "sentinel" signal to indicate we are out of data

Running out of data at the end of a file

Running out of tokens after tokenizing a string

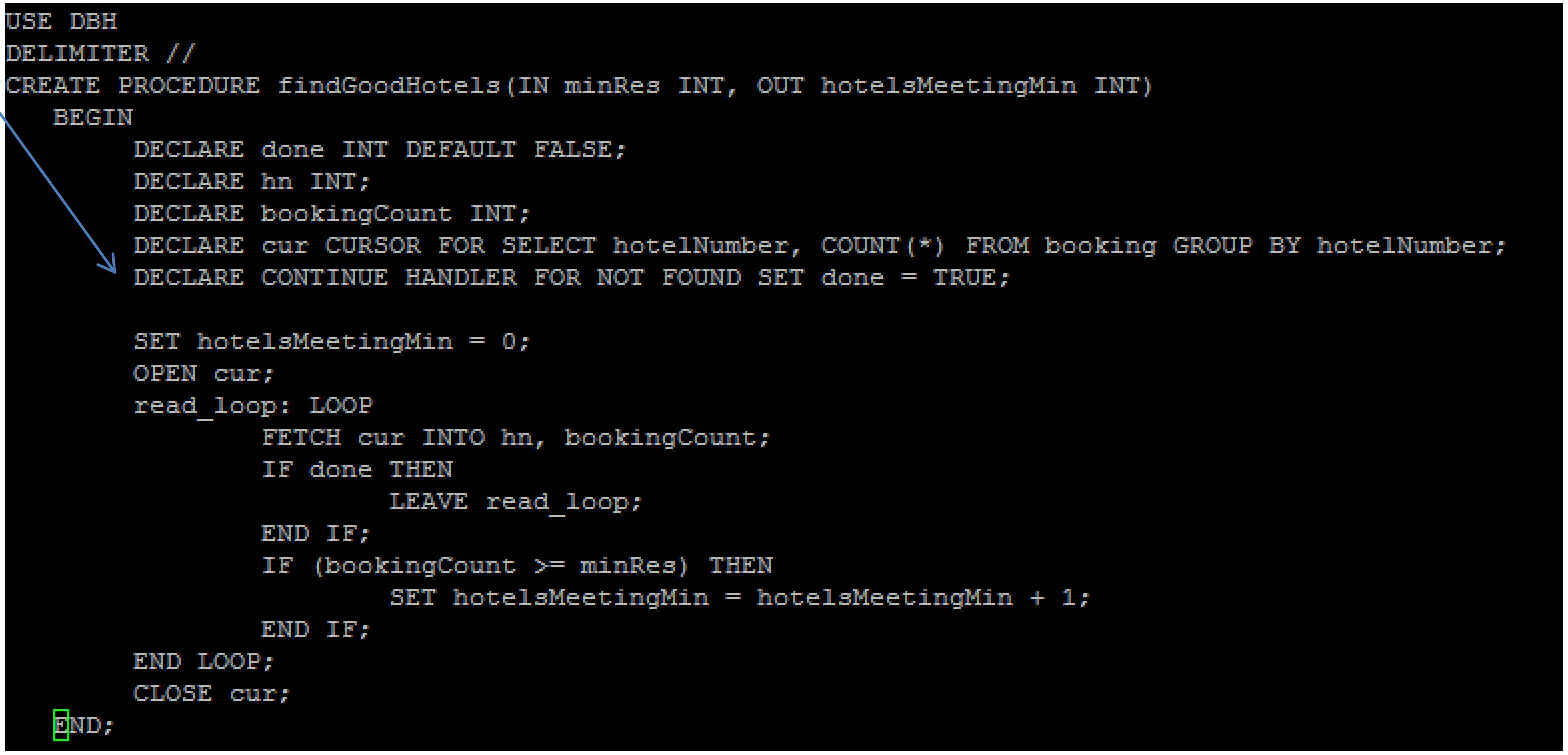
Iterating down an arraylist/vector

SQL Syntax: Cursors

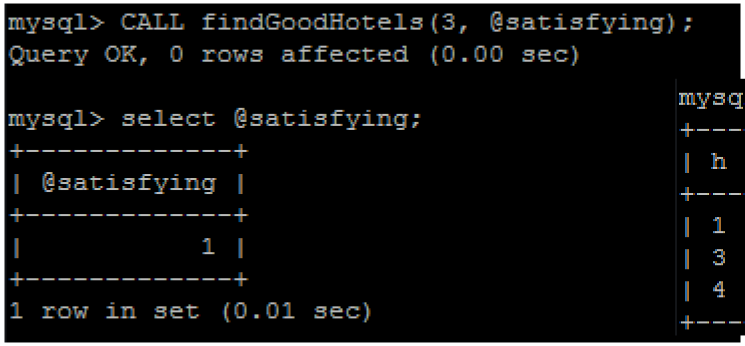
- When data runs out in cursor, the NOT FOUND condition is raised
  - If we handle that, set some variable/flag to indicate such, then continue in the stored procedure, checking for the flag to be set and calling LEAVE if it is set, we can exit gracefully

SQL Syntax: Cursors Example

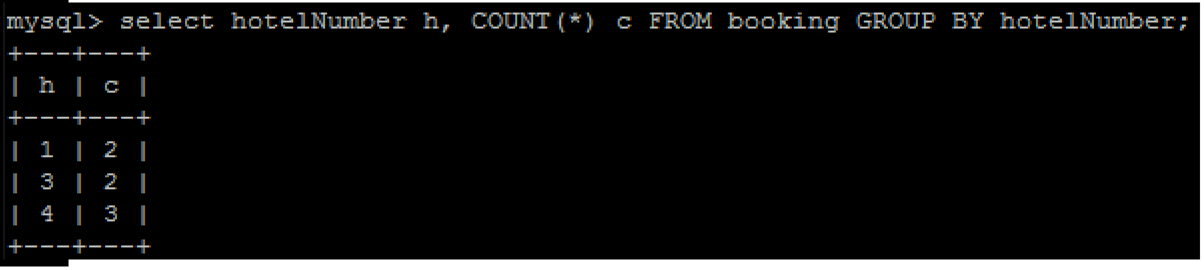
Note CONTINUE CURSOR



```
USE DBH
DELIMITER //
CREATE PROCEDURE findGoodHotels(IN minRes INT, OUT hotelsMeetingMin INT)
   BEGIN
        DECLARE done INT DEFAULT FALSE;
        DECLARE hn INT;
        DECLARE bookingCount INT;
        DECLARE cur CURSOR FOR SELECT hotelNumber, COUNT(*) FROM booking GROUP BY hotelNumber;
        DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;

        SET hotelsMeetingMin = 0;
        OPEN cur;
        read_loop: LOOP
                FETCH cur INTO hn, bookingCount;
                IF done THEN
                        LEAVE read_loop;
                END IF;
                IF (bookingCount >= minRes) THEN
                        SET hotelsMeetingMin = hotelsMeetingMin + 1;
                END IF;
        END LOOP;
        CLOSE cur;
   END;
```



```
mysql> CALL findGoodHotels(3, @satisfying);
Query OK, 0 rows affected (0.00 sec)

mysql> select @satisfying;
+-------------+
| @satisfying |
+-------------+
|           1 |
+-------------+
1 row in set (0.01 sec)
```



```
mysql> select hotelNumber h, COUNT(*) c FROM booking GROUP BY hotelNumber;
+---+---+
| h | c |
+---+---+
| 1 | 2 |
| 3 | 2 |
| 4 | 3 |
+---+---+
```

SQL Syntax: Cursors

- The notion of cursors propagates into client side APIs that allow you to make calls to databases like MySQL
- Java API: A ResultSet is returned from query, call next() to step through ResultSet
- You are not actually manipulating the cursor in the database, just emulating idea

```
public static void viewTable(Connection con, String dbName)
    throws SQLException {

    Statement stmt = null;
    String query =
        "select COF_NAME, SUP_ID, PRICE, " +
        "SALES, TOTAL " +
        "from " + dbName + ".COFFEES";

    try {
        stmt = con.createStatement();
        ResultSet rs = stmt.executeQuery(query);
        while (rs.next()) {
            String coffeeName = rs.getString("COF_NAME");
            int supplierID = rs.getInt("SUP_ID");
            float price = rs.getFloat("PRICE");
            int sales = rs.getInt("SALES");
            int total = rs.getInt("TOTAL");
            System.out.println(coffeeName + "\t" + supplierID +
                               "\t" + price + "\t" + sales +
                               "\t" + total);
        }
    } catch (SQLException e ) {
        JDBCTutorialUtilities.printSQLException(e);
    } finally {
        if (stmt != null) { stmt.close(); }
    }
}
```

Very similar to iterators, enumerations

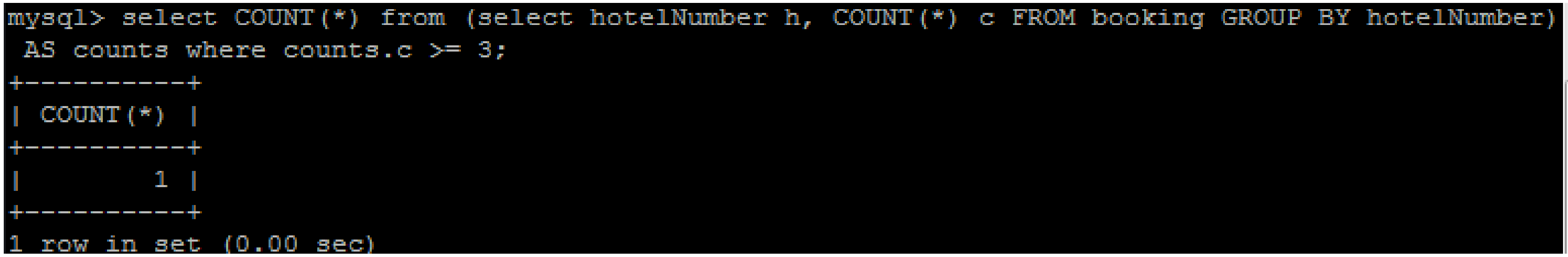
SQL Syntax: Cursors

- Just as an aside, the procedure on the previous page maps back to an SQL statement we easily could have written (not always true that we could come up with such a simple mapping)

What is that query?

SQL Syntax: Cursors

- Just as an aside, the procedure on the previous page maps back to an SQL statement we easily could have written (not always true that we could come up with such a simple mapping)



```
mysql> select COUNT(*) from (select hotelNumber h, COUNT(*) c FROM booking GROUP BY hotelNumber)
 AS counts where counts.c >= 3;
+----------+
| COUNT(*) |
+----------+
|        1 |
+----------+
1 row in set (0.00 sec)
```

SQL Syntax: Triggers

- A trigger allows one to define actions that are spawned (triggered) when certain events occur in the system
  - Not limited to error conditions/SQLSTATE changes

- To me, similar to requesting to “listen” for events in Java context

SQL Syntax: Triggers

- Triggers can be associated with data changes:
  - INSERT
  - UPDATE
  - DELETE

- Triggers can fire, relative to update:
  - Before
  - After

Section 18.3 of MySQL documentation

SQL Syntax: Triggers

- Typically want to use BEFORE triggers to support
  - Enforcing constraints
    - Verify conditions are met before actually do the change
  - Data manipulations
    - Massaging data into format as needed

- Typical use of AFTER triggers is to support
  - Auditing
    - Ensure the change has actually happened before recording

Section 18.3 of MySQL documentation

SQL Syntax: An Example Case For Triggers

- Assume we want to constrain a client from having more than five reservations at a time in our DBH Hotel Booking system
  - We can with a trigger, before adding a new reservation, count how many they already have and reject the insert

SQL Syntax: Trigger Motivation

- We could support enforcing constraints and auditing in *a client application itself.*

How might we do it, and what are the pros & cons of such an approach?

SQL Syntax: Trigger Motivation

- We could support enforcing constraints and auditing in a client application itself through multiple queries/calls from client side
  - Enforcing constraint:
    - Before inserting a new reservation, have the client make a call to see how many reservations are already present, and if at limit, not make 2nd call
  - Auditing:
    - Have the client make two updates calls, one to the table to be updated, one to the audit table

SQL Syntax: Trigger Motivation

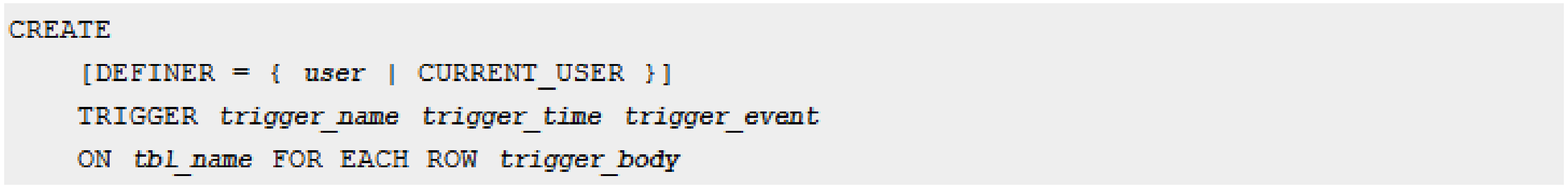
- By allowing triggers in the database, *what benefits do we receive?*
  - *Besides the client only having to make one call*

SQL Syntax: Trigger Motivation

- By allowing triggers in the database, we can:
  - Ensure all clients act the same
  - Only have to write logic in one language (SQL) in one place, not in multiple languages
  - Lower overhead on client side (but more on SQL server side)

SQL Syntax: Triggers

- Can only be applied to real tables (not views)



```
CREATE
    [DEFINER = { user | CURRENT_USER }]
    TRIGGER trigger_name trigger_time trigger_event
    ON tbl_name FOR EACH ROW trigger_body
```

- trigger_time: BEFORE, AFTER
- trigger_event: INSERT, UPDATE, DELETE

MySQL supports *row_level* triggers – fire once for each row affected by the triggering event.

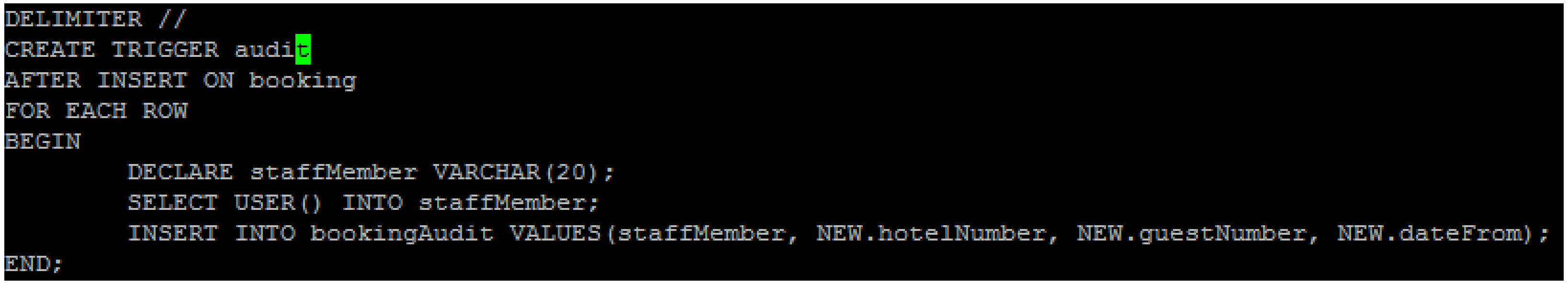
Can't, in MySQL, define two triggers for the same table, same event, same time ➔ use one trigger with compound statement instead

SQL Syntax: Triggers

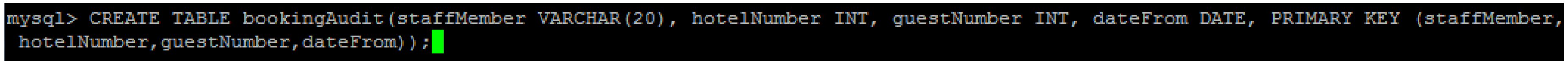
- For a given row being updated, can refer to pre-updated and post-updated versions (where applicable)
  - Pre-update, delete: OLD.columnName
  - Post-update, insert: NEW.columnName

SQL Syntax: An Auditing Trigger

- Whenever an insert is made, add information about who made the insert to an auditing table
  - USER() can be called as a function to return current user

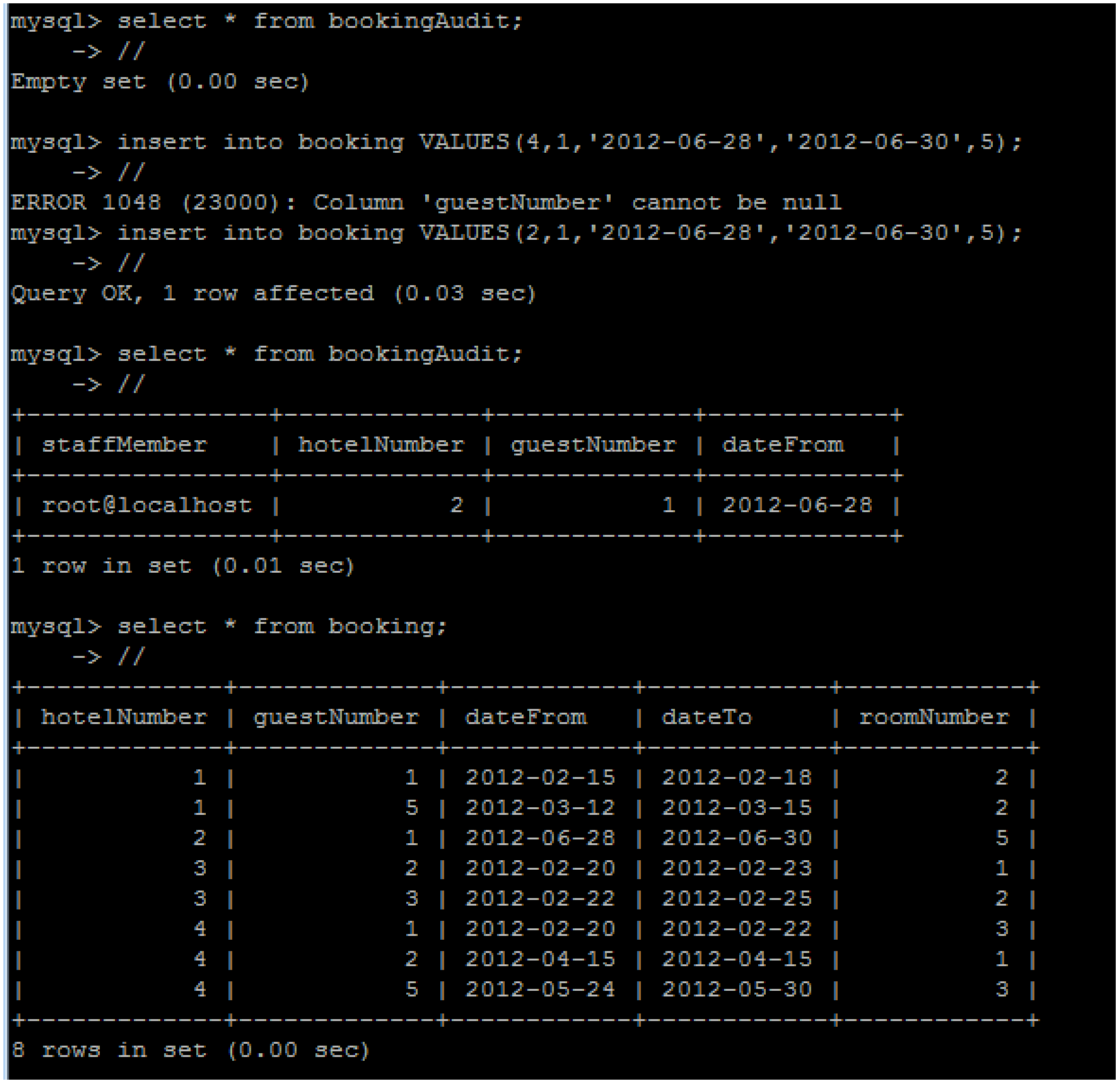


```
DELIMITER //
CREATE TRIGGER audit
AFTER INSERT ON booking
FOR EACH ROW
BEGIN
        DECLARE staffMember VARCHAR(20);
        SELECT USER() INTO staffMember;
        INSERT INTO bookingAudit VALUES(staffMember, NEW.hotelNumber, NEW.guestNumber, NEW.dateFrom);
END;
```



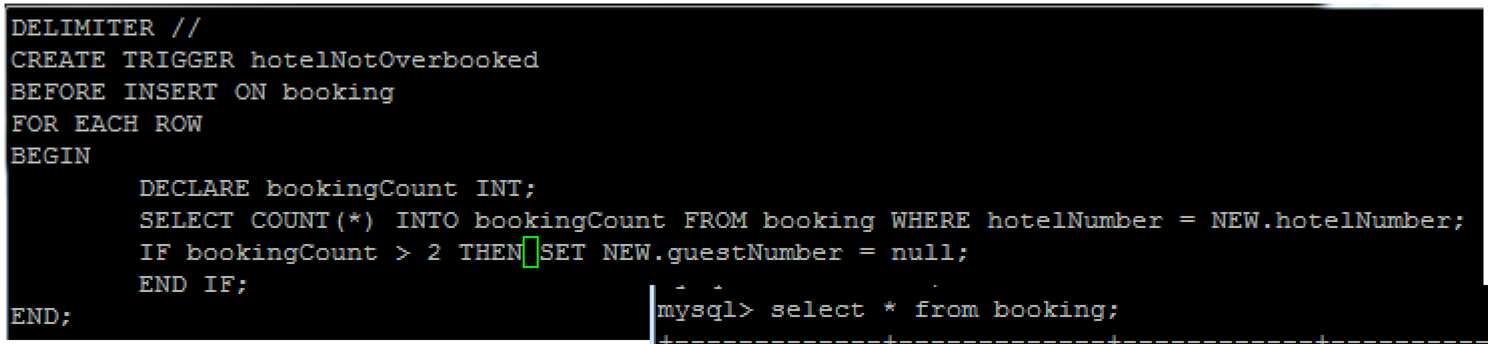
```
mysql> CREATE TABLE bookingAudit(staffMember VARCHAR(20), hotelNumber INT, guestNumber INT, dateFrom DATE, PRIMARY KEY (staffMember,
 hotelNumber,guestNumber,dateFrom));
```

SQL Syntax: An Auditing Trigger

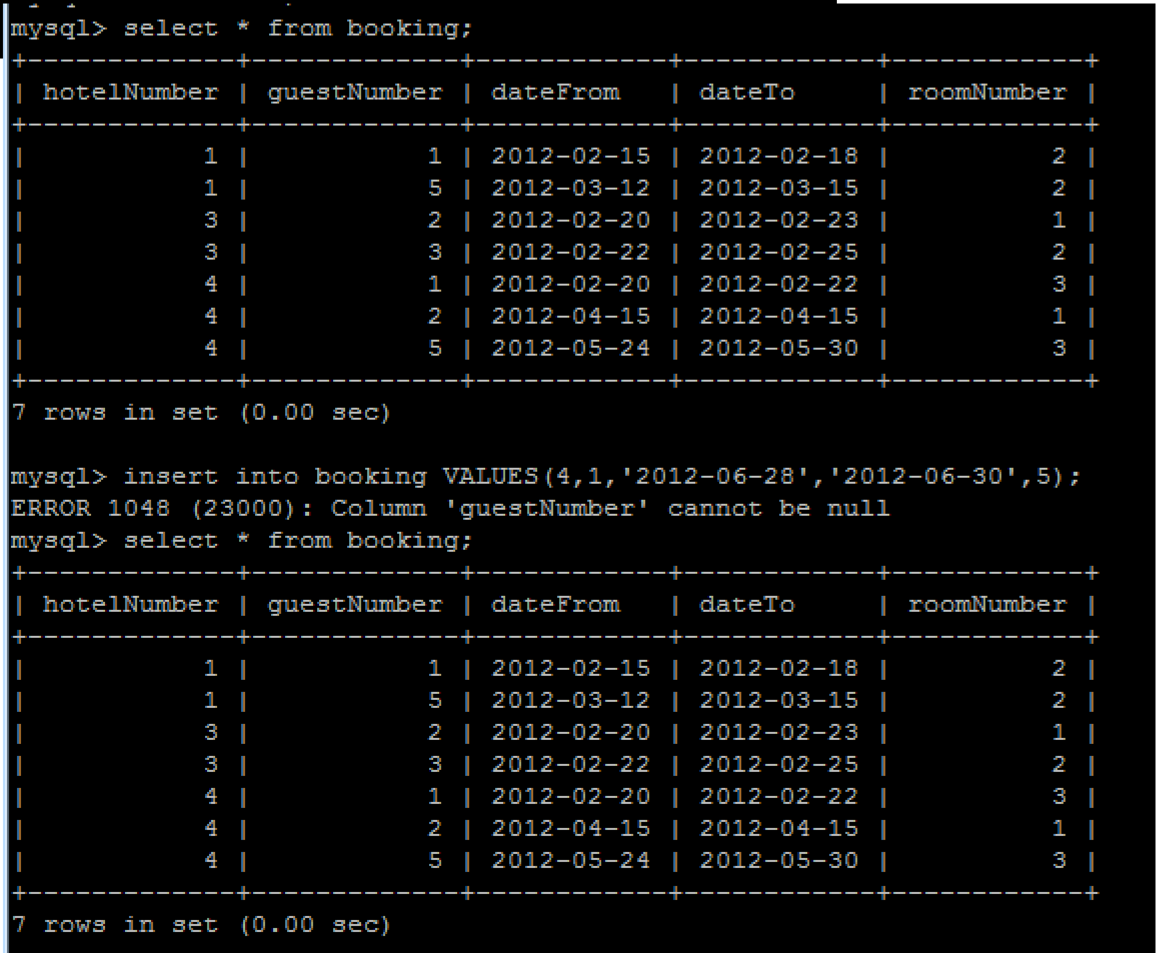


```
mysql> select * from bookingAudit;
    -> //
Empty set (0.00 sec)

mysql> insert into booking VALUES(4,1,'2012-06-28','2012-06-30',5);
    -> //
ERROR 1048 (23000): Column 'guestNumber' cannot be null
mysql> insert into booking VALUES(2,1,'2012-06-28','2012-06-30',5);
    -> //
Query OK, 1 row affected (0.03 sec)

mysql> select * from bookingAudit;
    -> //
+----------------+-------------+-------------+------------+
| staffMember    | hotelNumber | guestNumber | dateFrom   |
+----------------+-------------+-------------+------------+
| root@localhost |           2 |           1 | 2012-06-28 |
+----------------+-------------+-------------+------------+
1 row in set (0.01 sec)

mysql> select * from booking;
    -> //
+-------------+-------------+------------+------------+------------+
| hotelNumber | guestNumber | dateFrom   | dateTo     | roomNumber |
+-------------+-------------+------------+------------+------------+
|           1 |           1 | 2012-02-15 | 2012-02-18 |          2 |
|           1 |           5 | 2012-03-12 | 2012-03-15 |          2 |
|           2 |           1 | 2012-06-28 | 2012-06-30 |          5 |
|           3 |           2 | 2012-02-20 | 2012-02-23 |          1 |
|           3 |           3 | 2012-02-22 | 2012-02-25 |          2 |
|           4 |           1 | 2012-02-20 | 2012-02-22 |          3 |
|           4 |           2 | 2012-04-15 | 2012-04-15 |          1 |
|           4 |           5 | 2012-05-24 | 2012-05-30 |          3 |
+-------------+-------------+------------+------------+------------+
8 rows in set (0.00 sec)
```

SQL Syntax: Checking Trigger Example



```
DELIMITER //
CREATE TRIGGER hotelNotOverbooked
BEFORE INSERT ON booking
FOR EACH ROW
BEGIN
        DECLARE bookingCount INT;
        SELECT COUNT(*) INTO bookingCount FROM booking WHERE hotelNumber = NEW.hotelNumber;
        IF bookingCount > 2 THEN SET NEW.guestNumber = null;
        END IF;
END;
```



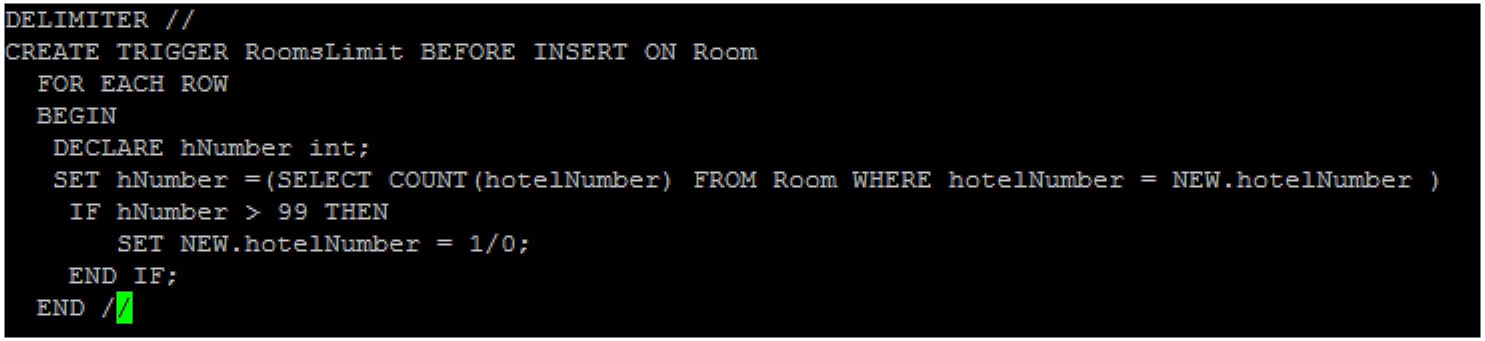
```
mysql> select * from booking;
+-------------+-------------+------------+------------+------------+
| hotelNumber | guestNumber | dateFrom   | dateTo     | roomNumber |
+-------------+-------------+------------+------------+------------+
|           1 |           1 | 2012-02-15 | 2012-02-18 |          2 |
|           1 |           5 | 2012-03-12 | 2012-03-15 |          2 |
|           3 |           2 | 2012-02-20 | 2012-02-23 |          1 |
|           3 |           3 | 2012-02-22 | 2012-02-25 |          2 |
|           4 |           1 | 2012-02-20 | 2012-02-22 |          3 |
|           4 |           2 | 2012-04-15 | 2012-04-15 |          1 |
|           4 |           5 | 2012-05-24 | 2012-05-30 |          3 |
+-------------+-------------+------------+------------+------------+
7 rows in set (0.00 sec)

mysql> insert into booking VALUES(4,1,'2012-06-28','2012-06-30',5);
ERROR 1048 (23000): Column 'guestNumber' cannot be null
mysql> select * from booking;
+-------------+-------------+------------+------------+------------+
| hotelNumber | guestNumber | dateFrom   | dateTo     | roomNumber |
+-------------+-------------+------------+------------+------------+
|           1 |           1 | 2012-02-15 | 2012-02-18 |          2 |
|           1 |           5 | 2012-03-12 | 2012-03-15 |          2 |
|           3 |           2 | 2012-02-20 | 2012-02-23 |          1 |
|           3 |           3 | 2012-02-22 | 2012-02-25 |          2 |
|           4 |           1 | 2012-02-20 | 2012-02-22 |          3 |
|           4 |           2 | 2012-04-15 | 2012-04-15 |          1 |
|           4 |           5 | 2012-05-24 | 2012-05-30 |          3 |
+-------------+-------------+------------+------------+------------+
7 rows in set (0.00 sec)
```

This is very hackish due to limitations of MySQL. MySQL does not have ability to raise programmer specified exceptions, so have to trigger another exception (which may be meaningless in reality)

But it gets the point across!

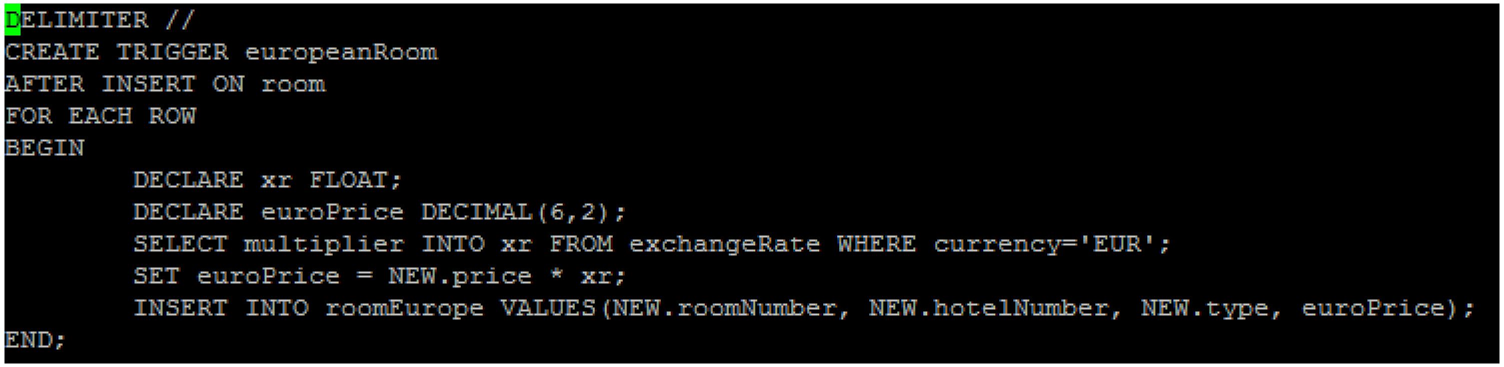
SQL Syntax: Checking Trigger Example

- What will this trigger do? (Courtesy of Percy Campos, Homework 1)

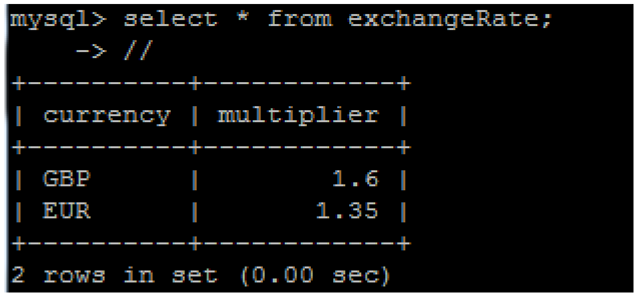


```
DELIMITER //
CREATE TRIGGER RoomsLimit BEFORE INSERT ON Room
  FOR EACH ROW
  BEGIN
   DECLARE hNumber int;
   SET hNumber =(SELECT COUNT(hotelNumber) FROM Room WHERE hotelNumber = NEW.hotelNumber )
    IF hNumber > 99 THEN
       SET NEW.hotelNumber = 1/0;
    END IF;
  END //
```

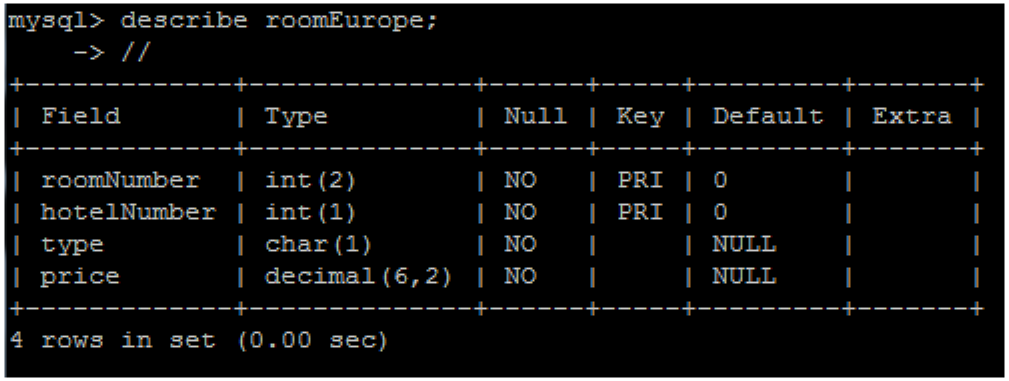
SQL Syntax: Massaging Data Trigger



```
DELIMITER //
CREATE TRIGGER europeanRoom
AFTER INSERT ON room
FOR EACH ROW
BEGIN
        DECLARE xr FLOAT;
        DECLARE euroPrice DECIMAL(6,2);
        SELECT multiplier INTO xr FROM exchangeRate WHERE currency='EUR';
        SET euroPrice = NEW.price * xr;
        INSERT INTO roomEurope VALUES(NEW.roomNumber, NEW.hotelNumber, NEW.type, euroPrice);
END;
```

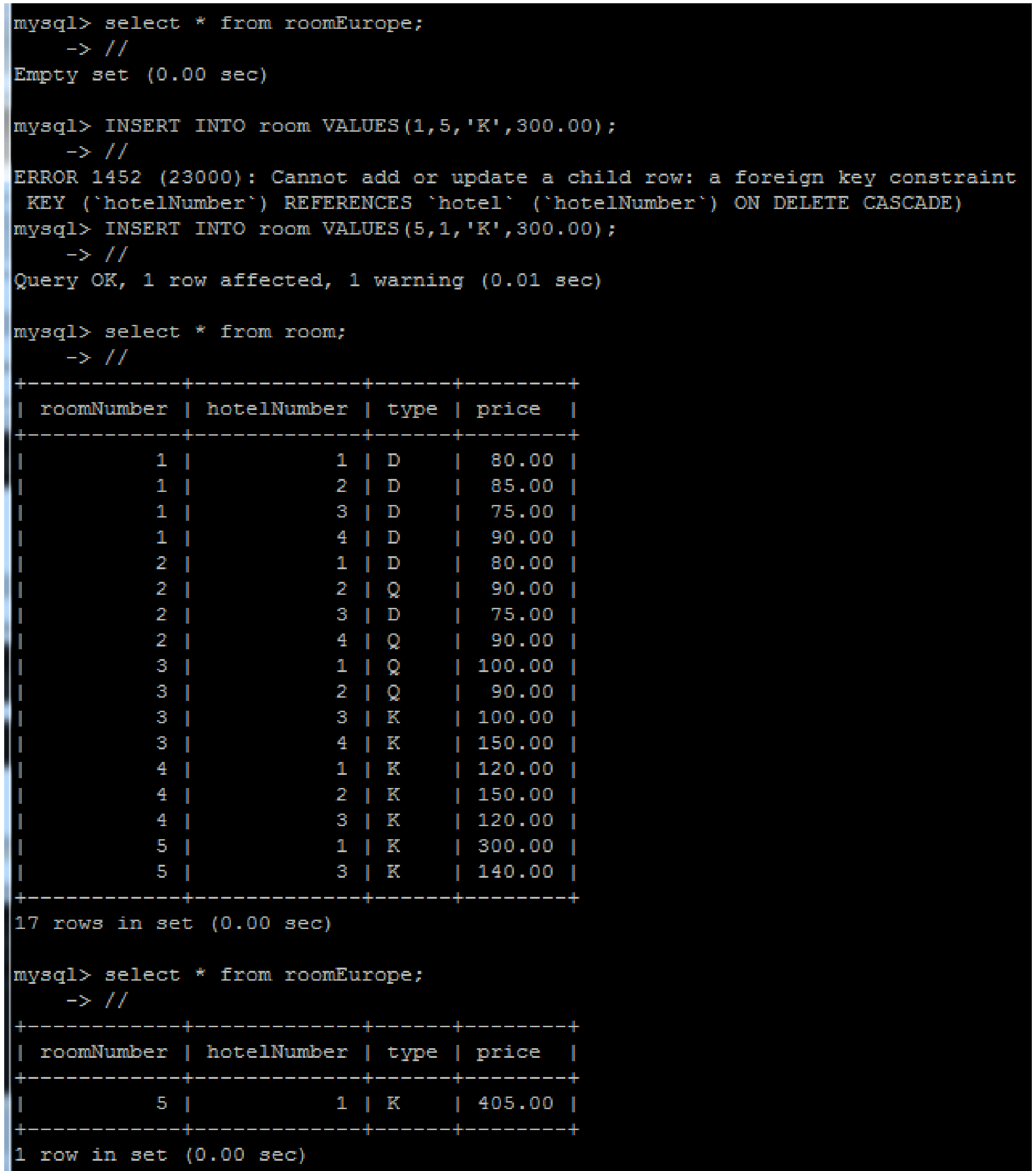


```
mysql> select * from exchangeRate;
    -> //
+----------+------------+
| currency | multiplier |
+----------+------------+
| GBP      |        1.6 |
| EUR      |       1.35 |
+----------+------------+
2 rows in set (0.00 sec)
```



```
mysql> describe roomEurope;
    -> //
+-------------+--------------+------+-----+---------+-------+
| Field       | Type         | Null | Key | Default | Extra |
+-------------+--------------+------+-----+---------+-------+
| roomNumber  | int(2)       | NO   | PRI | 0       |       |
| hotelNumber | int(1)       | NO   | PRI | 0       |       |
| type        | char(1)      | NO   |     | NULL    |       |
| price       | decimal(6,2) | NO   |     | NULL    |       |
+-------------+--------------+------+-----+---------+-------+
4 rows in set (0.00 sec)
```

SQL Syntax: Massaging Data Trigger



```
mysql> select * from roomEurope;
    -> //
Empty set (0.00 sec)

mysql> INSERT INTO room VALUES(1,5,'K',300.00);
    -> //
ERROR 1452 (23000): Cannot add or update a child row: a foreign key constraint
 KEY (`hotelNumber`) REFERENCES `hotel` (`hotelNumber`) ON DELETE CASCADE)
mysql> INSERT INTO room VALUES(5,1,'K',300.00);
    -> //
Query OK, 1 row affected, 1 warning (0.01 sec)

mysql> select * from room;
    -> //
+------------+-------------+------+--------+
| roomNumber | hotelNumber | type | price  |
+------------+-------------+------+--------+
|          1 |           1 | D    |  80.00 |
|          1 |           2 | D    |  85.00 |
|          1 |           3 | D    |  75.00 |
|          1 |           4 | D    |  90.00 |
|          2 |           1 | D    |  80.00 |
|          2 |           2 | Q    |  90.00 |
|          2 |           3 | D    |  75.00 |
|          2 |           4 | Q    |  90.00 |
|          3 |           1 | Q    | 100.00 |
|          3 |           2 | Q    |  90.00 |
|          3 |           3 | K    | 100.00 |
|          3 |           4 | K    | 150.00 |
|          4 |           1 | K    | 120.00 |
|          4 |           2 | K    | 150.00 |
|          4 |           3 | K    | 120.00 |
|          5 |           1 | K    | 300.00 |
|          5 |           3 | K    | 140.00 |
+------------+-------------+------+--------+
17 rows in set (0.00 sec)

mysql> select * from roomEurope;
    -> //
+------------+-------------+------+--------+
| roomNumber | hotelNumber | type | price  |
+------------+-------------+------+--------+
|          5 |           1 | K    | 405.00 |
+------------+-------------+------+--------+
1 row in set (0.00 sec)
```