



# STORED PROCEDURES

**Credit - Jason Carter**

# STORED PROCEDURES

- Database program modules that are stored and executed by the DBMS at the server

```
DELIMITER //  
CREATE PROCEDURE GetAllProducts()  
BEGIN  
    SELECT * FROM products;  
END //  
DELIMITER ;
```



# WHY STORED PROCEDURES

- Reduces Duplication of effort and improves software modularity
  - Multiple applications can use the stored procedure vs. the SQL statements being stored in the application language (PHP)

Reduces communication and data transfer cost between client and server (in certain situations)

- Instead of sending multiple lengthy SQL statements, the application only has to send the name and parameters of the Stored Procedure

Can be more secure than SQL statements

- Permission can be granted to certain stored procedures without granting access to database tables



# DISADVANTAGES OF STORED PROCEDURES

- Difficult to debug

- MySQL does not provide ways for debugging stored procedures

Many stored procedures can increase memory use

- The more stored procedures you use, the more memory is used

Can be difficult to maintain and develop stored procedures

- Another programming language to learn



# CREATING STORED PROCEDURES

```
DELIMITER //
```

```
CREATE PROCEDURE NAME
```

```
BEGIN
```

```
    SQL STATEMENT
```

```
END //
```

```
DELIMITER ;
```

```
DELIMITER //
```

```
CREATE PROCEDURE GetAllProducts()
```

```
BEGIN
```

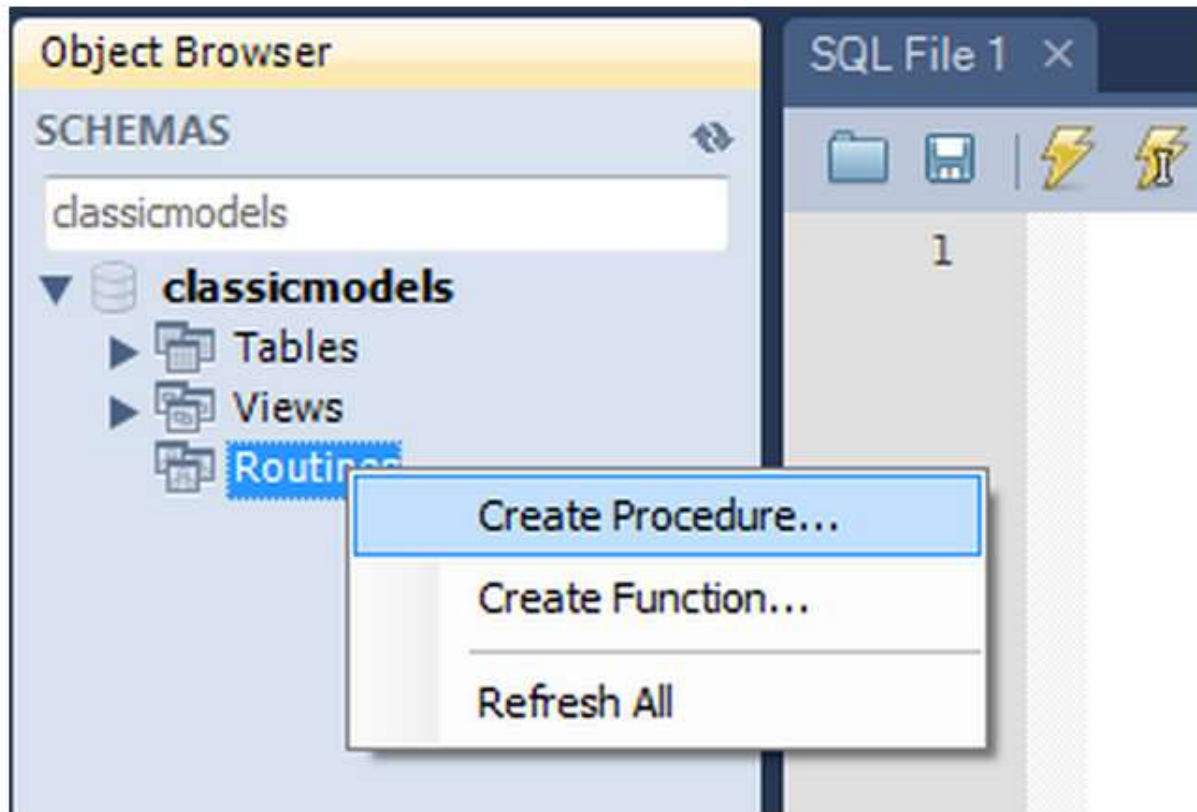
```
    SELECT * FROM products;
```

```
END //
```

```
DELIMITER ;
```



# STORED PROCEDURE IN WORKBENCH



**Right mouse click on the Routines and choose  
“Create Procedure...”**



Query 1 new\_procedure - Routine

Name: new\_procedure

Code:

```
1
2  -- Routine DDL
3  -- Note: comments before and after the routine body
4
5  DELIMITER //
6
7  CREATE PROCEDURE `classicmodels`.`GetAllProducts` ()
8  BEGIN
9
10     SELECT * FROM products;
11
12 END//
13
14 DELIMITER ;
```

100% 6:12

Routine Apply Revert

## Review the SQL Script to be Applied on the Database


Please review the following SQL script that will be applied to the database. Note that once applied, these statements may not be revertible without losing some of the data. You can also manually change the SQL statements before execution.

```
1  USE `classicmodels`;
2  DROP procedure IF EXISTS `GetAllProducts`;
3
4  DELIMITER $$
5  USE `classicmodels`$$
6  CREATE PROCEDURE `classicmodels`.`GetAllProducts` ()
7  BEGIN
8
9      SELECT * FROM products;
10
11  END$$
12
13  DELIMITER ;
14
15
```



## Applying SQL script to the database ...

The following tasks will now be executed. Please monitor the execution.  
Press Show Logs to see the execution logs.

 Execute SQL Statements

SQL script was successfully applied to the database.




# CALLING STORED PROCEDURES

CALL  
STORED\_PROCEDURE\_NAME

CALL GetAllProducts();








1 •

`CALL GetAllProducts();`

Filter:

File:  

Autosize:  A

	productCode	productName	productLine	productScale
	S10_1678	1969 Harley Davidson Ultimate Chopper	Motorcycles	1:10
	S10_1949	1952 Alpine Renault 1300	Classic Cars	1:10
	S10_2016	1996 Moto Guzzi 1100i	Motorcycles	1:10
	S10_4698	2003 Harley-Davidson Eagle Drag Bike	Motorcycles	1:10
	S10_4757	1972 Alfa Romeo GTA	Classic Cars	1:10



# VARIABLES

- A variable is a name that refers to a value

A name that represents a value stored in the computer memory

- MySQL

```
DECLARE name VARCHAR(255)
```

```
DECLARE age INT
```



# THREE TYPES OF PARAMETERS

- **IN**
  - **Default**
- **OUT**
- **INOUT**



# IN PARAMETER

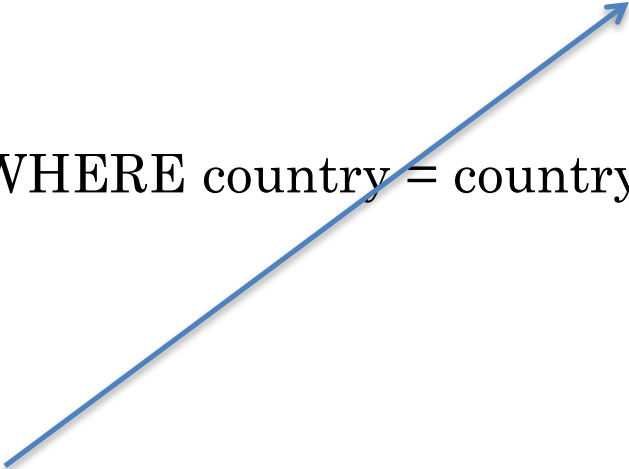
- Calling program has to pass an argument to the stored procedure.



# ARGUMENTS AND PARAMETERS

## Defining

```
DELIMITER //  
CREATE PROCEDURE GetOfficeByCountry(IN countryName  
VARCHAR(255))  
BEGIN  
SELECT * FROM offices WHERE country = countryName;  
END //  
DELIMITER ;
```



## Calling

```
CALL GetOfficeByCountry('USA')
```

The values being copied from the calling stored procedure are calling arguments.

The variables being copied into are called parameters.

# THREE TYPES OF PARAMETERS

- IN
  - Default
- **OUT**
- INOUT





# OUT PARAMETER

- OUT – the value of an OUT parameter can be changed inside the stored procedure and its new value is passed back to the calling program
- OUT is a keyword



# OUT PARAMETER

## Defining

DELIMITER //

CREATE PROCEDURE CountOrderByStatus(IN orderStatus  
VARCHAR(25), OUT total INT)

BEGIN

SELECT count(orderNumber) INTO total FROM orders WHERE  
status = orderStatus;

END//

DELIMITER ;

## Calling

CALL CountOrderByStatus('Shipped',@total);

SELECT @total;

The out parameter is used outside of the stored procedure.

# THREE TYPES OF PARAMETERS

- IN
  - Default
- OUT
- INOUT



# THE “IF” STATEMENT

## MySql Syntax

```
IF if_expression THEN commands  
  [ELSEIF elseif_expression THEN commands]  
  [ELSE commands]  
END IF;
```

First line is known as the IF clause

Includes the keyword **IF** followed by condition followed by the keyword **THEN**

- When the **IF** statement executes, the condition is tested, and if it is true the block statements are executed. Otherwise, block statements are skipped



# “IFEXPRESSION”: BOOLEAN EXPRESSIONS AND OPERATORS

Name	Description
<u>BETWEEN ... AND ...</u>	Check whether a value is within a range of values
<u>COALESCE ()</u>	Return the first non-NULL argument
<u>&lt;=&gt;</u>	NULL-safe equal to operator
<u>=</u>	Equal operator
<u>&gt;=</u>	Greater than or equal operator
<u>&gt;</u>	Greater than operator
<u>GREATEST ()</u>	Return the largest argument
<u>IN ()</u>	Check whether a value is within a set of values
<u>INTERVAL ()</u>	Return the index of the argument that is less than the first argument
<u>IS NOT NULL</u>	NOT NULL value test
<u>IS NOT</u>	Test a value against a boolean
<u>IS NULL</u>	NULL value test
<u>IS</u>	Test a value against a boolean
<u>ISNULL ()</u>	Test whether the argument is NULL
<u>LEAST ()</u>	Return the smallest argument
<u>&lt;=</u>	Less than or equal operator
<u>&lt;</u>	Less than operator
<u>LIKE</u>	Simple pattern matching
<u>NOT BETWEEN ... AND ...</u>	Check whether a value is not within a range of values
<u>!=, &lt;&gt;</u>	Not equal operator
<u>NOT IN ()</u>	Check whether a value is not within a set of values
<u>NOT LIKE</u>	Negation of simple pattern matching
<u>STRCMP ()</u>	Compare two strings

# IF STATEMENT

```
DELIMITER //
CREATE PROCEDURE
GetProductsInStockBasedOnQuantityLevel(IN
p_operator VARCHAR(255), IN p_quantityInStock INT)
BEGIN
    IF p_operator = "<" THEN
        select * from products WHERE quantityInStock <
p_quantityInStock;
    ELSEIF p_operator = ">" THEN
        select * from products WHERE quantityInStock >
p_quantityInStock;
    END IF;
END //
DELIMITER ;
```



# IF STATEMENT

- CREATE PROCEDURE

GetProductsInStockBasedOnQuantityLevel

(IN p\_operator VARCHAR(255), IN p\_quantityInStock  
INT)



The ooperator > or <

The number in stock



# THE IF STATEMENT

```
IF p_operator = "<" THEN
```

```
    select * from products WHERE quantityInStock  
< p_quantityInStock;
```

```
ELSEIF p_operator = ">" THEN
```

```
    select * from products WHERE quantityInStock  
> p_quantityInStock;  
END IF;
```





# LOOPS

- While
- Repeat
- Loop

Repeats a set of commands until some condition is met

Iteration: one execution of the body of a loop

If a condition is never met, we will have an infinite loop

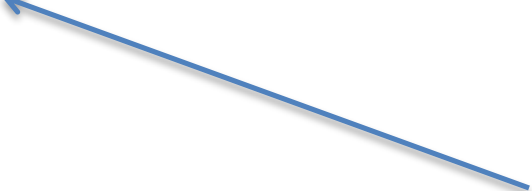


# WHILE LOOP

WHILE expression DO

Statements

END WHILE



The expression must evaluate  
to true or false

while loop is known as a *pretest* loop

Tests condition before performing an iteration

Will never execute if condition is false to start with

Requires performing some steps prior to the loop



# INFINITE LOOPS

- Loops must contain within themselves a way to terminate
  - Something inside a while loop must eventually make the condition false
- **Infinite loop**: loop that does not have a way of stopping
  - Repeats until program is interrupted
  - Occurs when programmer forgets to include stopping code in the loop



# WHILE LOOP

```
DELIMITER //
CREATE PROCEDURE WhileLoopProc()
BEGIN
    DECLARE x INT;
    DECLARE str VARCHAR(255);
    SET x = 1;
    SET str = "";
    WHILE x <= 5 DO
        SET str = CONCAT(str,x,',');
        SET x = x + 1;
    END WHILE;
    SELECT str;
END//
DELIMITER ;
```



# WHILE LOOP

- Creating Variables

```
DECLARE x INT;  
DECLARE str VARCHAR(255);  
SET x = 1;  
SET str = '';
```



# WHILE LOOP

```
WHILE x <= 5 DO
```

```
    SET str = CONCAT(str,x,',');
```

```
    SET x = x + 1;
```

```
END WHILE;
```



# TRIGGERS

- A set of SQL statements stored in the database catalog
- A SQL trigger is executed or fired whenever an event associated with a table occurs e.g., insert, update or delete
- A SQL trigger is a special type of stored procedure



# TRIGGERS VS STORED PROCEDURES

- A stored procedure is called explicitly
  - `CALL GetAllProducts()`
- A trigger is called implicitly and automatically
- When a data modification event is made against a table





# WHY TRIGGERS?

- Provide an alternative way to check the integrity of data
  - Uniqueness check: SQL query to check if value exists, if value doesn't exist, insert value
- Are very useful to audit the changes of data in tables
- Store business rules in the database



# DISADVANTAGES OF TRIGGERS

- May increase performance (overhead) of the database server
  - The trigger is being run in addition to the original SQL query and could take a large amount of time to execute
- Difficult to debug
  - Triggers are invoked and executed invisibly from client-applications therefore it is difficult to figure out what happen in the database layer
- Programmers don't have full control
  - Programmers don't have access to the database
  - Business rules are stored in database and hidden from application



# TRIGGERS

- A trigger is a set of SQL statements that is invoked automatically when a change is made to the data on the associated table
- A trigger can be defined to be invoked either before or after the data is changed by INSERT, UPDATE, or DELETE statement
- If you use any other statement than INSERT, UPDATE, or DELETE, the trigger is not invoked (For example TRUNCATE)



## WHY LEARN IT?

- Triggers allow specified actions to be performed automatically within the database, without having to write any extra application code.
- Triggers increase the power of the database, and the power of your application.
- You will learn much more about triggers in the following lessons.



# NEED FOR A TRIGGER

- Let's start with an example: a business rule states that whenever an employee's salary is changed, the change must be recorded in a logging table.
- We could create two procedures to do this:  
UPD\_EMP\_SAL to update the salary, and  
LOG\_SAL\_CHANGE to insert the row into the logging table. And we could invoke LOG\_SAL\_CHANGE from within UPD\_EMP\_SAL, or invoke LOG\_SAL\_CHANGE separately from the calling environment.
- But we don't have to do this. Instead, we create a trigger. The next slide shows how.



# EXAMPLE OF A SIMPLE TRIGGER

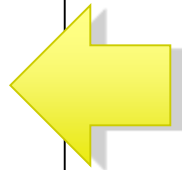
```
CREATE OR REPLACE TRIGGER log_sal_change_trigg  
AFTER UPDATE OF salary ON employees  
BEGIN  
    INSERT INTO log_table (user_id, logon_date)  
        VALUES (USER, SYSDATE);  
END;
```

- From now on, whenever a SQL statement updates a salary, this trigger executes automatically, inserting the row into the logging table.
- We say that the trigger automatically fires (i.e. executes) whenever the triggering event (updating a salary) occurs.
- Cause and effect: the event occurs, and the trigger fires.

# Triggers: Introduction

- The application constraints need to be captured inside the database
- **Some constraints can be captured by:**
  - Primary Keys, Foreign Keys, Unique, Not NULL, and domain constraints

```
CREATE TABLE Students
(sid: CHAR(20),
name: CHAR(20) NOT NULL,
login: CHAR(10),
age: INTEGER,
gpa: REAL Default 0,
Constraint pk Primary Key (sid),
Constraint u1 Unique (login),
Constraint gpaMax check (gpa <= 4.0) );
```



These constraints are defined in ***CREATE TABLE*** or ***ALTER TABLE***

# Triggers: Introduction

- **Other application constraints are more complex**
  - Need for assertions and triggers
- **Examples:**
  - Sum of loans taken by a customer does not exceed 100,000
  - Student cannot take the same course after getting a pass grade in it
  - Age field is derived automatically from the Date-of-Birth field



# Triggers

- A procedure that runs automatically when a certain **event** occurs in the DBMS
- **The procedure performs some actions, e.g.,**
  - Check certain values
  - Fill in some values
  - Inserts/deletes/updates other records
  - Check that some business constraints are satisfied
  - Commit (approve the transaction) or roll back (cancel the transaction)

# Trigger Components

- **Three components**

- **Event:** When this event happens, the trigger is activated
- **Condition (optional):** If the condition is true, the trigger executes, otherwise skipped
- **Action:** The actions performed by the trigger

- **Semantics**

- When the **Event** occurs and **Condition** is true, execute the **Action**

Lets see how to define these components

# Trigger: Events

- **Three event types**

- Insert
- Update
- Delete

- **Two triggering times**

- Before the event
- After the event

- **Two granularities**

- Execute for each row
- Execute for each statement

# 1) Trigger: Event

**Create Trigger** *<name>* **Before|After** **Insert|Update|Delete** **ON** *<tablename>*  
....

Trigger name

That is the event

## ● Example

**Create Trigger** *ABC*  
**Before Insert On** Students  
....

This trigger is activated when an insert statement is issued, but before the new record is inserted

**Create Trigger** *XYZ*  
**After Update On** Students  
....

This trigger is activated when an update statement is issued and after the update is executed

# Granularity of Event

- A single SQL statement may update, delete, or insert many records at the same time
  - E.g., Update student set gpa = gpa x 0.8;
- Does the trigger execute for each updated or deleted record, or once for the entire statement ?
  - We define such granularity

Create Trigger *<name>*

Before| After      Insert| Update| Delete

For Each Row | For Each Statement

....

This is the event

This is the granularity

# Example: Granularity of Event

**Create Trigger XYZ**  
**After Update ON** <tablename>

**For each statement**

....



This trigger is activated once (per UPDATE statement) after all records are updated

**Create Trigger XYZ**  
**Before Delete ON** <tablename>

**For each row**

....



This trigger is activated before deleting each record

## 2) Trigger: Condition

- This component is **optional**

**Create Trigger** *<name>*

**Before| After**      **Insert| Update| Delete On** *<tableName>*

**For Each Row | For Each Statement**

**When** *<condition>*      ← That is the condition

...

If the employee salary > 150,000 then some actions will be taken

**Create Trigger** *EmpSal*

**After Insert or Update On** *Employee*

**For Each Row**

**When** *(new.salary > 150,000)*

...

### 3) Trigger: Action

- **Action depends on what you want to do, e.g.:**
  - Check certain values
  - Fill in some values
  - Inserts/deletes/updates other records
  - Check that some business constraints are satisfied
  - Commit (approve the transaction) or roll back (cancel the transaction)
- **In the action, you may want to reference:**
  - The new values of inserted or updated records **(:new)**
  - The old values of deleted or updated records **(:old)**



# Trigger: Referencing Values

- In the action, you may want to reference:
  - The new values of inserted or updated records **(:new)**
  - The old values of deleted or updated records **(:old)**

**Trigger body** {

```
Create Trigger EmpSal
After Insert or Update On Employee
For Each Row
When (new.salary > 150,000)
Begin
    if (:new.salary < 100,000) ...
End;
```

Inside “When”, the “new” and “old” should not have “:.”

Inside the trigger body, they should have “:.”

# Trigger: Referencing Values (Cont'd)

- **Insert Event**
  - Has only :new defined
- **Delete Event**
  - Has only :old defined
- **Update Event**
  - Has both :new and :old defined
- **Before triggering (for insert/update)**
  - Can update the values in :new
  - Changing :old values does not make sense
- **After triggering**
  - Should not change :new because the event is already done

# Example 1

If the employee salary increased by more than 10%, make sure the 'rank' field is not empty and its value has changed, otherwise reject the update

If the trigger exists, then drop it first

**Create or Replace Trigger** *EmpSal*

**Before Update On** *Employee*

**For Each Row**

**Begin**

IF (:new.salary > (:old.salary \* 1.1)) Then

IF (:new.rank is null or :new.rank = :old.rank) Then

RAISE\_APPLICATION\_ERROR(-20004, 'rank field not correct');

End IF;

End IF;

**End;**

**/**

Compare the old and new salaries

Make sure to have the "/" to run the command

# Example 2

If the employee salary increased by more than 10%, then increment the rank field by 1.

In the case of **Update** event only, we can specify which columns

```
Create or Replace Trigger EmpSal  
Before Update Of salary On Employee  
For Each Row  
Begin
```

```
    IF (:new.salary > (:old.salary * 1.1)) Then
```

```
        :new.rank := :old.rank + 1;
```

```
    End IF;
```

```
End;
```

```
/
```

We changed the new value of **rank** field

The assignment operator has ":"

# Example 3: Using Temp Variable

If the newly inserted record in employee has null hireDate field, fill it in with the current date

```
Create Trigger EmpDate  
Before Insert On Employee  
For Each Row
```

Since we need to change values, then it should be "Before" event

```
Declare
```

Declare section to define variables

```
temp date;
```

```
Begin
```

```
  Select sysdate into temp from dual;
```

Oracle way to select the current date

```
  IF (:new.hireDate is null) Then
```

```
    :new.hireDate := temp;
```

Updating the new value of hireDate before inserting it

```
  End IF;
```

```
End;
```

```
/
```

# Example 4: Maintenance of Derived Attributes

Keep the bonus attribute in Employee table always 3% of the salary attribute

```
Create Trigger EmpBonus
Before Insert Or Update On Employee
For Each Row
Begin
    :new.bonus := :new.salary * 0.03;
End;
/
```

Indicate two events at the same time

The bonus value is always computed automatically

# Row-Level vs. Statement-Level Triggers


- **Example:** *Update emp set salary = 1.1 \* salary;*
  - Changes many rows (records)
- **Row-level triggers**
  - Check individual values and can update them
  - Have access to **:new** and **:old** vectors
- **Statement-level triggers**
  - Do not have access to **:new** or **:old** vectors (only for row-level)
  - Execute once for the entire statement regardless how many records are affected
  - Used for verification before or after the statement

# Example 5: Statement-level Trigger

Store the count of employees having salary > 100,000 in table R

```
Create Trigger EmpBonus
After Insert Or Update of salary Or Delete On Employee
For Each Statement
Begin
    delete from R;
    insert into R(cnt) Select count(*) from employee where salary > 100,000;
End;
/
```

Indicate three events at the same time



Delete the existing record in R, and then insert the new count.





# Order Of Trigger Firing

Loop over each affected record

***Before Trigger***  
(statement-level)

***Before Trigger***  
(row-level)

Even  
t  
(row-  
level)

***After Trigger***  
(row-level)

***After Trigger***  
(statement-level)

# Some Other Operations

- Dropping Trigger

```
SQL> Create Trigger <trigger name>;
```

- If creating trigger with errors

```
SQL > Show errors;
```

# POSSIBLE USES FOR TRIGGERS

- You can use triggers to:
  - Enhance complex database security rules
  - Create auditing records automatically
  - Enforce complex data integrity rules
  - Create logging records automatically
  - Prevent tables from being accidentally dropped
  - Prevent invalid DML transactions from occurring
  - And many other purposes!



# MORE USES FOR TRIGGERS

- You can use triggers to:
  - Generate derived column values automatically
  - Maintain synchronous table replication
  - Gather statistics on table access
  - Modify table data when DML statements are issued against views
  - And many other purposes !



## EXAMPLE 1:

### CREATING LOGGING RECORDS AUTOMATICALLY

- The Database Administrator wants to keep an automatic record (in a database table) of who logs onto the database, and when. He/she could create the log table and a suitable trigger as follows:

```
CREATE TABLE log_table (  
    user_id          VARCHAR2(30),  
    logon_date       DATE);  
  
CREATE OR REPLACE TRIGGER logon_trigg  
AFTER LOGON ON DATABASE  
BEGIN  
    INSERT INTO log_table (user_id, logon_date)  
        VALUES (USER, SYSDATE);  
END;
```

# WHEN A TRIGGER CAN BE INVOKED

- **BEFORE INSERT** – activated before data is inserted into the table.
- **AFTER INSERT** – activated after data is inserted into the table.
- **BEFORE UPDATE** – activated before data in the table is updated.
- **AFTER UPDATE** – activated after data in the table is updated.
- **BEFORE DELETE** – activated before data is removed from the table.
- **AFTER DELETE** – activated after data is removed from the table.



# NAMING A TRIGGER

- Triggers names for a table must be unique
- Can have the same trigger name defined for different tables
- Naming conventions

```
1 tablename_(BEFORE | AFTER)_(INSERT | UPDATE | DELETE)
```

order\_before\_update

A trigger invoked before a row in the order table is updated



# CREATE TRIGGERS

DELIMITER \$\$

CREATE TRIGGER trigger\_name trigger\_time  
trigger\_event

ON table\_name

FOR EACH ROW

BEGIN

...

END\$\$

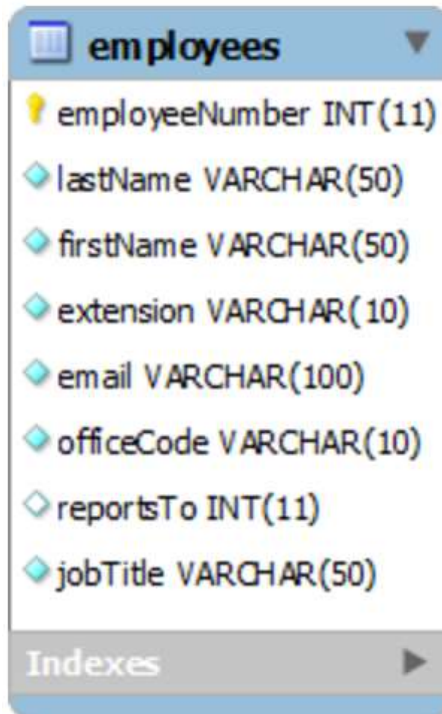
DELIMITER;





# CREATE TRIGGERS (CONTINUED)

- Create a trigger to log changes in the employees table



Need to create a table to store the changes **before** an **update** is made to **employees**

```
CREATE TABLE employees_audit (  
    id INT AUTO_INCREMENT PRIMARY KEY,  
    employeeNumber INT NOT NULL,  
    lastname VARCHAR(50) NOT NULL,  
    changedat DATETIME DEFAULT NULL,  
    action VARCHAR(50) DEFAULT NULL  
);
```

# WHAT SHOULD WE NAME THE TRIGGER?

```
1 tablename_(BEFORE | AFTER)_(INSERT | UPDATE | DELETE)
```

Need to create a table to store the changes **before** an **update** is made to **employees** table

Tablename = employee

Before or After = Before

Insert OR UPDATE OR DELETE = UPDATE

**employee\_before\_update**



# CREATE TRIGGERS

DELIMITER \$\$

CREATE TRIGGER **employee\_before\_update**

trigger\_time trigger\_event

ON table\_name

FOR EACH ROW

BEGIN

...

END\$\$

DELIMITER;



# WHAT IS THE TRIGGER TIME AND EVENT?

- Need to create a table to store the changes **before** an **update** is made to **employees** table
- **BEFORE INSERT**
- **AFTER INSERT**
- **BEFORE UPDATE**
- **AFTER UPDATE**
- **BEFORE DELETE**
- **AFTER DELETE**

## **BEFORE UPDATE**



# CREATE TRIGGERS

DELIMITER \$\$

CREATE TRIGGER **employee\_before\_update**  
**BEFORE\_UPDATE** ON **employees**

FOR EACH ROW

BEGIN

...

END\$\$

DELIMITER;

What SQL should go here?



# SQL IN TRIGGER BODY

- Goal is to store the changes **before** an **update** is made to **employees** table in the **employees\_audit** table
- Employees\_Audit
  - employeeNumber
  - lastname
  - changedat (date change was made)
  - action (what action was taken on the employees table)

```
INSERT INTO employees_audit  
SET action = 'update',  
employeeNumber = OLD.employeeNumber,  
lastname = OLD.lastname,  
changedat = NOW();
```



# WHAT DOES THE “OLD” KEYWORD MEAN?

- **OLD** keyword to access employeeNumber and lastname column of the row affected by the trigger
- **INSERT TRIGGER**
  - You can use **NEW** keyword only. You cannot use the **OLD** keyword.
- **DELETE Trigger**
  - There is no new row so you can use the **OLD** keyword only.
- **UPDATE Trigger**
  - **OLD** refers to the row before it is updated and **NEW** refers to the row after it is updated.



# CREATE TRIGGERS

DELIMITER \$\$

CREATE TRIGGER **employee\_before\_update**  
**BEFORE\_UPDATE** ON employees

FOR EACH ROW

BEGIN

INSERT INTO employees\_audit

SET action = 'update',

employeeNumber = OLD.employeeNumber,

lastname = OLD.lastname,

changedat = NOW();

END\$\$

DELIMITER;





# TEST TRIGGER

- Update the employees table to check whether the trigger is invoked

UPDATE employees

SET

lastName = 'Phan'

WHERE

employeeNumber = 1056;



# TEST TRIGGER

- Check if the trigger was invoked by the UPDATE statement

```
SELECT * FROM employees_audit;
```

	id	employeeNumber	lastname	changedat	action
▶	1	1056	Phan	2015-11-14 21:39:12	update

