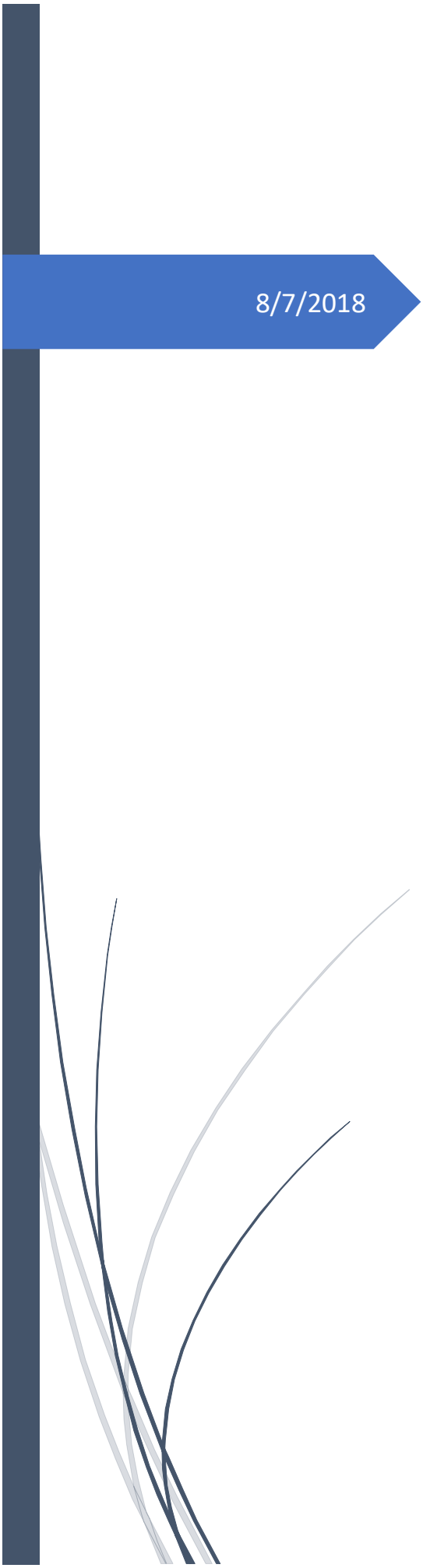
8/7/2018

MySQL Notes
V 1.0



Raj Solanki

Contents

MySQL Udemy Notes [1]

1. Creating Tables

a. **Null** – Means allow no values

1. CREATE TABLE USERS (id int not null, username text not null);
2. Show engines; #This shows engines available to use

b. Default engines are **MYISAM** and **INNODB**

c. **Autoincrement** – allows the column to update automatically such as id

d. Different Data Types

1. **Char(<size>)** – fixed length.
2. **Varchar(<size>)** – variable length and resizes to save memory
3. **Blob** – store binary types
4. **Boolean** – True/False
5. **Timestamp/year/date**
6. **Datetime** – combines date and time together
7. **ENUM (Value 1, Value 2…)** – use this keyword for create a list of values a column can take on. This is called an enumeration.

e. **Examples of Creating Tables**

1. **Example 1**

```
1. CREATE TABLE products (
2.     product VARCHAR (100),
3.     available BOOL DEFAULT FALSE
4. );
5. desc products;
```

2. **Example 2**

```
1. CREATE TABLE foods (
2.     food VARCHAR (50),
3.     flavor ENUM ('Sweet', 'Bitter') DEFAULT 'Sweet',
4.     texture ENUM ('Rough', 'Soft') DEFAULT 'Soft'
5. );
```

3. **Example 3**

```
1. CREATE TABLE products (
2.     id INT PRIMARY KEY,
3.     name VARCHAR (20),
4.     category VARCHAR (20),
5.     sell_by DATE,
6.     sold BOOL,
7.     moment_of_sale TIMESTAMP,
8.     quantity INT,
9.     weight DECIMAL (10, 2)
10. );
```

[1] (Notes are derived from MySQL, SQL and Stored Procedures from Beginner to Advanced course on Udemy)

f. **How to insert values into tables?**

1. **Syntax**
    1. INSERT into table_name(column 1,…,column n) VALUES (value1,..,value n),(value 1,…, value n);

g. **Foreign Keys**

1. A foreign key is a field in a table that matches another field (The primary key field) of another table. A foreign key places constraint on data in the related tables. This enforces referential integrity
    1. Referential Integrity – It states that table relationships must be consistent
        a. Any primary key changes much be applied to all foreign keys or not at all
        b. Any foreign key changes must be propagated to the primary parent keys

2. **Syntax**
    1. **CONSTRAINT constraint_name**
    2. **FOREIGN KEY foreign_key_name (columns)**
    3. **REFERENCES parent_table(columns)**
    4. **ON DELETE action**
    5. **ON UPDATE action**
        a. **FOREIGN KEY** – Specifies the column that the child table references to
        b. **ON DELETE** – allows you to define what happens to the child table rows when records in parent table deleted
            i. If not specified, MySQL will block any deletion
            ii. ON DELETE CASCADE – will delete the child rows that correspond to the deleted rows in parent table
            iii. ON DELETE SET NULL – child records will not be deleted
        c. **ON UPDATE** – specifies what happens to the child rows when the rows in the parent table are updated
            i. **ON UPDATE CASCADE** – this property allows cross table updates
            ii. **ON UPDATE SET NULL** – this property sets the rows in the child table to be NULL when rows in parent table are updated
            iii. **ON UPDATE RESTRICT** – reject any updates
3. **Adding Foreign Keys**
    1. **ALTER TABLE** *table_name* ADD constraint fk_name (column_name)
    2. **REFERENCES** *table_name(column_name);*
        a. **Note –** The reference table refers to the primary key of the related table.

h. **Adding/Dropping new columns to a table**

1. Syntax for adding
    1. ALTER TABLE table_name
    2. ADD column_name datatype [before|after] column_name;
2. Syntax for dropping
    1. ALTER TABLE table_name
    2. DROP column_name;

2. Global Modes

a. SELECT @@GLOBAL.sql_mode, @@SESSION.SQL_MODE;
    1. **Global** – affects all connections
    2. **Session** -affects only this connection
    3. Above is the syntax

3. Deleting data from tables

a. By default, MySQL will prevent you from deleting all rows without a condition
b. To all to delete all rows freely within session use
    1. SET sql_safe_updates=0;
    2. SELECT @@Session.SQL_SAFE_UPDATES; /*This is to vie this property */

4. Distinct Keyword

a. Return only the unique values for all columns selected which are the unique rows
b. **Syntax**
    1. SELECT DISTINCT *column_name*,*column_name*
       FROM *table_name*;
c. `With aggregate functions`
    1. `Count Distinct`
        1. `Example`
            a. select count (distinct age) from users;
    2. Can mix with other aggregate functions

5. Having Keyword

a. Use this to filter groups of rows or aggregates
b. Having is used with a Group by clause
c. **Example**
    1. SELECT
    2. country, gender, COUNT (*), AVG(weight)
    3. FROM
    4. survey
    5. GROUP BY country, gender
    6. HAVING COUNT (*) > 2
    7. ORDER BY country;

6. Joins

a. **Inner Join**
   1. Selects only matching values between tables
   2. Excludes all NULL values (Also NULL<> NULL)
   3. Specified using “INNER JOIN” or “JOIN”

b. **Left/Right Join**
   1. Left/Right join returns all the matched values with the Left/Right table and matched records from the other table
   2. Specified using “LEFT/RIGHT JOIN”

c. **Self-Join**
   1. A join with the same table itself

d. **Minus Join**
   1. Select only those values in that table. IE Remove any values both tables share.
   2. Syntax:
      1. Select * from table A left join table B on col A = col B where col B is null;

7. Unions/Union All

a. **Union –** This removes all duplicate rows so it’s less efficient
b. **Union All-** This keeps all rows and duplicate rows
c. Must have the same columns in each query in the same order with the same name
d. **Example: Select acctno, name from dda union all select accnto, name from cd;**

8. Inline Views

a. **Example:**

Select avg(respondents) from (select country, count (*) as respondents from survey group by country) as totals;

Note - (select country, count (*) as respondents from survey group by country) as totals) is the view

9. Information Schema

a. This provides more information about the database meta characteristics
b. **Examples**
   1. How to find all tables?
      1. USE information schema;
      2. SELECT TABLE_NAME from information_schema.TABLES;

10. Adding Indexes

a. It speeds up querying by adding an index to the column
b. A primary key already has an index

c. An index is a copy of the column and is ordered behind the scenes, so it allows MySQL to query the column very fast
d. It does take up disk space
    1. **Syntax**
        1. ALTER *table_name*
        2. add index *table_name(column_name)*;

11. Setting variables

a. Syntax using SET
    1. SET @var_name=value;
b. Syntax using Select
    1. 1st way – **Using the := operator**
        1. SELECT @var_name:= value;
            a. Note- You must use the **:=** operator to assign values.
    2. 2nd way – **Using the into operator**
        1. SELECT var1, var2 INTO @var_name_1 from tbl;
        2. SELECT @var_name_1;

12. Creating Users

a. Using the root account is bad since it has security risks
b. How to create users and grant privileges?
    1. Example 1
        1. CREATE USER 'user'@'localhost' IDENTIFIED by 'password';
        2. GRANT ALL PRIVILEGES ON *. * to 'Raider Nation'@'localhost'; //The *.* is a wildcard for all dbs
        3. FLUSH PRIVILEGES;
    2. Example 2
        1. GRANT ALL PRIVILEGES ON cra.* to 'Raider Nation'@'localhost';
            a. This guy can do anything to all tables in cra db.
    3. Example 3
        1. GRANT SELECT ON cra.* to 'Raider Nation'@'localhost';
            a. Can only query

13. Views

a. Views are useful as they are temporary tables but not as efficient as querying the db.
b. Great for giving another user the ability to view needed data and hide any confidential data
c. Can insert data in for a simple view
d. Views in MySQL are archived when updated and can be restored using
    1. View_name.frm-00001
e. Cannot create an index on a view
f. Syntax
    1. CREATE
    2. [OR REPLACE]
    3. [ALGORITHM = {UNDEFINED | MERGE | TEMPTABLE}]

4. [DEFINER = { user | CURRENT_USER }]
5. [SQL SECURITY { DEFINER | INVOKER }]
6. VIEW view_name [(column_list)]
7. AS select_statement
8. [WITH [CASCADED | LOCAL] CHECK OPTION]

g. Note – Specifying the WITH CHECK OPTION at the end will allow data to be updated and incorporated within the view.

14. Locking Tables

a. When importing data into a MySQL table, it locks the table to prevent any other edits from taking place.
b. It locks the table when importing because there can be multiple connetions to the same db to prevent changing the table when importing the data.
c. When locking tables, only that connection that locked the table has access to that table.
d. **Types of Locks**

1. **Exclusive Table Syntax**
    1. LOCK TABLE table_name [[AS] Alias] lock_type
    2. [, table_name [[AS] Alias] lock_type]…
    3. Lock_type:
        a. READ [LOCAL] | [LOW_PRIORITY] WRITE
            i. READ – Can only read the tables
            ii. Write – can only write to the tables
    4. UNLOCK TABLES ← This will unlock all tables

2. **Shared locks**
    1. **Read Lock** - means multiple connections have that lock to that table but they can't write to the table.
    2. Once a table is locked, that connection cannot query other unlocked tables.
    3. To go around the issue of locking all tables, you can set a read lock on the tables impacted and a write lock to only the tables being written to.
    4. This is typically used if your engine is MyISAM. If your engine is InnoDB you can lock individual rows called a transaction.

15. ACID DB Acronym

a. **A** = Atomicity which allows you to execute multiple SQL statements and functions as 1 statement
b. **C**= Consistency which prevents the user from writing code that leaves the dB in an inconsistent state like wrong values or dangling foreign keys.
c. **I** = Isolation whatever one user does we want to isolate it from the other user so that both users don't conflict each other
d. **D** = Durability or DB can be recovered easily from a crash

e. Notes
    1. InnoDB is ACID compliant while MyISAM isn't

f. **Row Level Locking** - Lock certain rows when doing an update within another transaction but you must have an index on the column your selecting your rows by. Other users can update other rows not locked (If there is no index, then it will default to table locking)
g. **Read Lock** (Shared Locking) – you can read the rows but not update
h. **Write Lock (**Exclusive Locking) – You can update the rows but no one else can read them.
i. **Transactions -** This ensures atomicity so if you have > 1 sql statement and you want to make sure all the statements commits/remains uncommitted you wrap them in a transaction

16. Transaction Syntax and Example

a. Data won't be inserted until its committed
b. Others won't see changes till its committed
c. **Rollback –** This will undo any changes from the last commit and will be rolled back.
d. **Set autocommit=0** – This will prevent commits from being automatic.
e. **Example 1**
    1. SET autocommit=0;
    2. INSERT into products (name, price,category_id, quantity_available) values ('DIck Haryy',5.35,1,500);
    3. DELETE from products where id=3;
    4. UPDATE products set name="Raji Roy" where category_id=3;
    5. SELECT* from products;
    6. COMMIT;
    7. SELECT * from products;
    8. ROLLBACK; #This will undo all the changes between SET and Commit.
f. **Example 2**
    1. START transaction;
    2. SELECT * from sales;
    3. UPDATE products set name="The 39 book" where id=1;
    4. COMMIT;
g. **Isolation and Row Locking**
    1. To check isolation level use SELECT @@Session.tx_isolation;
    2. Different isolation levels
        1. **SERIALIZABLE -** This means that this user who does a select on rows then other users won't be able to rollback/commit to those rows till he is done with the transaction as those selected rows get locked but will make users wait to update so efficiency is bad.

        2. **REPEATABLE READ -** Prevents 1 user from seeing the updates of other users. That is if the same users run the same select statement over and

over in a transaction, he will get the same results despite updates going on with other users using multiple connections.

3. **READ COMMITTED –** Your select statements within your transaction get updated with the commits/rollbacks from other connections.

4. **READ UNCOMMITED –** You can see all uncommitted transactions within your current transactions.

3. To set isolation level use
    1. SET [GLOBAL | SESSION] TRANSACTION
    2. transaction_characteristic [, transaction_characteristic] ...
    3.
    4. transaction_characteristic:
    5. ISOLATION LEVEL level
    6. | READ WRITE
    7. | READ ONLY
    8.
    9. level:
    10. REPEATABLE READ
    11. | READ COMMITTED
    12. | READ UNCOMMITTED
    13. | SERIALIZABLE
4. Example of setting isolation level
    1. SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

h. Rolling back transactions to savepoints
    1. Use **SAVEPOINT** save_name; within your transaction to create a savepoint to rollback to.
    2. **SAVEPOINT**s do not end your transaction or release any locks.
    3. Example:
        1. START transaction;
        2. INSERT t into person(name) values ("Tom"),("Dic"),("Harry");
        3. SELECT * from person;
        4. DELETE from person where id=4;
        5. **SAVEPOINT** test1;
        6. UPDATE person set name="Bad point" where id=1;
        7. SELECT * from person;
        8. ROLLBACK to test1;
        9. COMMIT;
        10. SELECT * from person;

17. String/Date built-in MYSQL Functions

a. **CONCAT (string1, string2)**
    1. Combines strings together
    2. Examples
        1. SELECT CONCAT(fname, lname) FROM people;

b. **CURDATE ()**
    1. Returns the current date

c. **CURDATE () – INTERVAL [time unit]**
    1. INTERVAL allows you to add or subtract a unit of time
    2. Examples:
        1. CURDATE () – INTERVAL 1 MONTH
        2. CURDATE () – INTERVAL 1 DAY
        3. SELECT date_sub('2010-06-15',interval 5 month);

d. **DAY (<date>)**
    1. Returns the numeric value of the date.

e. **DATEDIFF (date1, date2)**
    1. Returns the # of days between dates

f. **str_to_date(string, format="%m/%d/%Y")**
    1. Essentially takes a date in a string and converts it to a MYSQL Date type using the format code.
        1. Common format parameters:
            a. %D – day of month with suffix (1st, 2nd, etc.)
            b. %d - numeric day of the month
            c. %M – full month name
            d. %m – numeric month
            e. %M – full month name
            f. %y – numeric 2-digit year
            g. %Y – numeric 4-digit year
            h. %W – weekday name
            i. %a, %b – abbreviated day/year

g. **date_format(<Date>, format)**
    1. Will format a MySQL date type to format specified
    2. The format parameters are the same as str_to_date function

18. Cast function

a. Converts one data type to another

b. **Syntax**
    1. **Cast (VALUE as <TYPE>)**
        1. TYPE:
            a. Date
            b. Unsigned
            c. Binary
            d. Datetime
            e. Char

19. RAND () Function

a. Generates random numbers between 0 and 1.

20. Stored procedures

a. Gives your ability to group together multiple statements
b. These are efficient
c. Allows you to make db more secure as you can give the user the ability to run that procedure
d. **Example 1**

```
1. delimiter $$
2.
3. CREATE procedure HelloWorld()
4. begin
5. SELECT * from dates;
6. select "Hello World";
7. end$$
8. CALL HelloWorld();
9. DROP procedure HelloWorld;
```

e. **Listing all procedures**

```
1. SELECT cast (body as char) from mysql.proc; #CAST bits to char
```

f. **Defining permissions for procedures**

1. The stored procedure will only run with the permissions of the definer
2. Must grant the user **Execute** permission to run
3. If a user runs a stored procedure, everything inside will also run according to the permissions of the user
4. Any statements not covered by the user permissions won't run. CR
    1. **Example 1**

```
1. CREATE USER shopuser@localhost IDENTIFIED by 'hello';
2. delimiter $$
3.
4. CREATE definer=shopuser@localhost procedure
   ShowCustomer()
5. sql security definer # This will run with the permission with the
   permission of the definer
6. begin
7. SELECT * from customers;
8. end$$
9. delimiter;
10. CALL ShowCustomer();
```

    2. Example 2 – Covers how to grant procedure permissions to users

a. GRANT execute on procedure online_shop.ShowCustomer to prokUser@localhost;

g. **Passing parameters into procedures**

1. **Example 1- Single Parameter**

```
1. DELIMITER $$
2. CREATE PROCEDURE `procedure_name`(in max_id int) # This is how define the input parameters. Note that we use the keyword in and define the parameter var type.
3. begin
4. begin
5. select * from person where id < max_id;
6. end$$
7. DELIMITER;
8.
9. CALL ShowBooks(5);
```

2. **Example 2 – Multiple Parameters**

```
1. DELIMITER $$
2. CREATE PROCEDURE `theTitle` (in max_id int, in title varchar (20)) # This is how define many input parameters.
3. begin
4. select * from person where id < max_id;
5. select title;
6. end$$
7. DELIMITER ;
8.
9. CALL theTitle (5,'Halo Fire');
```

h. **Outputting from procedure**

1. Use the OUT parameter to output values
2. Don't prefix your OUTPUT parameter with a @ within procedure.
3. Example 1

```
1. DELIMITER $$
2. CREATE PROCEDURE `outPerson`(in the_id int , out outID int, out outName varchar(30))
3. begin
4. select id,name into outID,outName from person where id =the_id;
5. end$$
6. DELIMITER;
7.
8. call outPerson(3, @id, @title); #Passing in output variables
9. select @id, @title; #showing output variables
```

i. **INOUT keyword**

1. Combines the functionality of in and out parameters

2. It will take in both in and return the same variable
3. **Example 1** - Assume there are 8 records in this table

```
1. Delimiter //
2. CREATE PROCEDURE TEST1 (INOUT ID INT)
3. BEGIN
4. SELECT ID;
5. SELECT COUNT (*) INTO ID FROM PERSON
6. END//
7. SET DELIMETER;
8.
9. SET @value = 3;
10. CALL test1(@value);
11. SELECT @value; # Will return 8 now.
```

j. **IF Statement**

1. Can use if statement within procedure
2. **Syntax**

```
1.  if condition then
2.  <statement 1>
3. <Statement 2>
4. <Statement n>
5. END IF;
```

3. **Example 1**

```
1. Delimiter $$
2. Create procedure withdraw (IN flag bool)
3. Begin
4.    IF flag=true then
5.          SELECT “Hello”
6.     ELSE
7.        SELECT “bye”
8. END IF;
9. END$$
10. Delimiter;
```

k. **Variable scope**

1. **Session:** Any variables using @ are session variables
2. **Local:** Any variable without a prefix but are declared using **declare** within procedure
3. **Example 1**

```
1. Delimiter $$
2. CREATE PROCEDURE proc_name(in param1 var_type)
3. Begin
4. Declare var1 var_type; # This is a local variable
5. SET var1:=value; # This is how we initialize the variable
6. END $$
7. Delimiter;
```

l. **Errors and Warnings**

1. Want to be able to pick up errors and warnings such as id not found
2. If there is an error/warning, then we can continue/exit the procedure.
3. **Syntax**

```
1.  Delimiter $$
2.  CREATE PROCEDURE proc_name (in param1 var_type)
3.  Begin
4.  Declare var1 var_type;
5.  # Declare exit for procedure for any warning or exception.
6.
7.  DECLARE EXIT HANDLER FOR SQLEXCEPTION
8.  Begin
9.      SHOW errors;
10. End;
11. declare exit handler for sqlwarning
12.    begin
13.     SHOW warnings;
14.    end;
15. <Rest of procedure>
```

m. **While Loops**

1. Syntax

```
1.  [ label_name: ] WHILE condition DO
2.    {...statements...}
3.  END WHILE [ label_name ];
```

2. Example

```
1.
2.  delimiter $$
3.
4.  create procedure whiledemo()
5.  begin
6.
7.     declare count int default 0;
8.     declare numbers varchar (30) default "";
9.
10.    while count<10 do
11.
12.       set numbers:= concat (numbers,' ', count);
13.       set count := count+1; #We update the counter
14.
15.            end while;
16.
17.     select numbers;
18.
19. end$$
```

n. **Labeled loops**

1. Useful if you don't know how many iterations you need
2. Use LOOP keyword within procedure
3. Create a labeled name for loop as well
4. **Example**

```
1.  delimiter $$
2.
3.  CREATE procedure loopdemo()
4.  begin
5.
6.          declare count int default 0;
7.     declare numlist varchar(50) default "";
8.     the_loop: loop
9.                  if count=10 then
10.                         leave the_loop;
11.                 end if;
12.
13.     set numlist:= concat(numlist,',' , count);
14.
15.                 set count:= count+1;
16.
17.
18.         end loop;
19.
20.    select numlist;
21.
22. end $$
23.
24. delimiter ;
```

o. **Cursors**

1. A cursor allows you to iterate through a set of rows returned by a query.
2. Cursors are read only so cannot update underlying table through a cursor
3. Cursors only iterate through rows in the order the query returns
4. Cursors can point to either the actual or copy of data
5. **Syntax**
    1. Declare the cursor
        a. DECLARE cursor_name CURSOR FOR SELECT_statement;
    2. Initaiate the cursor
        a. OPEN cursor_name;
    3. Fetch the rows
        a. FETCH cursor_name INTO variables list;
    4. Close the cursor
        a. CLOSE cursor_name;

5. Declare a continue handler to handle the case when there are no more rows to iterate through
    a. DECLARE CONTINUE HANDLER FOR NOT FOUND SET finished = 1

6. You can have multiple cursors but use a nested begin…end block. This is because tables have different # of rows, so you won't know which one ends first if the cursors are used together.
    1. **Syntax**
        a. Create procedure name ()
        b. Begin
        c. …
        d. End
        e. Begin
        f. …
        g. End//
        h. Delimiter;

7. **Example 1**

```
1. delimiter //
2.
3. CREATE procedure cursortest1()
4. begin
5.
6. DECLARE the_email varchar(40);
7. DECLARE finished Boolean default false;
8. DECLARE cur1 cursor for select email from users where active=true and
   registered> date (now ())- interval 1 year; #Declare 1st cursor
9.
10. DECLARE continue handler for not found set finished: =true; #This will
    detect when the fetch below can't find any more data it throws a not
    found condition so this
11.   #handles that.
12.
13.   open cur1;
14.         the_loop : loop
15.
16.                   fetch cur1 into the_email; #This will only grab one row
17.
18.     if finished then
19.                         leave the_loop;
20.                   end if;
21.
22.     insert into leads (email) values (the_email);
23.
24.   end loop the_loop;
```

```
    close cur1;
        select * from leads;

end//

delimiter;
```

21. Triggers

a. Triggers are automated procedures that run when the data is changed
b. Triggers happen in response to an event. These events are
    1. Update
    2. Insert
    3. Delete
c. Triggers are set to happen **before** or **after** the event.
d. **Advantages**
    1. Alternative way to check data integrity
    2. Can catch errors in bizness logic in the database layer
    3. Very useful for auditing changes
e. **Disadvantages**
    1. Cannot perform all possible data integrity checks
    2. Can increase overhead
    3. Executed invisible from client apps making it hard to diagnose what is going on in the database layer.
f. **Syntax**
    1. CREATE TRIGGER trigger_name trigger_time trigger_event
    2. ON table_name
    3. FOR EACH ROW
    4. BEGIN
    5. ...
    6. END;
g. **Example 1**

```
1. delimiter $$
2.
3. create trigger before_sales_update before update on sales for each row #It will be run once for every changed row
4. begin
5. insert into sales_update(product_id, changed_at, before_value,after_value)
6.  value (old.id,now(),old.value,new.value);
7. end$$
8.
9. delimiter;
```

22. User Defined Functions (UDF)

a. **Example 1**

```
1. delimiter $$
2.create function sales_after_tax (tax float, day date) returns numeric (10,2)
3.begin
4.#Stuff goes here
5.return result;
6. end $$
7.delimiter;
```