

# DBMS Chap 4

Advanced SQL with integrity, Security and  
Authorization

# Nested Sub Queries

A subquery is a query, which is nested into another SQL query and embedded with SELECT, INSERT, UPDATE or DELETE statement along with the various operators.

We can also nest the subquery with another subquery.

A subquery is known as the **inner query**, and the query that contains subquery is known as the **outer query**.

The inner query executed first gives the result to the outer query, and then the main/outer query will be performed.

MySQL allows us to use subquery anywhere, but it must be closed within parenthesis.

All subquery forms and operations supported by the SQL standard will be supported in MySQL also.

# Rules to use sub-queries

- Subqueries should always use in **parentheses**.
- If the main query does not have multiple columns for subquery, then a subquery can have only one column in the SELECT command.
- We can use various comparison operators with the subquery, such as >, <, =, IN, ANY, SOME, and ALL. A multiple-row operator is very useful when the subquery returns more than one row.
- We cannot use the **ORDER BY** clause in a subquery, although it can be used inside the main query.
- If we use a subquery in a **set function**, it cannot be immediately enclosed in a set function.

# Advantages of using sub-queries

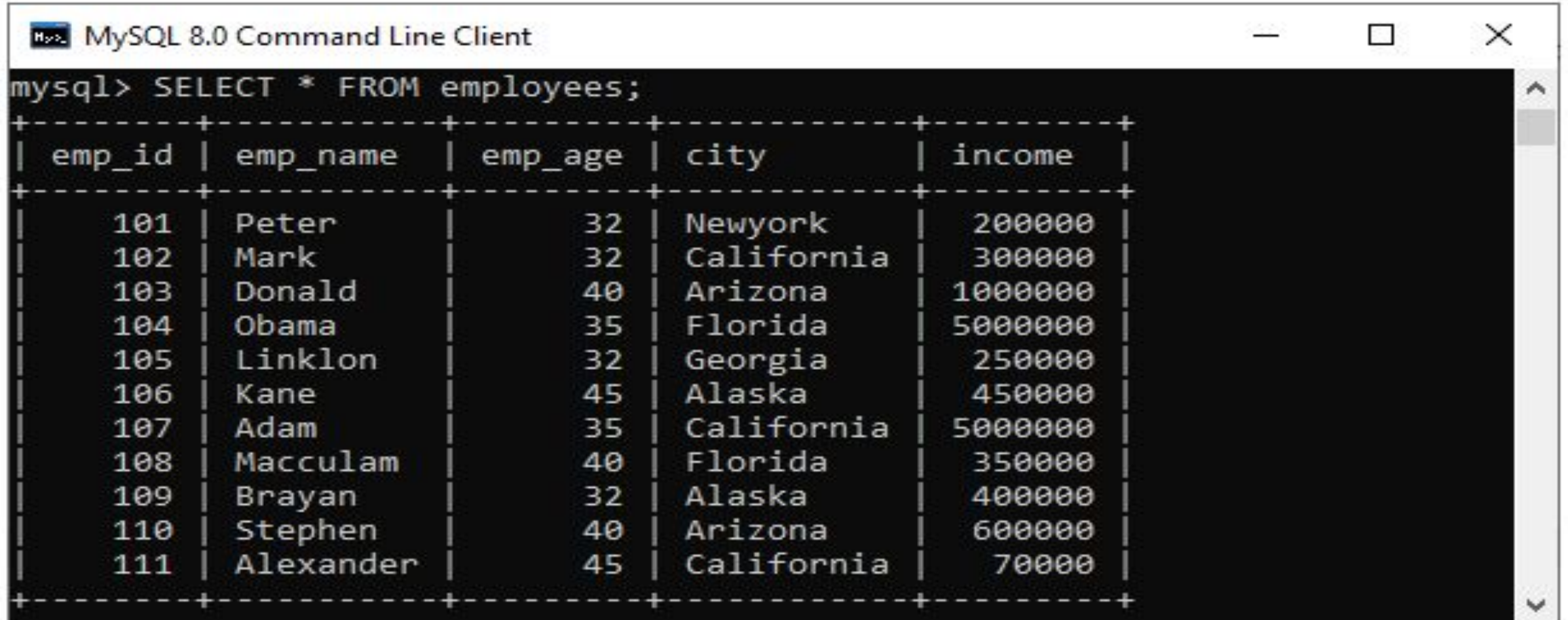
- The subqueries make the queries in a structured form that allows us to isolate each part of a statement.
- The subqueries provide alternative ways to query the data from the table; otherwise, we need to use complex joins and unions.
- The subqueries are more readable than complex join or union statements.

# MySQL Subquery Syntax

```
SELECT column_list (s) FROM table_name  
WHERE column_name OPERATOR  
(SELECT column_list (s) FROM table_name [WHERE]);
```

# MySQL Subquery Example

**Table: employees**



The image shows a screenshot of the MySQL 8.0 Command Line Client window. The title bar reads "MySQL 8.0 Command Line Client". The command prompt shows the command `mysql> SELECT * FROM employees;` followed by a table of results. The table has columns `emp_id`, `emp_name`, `emp_age`, `city`, and `income`. The results are displayed in a text-based table format with dashed lines separating the columns.

emp_id	emp_name	emp_age	city	income
101	Peter	32	Newyork	200000
102	Mark	32	California	300000
103	Donald	40	Arizona	1000000
104	Obama	35	Florida	5000000
105	Linklon	32	Georgia	250000
106	Kane	45	Alaska	450000
107	Adam	35	California	5000000
108	Macculam	40	Florida	350000
109	Brayan	32	Alaska	400000
110	Stephen	40	Arizona	600000
111	Alexander	45	California	70000

- SQL statement that returns the **employee detail whose id matches in a subquery**:

```
SELECT emp_name, city, income FROM employees  
WHERE emp_id IN (SELECT emp_id FROM employees);
```

Output

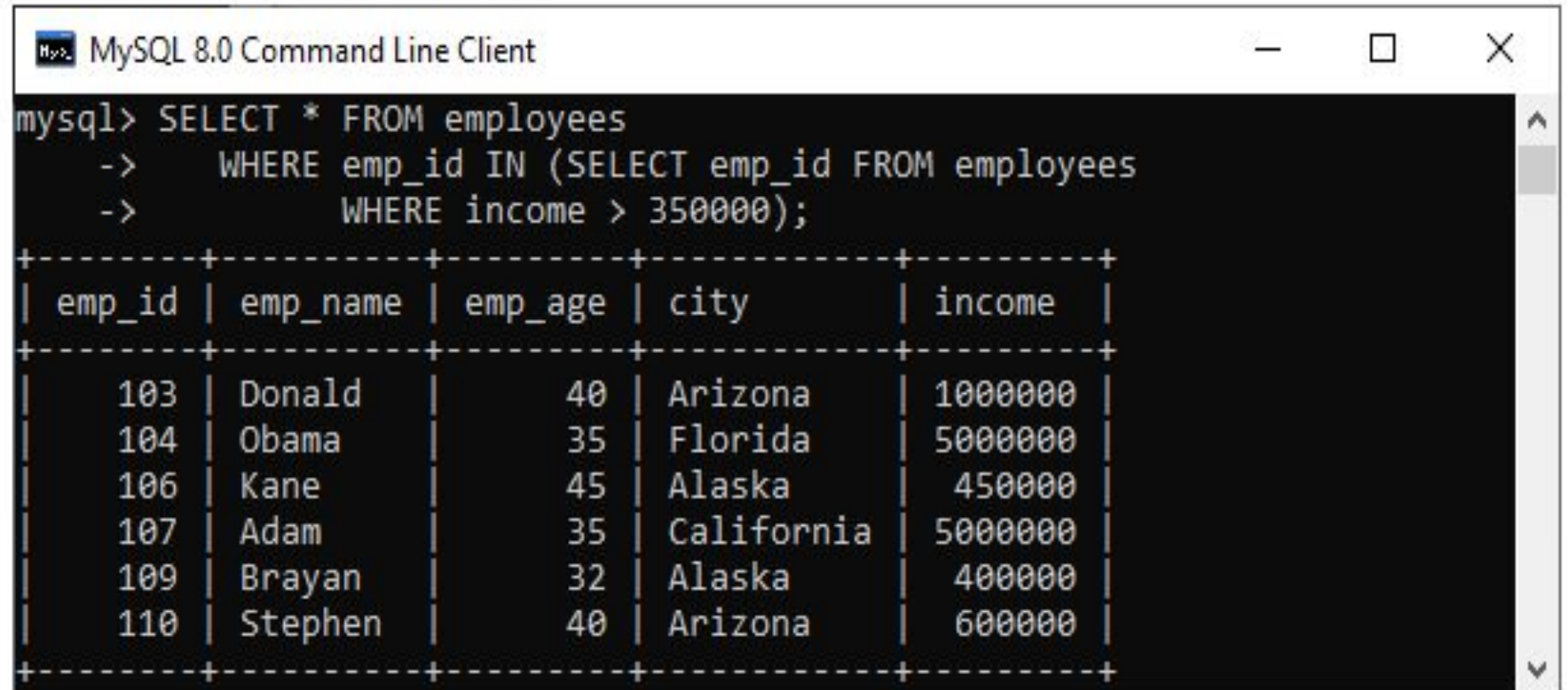
MySQL 8.0 Command Line Client

```
mysql> SELECT emp_name, city, income FROM employees  
-> WHERE emp_id IN (SELECT emp_id FROM employees);
```

emp_name	city	income
Peter	Newyork	200000
Mark	California	300000
Donald	Arizona	1000000
Obama	Florida	5000000
Linklon	Georgia	250000
Kane	Alaska	450000
Adam	California	5000000
Macculam	Florida	350000
Brayan	Alaska	400000
Stephen	Arizona	600000
Alexander	California	70000

- employee detail whose income is more than 350000

```
SELECT * FROM employees
WHERE emp_id IN (SELECT emp_id FROM employees
WHERE income > 350000);
```



The image shows a screenshot of a MySQL 8.0 Command Line Client window. The window title is "MySQL 8.0 Command Line Client". The command prompt shows the following SQL query being executed:

```
mysql> SELECT * FROM employees
-> WHERE emp_id IN (SELECT emp_id FROM employees
-> WHERE income > 350000);
```

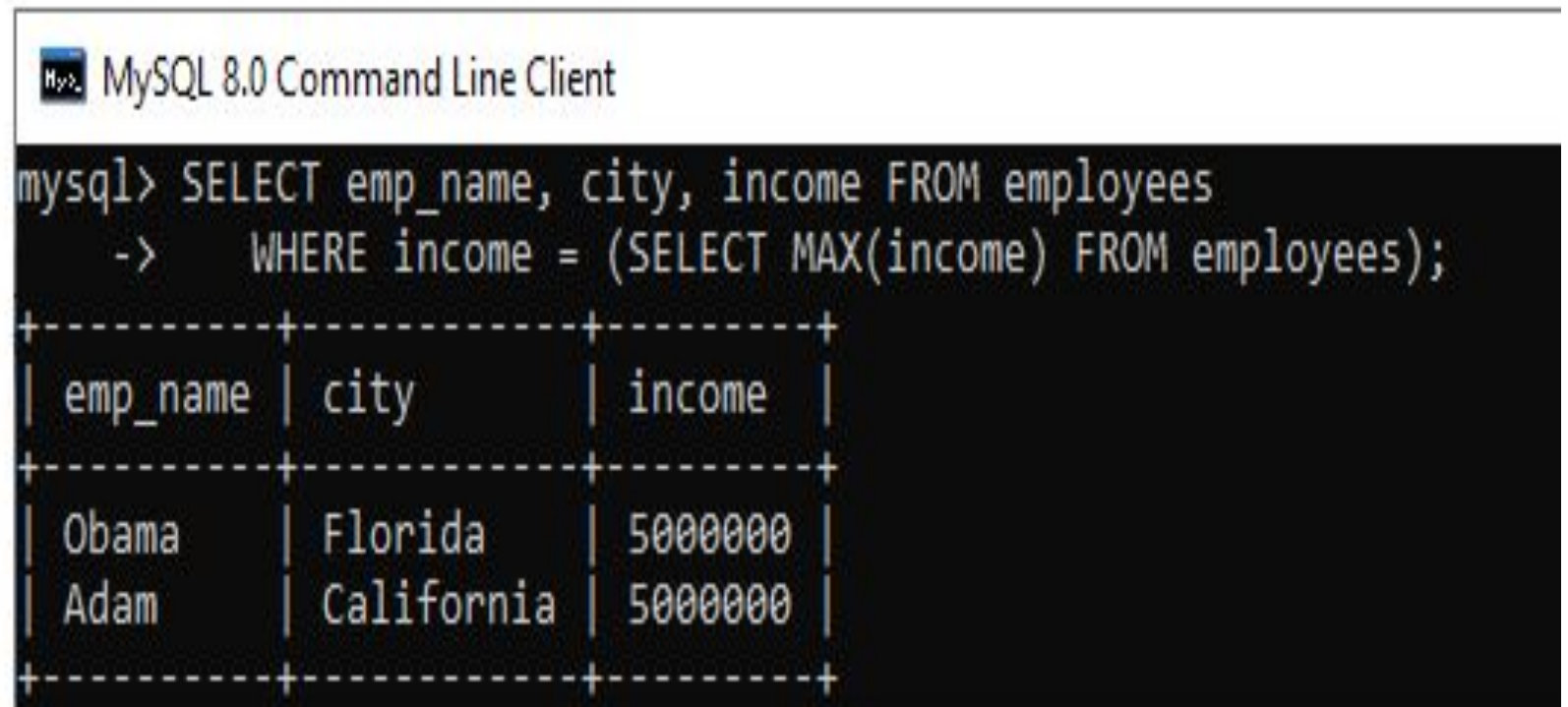
The results of the query are displayed in a table format with columns: emp\_id, emp\_name, emp\_age, city, and income. The table contains 6 rows of data.

emp_id	emp_name	emp_age	city	income
103	Donald	40	Arizona	1000000
104	Obama	35	Florida	5000000
106	Kane	45	Alaska	450000
107	Adam	35	California	5000000
109	Brayan	32	Alaska	400000
110	Stephen	40	Arizona	600000



- find employee details with **maximum income**

```
SELECT emp_name, city, income FROM employees  
WHERE income = (SELECT MAX(income) FROM employees);
```



MySQL 8.0 Command Line Client

```
mysql> SELECT emp_name, city, income FROM employees  
-> WHERE income = (SELECT MAX(income) FROM employees);
```

emp_name	city	income
Obama	Florida	5000000
Adam	California	5000000

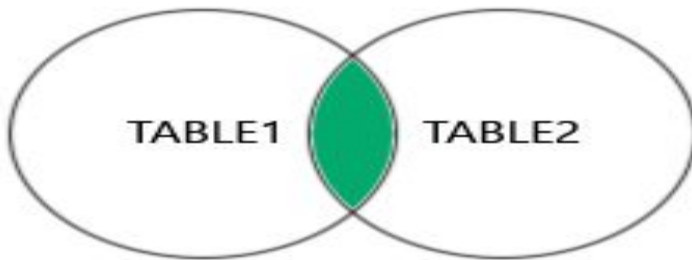
# SQL Joins

- **SQL Join** operation combines data or rows from two or more tables based on a common field between them.
- SQL JOIN clause is used to query and access data from multiple tables by establishing logical relationships between them.
- It can access data from multiple tables simultaneously using common key values shared across different tables.
- SQL JOIN can be used with multiple tables.
- It can also be paired with other clauses, the most popular use will be using JOIN with **WHERE clause** to filter data retrieval.

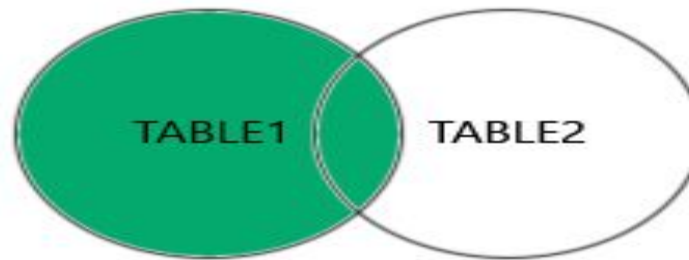
# Types of Joins

- INNER JOIN:
- LEFT JOIN:
- RIGHT JOIN:
- CROSS JOIN:

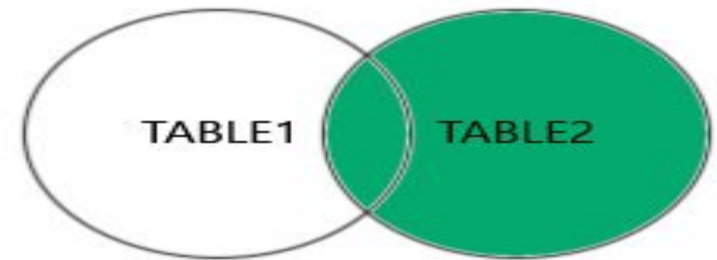
INNER JOIN



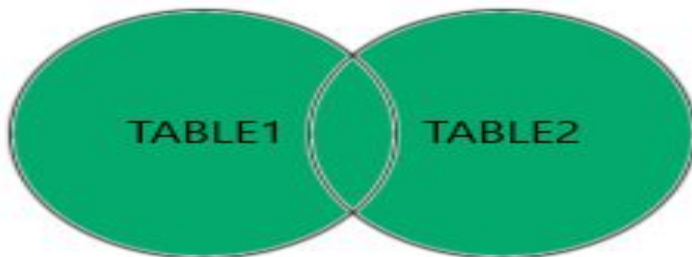
LEFT JOIN



RIGHT JOIN



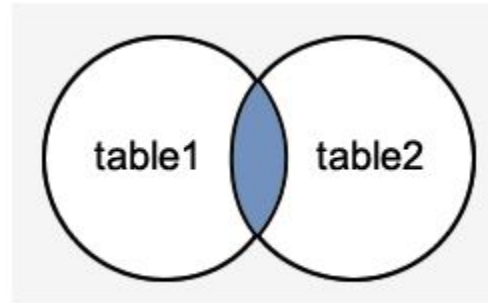
CROSS JOIN



# 1. INNER JOIN

- Returns records that have matching values in both tables.

```
SELECT columns  
FROM table1  
INNER JOIN table2  
ON table1.column = table2.column;
```



# 1. INNER JOIN

```
+-----+
4 rows in set (0.00 sec)

mysql> SELECT * FROM officers;
+-----+-----+-----+
| officer_id | officer_name | address |
+-----+-----+-----+
| 1 | Ajeet | Mau |
| 2 | Deepika | Lucknow |
| 3 | Uinal | Faizabad |
| 4 | Rahul | Lucknow |
+-----+-----+-----+
4 rows in set (0.00 sec)

mysql> SELECT * FROM students;
+-----+-----+-----+
| student_id | student_name | course_name |
+-----+-----+-----+
| 1 | Aryan | Java |
| 2 | Rohini | Hadoop |
| 3 | Lallu | MongoDB |
+-----+-----+-----+
3 rows in set (0.00 sec)
```

```
SELECT officers.officer_name, officers.address, students.course_name
FROM officers
INNER JOIN students
ON officers.officer_id = students.student_id;
```

# 1. INNER JOIN

```
mysql> SELECT officers.officer_name, officers.address, students.course_name  
-> FROM officers  
-> INNER JOIN students  
-> ON officers.officer_id = students.student_id;
```

officer_name	address	course_name
Ajeet	Mau	Java
Deepika	Lucknow	Hadoop
Uinal	Faizabad	MongoDB

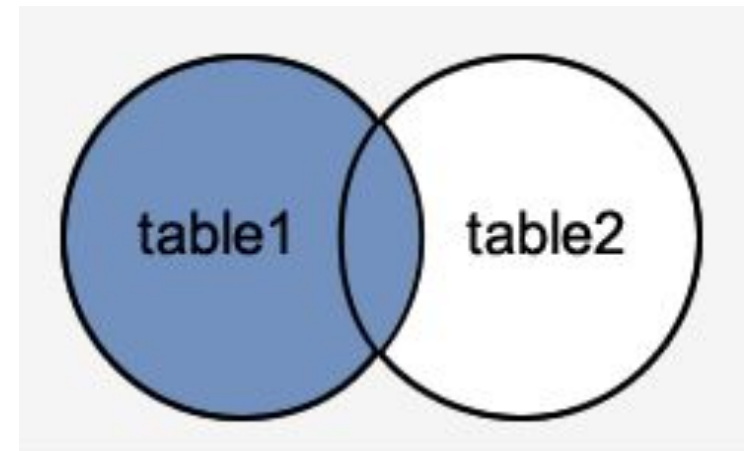
3 rows in set (0.00 sec)

```
mysql> _
```

## 2. LEFT JOIN

- Returns all records from the left table, and the matched records from the right table.

```
SELECT columns  
FROM table1  
LEFT [OUTER] JOIN table2  
ON table1.column = table2.column;
```





4 rows in set (0.00 sec)

```
mysql> SELECT * FROM officers;
```

officer_id	officer_name	address
1	Ajeet	Mau
2	Deepika	Lucknow
3	Vinal	Faizabad
4	Rahul	Lucknow

4 rows in set (0.00 sec)

```
mysql> SELECT * FROM students;
```

student_id	student_name	course_name
1	Aryan	Java
2	Rohini	Hadoop
3	Lallu	MongoDB

3 rows in set (0.00 sec)

## 2. LEFT JOIN

```
SELECT officers.officer_name, officers.address, students.course_name
FROM officers
LEFT JOIN students
ON officers.officer_id = students.student_id;
```



## 2. LEFT JOIN

```
mysql> SELECT officers.officer_name, officers.address, students.course_name  
-> FROM officers  
-> LEFT JOIN students  
-> ON officers.officer_id = students.student_id;
```

officer_name	address	course_name
Ajeet	Mau	Java
Deepika	Lucknow	Hadoop
Vinal	Faizabad	MongoDB
Rahul	Lucknow	NULL

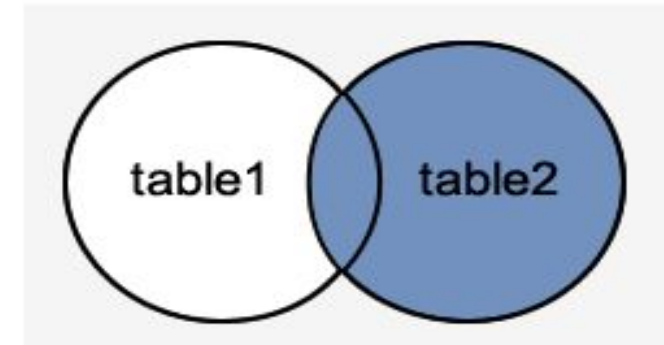
```
4 rows in set (0.01 sec)
```

```
mysql>
```

# 3. RIGHT JOIN

- Returns all records from the right table, and the matched records from the left table.

```
SELECT columns  
FROM table1  
RIGHT [OUTER] JOIN table2  
ON table1.column = table2.column;
```



### 3. RIGHT JOIN

```
4 rows in set (0.00 sec)

mysql> SELECT * FROM officers;
+-----+-----+-----+
| officer_id | officer_name | address |
+-----+-----+-----+
| 1 | Ajeet | Mau |
| 2 | Deepika | Lucknow |
| 3 | Uinal | Faizabad |
| 4 | Rahul | Lucknow |
+-----+-----+-----+
4 rows in set (0.00 sec)

mysql> SELECT * FROM students;
+-----+-----+-----+
| student_id | student_name | course_name |
+-----+-----+-----+
| 1 | Aryan | Java |
| 2 | Rohini | Hadoop |
| 3 | Lallu | MongoDB |
+-----+-----+-----+
3 rows in set (0.00 sec)
```

```
SELECT officers.officer_name, officers.address, students.course_name, students.student_name
FROM officers
RIGHT JOIN students
ON officers.officer_id = students.student_id;
```

### 3. RIGHT JOIN

```
mysql> SELECT officers.officer_name, officers.address, students.course_name, st
udents.student_name
-> FROM officers
-> RIGHT JOIN students
-> ON officers.officer_id = students.student_id;
```

officer_name	address	course_name	student_name
Ajeet	Mau	Java	Aryan
Deepika	Lucknow	Hadoop	Rohini
Vinal	Faizabad	MongoDB	Lallu

```
3 rows in set (0.00 sec)
```

```
mysql> _
```

## 4. CROSS Join

- Returns all records from both tables

```
SELECT column-lists  
FROM table1  
CROSS JOIN table2;
```

**Table: customers**

customer_id	cust_name	occupation	income	qualification
1	John Miller	Developer	20000	Btech
2	Mark Robert	Engineer	40000	Btech
3	Reyan Watson	Scientists	60000	MSc
4	Shane Trump	Businessman	10000	MBA
5	Adam Obama	Manager	80000	MBA
6	Rinky Ponting	Cricketer	200000	Btech

**Table: contacts**

contact_id	cellphone	homephone
1	6546645978	4565242557
2	8798634532	8652413954
3	8790744345	9874437396
4	7655654336	9934345363



```
SELECT *
FROM customers
CROSS JOIN contacts;
```

## 4. CROSS Join

customer_id	cust_name	occupation	income	qualification	contact_id	cellphone	homephone
1	John Miller	Developer	20000	Btech	1	6546645978	4565242557
1	John Miller	Developer	20000	Btech	2	8798634532	8652413954
1	John Miller	Developer	20000	Btech	3	8790744345	9874437396
1	John Miller	Developer	20000	Btech	4	7655654336	9934345363
1	John Miller	Developer	20000	Btech	5	NULL	6786507067
1	John Miller	Developer	20000	Btech	6	NULL	9086053684
2	Mark Robert	Engineer	40000	Btech	1	6546645978	4565242557
2	Mark Robert	Engineer	40000	Btech	2	8798634532	8652413954
2	Mark Robert	Engineer	40000	Btech	3	8790744345	9874437396
2	Mark Robert	Engineer	40000	Btech	4	7655654336	9934345363
2	Mark Robert	Engineer	40000	Btech	5	NULL	6786507067
2	Mark Robert	Engineer	40000	Btech	6	NULL	9086053684
3	Reyan Watson	Scientists	60000	MSc	1	6546645978	4565242557
3	Reyan Watson	Scientists	60000	MSc	2	8798634532	8652413954
3	Reyan Watson	Scientists	60000	MSc	3	8790744345	9874437396

# VIEWS

- a view is a virtual table based on the result-set of an SQL statement.
- A view also has rows and columns like tables, but a view doesn't store data on the disk like a table.
- 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.
- You can add SQL statements and functions to a view and present the data as if the data were coming from one single table.
- A view is created with the CREATE VIEW.
- A View can either have all the rows of a table or specific rows based on certain conditions.

# Benefit of Using Views

- **Simplicity:** Instead of writing complex joins & queries, views provide a way of writing simple SELECT statements.
- **Enhanced Security:** Views expose only the data to the third-party apps and hide the internal details like table structure, attributes, etc, thus adding extra security.
- **Consistency:** By writing views instead of common queries, we can write a view that avoids multiple declarations & definitions of the same queries and eventually provides a centralized way.



# 1. Create a View Based On Single Table

StudentDetails Table:

sid	sname	age	university
1	Girish	24	IIT Hyderabad
2	Aaditya	24	SRM University
3	Aashish	23	IIT Hyderabad
4	John	25	Mumbai University
5	Shruti	24	IIT Hyderabad
6	Leena	25	Mumbai University

```
CREATE VIEW <view_name> AS  
  
SELECT <column1>, <column2>....., <columnN>  
  
FROM <table-name>  
  
WHERE [conditions];
```

Let's us create a view named **"IITHyderabadStudentsView"** from the StudentDetails table. This view selects the students from the StudentDetails table who study in "IIT Hyderabad" university and outputs their details like student id, name, and age.

```
CREATE VIEW IITHyderabadStudentsView AS  
SELECT sid, sname, age  
FROM StudentDetails  
WHERE university = "IIT Hyderabad";
```

```
[mysql> SELECT * FROM IITHyderabadStudentsView;  
+-----+-----+-----+  
| sid | sname   | age |  
+-----+-----+-----+  
| 1   | Girish  | 24  |  
| 3   | Aashish | 23  |  
| 5   | Shruti  | 24  |  
+-----+-----+-----+  
3 rows in set (0.07 sec)
```

## 2. Create a View Based On Multiple Tables With JOIN Clause

```
CREATE VIEW <view_name> AS  
  
SELECT <column1>, <column2>....., <columnN>  
  
FROM <table1>  
  
[ INNER | LEFT | RIGHT | FULL ] JOIN <table2> ON <joining-column>  
  
WHERE [condition1 | condition2 | .....];
```

```
CREATE VIEW PythonEnrolledView AS  
  
SELECT S.sid, S.sname, S.age  
FROM StudentDetails S JOIN EnrolledIn E ON S.sid = E.sid  
JOIN CourseDetails C ON C.cid = E.cid  
WHERE C.cname = "Python Fundamentals";
```

Let's us create a view named "**PythonEnrolledView**" using the StudentDetails, CourseDetails, and EnrolledIn table. This view outputs the students who are enrolled in "Python Fundamentals" course the details as student id, name, and age.

```
SELECT * FROM PythonEnrolledView;
```

```
[mysql> SELECT * FROM PythonEnrolledView;
```

sid	sname	age
2	Aaditya	24
4	John	25

```
2 rows in set (0.01 sec)
```

# 3. Update View

There are certain conditions that need to be satisfied to update a view. If any one of these conditions is not met, then we are not allowed to update the view.

- The View should be created from a single table. If the view is created using multiple tables then we will not be allowed to update the view.
- The View should not be created using nested queries or complex queries.
- The View should have all **NOT NULL** values.
- The **SELECT** statement should not have the **DISTINCT** keyword.
- The **SELECT** statement which is used to create the view should not include **GROUP BY** clause or **ORDER BY** clause.

# Update View Definition/Structure

- To update the view for adding or remove columns and rows by changing WHERE clause condition, we can use **CREATE OR REPLACE VIEW** statement.

```
CREATE OR REPLACE VIEW <view_name> AS  
SELECT <column1>, <column2>, ....., <columnN>  
FROM <table_name>  
WHERE [condition];
```

```
CREATE OR REPLACE VIEW IITHyderabadStudentsView AS  
SELECT sname, age  
FROM StudentDetails  
WHERE university = "IIT Hyderabad";
```

## 4. Insert Into View

- To insert the new row into the view, we can do it in a similar way just like how we do it for normal tables.

```
INSERT INTO <view_name>(<column1>, <column2>, <column3>,.....)  
VALUES(<value1>, <value2>, <value3>,.....);
```



```
[mysql> SELECT * FROM StudentDetails;
```

sid	sname	age	university
1	Girish	24	IIT Hyderabad
2	Aaditya	24	SRM University
3	Aashish	23	IIT Hyderabad
4	John	25	Mumbai University
5	Shruti	24	IIT Hyderabad
6	Leena	25	Mumbai University

```
6 rows in set (0.00 sec)
```

```
mysql> INSERT INTO IITHyderabadStudentsView(sid, sname, age) VALUES(7, "Tenali Rama", 26);  
Query OK, 1 row affected (0.01 sec)
```

```
mysql> SELECT * FROM StudentDetails;
```

sid	sname	age	university
1	Girish	24	IIT Hyderabad
2	Aaditya	24	SRM University
3	Aashish	23	IIT Hyderabad
4	John	25	Mumbai University
5	Shruti	24	IIT Hyderabad
6	Leena	25	Mumbai University
7	Tenali Rama	26	IIT Hyderabad

```
7 rows in set (0.00 sec)
```

## 5. Delete From View

```
DELETE FROM <view_name> WHERE [condition];
```

```
DELETE FROM IITHyderabadStudentsView WHERE sname = "Tenali Rama";
```

```
[mysql> SELECT * FROM IITHyderabadStudentsView;
```

sid	sname	age
1	Girish	24
3	Aashish	23
5	Shruti	24
7	Tenali Rama	26

4 rows in set (0.01 sec)

```
[mysql> DELETE FROM IITHyderabadStudentsView WHERE sname = "Tenali Rama";  
Query OK, 1 row affected (0.01 sec)
```

```
[mysql> SELECT * FROM IITHyderabadStudentsView;
```

sid	sname	age
1	Girish	24
3	Aashish	23
5	Shruti	24

3 rows in set (0.00 sec)

## 6. Drop View

```
DROP VIEW <view_name>;
```

```
DROP VIEW PythonEnrolledView;
```

# Trigger

A trigger in MySQL is a set of SQL statements that reside in a system catalog. It is a special type of stored procedure that is invoked automatically in response to an event. Each trigger is associated with a table, which is activated on any DML statement such as INSERT, UPDATE, or DELETE.

A trigger is called a special procedure because it cannot be called directly like a stored procedure. The main difference between the trigger and procedure is that a trigger is called automatically when a data modification event is made against a table. In contrast, a stored procedure must be called explicitly.

triggers are of two types: row-level triggers and statement-level triggers.

Row-Level Trigger: It is a 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: It is a trigger, which is fired once for each event that occurs on a table regardless of how many rows are inserted, updated, or deleted.

# Use of Triggers

- Triggers help us to enforce business rules.
- Triggers help us to validate data even before they are inserted or updated.
- Triggers help us to keep a log of records like maintaining audit trails in tables.
- SQL triggers provide an alternative way to check the integrity of data.
- Triggers provide an alternative way to run the scheduled task.
- Triggers increases the performance of SQL queries because it does not need to compile each time the query is executed.
- Triggers reduce the client-side code that saves time and effort.
- Triggers help us to scale our application across different platforms.
- Triggers are easy to maintain.

# Limitations of Using Triggers

- MySQL triggers do not allow to use of all validations; they only provide extended validations. For example, we can use the NOT NULL, UNIQUE, CHECK and FOREIGN KEY constraints for simple validations.
- Triggers are invoked and executed invisibly from the client application. Therefore, it isn't easy to troubleshoot what happens in the database layer.
- Triggers may increase the overhead of the database server.

# Components of Trigger

1. Event: SQL statement that causes trigger to fire.
2. Condition: A condition that must be satisfied for execution of trigger.
3. Action: A code that will be executed when triggering condition satisfies and the trigger is activated.



# Types of Triggers

1. **Before Insert:** It is activated before the insertion of data into the table.
2. **After Insert:** It is activated after the insertion of data into the table.
3. **Before Update:** It is activated before the update of data in the table.
4. **After Update:** It is activated after the update of the data in the table.
5. **Before Delete:** It is activated before the data is removed from the table.
6. **After Delete:** It is activated after the deletion of data from the table.

# Example

```
mysql> CREATE TABLE account (acct_num INT, amount DECIMAL(10,2));  
Query OK, 0 rows affected (0.03 sec)
```

```
mysql> CREATE TRIGGER ins_sum BEFORE INSERT ON account  
FOR EACH ROW SET @sum = @sum + NEW.amount;  
Query OK, 0 rows affected (0.01 sec)
```

The [CREATE TRIGGER](#) statement creates a trigger named `ins_sum` that is associated with the `account` table. It also includes clauses that specify the trigger action time, the triggering event, and what to do when the trigger activates:

- The keyword **BEFORE** indicates the trigger action time. In this case, the trigger activates before each row inserted into the table. The other permitted keyword here is **AFTER**.
- The keyword **INSERT** indicates the trigger event; that is, the type of operation that activates the trigger. In the example, [INSERT](#) operations cause trigger activation. You can also create triggers for [DELETE](#) and [UPDATE](#) operations.
- The statement following **FOR EACH ROW** defines the trigger body; that is, the statement to execute each time the trigger activates, which occurs once for each row affected by the triggering event. In the example, the trigger body is a simple [SET](#) that accumulates into a user variable the values inserted into the `amount` column. The statement refers to the column as `NEW.amount` which means “the value of the `amount` column to be inserted into the new row.”

To use the trigger, set the accumulator variable to zero, execute an [INSERT](#) statement, and then see what value the variable has afterward:

```
mysql> SET @sum = 0;
mysql> INSERT INTO account VALUES(137,14.98),(141,1937.50),(97,-100.00);
mysql> SELECT @sum AS 'Total amount inserted';
+-----+
| Total amount inserted |
+-----+
|           1852.48    |
+-----+
```

In this case, the value of @sum after the [INSERT](#) statement has executed is  $14.98 + 1937.50 - 100$ , or 1852.48.

To destroy the trigger, use a [DROP TRIGGER](#) statement. You must specify the schema name if the trigger is not in the default schema:

```
mysql> DROP TRIGGER ins_sum;
```

If you drop a table, any triggers for the table are also dropped.

Trigger names exist in the schema namespace, meaning that all triggers must have unique names within a schema. Triggers in different schemas can have the same name.

It is possible to define multiple triggers for a given table that have the same trigger event and action time. For example, you can have two **BEFORE UPDATE** triggers for a table. By default, triggers that have the same trigger event and action time activate in the order they were created. To affect trigger order, specify a clause after **FOR EACH ROW** that indicates **FOLLOWS** or **PRECEDES** and the name of an

# Privileges

The authority or permission to access a named object as advised manner, for example, permission to access a table. Privileges can allow permitting a particular user to connect to the database. In, other words privileges are the allowance to the database by the database object.

- **Database privileges** — A privilege is permission to execute one particular type of [SQL](#) statement or access a second persons' object. Database privilege controls the use of computing resources. Database privilege does not apply to the Database administrator of the database.
- **System privileges** — A system privilege is the right to perform an activity on a specific type of object. for example, the privilege to delete rows of any table in a database is system privilege. There are a total of 60 different system privileges. System privileges allow users to CREATE, ALTER, or DROP the database objects.
- **Object privilege** — An object privilege is a privilege to perform a specific action on a particular table, function, or package. For example, the right to delete rows from a table is an object privilege. For example, let us consider a row of table GEEKSFORGEEKS that contains the name of the employee who is no longer a part of the organization, then deleting that row is considered as an object privilege. Object privilege allows the user to INSERT, DELETE, UPDATE, or SELECT the data in the database object

# Roles :

A role is a mechanism that can be used to allow authorization. A person or a group of people can be allowed a role or group of roles. By many roles, the head can manage access privileges very easily. The roles are provided by the [database management system](#) for easy and managed or controlled privilege management.

## Properties

The following are the properties of the roles which allow easy privilege management inside a database:

- **Reduced privilege administration** — The user can grant the privilege for a group of users who are related instead of granting the same set of privileges to the users explicitly.
- **Dynamic privilege management** — If the privilege of the group changes then, only the right of role needs to be changed.
- **Application-specific security** — The user can also protect the use of a role by using a password. Applications can be created to allow a role when entering the correct and best password. Users are not allowed the role if they do not know about the password.

# CREATE USER Statement

The **CREATE USER** statement in **MySQL** is an essential command used to create new user accounts for database access. It enables database administrators to define which users can connect to the MySQL database server and specify their login credentials.

## CREATE USER in MySQL

- The **CREATE USER** statement in [SQL](#) is used to create a new user and a password to access that user.
- [MySQL](#) allows us to specify which user account can connect to a database server. The user account details in MySQL contain two information – **username** and **host** from which the user is trying to connect in the format **username@host-name**.
- If the admin user is connecting through localhost then the user account will be **admin@localhost**. MySQL stores the user account in the **user** grant table of the **MySQL** [database](#).
- The **CREATE USER** statement in MySQL allows us to create new MySQL accounts or in other words, the **CREATE USER** statement is used to create a database account that allows the user to log into the MySQL database.

**Syntax:** *CREATE USER user\_account IDENTIFIED BY password;*

## Parameters

1. **user\_account:** It is the name that the user wants to give to the database account. The user\_account should be in the format '**username**'@'**hostname**'
2. **password:** It is the password used to assign to the user\_account. The password is specified in the IDENTIFIED BY clause.

## Examples of MySQL CREATE USER Statement

Let's look at some examples of the CREATE USER command in MySQL, and understand it's working.

### Example 1: MySQL Create Single User

In this example, we will create a new user "gfguser1" that connects to the MySQL database server from the localhost with the password "abcd".

```
CREATE USER gfguser1@localhost IDENTIFIED BY 'abcd';
```



# Security

## ■ **Security** - protection from malicious attempts to steal or modify data.

### 👉 Database system level

📄 **Authentication** and **authorization** mechanisms to allow specific users access only to required data

📄 We concentrate on authorization in the rest of this chapter

### 👉 Operating system level

📄 Operating system super-users can do anything they want to the database! Good operating system level security is required.

### 👉 Network level: must use encryption to prevent



📄 Eavesdropping (unauthorized reading of messages)

📄 Masquerading (pretending to be an authorized user or sending messages supposedly from authorized users)





# Security

## Physical level

-  Physical access to computers allows destruction of data by intruders; traditional lock-and-key security is needed
-  Computers must also be protected from floods, fire, etc.
  - More in Chapter 17 (Recovery)

## Human level

-  Users must be screened to ensure that an authorized users do not give access to intruders
-  Users should be trained on password selection and secrecy



# Authorization

Forms of authorization on parts of the database:

- **Read authorization** - allows reading, but not modification of data.
- **Insert authorization** - allows insertion of new data, but not modification of existing data.
- **Update authorization** - allows modification, but not deletion of data.
- **Delete authorization** - allows deletion of data

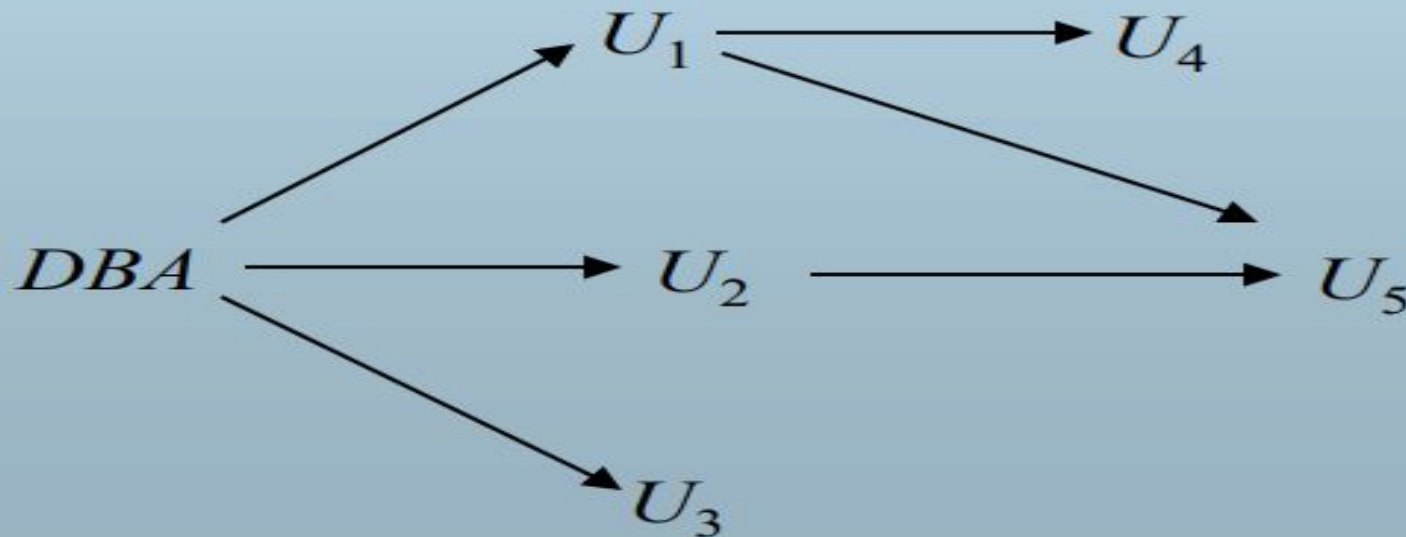
# Authorization

Forms of authorization to modify the database schema:

- **Index authorization** - allows creation and deletion of indices.
- **Resources authorization** - allows creation of new relations.
- **Alteration authorization** - allows addition or deletion of attributes in a relation.
- **Drop authorization** - allows deletion of relations.

# Granting of Privileges

- The passage of authorization from one user to another may be represented by an authorization graph.
- The nodes of this graph are the users.
- The root of the graph is the database administrator.
- Consider graph for update authorization on loan.
- An edge  $U_i \rightarrow U_j$  indicates that user  $U_i$  has granted update authorization on loan to  $U_j$ .





# Authorization Graph

- *Requirement:* All edges in an authorization graph must be part of some path originating with the database administrator
- If DBA revokes grant from  $U_1$ :
  - 👉 Grant must be revoked from  $U_4$  since  $U_1$  no longer has authorization
  - 👉 Grant must not be revoked from  $U_5$  since  $U_5$  has another authorization path from DBA through  $U_2$
- Must prevent cycles of grants with no path from the root:
  - 👉 DBA grants authorization to  $U_7$
  - 👉  $U_7$  grants authorization to  $U_8$
  - 👉  $U_8$  grants authorization to  $U_7$
  - 👉 DBA revokes authorization from  $U_7$
- Must revoke grant  $U_7$  to  $U_8$  and from  $U_8$  to  $U_7$  since there is no path from DBA to  $U_7$  or to  $U_8$  anymore.



# Security

- The grant statement is used to confer authorization  
**grant** <privilege list>  
**on** <relation name or view name> to <user list>
- <user list> is:
  - 👉 a user-id
  - 👉 *public*, which allows all valid users the privilege granted
  - 👉 A role (more on this later)
- 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 *branch* relation:

**grant select on *branch* 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.
- **references:** ability to declare foreign keys when creating relations.
- **usage:** In SQL-92; authorizes a user to use a specified domain
- **all privileges:** used as a short form for all the allowable privileges



# Grant Privilege

- **with grant option:** allows a user who is granted a privilege to pass the privilege on to other users.

👉 Example:

**grant select on *branch* to  $U_1$  with grant option**

gives  $U_1$  the **select** privileges on *branch* and allows  $U_1$  to grant this privilege to others

# Roles

- Roles permit common privileges for a class of users can be specified just once by creating a corresponding “role”
- Privileges can be granted to or revoked from roles, just like user
- Roles can be assigned to users, and even to other roles
- SQL:1999 supports roles

```
create role teller  
create role manager
```

```
grant select on branch to teller  
grant update (balance) on account to teller  
grant all privileges on account to manager
```

```
grant teller to manager
```

```
grant teller to alice, bob  
grant manager to avi
```





# Revoking Authorization

- The **revoke** statement is used to revoke authorization.  
**revoke**<privilege list>  
**on** <relation name or view name> **from** <user list> [**restrict**|**cascade**]
- Example:  
**revoke select on branch from  $U_1, U_2, U_3$  cascade**
- Revocation of a privilege from a user may cause other users also to lose that privilege; referred to as cascading of the **revoke**.
- We can prevent cascading by specifying **restrict**:  
**revoke select on branch from  $U_1, U_2, U_3$  restrict**  
With **restrict**, the **revoke** command fails if cascading revokes are required.

# Revoking Authorization

- <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.
- All privileges that depend on the privilege being revoked are also revoked.

# Assertions

- It is a statement in database that ensures certain conditions will always exist in database.

- Syntax:

Create Assertion `assertion_name` check condition;

# Assertions

- Example:

Ensuring licenses are recent or issued after the date 01-01-2023

Create Assertion rec\_lic

Check (select count(\*) from licenses

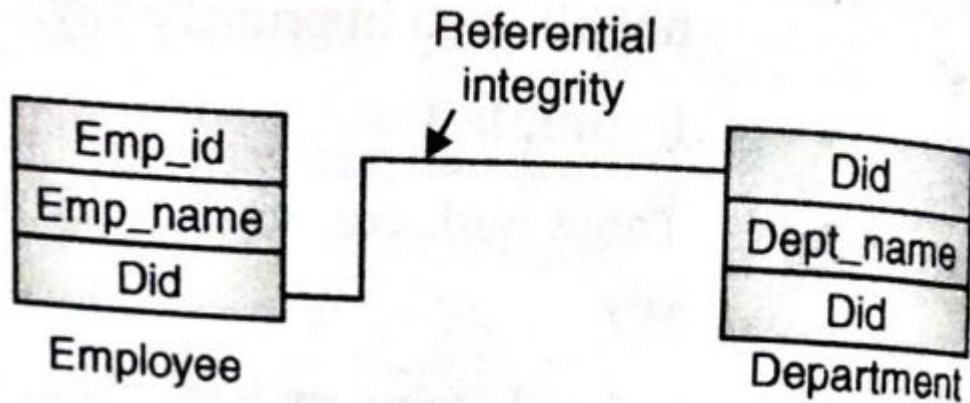
where lic\_renewal\_date<'2023-01-01'>);

# Referential Integrity

- A value appearing in one relation (table) for given set of attributes also appears in another table for another set of attributes is known as referential integrity.
- It is used to maintain the consistency between two tables.
- The tuple in one relation refers only to existing tuple in another relation.



# Referential Integrity



- Employee has Did as foreign key so this is known as referential integrity.
- Here when we want to insert data in Employee table it has to be checked first with Department table Did.

Emp Table		
Emp_Id	Emp_name	Did
1	Sachin	20
2	Suhas	10
3	Jay	20
4	Om	10

Department Table	
Did	Dept_name
10	HR
20	TIS
30	L&D