UNIT 2.2

Intermediate and Advanced SQL

Outline

- Join Expressions,
- Views,
- Integrity Constraints
- SQL Data TYpes
- Authorization,
- Functions
- Procedures,
- Triggers.

Join Expressions

- A SQL Join statement is used to combine data or rows from two or more tables based on a common field between them.
- Different types of Joins are:
    - Natural join
    - Inner join
    - Outer join
    - Cartesian Join
    - Self join

student(rol_no:number, name: string, address:string, phno:number,age:number)

studentcourse(cid:number,rol_no:number)

Consider the two tables below:

Student

ROLL_NO	NAME	ADDRESS	PHONE	Age
1	HARSH	DELHI	xxxxxxxxxx	18
2	PRATIK	BIHAR	xxxxxxxxxx	19
3	RIYANKA	SILIGURI	xxxxxxxxxx	20
4	DEEP	RAMNAGAR	xxxxxxxxxx	18
5	SAPTARHI	KOLKATA	xxxxxxxxxx	19
6	DHANRAJ	BARABAJAR	xxxxxxxxxx	20
7	ROHIT	BALURGHAT	xxxxxxxxxx	18
8	NIRAJ	ALIPUR	xxxxxxxxxx	19

StudentCourse

COURSE_ID	ROLL_NO
1	1
2	2
2	3
3	4
1	5
4	9
5	10
4	11

Natural Join

- Natural join matches tuples with the same values for all common attributes, and retains only one copy of each common column.

- Syntax

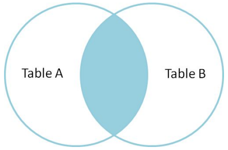
```
SELECT *

FROM TABLE1

NATURAL JOIN TABLE2;
```

Example:

```
SELECT StudentCourse.COURSE_ID,
Student.NAME, Student.AGE
FROM Student
NATURAL JOIN StudentCourse
```

INNER JOIN:

- The INNER JOIN keyword selects all rows from both the tables as long as the condition satisfies
- This keyword will create the result-set by combining all rows from both the tables where the condition satisfies i.e value of the common field will be same.
- Syntax
  - SELECT table1.column1,table1.column2,table2.column1,....
  - FROM table1
  - INNER JOIN table2
  - ON table1.matching_column = table2.matching_column;



- We can also write JOIN instead of INNER JOIN. JOIN is same as INNER JOIN.

```
SELECT StudentCourse.COURSE_ID, Student.NAME, Student.AGE
FROM Student
INNER JOIN StudentCourse
ON Student.ROLL_NO = StudentCourse.ROLL_NO;
```

Output:

COURSE_ID	NAME	Age
1	HARSH	18
2	PRATIK	19
2	RIYANKA	20
3	DEEP	18
1	SAPTARHI	19

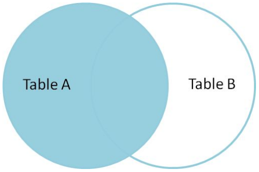
Outer Join

- An extension of the join operation that avoids loss of information.
- Computes the join and then adds tuples form one relation that does not match tuples in the other relation to the result of the join.
- Uses *null* values.
- Three forms of outer join:
  - left outer join
  - right outer join
  - full outer join

LEFT OUTER JOIN:

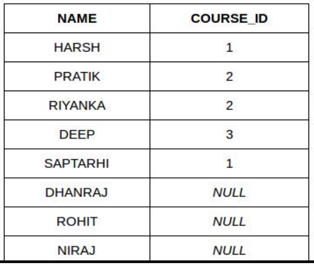
- This join returns all the rows of the table on the left side of the join and matching rows for the table on the right side of join.
- The rows for which there is no matching row on right side, the result-set will contain *null*.
- LEFT JOIN is also known as LEFT OUTER JOIN.
- **Syntax:**

```
SELECT table1.column1,table1.column2,table2.column1,....
FROM table1
LEFT JOIN table2
ON table1.matching_column = table2.matching_column;
```



```
SELECT Student.NAME,StudentCourse.COURSE_ID
FROM Student
LEFT JOIN StudentCourse
  ON StudentCourse.ROLL_NO = Student.ROLL_NO;
```

Output:

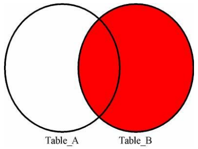


NAME	COURSE_ID
HARSH	1
PRATIK	2
RIYANKA	2
DEEP	3
SAPTARHI	1
DHANRAJ	*NULL*
ROHIT	*NULL*
NIRAJ	*NULL*

RIGHT JOIN:

- This join returns all the rows of the table on the right side of the join and matching rows for the table on the left side of join.
- The rows for which there is no matching row on left side, the result-set will contain *null*.
- RIGHT JOIN is also known as RIGHT OUTER JOIN.
- **Syntax:**

```
SELECT table1.column1,table1.column2,table2.column1,....
FROM table1
RIGHT JOIN table2
ON table1.matching_column = table2.matching_column;
```



```
SELECT Student.NAME,StudentCourse.COURSE_ID
FROM Student
RIGHT JOIN StudentCourse
   ON StudentCourse.ROLL_NO = Student.ROLL_NO;
```

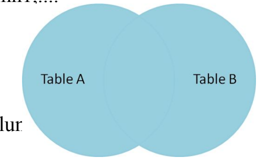
Output:

NAME	COURSE_ID
HARSH	1
PRATIK	2
RIYANKA	2
DEEP	3
SAPTARHI	1
NULL	4
NULL	5
NULL	4

FULL OUTER JOIN:

- FULL JOIN creates the result-set by combining result of both LEFT JOIN and RIGHT JOIN.
- The result-set will contain all the rows from both the tables.
- The rows for which there is no matching, the result-set will contain *NULL* values.
- **Syntax:**

```
SELECT table1.column1,table1.column2,table2.column1,....

FROM table1

FULL JOIN table2

   ON table1.matching_column = table2.matching_colur
```



```
SELECT Student.NAME,StudentCourse.COURSE_ID
FROM Student
FULL JOIN StudentCourse
   ON StudentCourse.ROLL_NO = Student.ROLL_NO;
```

Output:

NAME	COURSE_ID
HARSH	1
PRATIK	2
RIYANKA	2
DEEP	3
SAPTARHI	1
DHANRAJ	*NULL*
ROHIT	*NULL*
NIRAJ	*NULL*
NULL	9
NULL	10
NULL	11

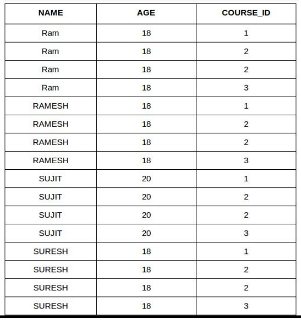
CARTESIAN JOIN:

- The CARTESIAN JOIN is also known as CROSS JOIN.
- In a CARTESIAN JOIN there is a join for each row of one table to every row of another table.
- This usually happens when the matching column or WHERE condition is not specified.
  - In the absence of a WHERE condition the CARTESIAN JOIN will behave like a CARTESIAN PRODUCT . i.e., the number of rows in the result-set is the product of the number of rows of the two tables.
  - In the presence of WHERE condition this JOIN will function like a INNER JOIN.
- **Syntax:**

```
SELECT table1.column1 , table1.column2, table2.column1...FROM table1   CROSS JOIN
table2;
```

```
SELECT Student.NAME, Student.AGE, StudentCourse.COURSE_ID

FROM Student

   CROSS JOIN StudentCourse;
```

Output:



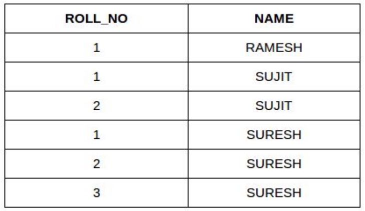
NAME	AGE	COURSE_ID
Ram	18	1
Ram	18	2
Ram	18	2
Ram	18	3
RAMESH	18	1
RAMESH	18	2
RAMESH	18	2
RAMESH	18	3
SUJIT	20	1
SUJIT	20	2
SUJIT	20	2
SUJIT	20	3
SURESH	18	1
SURESH	18	2
SURESH	18	2
SURESH	18	3

SELF JOIN:

- in SELF JOIN a table is joined to itself.
- Each row of the table is joined with itself and all other rows depending on some conditions.
- it is a join between two copies of the same table.
- **Syntax:**
    - SELECT a.coulmn1 , b.column2
    - FROM table_name a, table_name b
    - WHERE some_condition;

```
SELECT a.ROLL_NO , b.NAME
FROM Student a, Student b
   WHERE a.ROLL_NO < b.ROLL_NO;
```

Output:



ROLL_NO	NAME
1	RAMESH
1	SUJIT
2	SUJIT
1	SURESH
2	SURESH
3	SURESH

Views

- A **view** provides a mechanism to hide certain data from the view of certain users.
- In SQL, a view is a virtual table based on the result-set of an SQL statement.
- A view contains rows and columns, just like a real table. The fields in a view are fields from one or more real tables in the database.
- A view is created with the CREATE VIEW statement.
- Syntax:
    - CREATE VIEW *view_name* AS
    - SELECT *column1*, *column2*, ...
    - FROM *table_name*
    - WHERE *condition*;

- Example:
- CREATE VIEW Eligble_students
  AS
- SELECT sid,name, cgpa
- FROM Student
- WHERE cgpa > 9.0;

- To see the data in the View, we can query the view in the same manner as we query a table.
  - SELECT *

    FROM

    Eligble_students;

- Example2:
- CREATE VIEW Eligble_students2 AS
- SELECT sid,name,
- FROM Student
- Order by cgpa desc;

- **UPDATING VIEWS**
  - CREATE OR REPLACE VIEW view_name AS
  - SELECT column1,coulmn2,..
  - FROM table_name
  - WHERE condition;
- Most SQL implementations allow updates only on simple views
  - The **from** clause has only one database relation.
  - The **select** clause contains only attribute names of the relation, and does not have any expressions, aggregates, or **distinct** specification.
  - Any attribute not listed in the **select** clause can be set to null
  - The query does not have a **group** by or **having** clause.

- **Inserting a row in a view**
  - INSERT INTO view_name(column1, column2 , column3,..)

    VALUES(value1, value2, value3..);
- **Deleting a row from a View**
  - DELETE FROM view_name

    WHERE condition;

- We can delete or drop a View using the DROP statement.
  - DROP VIEW view_name;

Integrity Constraints

- Integrity constraints guard against accidental damage to the database, by ensuring that authorized changes to the database do not result in a loss of data consistency.
  - A checking account must have a balance greater than $10,000.00
  - A salary of a bank employee must be at least $4.00 an hour
  - A customer must have a (non-null) phone number

Constraints on a Single Relation

- **not null**
- **primary key**
- **unique**
- **check** (P), where P is a predicate

- **not null**
  - Declare *name* and *budget* to be **not null**

    name **varchar**(20) **not null**
    budget **numeric**(12,2) **not null**

- **unique** ($A_1, A_2, \ldots, A_m$)
  - The unique specification states that the attributes $A_1, A_2, \ldots, A_m$ form a candidate key.
  - Candidate keys are permitted to be null (in contrast to primary keys).

- The PRIMARY KEY constraint uniquely identifies each record in a table.
- Primary keys must contain UNIQUE values, and cannot contain NULL values.
- A table can have only ONE primary key; and in the table, this primary key can consist of single or multiple columns (fields).
- Example:
  - ○ CREATE TABLE Persons (ID int,LastName varchar(255) NOT NULL,FirstName varchar(255),Age int, PRIMARY KEY (ID));
  - ○ CREATE TABLE Persons (ID int , LastName varchar(255) NOT NULL, FirstName varchar(255), Age int, CONSTRAINT PK_Person PRIMARY KEY (ID,LastName));

The check clause

- The **check** (P) clause specifies a predicate P that must be satisfied by every tuple in a relation.
- Example: ensure that semester is one of fall, winter, spring or summer

```
create table section
(course_id varchar (8),
 sec_id varchar (8),
 semester varchar (6),
 year numeric (4,0),
 building varchar (15),
 room_number varchar (7),
 time_slot_id varchar (4),
 primary key (course_id, sec_id, semester, year),
 check (semester in ('Fall', 'Winter', 'Spring', 'Summer')))
```

Referential Integrity

- Ensures that a value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation.
  - Example: If "CSE" is a department name appearing in one of the tuples in the *instructor* relation, then there exists a tuple in the *department* relation for "CSE".
- Let A be a set of attributes.

  Let R and S be two relations that contain attributes A and where A is the primary key of S.

  A is said to be a **foreign key** of R if for any values of A appearing in R these values also appear in S.

- Foreign *keys can be* specified as part of the SQL **create table** statement

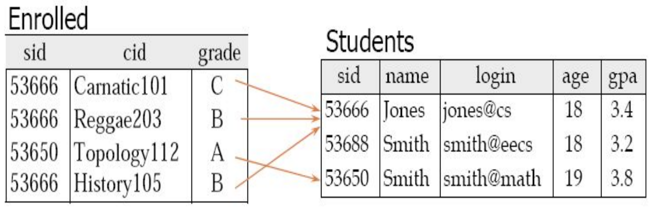
  foreign key (*dept_name*) **references** *department*
- By default, a foreign key references the primary-key attributes of the referenced table.
- SQL allows a list of attributes of the referenced relation to be specified explicitly.

  foreign key (*dept_name*) **references** *department* (*dept_name*)
- Like a `logical pointer'.
- If one of the relations is modified, the other must be checked, and perhaps modified, to keep the data consistent

---

- Only students listed in the Students relation should be allowed to enroll for courses
- CREATE TABLE Enrolled (

```
    sid CHAR(20),
     cid CHAR(20),
    grade CHAR(2),
     PRIMARY KEY  (sid,cid),
    FOREIGN KEY (sid) REFERENCES Students);
```



Enrolled

sid	cid	grade
53666	Carnatic101	C
53666	Reggae203	B
53650	Topology112	A
53666	History105	B

Students

sid	name	login	age	gpa
53666	Jones	jones@cs	18	3.4
53688	Smith	smith@eecs	18	3.2
53650	Smith	smith@math	19	3.8

- In translating a relationship set to a relation, attributes of the relation must include:
  - Keys for each participating entity set (as foreign keys).
    - This set of attributes forms a *superkey* for the relation.
  - All descriptive attributes

```
CREATE TABLE Works_In(
 ssn  CHAR(1),
 did  INTEGER,
 since  DATE,
 PRIMARY KEY (ssn, did),
 FOREIGN KEY (ssn)
    REFERENCES Employees,
 FOREIGN KEY (did)
    REFERENCES Departments)
```

ENFORCING INTEGRITY CONSTRAINTS

- Consider Students and Enrolled; *sid* in Enrolled is a foreign key that references Students.
- What should be done if an Enrolled tuple with a non-existent student id is inserted? (*Reject it!*)
- What should be done if a Students tuple is deleted?
  - Also delete all Enrolled tuples that refer to it.
  - Disallow deletion of a Students tuple that is referred to.
  - Set sid in Enrolled tuples that refer to it to a *default sid*.
- Similar if primary key of Students tuple is updated.

- INSERT INTO Students (sid, name, login, age, gpa) VALUES (53688, `Mike', `mike@ee', 17, 3.4);
  - violates the primary key constraint as duplicates are not allowed
- INSERT INTO Students (sid, name, login, age, gpa) VALUES (*null, `Mike', `mike@ee', 17, 3.4)*
  - violates the constraint that the primary key cannot contain null:
- UPDATE Students S SET S.sid = 50000 WHERE S.sid = 53688
  - violates the constraint

Cascading Actions in Referential Integrity

```
CREATE TABLE Enrolled (
  sid CHAR(20) default ‘11111’,
cid CHAR(20),
 grade CHAR(2),
 PRIMARY KEY  (sid,cid),
FOREIGN KEY (sid)
REFERENCES Students
  ON DELETE CASCADE
  ON UPDATE SET DEFAULT )
```

NO ACTION, which means that the action (DELETE or UPDATE) is to be rejected

CASCADE keyword says that if a Students row is deleted, all Enrolled rows that refer to it are to be deleted as well.

If a Students row is deleted, we can switch the enrollment to a `default' student by using ON DELETE SET DEFAULT.

Built-in Data Types in SQL

- **date:** Dates, containing a (4 digit) year, month and date
  - Example: **date** '2005-07-27'
- **time:** Time of day, in hours, minutes and seconds.
  - Example: **time** '09:00:30' **time** '09:00:30.75'
- **timestamp:** date plus time of day
  - Example: **timestamp** '2005-7-27 09:00:30.75'
- **interval:** period of time
  - Example: interval '1' day
  - Subtracting a date/time/timestamp value from another gives an interval value
  - Interval values can be added to date/time/timestamp values

Date and Time functions

- ADDDATE(date,INTERVAL expr unit): to add days to a date
  - SELECT DATE_ADD('1998-01-02', INTERVAL 31 DAY);
- ADDTIME(expr1,expr2) : adds expr2 to expr1 and returns the result.
  - The expr1 is a time or datetime expression, while the expr2 is a time expression.
  - SELECT ADDTIME('1997-12-31 23:59:59.999999','1 1:1:1.000002');
- CURDATE(): Returns the current date as a value in 'YYYY-MM-DD'
  - SELECT CURDATE();
- CURTIME() :Returns the current time as a value in 'HH:MM:SS'
  - SELECT CURTIME();
- DATEDIFF(expr1,expr2):Returns differences between two dates
  - Both expr1 and expr2 are date or date-and-time expressions
  - SELECT DATEDIFF('1997-12-31 23:59:59','1997-12-30');

- SELECT DATE_FORMAT('1997-10-04 22:23:00', '%W %M %Y');
  - Returns Saturday October 1997
- DAYNAME(date):Returns the name of the weekday for date.
  - SELECT DAYNAME('1998-02-05');
- DAYOFMONTH(date) Returns the day of the month for date, in the range 1 to 31.
- DAYOFWEEK(date):Returns the weekday index for date (1 = Sunday, 2 = Monday, ., 7 = Saturday).
- DAYOFYEAR(date):Returns the day of the year for date, in the range 1 to 366.
- EXTRACT(unit FROM date): returns the unit from date
  - SELECT EXTRACT(YEAR FROM '1999-07-02');
- MINUTE(time): Returns the minute for time, in the range 0 to 59.
  - SELECT MINUTE('98-02-03 10:05:03');
- HOUR(time):Returns the hour for time.
- SECOND(time):Returns the second for time, in the range 0 to 59.
-

- MONTH(date):Returns the month for date, in the range 1 to 12.
- MONTHNAME(date): Returns the full name of the month for a date.
- YEAR(date): Returns the year for date, in the range 1000 to 9999
- QUARTER(date):Returns the quarter of the year for date, in the range 1 to 4.
- STR_TO_DATE(str,format): Convert string to date format
  - SELECT STR_TO_DATE('04/31/2004', '%m/%d/%Y');
- SUBDATE(date,INTERVAL expr unit)
  - SELECT DATE_SUB('1998-01-02', INTERVAL 31 DAY);
- SYSDATE():Returns the current date and time as a value in 'YYYY-MM-DD HH:MM:SS'
- TIME(expr): Extracts the time part of the time or datetime expression expr and returns it as a string.
  - SELECT TIME('2003-12-31 01:02:03');
- TIMEDIFF(expr1,expr2)
  - SELECT TIMEDIFF('1997-12-31 23:59:59.000001', '1997-12-30 01:01:01.000002');
- TIME_TO_SEC(time): Returns the time argument converted to seconds.
- TO_DAYS(date):Given a date, returns a day number (the number of days since year 0).
-

Large-Object Types

- Large objects (photos, videos, CAD files, etc.) are stored as a *large object*:
  - ○ **blob**: binary large object -- object is a large collection of uninterpreted binary data (whose interpretation is left to an application outside of the database system)
  - ○ **clob**: character large object -- object is a large collection of character data
- When a query returns a large object, a pointer is returned rather than the large object itself.

User-Defined Types

- **create type** construct in SQL creates user-defined type
  - **create type** *Dollars* **as numeric (12,2) final**
- Example:
- **create table** *department*
  (*dept_name* **varchar** (20),
  building **varchar** (15),
  budget Dollars);

Domains

- **create domain** construct in SQL-92 creates user-defined domain types

  create domain *person_name* **char**(20) **not null**

- Types and domains are similar. Domains can have constraints, such as **not null**, specified on them.
- Example:

  create domain *degree_level* **varchar**(10)
  constraint *degree_level_test*
  check (**value in** ('Bachelors', 'Masters', 'Doctorate'));

Index Creation

- Indexes are used to retrieve data from the database more quickly than otherwise.
- CREATE INDEX *index_name* ON *table_name* (*column1*, *column2*, ...);

Authorization

- We may assign a user several forms of authorizations on parts of the database.
    - Authorization to read data.
    - Authorization to insert new data.
    - Authorization to update data.
    - Authorization to delete data
- **Each** of these types of authorizations is called a **privilege**.
- We may authorize the user all, none, or a combination of these types of privileges on specified parts of a database, such as a relation or a view.

- The **grant** statement is used to confer authorization
- **grant** <privilege list> **on** <relation or view > **to** <user list>
- <user list> is:
    - a user-id
    - **public**, which allows all valid users the privilege granted
    - A role (more on this later)
- Example:
    - **grant select on** *department* **to** Amit, Santoshi
- Granting a privilege on a view does not imply granting any privileges on the underlying relations.
- The grantor of the privilege must already hold the privilege on the specified item (or be the database administrator).

Privileges in SQL

- **select**: allows read access to relation, or the ability to query using the view
    - Example: grant users U_1, U_2, and U_3 **select** authorization on the *instructor* relation:
    - **grant select on** *instructor* **to** U_1, U_2, U_3
- **insert**: the ability to insert tuples
- **update**: the ability to update using the SQL update statement
- **delete**: the ability to delete tuples.
- **all privileges**: used as a short form for all the allowable privileges

---

Revoking Authorization in SQL

- The **revoke** statement is used to revoke authorization.
    - **revoke** <privilege list> **on** <relation or view> **from** <user list>
- Example:
    - **revoke select on** *student* **from** U_1, U_2, U_3
- <privilege-list> may be **all** to revoke all privileges the revokee may hold.
- If <revokee-list> includes **public,** all users lose the privilege except those granted it explicitly.
- If the same privilege was granted twice to the same user by different grantees, the user may retain the privilege after the revocation.

Roles

- A **role** is a way to distinguish among various users as far as what these users can access/update in the database.
- To create a role we use:

  create a role <name>
- Example:
  - **create role** instructor
- Once a role is created we can assign “users” to the role using:
  - **grant** <role> **to** <users>

Roles Example

- **create role** instructor;
- **grant** *instructor* **to** Amit**;**
- Privileges can be granted to roles:
  - **grant select on** *takes* **to** *instructor*;
- Roles can be granted to users, as well as to other roles
  - **create role** *teaching_assistant*
  - **grant** *teaching_assistant* **to** *instructor*;
    - *Instructor* inherits all privileges of *teaching_assistant*
- Chain of roles
  - **create role** *dean*;
  - **grant** *instructor* **to** *dean*;
  - **grant** *dean* **to** Satoshi;

- SQL includes a references privilege that permits a user to declare foreign keys when creating relations
  - grant references (dept name) on department to Ravi;
- A user who has been granted some form of authorization may be allowed to pass on this authorization to other users
  - grant select on department to Amit with grant option;
    - Amit the select privilege on department and allow Amit to grant this privilege to others
-

TCL command

- Transaction Control Language(TCL) commands are used to manage transactions in database
- **Commit command**
  - Commit command is used to permanently save any transaction into database.
  - commit;
- **Rollback command**
  - This command restores the database to last committed state.
  - It is also use with savepoint command to jump to a savepoint in a transaction.
  - rollback to savepoint-name;
- **Savepoint command**
  - **savepoint** command is used to temporarily save a transaction so that you can rollback to that point whenever necessary.

Stored Procedures

- A procedure (often called a stored procedure) is a collection of pre-compiled SQL statements stored inside the database.
- It is a subroutine or a subprogram in the regular computing language.
- A procedure always contains a name, parameter lists, and SQL statements.
- We can invoke the procedures by using triggers, other procedures and applications such as Java, Python, PHP, etc.

- Stored Procedure Features
  - Stored Procedure increases the performance of the applications.
  - Once stored procedures are created, they are compiled and stored in the database.
  - Stored procedure reduces the traffic between application and database server.
    - Because the application has to send only the stored procedure's name and parameters instead of sending multiple SQL statements.
  - Stored procedures are reusable and transparent to any applications.
  - A procedure is always secure.
    - The database administrator can grant permissions to applications that access stored procedures in the database

Syntax

```
DELIMITER //
CREATE PROCEDURE procedure_name ([IN | OUT | INOUT] parameter_name dataty
pe [, parameter_name datatype])
BEGIN
   Declaration_section
   Executable_section
END ;//
DELIMITER ;
```

- **Procedure_name**
  - It represents the name of the stored procedure.
- **Parameter_name**
  - It represents the number of parameters. It can be one o more than one.
- **Declaration_section**
  - It represents the declarations of all variables.
- **Executable_section**
  - It represents the code for the function execution.

- **IN parameter**
  - It is the default mode.
  - It takes a parameter as input, such as an attribute.
  - When we define it, the calling program has to pass an argument to the stored procedure.
  - This parameter's value is always protected.
- **OUT parameters**
  - It is used to pass a parameter as output.
  - Its value can be changed inside the stored procedure, and the changed value is passed back to the calling program.
- **INOUT parameters**
  - It is a combination of IN and OUT parameters.
  - calling program can pass the argument,
  - and the procedure can modify the INOUT parameter,
  - and then passes the new value back to the calling program.

- We can use the **CALL statement** to call a stored procedure.
  - This statement returns the values to its caller through its parameters (IN, OUT, or INOUT).
  - The following syntax is used to call the stored procedure in MySQL:
    - **CALL procedure_name (parameter(s))**

- MySQL also allows a command to drop the procedure.
  - When the procedure is dropped, it is removed from the database server also.
  - The following statement is used to drop a stored procedure in MySQL:
    - **DROP PROCEDURE [IF EXISTS] procedure_name;**

- DECLARE Local Variables
  - DECLARE var_name[,...] type [DEFAULT value]
- Variable SET Statement
  - SET var_name = expr [, var_name = expr] …
- SELECT ... INTO Statement
  - SELECT col_name[,...] INTO var_name[,...] from table_name where condition;

Procedure to find addition of two numbers using IN parameters

```
mysql> delimiter //
mysql> create procedure adddemo(in a int,in b int)
    -> begin
    -> declare c int;
    -> set @c=a+b;
    -> end;
    -> //
Query OK, 0 rows affected (0.03 sec)
```

input:

```
mysql> set @a=12
    -> //
Query OK, 0 rows affected (0.01 sec)

mysql> set @b=35
    -> //
Query OK, 0 rows affected (0.00 sec)
```

Call the procedure:

```
mysql> call adddemo(@a,@b);
    -> //
Query OK, 0 rows affected (0.01 sec)

mysql> select @c;
    -> //
```

ouput

```
+------+
| @c   |
+------+
|   47 |
+------+
1 row in set (0.00 sec)
```

```
create procedure addtwonos2(out c int)
begin
declare a int;
declare b int;
set a=5;
set b=6;
set c=a+b;
end;
```

```
mysql> call addtwonos2(@c);
    -> //
Query OK, 0 rows affected (0.00 sec)

mysql> select @c;
    -> //
+------+
| @c   |
+------+
|   11 |
+------+
1 row in set (0.00 sec)
```

```
mysql> CREATE PROCEDURE p (OUT cgpa_var decimal(2,1), INOUT incr_param INT)
   -> BEGIN
   ->   SELECT cgpa INTO cgpa_var from student where sid='182';
   ->   SET incr_param = incr_param + 1;
   -> END;
   -> //
Query OK, 0 rows affected (0.01 sec)


  mysql> set @a=21;
     -> //
  Query OK, 0 rows affected (0.00
  sec)

  mysql> select @O,@a;
     -> //
  +------------+------+
  | @O         | @a   |
  +------------+------+
  | NULL       |   21 |
  +------------+------+
  1 row in set (0.00 sec)
```

```
CREATE PROCEDURE pdata2 (in ht varchar(20),OUT cgpa_var decimal(2,1), INOUT incr_param INT)
  BEGIN
     SELECT cgpa INTO cgpa_var from student where sid=ht;
   SET incr_param = incr_param + 1;
   END;
    //
```

Functions

- In Sql, Function can also be created.
- A function always returns a value using the return statement.
- The function can be used in SQL queries.

Syntax:

```
CREATE FUNCTION function_name  (parameter datatype [, parameter datatype])
RETURNS return_datatype  determinstic
BEGIN
Declaration_section
Executable_section
END;
```

Function_name: name of the function

Parameter: number of parameter. It can be one or more than one.

return_datatype: return value datatype of the function

declaration_section: all variables are declared.

executable_section: code for the function is written here.

Drop a function

- When A function is dropped, it is removed from the database.
- Syntax:
  - **Drop function** [IF EXISTS] function_name;

A ***deterministic function*** always returns the same results if given the same input values.

A ***nondeterministic function*** may return different results every time it is called, even when the same input values are provided.

```
create function addtwonos(a int,b int)
returns int deterministic
return a+b;
//
Execution:
Select addtwonos(12,13);
//
+-------------------+
| addtwonos(12,13) |
+-------------------+
|               25 |
+-------------------+
1 row in set (0.01 sec)
```

```
create function addtwo(a int,b int)
    returns int deterministic
  begin
  declare c int;
    set c=a+b;
    return c;
end;
    //
Execution:
Select  addtwo(12,31);
//
+---------------+
| addtwo(12,31) |
+---------------+
|            43 |
+---------------+
1 row in set (0.01 sec)
```

```
create function dept_count (dept_name varchar(20))
                  returns integer determinstic
                 begin
                 declare d_count  integer;
                       select count (* ) into d_count
                       from instructor
                       where instructor.dept_name = dept_name
                 return d_count;
          end
```

IF Statement:

Syntax:

IF search_condition THEN statement_list

[ELSEIF search_condition THEN statement_list] ...

[ELSE statement_list]

END IF

Example:

```
CREATE FUNCTION SimpleCompare(n INT, m INT)
  RETURNS VARCHAR(20) deterministic

  BEGIN
    DECLARE s VARCHAR(20);

    IF n > m THEN SET s = '>';
    ELSEIF n = m THEN SET s = '=';
    ELSE SET s = '<';
    END IF;

    SET s = CONCAT(n, ' ', s, ' ', m);

    RETURN s;
  END //
```

- **LOOP Statement**

  [begin_label:] LOOP

    statement_list

  END LOOP [end_label]

- **LEAVE Statement**

  LEAVE label

  - This statement is used to exit any labeled flow control construct. It can be used within BEGIN ... END or loop constructs (LOOP, REPEAT, WHILE).

- **ITERATE Statement**

  ITERATE label

  - ITERATE can appear only within LOOP, REPEAT, and WHILE statements. ITERATE means “do the loop again.”

```
CREATE PROCEDURE doiterate(p1 INT)
BEGIN
  label1: LOOP
    SET p1 = p1 + 1;
    IF p1 < 10 THEN ITERATE label1;
 END IF;
    LEAVE label1;
  END LOOP label1;
  SET @x = p1;
END
```

```
CREATE PROCEDURE test_mysql_loop()
 BEGIN
 DECLARE x  INT;
        DECLARE str  VARCHAR(255);

 SET x = 1;
        SET str =  '';

 loop_label:  LOOP
 IF  x > 10 THEN
 LEAVE  loop_label;
 END  IF;

 SET  x = x + 1;
 IF  (x mod 2) THEN
 ITERATE  loop_label;
 ELSE
             SET  str = CONCAT(str,x,',');
 END  IF;
        END LOOP;

        SELECT str;

 END;
```

REPEAT Statement:

Syntax:

The statement list within a REPEAT statement is repeated until the search_condition is true.
Thus, a REPEAT always enters the loop at least once

```
[begin_label:] REPEAT

    statement_list

UNTIL search_condition

END REPEAT [end_label]
```

```
CREATE PROCEDURE dorepeat(p1 INT)
     BEGIN
      SET @x = 0;
      REPEAT SET @x = @x + 1;
 UNTIL @x > p1
END REPEAT;
    END
    //
```

WHILE Statement:
Syntax:

```
[begin_label:] WHILE search_condition DO
    statement_list
END WHILE [end_label]
```

The statement list within a WHILE statement is repeated as long as the search_condition is true.

```
CREATE PROCEDURE whiledemo()
BEGIN
  set @x=10;

  WHILE @x > 0 DO
      SET @x = @x - 1;
  END WHILE;
END
```

```
CREATE PROCEDURE
test_mysql_while_loop()
          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;
```

Differences between Function and Procedure

Function	Procedure
Used mainly to perform some calculation	Used mainly to a execute certain process
A Function that contains no DML statements can be called in SELECT statement	Cannot call in SELECT statement
Use RETURN to return the value	Use OUT parameter to return the value
It is mandatory to return the value	It is not mandatory to return the value
RETURN will exit the control from subprogram and also returns the value	RETURN will simply exit the control from subprogram.
Return data type is mandatory at the time of creation	Return data type will not be specified at the time of creation

Triggers

- A **database trigger** is a procedural code that is automatically executed in response to certain events on a particular table or view in database.
- Analog to a “daemon” that monitors a database from certain events to occur.
- Each trigger is associated with a table, which is activated on any DML statement such as INSERT, UPDATE, or DELETE.

Triggers are of two types

- **Row-Level Trigger:** which is activated for each row by a triggering statement such as insert, update, or delete.
  - For example, if a table has inserted, updated, or deleted multiple rows, the row trigger is fired automatically for each row affected by the insert, update, or delete statement.
- **Statement-Level Trigger:**which is fired once for each event that occurs on a table regardless of how many rows are inserted, updated, or deleted.
- We can create a new trigger in MySQL by using the CREATE TRIGGER statement.

<u>Syntax</u> to create a trigger:

```
CREATE TRIGGER trigger_name  trigger_time trigger_event  ON table_name
FOR EACH ROW
BEGIN
   --variable declarations
   --trigger code
END;
```

- **trigger_name:** It is the name of the trigger that we want to create
- **trigger_time:** It is the trigger action time, which should be either BEFORE or AFTER.
- **trigger_event:** It is the type of operation name that activates the trigger.(insert, update,delete)

- **table_name**: It is the name of the table to which the trigger is associated.
- **BEGIN END Block**: Finally, we will specify the statement for execution when the trigger is activated
- The trigger body can access the column's values, which are affected by the DML statement.
  - The NEW and OLD modifiers are used to distinguish the column values BEFORE and AFTER the execution of the DML statement.
  - We can use the column name with NEW and OLD modifiers as OLD.col_name and NEW.col_name.
  - The OLD.column_name indicates the column of an existing row before the updation or deletion occurs.
  - NEW.col_name indicates the column of a new row that will be inserted or an existing row after it is updated.

We can define the maximum six types of actions or events in the form of triggers:

- **Before Insert:** It is activated before the insertion of data into the table.
- **After Insert:** It is activated after the insertion of data into the table.
- **Before Update:** It is activated before the update of data in the table.
- **After Update:** It is activated after the update of the data in the table.
- **Before Delete:** It is activated before the data is removed from the table.
- **After Delete:** It is activated after the deletion of data from the table.

DROP TRIGGER

To delete the triggers from the database drop trigger command is used
Syntax:

- Drop trigger <trigger_name>;
  - Ex:
    Drop trigger t1;

Row – level trigger

- Execute once for each row
- They are often used in data auditing applications.
- Row-level trigger is identified by the **FOR EACH ROW** clause in the CREATE TRIGGER command.

Statement – level trigger

- Execute once for each transaction
- They are often used to enforce additional security measures
- Statement-level triggers are the default type of triggers created

- **CREATE TABLE** employee(**name varchar**(45) NOT NULL,occupation **varchar**(35)

  NOT NULL, working_date **date**, working_hours **varchar**(10));

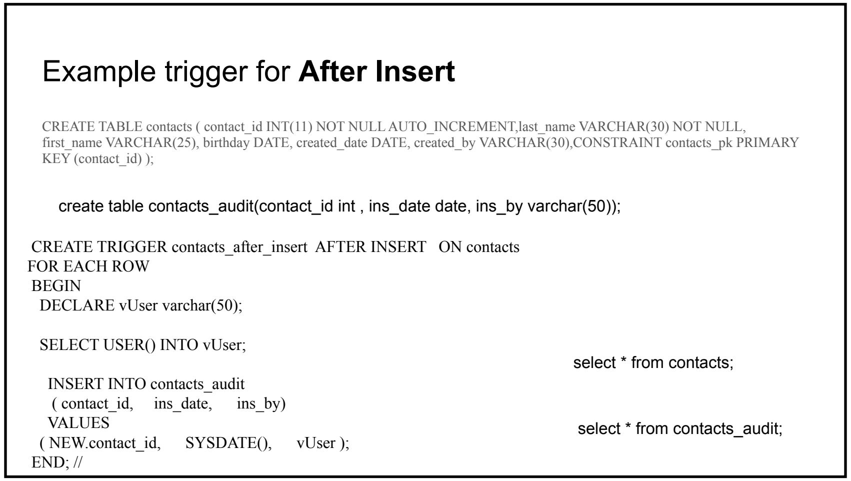
- **INSERT INTO** employee **VALUES** ('Rohith', 'Scientist', '2020-10-04', 12),

  ('Raj', 'Engineer', '2020-10-04', 10), ('Rajani', 'Actor', '2020-10-04', -13), ('Rithu', 'Doctor', '2020-10-04', 14),

- **<u>Example trigger for before insert:</u>**

```
Create Trigger before_insert_empworkinghours  BEFORE INSERT ON employee
FOR EACH ROW
BEGIN
IF NEW.working_hours < 0 THEN SET NEW.working_hours = 0;
 END IF;
  END //
```

- Now insert the row as shown below:
  INSERT INTO employee VALUES   ('Alexander', 'Actor', '2020-10-012', -13);



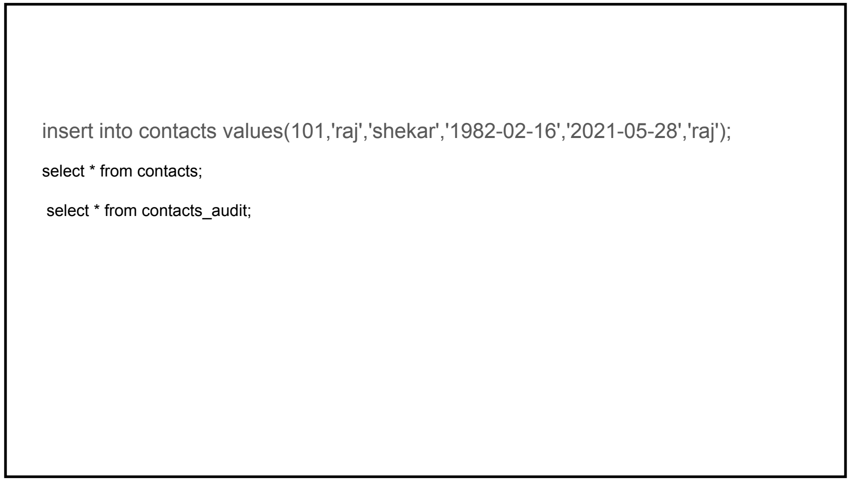
Example trigger for **After Insert**

CREATE TABLE contacts (contact_id INT(11) NOT NULL AUTO_INCREMENT,last_name VARCHAR(30) NOT NULL, first_name VARCHAR(25), birthday DATE, created_date DATE, created_by VARCHAR(30),CONSTRAINT contacts_pk PRIMARY KEY (contact_id));

```
create table contacts_audit(contact_id int , ins_date date, ins_by varchar(50));

 CREATE TRIGGER contacts_after_insert  AFTER INSERT   ON contacts
FOR EACH ROW
 BEGIN
   DECLARE vUser varchar(50);

   SELECT USER() INTO vUser;

     INSERT INTO contacts_audit
      ( contact_id,     ins_date,      ins_by)
     VALUES
   ( NEW.contact_id,      SYSDATE(),      vUser );
 END; //
```

select * from contacts;

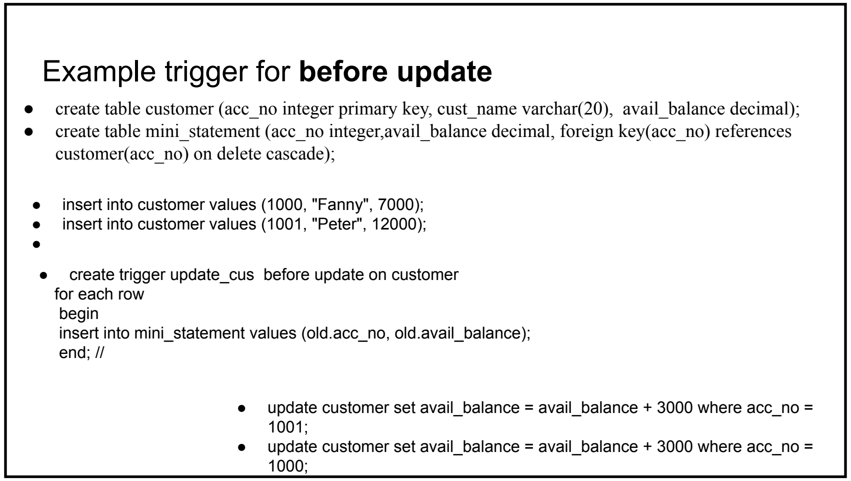
select * from contacts_audit;



insert into contacts values(101,'raj','shekar','1982-02-16','2021-05-28','raj');

select * from contacts;

select * from contacts_audit;



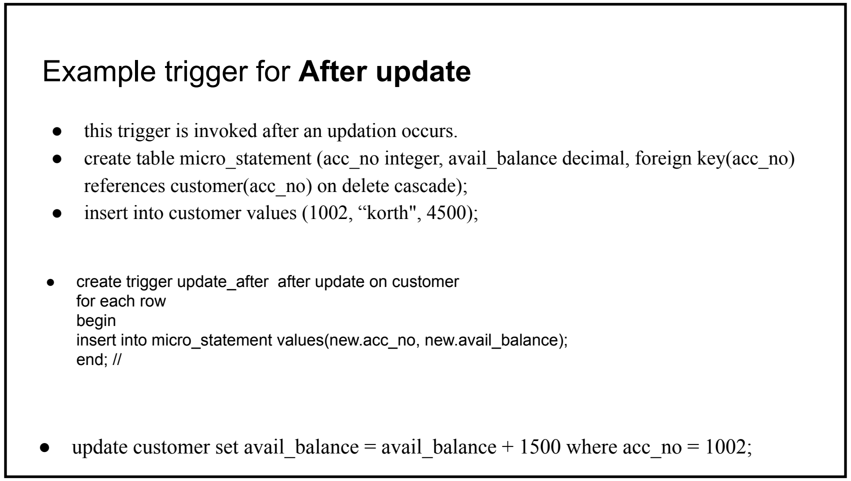
Example trigger for **before update**

- create table customer (acc_no integer primary key, cust_name varchar(20), avail_balance decimal);
- create table mini_statement (acc_no integer,avail_balance decimal, foreign key(acc_no) references customer(acc_no) on delete cascade);

- insert into customer values (1000, "Fanny", 7000);
- insert into customer values (1001, "Peter", 12000);
-

- create trigger update_cus before update on customer
  for each row
  begin
  insert into mini_statement values (old.acc_no, old.avail_balance);
  end; //

- update customer set avail_balance = avail_balance + 3000 where acc_no = 1001;
- update customer set avail_balance = avail_balance + 3000 where acc_no = 1000;



Example trigger for **After update**

- this trigger is invoked after an updation occurs.
- create table micro_statement (acc_no integer, avail_balance decimal, foreign key(acc_no) references customer(acc_no) on delete cascade);
- insert into customer values (1002, “korth", 4500);

- create trigger update_after after update on customer
  for each row
  begin
  insert into micro_statement values(new.acc_no, new.avail_balance);
  end; //

- update customer set avail_balance = avail_balance + 1500 where acc_no = 1002;

Example trigger for **Before delete**

- CREATE TABLE emp (id int,name varchar(10),salary int,PRIMARY KEY (id));
- CREATE TABLE emp_log (name varchar(10),salary int);
- insert into emp values(100,'Raj',55000),(200,’swathi’',50000),(300,'Ravi',45000);

```
CREATE TRIGGER  B_DEL     BEFORE DELETE  ON emp
 FOR EACH ROW
         BEGIN
           INSERT INTO emp_log values(upper(old.name),old.salary);
END;
```

- DELETE from emp where id=200;
  - select * from emp;
  - select * from emp_log;

Example trigger for **After delete**

```
CREATE TRIGGER  AFT_DEL  AFTER DELETE  ON emp
FOR EACH ROW
         BEGIN
   INSERT INTO emp_log values(upper(old.name),old.salary);
END; //
```

- DELETE from emp where id=300;
  - select * from emp;
  - select * from emp_log;