DAY 3

Day 3 Contents

- Transactions
- Built in functions
- Stored functions.
- Triggers
- Backup and Restore

Transactions

- The example of this involves a financial transfer where money from one account is placed into another account. Suppose that Moataz writes a check to ali for $100.00 and ali cashes the check. Moataz's account should be decremented by $100.00 and ali account incremented by the same amount:

```
UPDATE account SET balance = balance - 100 WHERE name =
'Moataz';
UPDATE account SET balance = balance + 100 WHERE name =
'ali';
```

Transactions

- If a crash occurs between the two statements, the operation is incomplete.

  Depending on which statement executes first, Moataz is $100 short without Islam having been credited, or Islam is given $100 without Motaz having been debited.
- Another use for transactions is to make sure that the rows involved in an operation are not modified by other clients while you're working with them.

Transactions

- A transaction is a set of SQL statements that execute as a unit. Either all the statements execute successfully, or none of them have any effect.
- This is achieved through the use of commit and rollback capabilities. If all of the statements in the transaction succeed, you commit it to record their effects permanently in the database. If an error occurs during the transaction, you roll it back to cancel it. Any statements executed up to that point within the transaction are undone, leaving the database in the state it was in prior to the point at which the transaction began.

Transactions

One way to perform a transaction is to issue a **`BEGIN TRANSACTION`** (or **`BEGIN`**) statement, execute the statements that make up the transaction, and end the transaction with a **`COMMIT`** or (**`END TRANSACTION`**) statement to make the changes permanent.

If an error occurs during the transaction, cancel it by issuing a **`ROLLBACK`** statement instead to undo the changes.

Transactions

- The following example illustrates this approach. First, create a table to use

```
CREATE TABLE t (name CHAR(20), UNIQUE (name));
```

- The statement creates a table, Next, initiate a transaction with BEGIN TRANSACTION, add a couple of rows to the table, commit the transaction, and then see what the table looks like:

```
BEGIN TRANSACTION;

INSERT INTO t SET name = ‘Islam';

INSERT INTO t SET name = ‘Moataz';

COMMIT;

SELECT * FROM t;
```

Transactions

- if you issue any of the following statements while a transaction is in progress, the server wait until the transaction ended before executing the statement:

```
ALTER TABLE

CREATE INDEX

DROP DATABASE

DROP INDEX

DROP TABLE

RENAME TABLE

TRUNCATE TABLE
```

Transactions

- Postgres enables you to perform a partial rollback of a transaction. To do this, issue a SAVEPOINT statement within the transaction to set a marker. To roll back to just that point in the transaction later, use a ROLLBACK statement that names the savepoint.

```
CREATE TABLE t (name CHAR(20), UNIQUE (name));
```

```
CREATE TABLE t (i INT);
BEGIN TRANSACTION;
INSERT INTO t VALUES(1);
SAVEPOINT my_savepoint;
INSERT INTO t VALUES(2);
ROLLBACK TO SAVEPOINT my_savepoint;
INSERT INTO t VALUES(3);
COMMIT;
```

Transactions

- To use transactions, you *must* use a transactional storage engine such as InnoDB Engines such as MyISAM and MEMORY will not work.
- By default, MySQL runs in **autocommit** *mode*, which means that changes made by individual statements are committed to the database immediately to make them permanent.

Transactions

- To perform transactions explicitly, **disable autocommit** mode and then tell MySQL when to commit or roll back changes.
- One way to perform a transaction is to issue a `START TRANSACTION` (or BEGIN) statement to suspend autocommit mode, execute the statements that make up the transaction, and end the transaction with a `COMMIT` statement to make the changes permanent. If an error occurs during the transaction, cancel it by issuing a ROLLBACK statement instead to undo the changes.

Transactions

- The following example illustrates this approach. First, create a table to use

```
CREATE TABLE t (name CHAR(20), UNIQUE (name)) ENGINE = InnoDB;
```

- The statement creates an InnoDB table, but you can use a different transactional storage engine if you like. Next, initiate a transaction with START TRANSACTION, add a couple of rows to the table, commit the transaction, and then see what the table looks like:

```
START TRANSACTION;
INSERT INTO t SET name = ‘Islam';
INSERT INTO t SET name = ‘Moataz';
COMMIT;
SELECT * FROM t;
```

Transactions

- Another way to perform transactions is to manipulate the autocommit mode directly using SET statements:

```
SET autocommit = 0;
SET autocommit = 1;
```

- Setting the autocommit variable to zero disables autocommit mode. The effect of any statements that follow becomes part of the current transaction, which you end by issuing a COMMIT or ROLLBACK statement to commit or cancel it. With this method, autocommit mode remains off until you turn it back on, so ending one transaction also begins the next one. You can also commit a transaction by re-enabling autocommit mode.

Transactions

- To see how this approach works, begin with the same table as for the previous examples:

```
DROP TABLE t;
CREATE TABLE t (name CHAR(20), UNIQUE (name)) ENGINE =
InnoDB;
```

- Then disable autocommit mode, insert some rows, and commit the transaction:

```
SET autocommit = 0;
INSERT INTO t SET name = ‘Islam';
INSERT INTO t SET name = ‘Moataz';
COMMIT;
SELECT * FROM t;
```

Built in Functions

- Sometimes you need to modify and format your displayed result set in your query , this group of functions help in modifying and formatting such result set
- According to the input and output of those functions they can be classified into two main categories :
  - Single Row Functions <Scalar Functions> : work on a set of rows and return one row - Aggregation Functions.
  - Multi Row Functions : work on a set of rows in a row by row interaction mode fashion.

Built in Functions

- The Multi row functions are categorized according to the mode of action and argument`s data type into the following :
  - Comparison Functions
  - Control Flow Functions
  - Cast Functions
  - Managing Different Types of Data

Comparison Functions

- These functions allow you to compare different values and, from those comparisons, return one of the values or return a condition of true, false, or NULL. If either argument or both arguments in a comparison are NULL, NULL is returned

ELT()

`ELT(n,str1,str2,...)`

- Returns the n-th string from the list of strings str1, str2, ... Returns NULL if n is NULL, the n-th string is NULL, or there is no n-th string.The index of the first string is 1. `ELT()` is complementary to `FIELD()`.

ELT()

`ELT(3,'a','b','c','d','e')` → 'c'

`ELT(0,'a','b','c','d','e')` → NULL

`ELT(6,'a','b','c','d','e')` → NULL

`ELT(FIELD('b','a','b','c'),'a','b','c')` → 'b'

FIELD()

`FIELD(arg0,arg1,arg2,...)`

- Finds arg0 in the list of arguments arg1, arg2, ... and returns the index of the matching argument (beginning with 1). Returns 0 if there is no match or if arg0 is NULL. String comparison is used if all arguments are strings, numeric comparison if all arguments are numbers, and double-precision comparison otherwise. `FIELD()` is complementary to `ELT()`.

FIELD()


```
FIELD('b','a','b','c') →2
FIELD('d','a','b','c') →0
FIELD(NULL,'a','b','c') →0
FIELD(ELT(2,'a','b','c'),'a','b','c') →2
```

GREATEST() & LEAST()

- The values specified can be numeric, string, or date/time values and are compared based on the current character set
- allow you to compare two or more values and return the value that is either the highest or lowest , depending on the function used.
- When you use this function, you must specify at least two values, although you can specify as many additional values as necessary.

```
SELECT GREATEST(4, 83, 0, 9, -3);
SELECT LEAST(4, 83, 0, 9, -3);
```

ISNULL()

- The `ISNULL()` returns a value of 1 if the expression evaluates to NULL; otherwise, the function returns a value of 0

```
SELECT ISNULL(1  *NULL); □ 1
```

STRCMP()

- The STRCMP( ) returns 0 if the strings are the same, -1 if the first argument is smaller than the second according to the current sort order, and 1 otherwise.

```
STRCMP('a','a') → 0
STRCMP('a','h') → -1
```

Cast Functions

- Cast functions allow you to convert values to a specific type of data or to assign a character set to a value.
- The type value may be CHAR(n) (non-binary string), DATE, DATETIME, TIME, SIGNED [INTEGER], UNSIGNED [INTEGER]

CONVERT()

- The following are the casting functions:
  - CONVERT();

```
SELECT CONVERT(20041031, DATE);
SELECT CONVERT(20041031, DATETIME);
SELECT CONVERT(20041031, TIME);
```

Managing Data Types

- According to the input data type they can be classified into :
  - -String functions
  - Numeric functions
  - Date time functions

String functions - ASCII()

- The ASCII( ) function allows you to identify the numeric value of the first character in a string works only for single-byte characters (with values from 0 to 255).

```
SELECT ASCII('book'); □ 98.
```

CHAR_LENGTH() & LENGTH()

- The CHAR_LENGTH( ) and CHARACTER_LENGTH( ) functions, which are synonymous, return the number of characters in the specified string.
- The LENGTH( ) function also returns the length of a string, only the length is measured in bytes, rather than characters.

```
SELECT CHAR_LENGTH('cats and dogs'); □ 13
SELECT LENGTH('cats and dogs'); □ 13
```

CHARSET() & COLLATION()

- The CHARSET( ) function identifies the character set used for a specified string

- Using the CHARSET() functions to identify the character set of a string can be useful when you want to find this information quickly, without having to search column, table, database, and system settings.

```
SELECT CHARSET('cats and dogs'); □ latin1
```

CONCAT() & CONCAT_WS()


```
SELECT CONCAT('cats', ' ', 'and', ' ', 'dogs'); # cats and
dogs


SELECT CONCAT_WS(' ', 'cats', 'and', 'dogs');       # cats
and dogs
```

INSTR() & LOCATE()

- Functions identifies where the substring is located in the string and returns the position number.

```
INSTR(<string>, <substring>)
SELECT INSTR('cats and dogs', 'dogs');  10

LOCATE(<substring>, <string>)
SELECT LOCATE('dogs', 'cats and dogs');  10
```

FIND_IN_SET()

- `FIND_IN_SET()` returns the index of str within str_list. Returns 0 if str is not present in str_list, or NULL if either argument is NULL. The index of the first substring is 1.

```
FIND_IN_SET('cow','moose,cow,pig') → 2
FIND_IN_SET('dog','moose,cow,pig') → 0
```

LCASE(), LOWER(), UCASE(), & UPPER()

- `SELECT LOWER('Cats and Dogs');  -> cats and dogs`
- `SELECT UPPER('cats and dogs');  -> CATS AND DOGS`
- `SELECT LEFT('cats and dogs', 4); -> cats`
- `SELECT RIGHT('cats and dogs',4); -> dogs`

REPEAT() & REVERSE()


```
SELECT REPEAT('CatsDogs', 3);  CatsDogsCatsDogsCatsDogs

SELECT REVERSE('bad');  dab
```

SUBSTRING()

- The function, which includes several forms, returns a substring from the identified string.

```
SUBSTRING(<string>, <position>)
SUBSTRING(<string>, <position>, <length>)

SELECT SUBSTRING('cats and dogs', 10);
SELECT SUBSTRING('cats and dogs and more dogs', 10, 4);
```

Numeric Functions

- Numeric functions return NULL if you pass arguments that are out of range or otherwise invalid.

ABS(x)

- Returns the absolute value of x.

ABS(13.5) → 13.5

ABS(-13.5) → 13.5

- `SELECT CEILING(9.327);` → `10`
- `SELECT FLOOR(9.327);` → `9`

Numeric Function

`SIN(x)`

- Returns the sine of x, where x is measured in radians.

`SIN(0) → 0`

`SIN(PI()/2) → 1`

`SQRT()`

- The `SQRT()` function returns to the square root of a specified number:

`SELECT SQRT(36); → 6`

Numeric Functions

`COS(x)`

- Returns the cosine of x, where x is measured in radians.

`COS(0)` → `1`

`COT(x)`

- Returns the cotangent of x, where x is measured in radians.

`COT(PI()/4)` → `1`

Numeric Function

`TAN(x)`

- Returns the tangent of x, where x is measured in radians.

`TAN(0) → 0`

`TAN(PI()/4) → 1`

`TRUNCATE(x,d)`

- Returns the value x, with the fractional part truncated to d decimal places. If d is 0, the result has no decimal point or fractional part. If d is greater than the number of decimal places in x, the fractional part is right-padded with trailing zeros to the desired width.

`TRUNCATE(1.23,1) → 1.2`

`TRUNCATE(1.23,0) → 1`

`TRUNCATE(1.23,4) → 1.2300`

Numeric Functions

`CEILING(x)`

`CEIL(x)`

- Returns the smallest integer not less than x. If the argument has an exact-value numeric type, the return value does, too. Otherwise the return value has a floating point (approximate-value) type. This is true even though the value has no fractional part.
- `CEILING(3.8)` → `4`
- `CEILING(-3.8)` → `-3`

Numeric Functions

`FLOOR(x)`

- Returns the largest integer not greater than x. If the argument has an exact-value numeric type, the return value does, too. Otherwise the return value has a floatingpoint (approximate-value) type. This is true even though the value has no fractional part.
- `FLOOR(3.8)` → 3
- `FLOOR(-3.8)` → -4

ROUND() and TRUNCATE()


```
SELECT ROUND(4.27943, 2); □ 4.28

SELECT TRUNCATE(4.27943, 2); □ 4.27
```

Numeric functions


```
SELECT MOD(22, 7); → 1

SELECT POW(4, 2); → 16

PI() → 3.141593
```

Date/Time Functions

- The ADDDATE() and DATE_ADD() functions, which are synonymous, allow you to add date-related intervals to your date values.

- The following table lists the types that you can specify in the INTERVAL clause and the format for the expression used with that type:

```
ADDDATE(<date>, <days>)
SELECT ADDDATE('2004-11-30 23:59:59', 31);
```

CURDATE(), CURTIME() & NOW()


```
SELECT CURDATE(); → 2004-09-08

SELECT CURTIME(); → 16:07:46

SELECT NOW(); → 2004-09-08 16:08:00
```

The CURDATE(), CURTIME(), and NOW() functions are particularly useful if you need to insert a date in a column that is based on the current date or time.

DATE(), MONTH(), MONTHNAME() &YEAR()


```
SELECT DATE('2004-12-31 23:59:59'); → 2004-12-31
SELECT MONTH('2004-12-31 23:59:59');  → 12
SELECT MONTHNAME('2004-12-31 23:59:59'); → December
SELECT YEAR('2004-12-31 23:59:59'); → 2004.
```

- The DATE(), MONTH(), MONTHNAME(), and YEAR() functions are helpful when you want to retrieve a portion of a date or a related value based on the date and use it in your application. That way, you don't have to store all the month names in your database, but they're readily available when you retrieve an order.

DATEDIFF() & TIMEDIFF()


```
SELECT DATEDIFF('2004-12-31 23:59:59', '2003-12-31
   23:59:59');  366 (because 2004 is a leap year).
SELECT TIMEDIFF('2004-12-31 23:59:59', '2004-12-30
   23:59:59');  24:00:00
```

The DATEDIFF() and TIMEDIFF() functions are useful when you have a table that includes two time/date columns.

DAY(), DAYNAME(), DAYOFWEEK(), and DAYOFYEAR()

- MySQL also allows you to pull day-related values out of date or date/time values.

```
SELECT DAY('2004-12-31 23:59:59'); □ 31
SELECT DAYNAME('2004-12-31 23:59:59'); □ Friday
SELECT DAYOFWEEK('2004-12-31 23:59:59'); □ 6
SELECT DAYOFYEAR('2004-12-31 23:59:59'); □ 366
```

- The DAY(), DAYOFMONTH(), DAYNAME(), DAYOFWEEK(), and DAYOFYEAR() functions can be useful if you want to extract specific types of information from a date.

SECOND(), MINUTE(), HOUR(), and TIME()


```
SELECT SECOND('2004-12-31 23:59:59'); → 59

SELECT MINUTE('2004-12-31 23:59:59'); → 59

SELECT HOUR('2004-12-31 23:59:59'); → 23

SELECT TIME('2004-12-31 23:59:59'); → 23:59:59
```

Stored functions

- Stored functions calculate a value to be returned to the caller for use in expressions, just like built-in functions such as COS() or SIN().

Stored functions


```
delimiter $
CREATE FUNCTION new_function (p_x INT)
RETURNS INT
BEGIN
RETURN (SELECT COUNT(*) FROM tbl WHERE col = p_x);
END$
delimiter ;
```

Stored functions

- You can use p_ as a prefix to distinguish parameter names from other names such as those of tables or columns.
- The function can be invoked just like a built-in function:

```
Select new_function(5);
```

Variables

- Variables are used to store the immediate result.

```
DECLARE variable_name datatype(size) DEFAULT
default_value;
```

- The **variable name** should follow the naming convention and should not be the same name of table or column in a database
- The **data type of the variable**, it can be any primitive type which MySQL supports such as INT, VARCHAR and DATETIME… along with the data type is the size of the variable. When you declare a variable, its initial value is NULL.

Variables in Stored Procedures

- You can also assign the **default value** for the variable by using DEFAULT statement.

```
DECLARE total_sale INT DEFAULT 0

DECLARE x, y INT DEFAULT 0
```

Assigning variables

- Once you declared a variable, you can start using it. To assign other value to a variable you can use
  - **SET** statement
  - **SELECT ... INTO** to assign a query result to a variable.

- Variables scope:
  - A variable has its own scope.
  - If you declare a variable inside a stored procedure, it will be out of scope when the END of stored procedure reached.
  - You can declare two variables or more variables with the same name in different scopes; the variable only is effective in its scope.

Assigning variables


```
DECLARE total_count INT DEFAULT 0 SET total_count = 10;
```

```
DECLARE total_products INT DEFAULT 0 SELECT COUNT(*) INTO
total_products FROM products;
```

Looping in Stored Procedures

- Three main types of looping:
  - WHILE
  - REPEAT

Looping


```
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;
```

While

Looping


```
DECLARE x INT;
DECLARE str VARCHAR(255);
SET x = 1;
SET str = '';
 REPEAT
SET str = CONCAT(str,x,',');
SET x = x + 1;
 UNTIL x > 5
END REPEAT;
SELECT str;
```

Repeat..Until

Triggers

- SQL trigger is an SQL statements or a set of SQL statements which is stored to be activated or fired when an event associating with a database table occurs. The event can be any event including INSERT, UPDATE and DELETE.
- Sometimes a trigger is referred as a special kind of stored procedure in term of procedural code inside its body.
- The difference between a trigger and a stored procedure is that a trigger is activated or called when an event happens in a database table, a stored procedure must be called explicitly.

Create Trigger

General form:

```
CREATE TRIGGER trigger_name

trigger_time trigger_event

ON table_name

FOR EACH ROW BEGIN ... END
```

Triggers

- SQL trigger is an SQL statements or a set of SQL statements which is stored to be activated or fired when an event associating with a database table occurs. The event can be any event including INSERT, UPDATE and DELETE.
- Sometimes a trigger is referred as a special kind of stored procedure in term of procedural code inside its body.
- The difference between a trigger and a stored procedure is that a trigger is activated or called when an event happens in a database table, a stored procedure must be called explicitly.

Create Trigger

- A trigger must be associated with a **specific table.** Without a table trigger does not exist so you have to specify the table name after the ON keyword.
- You can write the **logic between BEGIN and END** block of the trigger.
- MySQL gives you **OLD and NEW** keyword to help you write trigger more efficient.
  - The **OLD** keyword refers to the existing row before you update data and
  - The **NEW** keyword refers to the new row after you update data

Create Trigger

- In order to keep track the changes of last name of employee we can create a trigger that is fired before we make any update on the *employees* table

```
CREATE TABLE employees_audit
(  id int(11) NOT NULL AUTO_INCREMENT,  employeeNumber int(11) NOT
   NULL,  lastname varchar(50) NOT NULL,  changedon datetime DEFAULT
   NULL,  action varchar(50) DEFAULT NULL,  PRIMARY KEY (id)  );


DELIMITER $$
CREATE TRIGGER before_employee_update  BEFORE UPDATE ON employees
 FOR EACH ROW BEGIN
 INSERT INTO employees_audit
SET action = 'update', employeeNumber = OLD.employeeNumber, lastname
   = OLD.lastname, changedon = NOW();
END
$$ DELIMITER ;
```

Managing Triggers

- To remove a trigger you can use the SQL statement

```
DROP TRIGGER trigger_name;
```

- To modify a trigger, you have to delete it first and recreate it. MySQL doesn't provide you SQL statement to alter an existing trigger like altering other database objects such as tables or stored procedures.

Backup Database into file

- Create a dump file using:

```
mysqldump --databases sampdb > sampdb.sql
```

- load the dump file into the MySQL

```
mysql < /tmp/sampdb.sql
```

GUI Tools