MySQL Programming

Mimi Opkins & David Brown

CECS 323

General Introduction

- There are several different ways to protect a database from corruption:
  - Datatypes for the individual columns
  - Primary key and other uniqueness constraints
  - Referential integrity constraints:
    - Implement relationships between tables
    - Ensure that enumerated values are valid
    - Implementing reference data
  - Database code that implements complex (non declarative) constraints
- One major benefit to doing all of this in the database is that there is no way to "back door" the database.
- The other benefit is that the stored procedure/function/trigger runs on the server, which saves on network traffic.

Other Approaches

- Alternatively, you could require that all access to a given database (using that term loosely) has to be brokered by an application.
- It's not uncommon to have an application start off "owning" the data, then that data becomes of interest to other applications.
- That means that outside applications coming in through an interface somehow need to use that application so that the business rules are only implemented once.
- Or, the business rules need to be published and agreed to by all users of the data to prevent corruption.

Why MySQL?

- SQL Server has T-SQL, Oracle uses PL/SQL for their programmatic interface to the database.
- The programming concepts are very similar to MySQL.
- We're using MySQL for your term project, and this is a great way to apply the notion of programmatic constraints to the data.
- The syntax, however, will be significantly different from one platform to the next.
- We have to use **some** RDBMS to implement these non-declarative constraints.
- MySQL is a mature, industrial strength RDBMS that you are very likely to run into in your work, particularly now that Oracle has bought it.
- You will see first hand the stark contrast between the constraints that can be captured declaratively in the referential integrity constraints versus those that you have to program.

Before we get started on triggers, here are some handy MySQL features

- The next few slides go over some features that MySQL provides you that will prove useful.
- This is by no means systemic. These are just features that have cropped up in work that has been turned in by other students.
- Bear in mind that many of these are either not implemented at all in other relational database management systems using this syntax, or the underlying functionality might be absent altogether.

Group concat

- Allows you to gather up a given column from a set of rows, and concatenate the value of that column for each row together into a single string.
- This is handy for creating a multi-valued attribute from a junction table.
- Read more about it at: https://dev.mysql.com/doc/refman/5.7/en/aggregate-functions.html#function_group-concat
- Note that you can nest functions, so you could have a complex function and then group concat over that.
- Group concat is a great way to do things like reassemble a repeating value attribute that you have normalized.
  - select cLastName, cFirstName, group_concat(distinct hobby)
  - from contacts inner join hobbies using(contactID)
  - group by contactID;

Extract Function

- Can be used to get at any component of a given date value.
- For instance, in Derby we used the year() function to get the year of a given date value.
- To do that in MySQL, you would do something like select … from orders where extract (year from orderDate) = 2015 …
- To learn more, see: http://www.mysqltutorial.org/mysql-extract/ for a complete explanation. It turns out that there is a wide variety of elements that you can extract from the date in MySQL.

Other date operations

- select datediff("2016-06-15", "2017-06-15");
  - Yields -365
  - Note that you need to use **double** quotes on the date literals.
  - now() yields the current date and time.
- Select date_add("2017-06-15", interval 10 day);
  - Returns a date object that is 10 days after the supplied date.
  - select date_add(now(), interval -10 day); goes back 10 days.
- select "2020-12-31" < now();
  - Returns an integer "1" just like Java.
  - You can use this sort of expression in your SQL where clauses.

First N rows of a query

- If you want a query to only report out the first few rows of a query, use the limit keyword.
- The limit clause is the last thing in your query.
- It takes two arguments or one.
  - If only one argument, that argument is the number of rows to return.
  - If both arguments are supplied, then MySQL returns the second argument (the number of rows) starting at the offset (the first argument.
- Note that this can be used with an order by so that you can get the last few or the first few rows by setting up the proper order by.

Explicit casting

- The convert function (documented at: https://dev.mysql.com/doc/refman/5.7/en/cast-functions.html) allows you to cast an expression or a column value to a different type.
- It also allows you to set the format: convert(sum(service_instance.totalHours)* 50, decimal(10,2)) would provide the result of that calculation with two digits of decimal precision.

Any and some

- We've been using the *in* keyword to match on any value in a collection. Any and some are related:
  - expression comparison_operator ANY (subquery)
  - expression IN (subquery)
  - expression comparison_operator SOME (subquery)
- Any is true if the operand value meets the comparison operator for any of the values in the subquery.
- Some is true if the operand value meets the comparison operator for some of the values in the subquery.
- The comparison operator can be =, >, <, <=, >=, <>, or !=
- The in operator is actually just a special case of = any.
- Some is just an alias for any. I don't make this stuff up, I just report it. Don't shoot the messenger.
- Look at: https://dev.mysql.com/doc/refman/5.7/en/any-in-some-subqueries.html for more information.

The use “command”

- When entering a trigger, function or procedure in at the command prompt, if you preface that with the “use <schema name>;” “command”, then MySQL will store anything created after that in the given schema.
- This makes your DDL simpler since you do not have to fully qualify all your names.

auto_Increment

- It is possible to automatically assign a surrogate key to new rows in a table.
- In MySQL, the mechanics are:

Create table <table name> (<id column name> integer not null auto_increment, …)

- The auto_increment keyword ensures that the next integer value for the id column (whatever you choose to name it) is doled out each time an insert occurs.

Multiple columns in in

- The in Boolean function for most RDBMSs only allows one expression to be searched for from a list. But MySQL allows for you to use more than one expression in a tuple. For example:

```
select    lastName, firstName from student
          natural join Membership natural Join OficeUse
 where    oficeTYpe = 'President' and (lastname, firstName) in
          (select    lastName, firstName
          from       student natural join Membership
                     natural join officeUse
          where      officeType = 'Vice President');
```

Coalesce

- Works exactly like the coalesce function in Derby.
- Coalesce (expr1, expr2, […]) returns the first expression that does **not** evaluate to a null.
- This way, if you have an optional attribute, and you want to put in something to indicate a null, you can essentially report out a default value.
- Remember that the expressions that you pass in as arguments to coalesce can be arbitrarily complex, and include select statements if you need to.

MySQL Enumerations

- MySQL has the enumeration type which looks like:

```
CREATE TABLE shirts (
    name VARCHAR(40),
    size ENUM('x-small', 'small', 'medium', 'large', 'x-large'));
INSERT INTO shirts (name, size) VALUES ('dress shirt','large'), ('t-shirt','medium'),
  ('polo shirt','small');
SELECT name, size FROM shirts WHERE size = 'medium';
```

Problems with the MySQL Enumerations

- Things like state code, gender, titles, are really data, and the enumeration treats it like **meta** data (data about the data).
- Changing the member list is expensive – full table scan of the table using the enumeration.
- It is impossible to add related data. For instance, the full name of the state cannot be added to the state code.
- You have a very difficult time using the enumeration values to populate a drop down list in a GUI control.
- The performance benefits seldom pan out in practice.
- You cannot share that enumeration with other tables.

Shamelessly borrowed from: http://komlenic.com/244/8-reasons-why-mysqls-enum-data-type-is-evil/

MySQL Enumeration Evils (continued)

- MySQL will truncate the incorrect value (unless it's not null) whereas a foreign key constraint would prohibit the entire insert.
- MySQL stores the actual value as an integer lookup, which means that you can inadvertently store an integer as well.
- This particular feature of MySQL is proprietary and will not easily transfer to other DBMSs.
- The larger point of this discussion is when you are looking at a particular feature of a specific RDBMS, ask yourself whether the benefit of that feature outweighs the loss of portability, and whether the feature really serves your needs long term.

Bottom line for MySQL enums:

- Don’t do it!

Alternatives

- Make the enumeration a lookup table as we have talked about in class.
- Use a MySQL check constraint:
  - Create table persons (
    - …,
    - Constraint <constraint name> check (…))
- The expression in () after check can be as complex as needed, it can only reference columns in that table.
- Bottom line, **I will dock you** for using the MySQL enum. I only brief you on it because you are likely to run into in practice, and because I've had students use this in the past and I want to head it off.

The IF() Function in MySQL Select

- IF(exp,exp_result1, exp_result2) will return exp_result1 if the expression evaluates to true, and exp_result2 otherwise.
- One application of the if function is for handling null values: if(state is null, 'N/A', state) will make sure that you have **something** in your report for every record.
- And, you can nest these, so that the first argument to the if could be another if expression if need be.

Ex. Select If(1<2,2,3); (returns 2)

Ex. SELECT IF(STRCMP('test','test1'),'no','yes');

Reverse Engineering using the MSQL workbench

- I strongly encourage you to be very careful to always have an up to date set of scripts to create all your tables, constraints, and perform the inserts that you need to populate your database.
- However, it can be useful at times to have the database generate a script for you based on what you have in the database.
- To do this in MySQL Workbench, select Database | Reverse Engineer. That will prompt you for a connection to the database (even if you already connected in MySQL).
- The wizard will prompt you through and eventually generate a script for you. Just bear in mind that it will use the `quotes` around all your table and column names, which means that if you use that script to create everything, you will have to use the `quotes` as well.

Basic Programming Structures

- Stored Procedures
  - Blocks of code stored in the database that are pre-compiled.
  - They can operate on the tables within the database and return scalars or results sets.
- Functions
  - Can be used like a built-in function to provide expanded capability to your SQL statements.
  - They can take any number of arguments and return a single value.
- Triggers
  - Kick off in response to standard database operations on a specified table.
  - Can be used to automatically perform additional database operations when the triggering event occurs.

Basic Programming Structures Reference

- None of this is original, look at it as a digest from: http://dev.mysql.com/doc/.
- More specifically, we'll be talking about material found in: https://dev.mysql.com/doc/refman/5.7/en/sql-compound-statements.html.

Stored Procedures in MySQL

- A stored procedure contains a sequence of SQL commands stored in the database catalog so that it can be invoked later by a program
- Stored procedures are declared using the following syntax:

Create Procedure <proc-name>

(param_spec$_1$, param_spec$_2$, …, param_spec$_n$)

begin

-- execution code

end;

where each param_spec is of the form:

[in | out | inout] <param_name> <param_type>

- in mode: allows you to pass values into the procedure,
- out mode: allows you to pass value back from procedure to the calling program

More about Stored Procedures

- You can declare variables in stored procedures
- Can have any number of parameters.
- Each parameter must specify whether it's in, out, or inout.
  - The typical argument list will look like

  (**out** ver_param varchar(25), **inout** incr_param int …)
  - Be careful of output parameters for side effects.
- Your varchar declarations for the parameters must specify the maximum length.
- The individual parameters can have any supported MySQL datatype.
- They can be called using the call command, followed by the procedure name, and the arguments.
- You can use flow control statements (conditional IF-THEN-ELSE or loops such as WHILE and REPEAT)

Conditions and Handlers

- A condition is somewhat like an exception.
  - You can declare your own conditions, but we're not going to get into that for the purposes of this course.
- A handler is somewhat like the catch block in a try/catch construct.
  - The "canned" conditions that MySQL has will prove to be enough for our purposes.
  - We should be able to get by with just a few conditions, we'll see as we go along.

IF

- Note that <condition> is a generic Boolean expression, not a condition in the MySQL sense of the word.

IF <condition> then

    <statements>

ELSEIF <condition> then

    <statements>

ELSE

    <statements>

END IF

- Note the annoying syntax: END IF has an embedded blank, ELSEIF does not.
- There can be any number of ELSIF clauses in your IF statement.

Case Statement

- Two different syntaxes:

```
CASE <expression>
    WHEN <value> then
        <statements>
    WHEN <value> then
        <statements>
    …
    ELSE
        <statements>
END CASE;
```

CASE Statement (Continued)

```
CASE
    WHEN <condition> then
        <statements>
    WHEN <condition> then
        <statements>
    …
    ELSE
        <statements>
END CASE;
```

Looping

- [begin_label:] LOOP
  - <statement list>
- END LOOP [end_label]
- Note that the end_label has to = the begin_label
- Both are optional
- [begin_label:] REPEAT
  - <statement list>
- UNTIL <search_condition>
- END REPEAT [end_label]

Repeat Until Example

```
DELIMITER //
CREATE FUNCTION CalcIncome ( starting_value INT )
RETURNS INT
BEGIN
    DECLARE income INT;
    SET income = 0;
    label1: REPEAT
        SET income = income + starting_value;
        UNTIL income >= 4000
    END REPEAT label1;
    RETURN income;
END; //
DELIMITER ;
```

Notes on the previous example

- The DELIMITER // statement sets a session variable so that the // becomes the statement terminator.
- For the purposes of that session, the ";" within the stored procedure are just like any other character.
- When the stored procedure is run, however, the ";" function the way that they normally do in MySQL.
- You always want to make the delimiter a ";" again when you change it.

While

- [begin_label:] WHILE <condition> DO
  - <statements>
- END WHILE [end_label]

Loop Control Flow

- Iterate <label> – start the loop again
  - Can only be issued within LOOP, REPEAT, or WHILE statements
  - Works much like the "continue" statement in Java or C++.
- Leave <label> – jumps out of the control construct that has the given label.
  - Can only be issued within LOOP, REPEAT, or WHILE statements, just like the iterate statement.
  - You can use this at any level of nesting, → you can jump out to the out of the outermost loop if you desire.

Example

```
mysql> select * from employee;
+----+-------+---------+--------+------------+------+
| id | name  | superid | salary | bdate      | dno  |
+----+-------+---------+--------+------------+------+
|  1 | john  |       3 | 100000 | 1960-01-01 |    1 |
|  2 | mary  |       3 |  50000 | 1964-12-01 |    3 |
|  3 | bob   |    NULL |  80000 | 1974-02-07 |    3 |
|  4 | tom   |       1 |  50000 | 1978-01-17 |    2 |
|  5 | bill  |    NULL |   NULL | 1985-01-20 |    1 |
+----+-------+---------+--------+------------+------+
```

```
mysql> select * from department;
+---------+------------+
| dnumber | dname      |
+---------+------------+
|       1 | Payroll    |
|       2 | TechSupport |
|       3 | Research   |
+---------+------------+
```

- Suppose we want to keep track of the total salaries of employees working for each department

```
mysql> create table deptsal as
    -> select dnumber, 0 as totalsalary from department;
Query OK, 3 rows affected (0.00 sec)
Records: 3  Duplicates: 0  Warnings: 0

mysql> select * from deptsal;
+---------+-------------+
| dnumber | totalsalary |
+---------+-------------+
|       1 |           0 |
|       2 |           0 |
|       3 |           0 |
+---------+-------------+
```

We need to write a procedure to update the salaries in the deptsal table

Example – Step 1

Step 1: Change the delimiter (i.e., terminating character) of SQL statement from semicolon (;) to something else (e.g., //) So that you can distinguish between the semicolon of the SQL statements in the procedure and the terminating character of the procedure definition

```
mysql> delimiter //
```

Example – Step 2

Step 2:

1. Define a procedure called updateSalary which takes as input a department number.
2. The body of the procedure is an SQL command to update the totalsalary column of the deptsal table.
3. Terminate the procedure definition using the delimiter you had defined in step 1 (//)

```
mysql> delimiter //
mysql> create procedure updateSalary (IN param1 int)
    -> begin
    ->     update deptsal
    ->     set totalsalary = (select sum(salary) from employee where dno = param1)
    ->     where dnumber = param1;
    -> end; //
Query OK, 0 rows affected (0.01 sec)
```

Example – Step 3

Step 3: Change the delimiter back to semicolon (;)

```
mysql> delimiter //
mysql> create procedure updateSalary (IN param1 int)
    -> begin
    ->     update deptsal
    ->     set totalsalary = (select sum(salary) from employee where dno = param1)
    ->     where dnumber = param1;
    -> end; //
Query OK, 0 rows affected (0.01 sec)

mysql> delimiter ;
```

Example – Step 4

Step 4: Call the procedure to update the totalsalary for each department

```
mysql> call updateSalary(1);
Query OK, 0 rows affected (0.00 sec)

mysql> call updateSalary(2);
Query OK, 1 row affected (0.00 sec)

mysql> call updateSalary(3);
Query OK, 1 row affected (0.00 sec)
```

Example – Step 5

Step 5: Show the updated total salary in the deptsal table

```
mysql> select * from deptsal;
+---------+-------------+
| dnumber | totalsalary |
+---------+-------------+
|       1 |      100000 |
|       2 |       50000 |
|       3 |      130000 |
+---------+-------------+
3 rows in set (0.00 sec)
```

Stored Procedures in MySQL

- Use show procedure status to display the list of stored procedures you have created

```
mysql> show procedure status;
+------+---------------+-----------+---------+---------------------+---------------------+---------
-----+---------+----------------------+----------------------+--------------------+
| Db   | Name          | Type      | Definer | Modified            | Created             | Security_
type | Comment | character_set_client | collation_connection | Database Collation |
+------+---------------+-----------+---------+---------------------+---------------------+---------
-----+---------+----------------------+----------------------+--------------------+
| ptan | updateSalary0 | PROCEDURE | ptan@%  | 2010-03-16 12:21:55 | 2010-03-16 12:21:55 | DEFINER
     |         | latin1               | latin1_swedish_ci    | latin1_swedish_ci  |
+------+---------------+-----------+---------+---------------------+---------------------+---------
-----+---------+----------------------+----------------------+--------------------+
1 row in set (0.02 sec)
```

- Use drop procedure to remove a stored procedure

```
mysql> drop procedure updateSalary;
Query OK, 0 rows affected (0.00 sec)
```

Debugging your stored procedures

- Using the select statement
  - SELECT ‘Comment’; -- Put the literal Comment out to console
  - SELECT concat(‘myvar is ‘, myvar); -- Put the literal prompt out, followed by the current value of a variable named myvar.
    - Note, you cannot do this in a function as that is regarded as returning a result set.
- Insert into a table. Putting the current time and date stamp into a column with the message would be good too.
- Log messages to an output file: select … into outfile ‘<file_name>’;
  - Which might be blocked by the secure-file-priv option in MySQL.

Stored Procedures in MySQL

- MySQL also supports cursors in stored procedures.
  - A cursor is used to iterate through a set of rows returned by a query so that we can process each individual row.
- To learn more about stored procedures, go to:

  http://www.mysqltutorial.org/mysql-stored-procedure-tutorial.aspx

Example using Cursors

- The previous procedure updates one row in deptsal table based on input parameter
- Suppose we want to update all the rows in deptsal simultaneously
  - First, let's reset the totalsalary in deptsal to zero

```
mysql> update deptsal set totalsalary = 0;
Query OK, 0 rows affected (0.00 sec)
Rows matched: 3  Changed: 0  Warnings: 0

mysql> select * from deptsal;
+---------+-------------+
| dnumber | totalsalary |
+---------+-------------+
|       1 |           0 |
|       2 |           0 |
|       3 |           0 |
+---------+-------------+
3 rows in set (0.00 sec)
```

Example using Cursors – Part 2

```
mysql> delimiter $$
mysql> drop procedure if exists updateSalary$$
Query OK, 0 rows affected (0.00 sec)

mysql> create procedure updateSalary()
    -> begin
    ->          declare done int default 0;
    ->          declare current_dnum int;
    ->          declare dnumcur cursor for select dnumber from deptsal;
    ->          declare continue handler for not found set done = 1;
    ->
    ->          open dnumcur;
    ->
    ->          repeat
    ->                fetch dnumcur into current_dnum;
    ->                update deptsal
    ->                set totalsalary = (select sum(salary) from employee
    ->                                   where dno = current_dnum)
    ->                where dnumber = current_dnum;
    ->          until done
    ->          end repeat;
    ->
    ->          close dnumcur;
    -> end$$
Query OK, 0 rows affected (0.00 sec)

mysql> delimiter ;
```

Drop the old procedure

Use cursor to iterate the rows

Example using Cursors – Part 3

- Call procedure

```
mysql> select * from deptsal;
+---------+-------------+
| dnumber | totalsalary |
+---------+-------------+
|       1 |           0 |
|       2 |           0 |
|       3 |           0 |
+---------+-------------+
3 rows in set (0.01 sec)

mysql> call updateSalary;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from deptsal;
+---------+-------------+
| dnumber | totalsalary |
+---------+-------------+
|       1 |      100000 |
|       2 |       50000 |
|       3 |      130000 |
+---------+-------------+
3 rows in set (0.00 sec)
```

Another Example

- Create a procedure to give a raise to all employees

```
mysql> select * from emp;
+----+--------+---------+--------+------------+------+
| id | name   | superid | salary | bdate      | dno  |
+----+--------+---------+--------+------------+------+
|  1 | john   |       3 | 100000 | 1960-01-01 |    1 |
|  2 | mary   |       3 |  50000 | 1964-12-01 |    3 |
|  3 | bob    |    NULL |  80000 | 1974-02-07 |    3 |
|  4 | tom    |       1 |  50000 | 1978-01-17 |    2 |
|  5 | bill   |    NULL |   NULL | 1985-01-20 |    1 |
|  6 | lucy   |    NULL |  90000 | 1981-01-01 |    1 |
|  7 | george |    NULL |  45000 | 1971-11-11 | NULL |
+----+--------+---------+--------+------------+------+
7 rows in set (0.00 sec)
```

Another Example – Part 2

```
mysql> delimiter |
mysql> create procedure giveRaise (in amount double)
    -> begin
    ->         declare done int default 0;
    ->         declare eid int;
    ->         declare sal int;
    ->         declare emprec cursor for select id, salary from employee;
    ->         declare continue handler for not found set done = 1;
    ->
    ->         open emprec;
    ->         repeat
    ->                 fetch emprec into eid, sal;
    ->                 update employee
    ->                 set salary = sal + round(sal * amount)
    ->                 where id = eid;
    ->         until done
    ->         end repeat;
    -> end |
Query OK, 0 rows affected (0.00 sec)
```

Another Example – Part 3

```
mysql> delimiter ;
mysql> call giveRaise(0.1);
Query OK, 0 rows affected (0.00 sec)

mysql> select * from employee;
+----+-------+---------+--------+------------+------+
| id | name  | superid | salary | bdate      | dno  |
+----+-------+---------+--------+------------+------+
|  1 | john  |       3 | 110000 | 1960-01-01 |    1 |
|  2 | mary  |       3 |  55000 | 1964-12-01 |    3 |
|  3 | bob   |    NULL |  88000 | 1974-02-07 |    3 |
|  4 | tom   |       1 |  55000 | 1978-01-17 |    2 |
|  5 | bill  |    NULL |   NULL | 1985-01-20 |    1 |
+----+-------+---------+--------+------------+------+
5 rows in set (0.00 sec)
```

Functions

- Your user-defined functions can act just like a function defined in the database.
- They take arguments and return a single output.
- The general syntax is: create function <name> (<arg1> <type1>, [<arg2> <type2> [,…]) returns <return type> [deterministic]
  - Deterministic means that the output from the function is strictly a consequence of the arguments.
  - Same values input → same values output.
  - Like a static method in Java.
  - Note that the arguments cannot be changed and the new values passed back to the caller.
- Follow that with begin … end and you have a function.

Functions

- You need ADMIN privilege to create functions on mysql-user server
- Functions are declared using the following syntax:

function <function-name> (param_spec$_1$, …, param_spec$_k$)

returns <return_type>

[not] deterministic allow optimization if same output for the same input (use RAND not deterministic)

Begin

-- execution code

end;

where param_spec is:

[in | out | in out] <param_name> <param_type>

Example of Functions

```
mysql> select * from employee;
+----+------+----------+--------+------------+------+
| id | name | superid  | salary | bdate      | dno  |
+----+------+----------+--------+------------+------+
|  1 | john |        3 | 100000 | 1960-01-01 |    1 |
|  2 | mary |        3 |  50000 | 1964-12-01 |    3 |
|  3 | bob  |     NULL |  80000 | 1974-02-07 |    3 |
|  4 | tom  |        1 |  50000 | 1970-01-17 |    2 |
|  5 | bill |     NULL |   NULL | 1985-01-20 |    1 |
+----+------+----------+--------+------------+------+
5 rows in set (0.00 sec)

mysql> deliniter |
mysql> create function giveRaise (oldval double, amount double
    -> returns double
    -> deterministic
    -> begin
    ->       declare newval double;
    ->       set newval = oldval * (1 + amount);
    ->       return newval;
    -> end |
Query OK, 0 rows affected (0.00 sec)

mysql> deliniter ;
```

Another Example of Functions

```
mysql> select name, salary, giveRaise(salary, 0.1) as newsal
    -> from employee;
+------+--------+--------+
| name | salary | newsal |
+------+--------+--------+
| john | 100000 | 110000 |
| mary |  50000 |  55000 |
| bob  |  80000 |  88000 |
| tom  |  50000 |  55000 |
| bill |   NULL |   NULL |
+------+--------+--------+
5 rows in set (0.00 sec)
```

SQL Triggers

- To monitor a database and take a corrective action when a condition occurs
  - Examples:
    - Charge $10 overdraft fee if the balance of an account after a withdrawal transaction is less than $500
    - Limit the salary increase of an employee to no more than 5% raise

```
CREATE TRIGGER trigger-name
 trigger-time trigger-event
 ON table-name
 FOR EACH ROW
    trigger-action;
```

trigger-time ∈ {BEFORE, AFTER}

trigger-event ∈ {INSERT,DELETE,UPDATE}

Triggers

- Please see: http://dev.mysql.com/doc/refman/5.7/en/create-trigger.html for the complete specification for triggers.

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

trigger_time: { BEFORE | AFTER }

trigger_event: { INSERT | UPDATE | DELETE }

trigger_order: { FOLLOWS | PRECEDES } other_trigger_name
```

SQL Triggers: An Example

- We want to create a trigger to update the total salary of a department when a new employee is hired

```
mysql> select * from employee;
+----+------+---------+--------+------------+------+
| id | name | superid | salary | bdate      | dno  |
+----+------+---------+--------+------------+------+
|  1 | john |       3 | 100000 | 1960-01-01 |    1 |
|  2 | mary |       3 |  50000 | 1964-12-01 |    3 |
|  3 | bob  |    NULL |  80000 | 1974-02-07 |    3 |
|  4 | tom  |       1 |  50000 | 1970-01-17 |    2 |
|  5 | bill |    NULL |   NULL | 1985-01-20 |    1 |
+----+------+---------+--------+------------+------+
5 rows in set (0.00 sec)

mysql> select * from deptsal;
+---------+-------------+
| dnumber | totalsalary |
+---------+-------------+
|       1 |      100000 |
|       2 |       50000 |
|       3 |      130000 |
+---------+-------------+
3 rows in set (0.00 sec)
```

SQL Triggers: Another Example

- Create a trigger to update the total salary of a department when a new employee is hired:

```
mysql> delimiter !
mysql> create trigger update_salary
    -> after insert on employee
    -> for each row
    -> begin
    ->       if new.dno is not null then
    ->           update deptsal
    ->           set totalsalary = totalsalary + new.salary
    ->           where dnumber = new.dno;
    ->       end if;
    -> end !
Query OK, 0 rows affected (0.06 sec)

mysql> delimiter ;
```

- The keyword “new” refers to the new row inserted

SQL Triggers: Another Example – Part 2

```
mysql> select * from deptsal;
+---------+-------------+
| dnumber | totalsalary |
+---------+-------------+
|       1 |      100000 |
|       2 |       50000 |
|       3 |      130000 |
+---------+-------------+
3 rows in set (0.00 sec)

mysql> insert into employee values (6,'lucy',null,90000,'1981-01-01',1);
Query OK, 1 row affected (0.08 sec)

mysql> select * from deptsal;
+---------+-------------+
| dnumber | totalsalary |
+---------+-------------+
|       1 |      190000 |
|       2 |       50000 |
|       3 |      130000 |
+---------+-------------+
3 rows in set (0.00 sec)

mysql> insert into employee values (7,'george',null,45000,'1971-11-11',null);
Query OK, 1 row affected (0.02 sec)

mysql> select * from deptsal;
+---------+-------------+
| dnumber | totalsalary |
+---------+-------------+
|       1 |      190000 |
|       2 |       50000 |
|       3 |      130000 |
+---------+-------------+
3 rows in set (0.00 sec)

mysql> drop trigger update_salary;
Query OK, 0 rows affected (0.00 sec)
```

totalsalary increases by 90K

totalsalary did not change

SQL Triggers: Another Example – Part 3

- A trigger to update the total salary of a department when an employee tuple is modified:

```
mysql> delimiter !
mysql> create trigger update_salary2
    -> after update on employee
    -> for each row
    -> begin
    ->       if old.dno is not null then
    ->          update deptsal
    ->          set totalsalary = totalsalary - old.salary
    ->          where dnumber = old.dno;
    ->       end if;
    ->       if new.dno is not null then
    ->          update deptsal
    ->          set totalsalary = totalsalary + new.salary
    ->          where dnumber = new.dno;
    ->       end if;
    -> end !
Query OK, 0 rows affected (0.06 sec)
```

SQL Triggers: An Example – Part 4

```
mysql> delimiter ;
mysql> select * from employee;
+----+--------+---------+--------+------------+------+
| id | name   | superid | salary | bdate      | dno  |
+----+--------+---------+--------+------------+------+
|  1 | john   |       3 | 100000 | 1960-01-01 |    1 |
|  2 | mary   |       3 |  50000 | 1964-12-01 |    3 |
|  3 | bob    |    NULL |  80000 | 1974-02-07 |    3 |
|  4 | tom    |       1 |  50000 | 1970-01-17 |    2 |
|  5 | bill   |    NULL |   NULL | 1985-01-20 |    1 |
|  6 | lucy   |    NULL |  90000 | 1981-01-01 |    1 |
|  7 | george |    NULL |  45000 | 1971-11-11 | NULL |
+----+--------+---------+--------+------------+------+
7 rows in set (0.00 sec)

mysql> select * from deptsal;
+---------+-------------+
| dnumber | totalsalary |
+---------+-------------+
|       1 |      190000 |
|       2 |       50000 |
|       3 |      130000 |
+---------+-------------+
3 rows in set (0.00 sec)

mysql> update employee set salary = 100000 where id = 6;
Query OK, 1 row affected (0.03 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> select * from deptsal;
+---------+-------------+
| dnumber | totalsalary |
+---------+-------------+
|       1 |      200000 |
|       2 |       50000 |
|       3 |      130000 |
+---------+-------------+
3 rows in set (0.00 sec)
```

SQL Triggers: Another Example – Part 5

- A trigger to update the total salary of a department when an employee tuple is deleted:

```
mysql> delimiter !
mysql> create trigger update_salary3
    -> before delete on employee
    -> for each row
    -> begin
    ->       if (old.dno is not null) then
    ->           update deptsal
    ->           set totalsalary = totalsalary - old.salary
    ->           where dnumber = old.dno;
    ->       end if;
    -> end !
Query OK, 0 rows affected (0.08 sec)

mysql> delimiter ;
```

SQL Triggers: Another Example – Part 6

```
mysql> select * from employee;
+----+--------+---------+--------+------------+------+
| id | name   | superid | salary | bdate      | dno  |
+----+--------+---------+--------+------------+------+
|  1 | john   |       3 | 100000 | 1960-01-01 |    1 |
|  2 | mary   |       3 |  50000 | 1964-12-01 |    3 |
|  3 | bob    |    NULL |  80000 | 1974-02-07 |    3 |
|  4 | tom    |       1 |  50000 | 1970-01-17 |    2 |
|  5 | bill   |    NULL |   NULL | 1985-01-20 |    1 |
|  6 | lucy   |    NULL | 100000 | 1981-01-01 |    1 |
|  7 | george |    NULL |  45000 | 1971-11-11 | NULL |
+----+--------+---------+--------+------------+------+
7 rows in set (0.00 sec)
```

```
mysql> select * from deptsal;
+---------+-------------+
| dnumber | totalsalary |
+---------+-------------+
|       1 |      200000 |
|       2 |       50000 |
|       3 |      130000 |
+---------+-------------+
3 rows in set (0.00 sec)
```

```
mysql> delete from employee where id = 6;
Query OK, 1 row affected (0.02 sec)

mysql> delete from employee where id = 7;
Query OK, 1 row affected (0.03 sec)

mysql> select * from deptsal;
+---------+-------------+
| dnumber | totalsalary |
+---------+-------------+
|       1 |      100000 |
|       2 |       50000 |
|       3 |      130000 |
+---------+-------------+
3 rows in set (0.00 sec)
```

A Few Things to Note

- A given trigger can only have one event.
- If you have the same or similar processing that has to go on during insert and delete, then it's best to have that in a procedure or function and then call it from the trigger.
- A good naming standard for a trigger is <table_name>_event if you have the room for that in the name.
- Just like a function or a procedure, the trigger body will need a begin … end unless it is a single statement trigger.

The Special Powers of a Trigger

- While in the body of a trigger, there are potentially two sets of column values available to you, with special syntax for denoting them.
  - old.<column name> will give you the value of the column before the DML statement executed.
  - new.<column name> will give you the value of that column **after** the DML statement executed.
- Insert triggers have no old values available, and delete triggers have no new values available for obvious reasons. Only update triggers have both the old and the new values available.
- Only triggers can access these values this way.

Changing columns in a trigger

- In the body of a trigger, it is possible to change the values for the columns in the current row.
- Just use the “set” verb to change them.
- You can only do this for an update or insert trigger.
- You can only change the values of new.<column name> since there is no point to changing the old values.

More Examples

- Simplified example of a parent table: hospital_room as the parent and hospital_bed as the child.
- The room has a column: max_beds that dictates the maximum number of beds for that room.
- The hospital_bed table has a before insert trigger that checks to make sure that the hospital room does not already have its allotted number of beds.

The Trigger

```
CREATE DEFINER=`root`@`localhost`
TRIGGER `programming`.`hospital_bed_BEFORE_INSERT`
BEFORE INSERT ON `hospital_bed` FOR EACH ROW
BEGIN
	declare max_beds_per_room int;
	declare current_count int;
	select	max_beds into max_beds_per_room
	from	hospital_room
	where	hospital_room_no = new.room_id;
	select	count(*) into current_count
	from	hospital_bed
	where	room_id = new.room_id;
	if current_count >= max_beds_per_room then
		signal sqlstate '45000' set message_text='Too many beds in that room already!';
	end if;
END;
```

Firing the trigger

```
insertinto hospital_bed (room_id, hospital_bed_id)
values   ('323B', 1);
insertinto hospital_bed (room_id, hospital_bed_id)
values   ('323B', 2);
insertinto hospital_bed (room_id, hospital_bed_id)
values   ('323B', 3);
insertinto hospital_bed (room_id, hospital_bed_id)
values   ('323B', 4);
insertinto hospital_bed (room_id, hospital_bed_id)
values   ('323B', 5);
```

Error Code: 1644. Too many beds in that room already!

Using a Stored Procedure Instead

```
CREATE DEFINER=`root`@`localhost` PROCEDURE `too_many_beds`(in room_id varchar(45))
BEGIN
	declare max_beds_per_room int;
	declare current_count int;
	declare room_count int;
	-- see if the hospital room exists
	select	count(*) into room_count
	from hospital_room
	where	hospital_room_no = room_id;
	if room_count = 1 then -- we can see if room for 1 more bed
	begin
		select	max_beds into max_beds_per_room
		from hospital_room
		where	hospital_room_no = room_id;
		-- count the beds in this room
		select	count(*) into current_count
		from hospital_bed
		where	room_id = room_id;
		if current_count >= max_beds_per_room then
			-- flag an error to abort if necessary
			signal sqlstate '45000' set message_text='Too many beds in that room already!';
		end if;
	end;
	end if;
END
```

The new & improved trigger

```
CREATE DEFINER=`root`@`localhost`
TRIGGER
`programming`.`hospital_bed_BEFORE_INSERT`
BEFORE INSERT ON `hospital_bed` FOR EACH
ROW
BEGIN
    call `too many beds`(new.room_id);
END;
```

Comments on the Procedure

- Because that is in isolation from the beds table, we have to check to make sure that the room number is viable.
- As a stored procedure, this can be called directly from the command line as a means of unit testing.
- I'm still not too sure how exacting the typing of the parameters has to be. For instance, does that one argument have to be exactly a varchar(45) in order for it to work, or not?

Viewing Your Triggers

- MySQL has a schema that has tables for all of the information that is needed to define and run the data in the database. This is meta data.
- select * from information_schema.triggers where trigger_schema='<your schema name>'; -- retrieve the trigger information for the triggers in <your schema name>.
- Alternatively, you can use the "show triggers" command (this is not SQL) that will display a report of your triggers from the default schema.

  mysql> show triggers;

Viewing Your Triggers (Continued)

- If you're using MySQL Workbench, the IDE provides access to your triggers:
  - In the navigator pane, right click the table that has the trigger.
  - Select "Alter Table"
  - This will open up a rather lavish dialog which has tabs down near the bottom. One of those tabs is "Triggers". Select that.
  - That will open up **another** dialog, and over to the left will be the list of events that you can define triggers for.
    - At this point, you can right click one of those events and it will pop up a menu that will give you the option to create a **new** trigger for that event.
    - Or you can double click an existing trigger to get into an editor on that particular trigger. This will allow you to update the trigger in place as it were, rather than drop and recreate it.

Dynamic SQL

- Sometimes you need to operate against a table or columns that are not known at compile time. MySQL has a process using set, prepare, execute, and deallocate.

```
CREATE DEFINER=`root`@`localhost` PROCEDURE `dynamic`(in tableName varchar(40))
begin
	set @statement = concat('select * from ', tableName);
	prepare stmt from @statement;
	execute stmt;
	set @statement = concat('select count(*) from ', tableName, ' into @count');
	prepare stmt from @statement;
	execute stmt;
	select concat('Count was: ', @count, ' from table: ', tableName);
	deallocate prepare stmt;
end
```

Dynamic SQL (Continued)

- The @ in front of a name makes it a user variable, which is shared between the command session and the stored procedure.
- concat will take any number of arguments.
- Just like the Java API, you can have bind variables in the SQL that you submit, then use the using clause in the execute statement.
  - The bind variables have to map one for one to the variables in the using clause: execute stmt using @var1, @var2, …

Configuration Management

- Remember that all objects that you create, be they tables, indexes, constraints, triggers, stored procedures, functions, … exist in the database.
- The data dictionary has the definition of these objects, and generally speaking, you can use utilities (either in the IDE or the RDBMS) to reverse engineer those objects.
- MySQL Workbench has a way to reverse engineer an entire schema. Read about it at: https://dev.mysql.com/doc/workbench/en/wb-reverse-engineer-live.html.

Credits

Presentation taken from:

- *www.cse.msu.edu/~pramanik/teaching/courses/cse480/14s/lectures/12/lecture13.ppt* by Sakti Pramanik at Michigan State University
- MySQL Procedural Language by David Brown at California State University Long Beach