Advanced SQL

- By Jyoti Tryambake

Join

Types:

1. Inner Joins
2. Outer Joins
    1. Left outer
    2. Right outer
    3. Full outer
3. Natural Join
4. Cross Join
5. Self joins

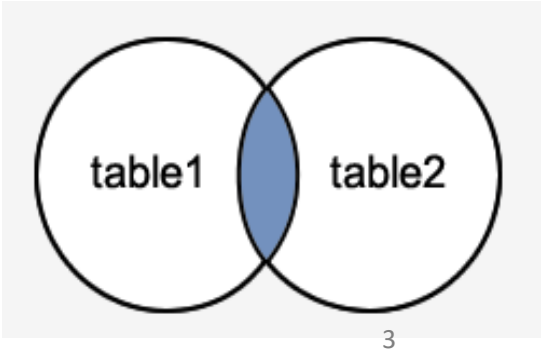
Join (cont..)

- A SQL JOIN combines records from two tables.
- A JOIN locates related column values in the two tables.
- A query can contain zero, one, or multiple JOIN operations.
- INNER JOIN is the same as JOIN; the keyword INNER is optional.

1. Inner Join

SELECT columns FROM table1 INNER JOIN table2

ON table1.column = table2.column;



Join (cont..)

Example:

customer_id	last_name	first_name	favorite_website
4000	Jackson	Joe	techonthenet.com
5000	Smith	Jane	digminecraft.com
6000	Ferguson	Samantha	bigactivities.com
7000	Reynolds	Allen	checkyourmath.com
8000	Anderson	Paige	NULL
9000	Johnson	Derek	techonthenet.com

order_id	customer_id	order_date
1	7000	2016/04/18
2	5000	2016/04/18
3	8000	2016/04/19
4	4000	2016/04/20
5	NULL	2016/05/01

Join (cont..)

Example:

SELECT customers.customer_id, orders.order_id, orders.order_date FROM customers INNER JOIN orders ON customers.customer_id = orders.customer_id ORDER BY customers.customer_id;

customer_id	order_id	order_date
4000	4	2016/04/20
5000	2	2016/04/18
7000	1	2016/04/18
8000	3	2016/04/19

Join (cont..)

Outer Join:

- returns all rows from the participating tables which satisfy the condition and also those rows which do not match the condition will appear in this operation. This result set can appear in three types of format -
- LEFT OUTER JOIN - all the rows from a left table of JOIN clause and the unmatched rows from a right table with NULL values for selected columns.
- RIGHT OUTER JOIN - all rows from the right of JOIN cause and the unmatched rows from the left table with NULL values for selected columns.
- FULL OUTER JOIN -includes the matching rows from the left and right tables of JOIN clause and the unmatched rows from left and right table with NULL values for selected columns.

Join (cont..)

Left Outer Join:

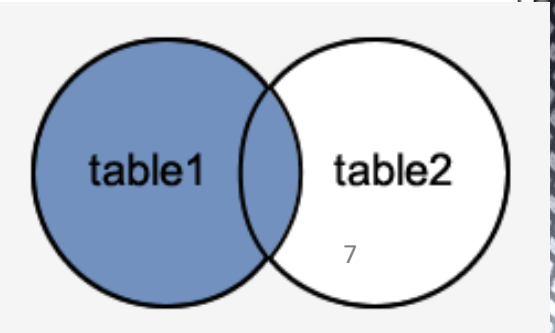
This type of join returns all rows from the LEFT-hand table specified in the ON condition and **only** those rows from the other table where the joined fields are equal (join condition is met).

Syntax:

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

Query:

SELECT customers.customer_id, orders.order_id, orders.order_date FROM customers LEFT OUTER JOIN orders ON customers.customer_id = orders.customer_id ORDER BY customers.customer_id;



Join (cont..)

Left Outer Join:

customer_id	last_name	first_name	favorite_website
4000	Jackson	Joe	techonthenet.com
5000	Smith	Jane	digminecraft.com
6000	Ferguson	Samantha	bigactivities.com
7000	Reynolds	Allen	checkyourmath.com
8000	Anderson	Paige	NULL
9000	Johnson	Derek	techonthenet.com

order_id	customer_id	order_date
1	7000	2016/04/18
2	5000	2016/04/18
3	8000	2016/04/19
4	4000	2016/04/20
5	NULL	2016/05/01

customer_id	order_id	order_date
4000	4	2016/04/20
5000	2	2016/04/18
6000	NULL	NULL
7000	1	2016/04/18
8000	3	2016/04/19
9000	NULL	NULL

Join (cont..)

Right Outer Join:

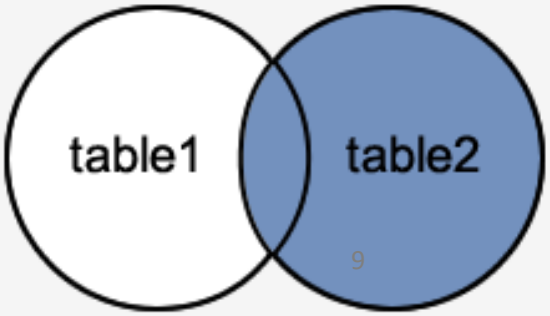
This type of join returns all rows from the RIGHT-hand table specified in the ON condition and **only** those rows from the other table where the joined fields are equal (join condition is met).

Syntax:

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

Query:

SELECT customers.customer_id, orders.order_id, orders.order_date FROM customers RIGHT OUTER JOIN orders ON customers.customer_id = orders.customer_id ORDER BY customers.customer_id;



Join (cont..)

Right Outer Join:

customer_id	last_name	first_name	favorite_website
4000	Jackson	Joe	techonthenet.com
5000	Smith	Jane	digminecraft.com
6000	Ferguson	Samantha	bigactivities.com
7000	Reynolds	Allen	checkyourmath.com
8000	Anderson	Paige	NULL
9000	Johnson	Derek	techonthenet.com

order_id	customer_id	order_date
1	7000	2016/04/18
2	5000	2016/04/18
3	8000	2016/04/19
4	4000	2016/04/20
5	NULL	2016/05/01

customer_id	order_id	order_date
NULL	5	2016/05/01
4000	4	2016/04/20
5000	2	2016/04/18
7000	1	2016/04/18
8000	3	2016/04/19

Join (cont..)

Full Outer Join:

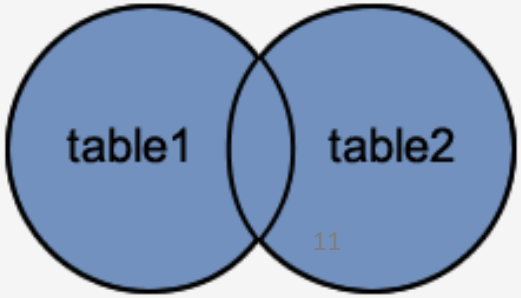
This type of join returns all rows from the LEFT-hand table and RIGHT-hand table with NULL values in place where the join condition is not met.

Syntax:

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

Query:

SELECT customers.customer_id, orders.order_id, orders.order_date FROM customers FULL OUTER JOIN orders ON customers.customer_id = orders.customer_id ORDER BY customers.customer_id;



Join (cont..)

Full Outer Join:

customer_id	last_name	first_name	favorite_website
4000	Jackson	Joe	techonthenet.com
5000	Smith	Jane	digminecraft.com
6000	Ferguson	Samantha	bigactivities.com
7000	Reynolds	Allen	checkyourmath.com
8000	Anderson	Paige	NULL
9000	Johnson	Derek	techonthenet.com

order_id	customer_id	order_date
1	7000	2016/04/18
2	5000	2016/04/18
3	8000	2016/04/19
4	4000	2016/04/20
5	NULL	2016/05/01

customer_id	order_id	order_date
NULL	5	2016/05/01
4000	4	2016/04/20
5000	2	2016/04/18
6000	NULL	NULL
7000	1	2016/04/18
8000	3	2016/04/19
9000	NULL	NULL

Join (cont..)

Natural join:

The SQL NATURAL JOIN is a type of EQUI JOIN and is structured in such a way that, columns with the same name of associated tables will appear once only.

Query:

SELECT * FROM foods NATURAL JOIN company

Join (cont..)

Natural join example:

ITEM_ID	ITEM_NAME	ITEM_UNIT	COMPANY_ID
1	Chex Mix	Pcs	16
6	Cheez-It	Pcs	15
2	BN Biscuit	Pcs	15
3	Mighty Munch	Pcs	17
4	Pot Rice	Pcs	15
5	Jaffa Cakes	Pcs	18
7	Salt n Shake	Pcs	-

COMPANY_ID	COMPANY_NAME	COMPANY_CITY
18	Order All	Boston
15	Jack Hill Ltd	London
16	Akas Foods	Delhi
17	Foodies.	London
19	sip-n-Bite.	New York

** Same column came once

COMPANY_ID	ITEM_ID	ITEM_NAME	ITEM_UNIT	COMPANY_NAME	COMPANY_CITY
16	1	Chex Mix	Pcs	Akas Foods	Delhi
15	6	Cheez-It	Pcs	Jack Hill Ltd	London
15	2	BN Biscuit	Pcs	Jack Hill Ltd	London
17	3	Mighty Munch	Pcs	Foodies.	London
15	4	Pot Rice	Pcs	Jack Hill Ltd	London
18	5	Jaffa Cakes	Pcs	Order All	Boston

Join (cont..)

Cross join:

- The number of rows in the first table multiplied by the number of rows in the second table if no WHERE clause is used along with CROSS JOIN.
- This kind of result is called as Cartesian Product.
- If WHERE clause is used with CROSS JOIN, it functions like an INNER JOIN.
- An alternative way of achieving the same result is to use column names separated by commas after SELECT and mentioning the table names involved, after a FROM clause.

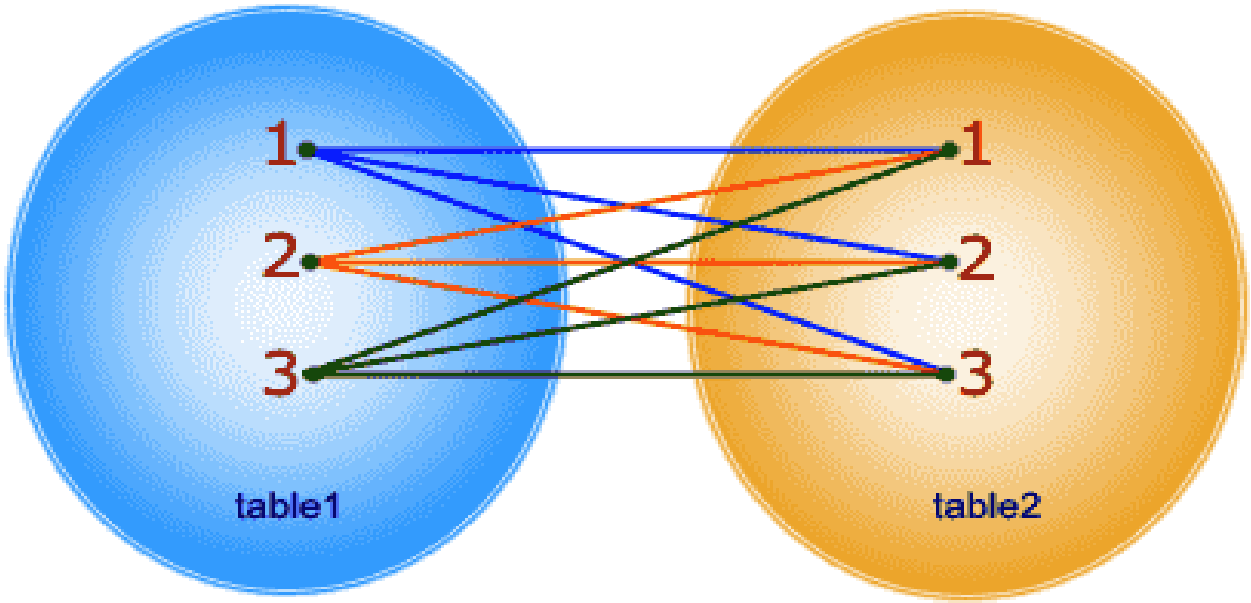
Syntax:

```
SELECT * FROM table1 CROSS JOIN table2;
```

Join (cont..)

Cross join:

SELECT * FROM table1 CROSS JOIN table2;



In CROSS JOIN, each row from 1st table joins with all the rows of another table.
If 1st table contain x rows and y rows in 2nd one the result set will be x * y rows.

Join (cont..)

Cross join:

- **SQL Code:**

```
SELECT foods.item_name,foods.item_unit,
company.company_name,company.company_city FROM foods
CROSS JOIN company;
```

or

- **SQL Code:**

```
SELECT foods.item_name,foods.item_unit,
company.company_name,company.company_city FROM
foods,company;
```

Join (cont..)

Cross join:

```
SELECT foods.item_name,foods.item_unit,
company.company_name,company.company_city
FROM foods
CROSS JOIN company;
```

Co.ID	Co.Name	Co.City
18	Order All	Boston
15	Jack Hill Ltd	London
16	Akas Foods	Delhi
17	Foodies.	London
19	sip-n-Bite.	New York

Company

I.ID	ITEM_NAME	I.Unit	Co.ID
1	Chex Mix	Pcs	16
6	Cheez-It	Pcs	15
2	BN Biscuit	Pcs	15
3	Mighty Munch	Pcs	17
4	Pot Rice	Pcs	15
5	Jaffa Cakes	Pcs	18
7	Salt n Shake	Pcs	-

Foods

Join (cont..)

Self join:

- A table is joined with itself (which is also called Unary relationships), especially when the table has a FOREIGN KEY which references its own PRIMARY KEY.
- To join a table itself means that each row of the table is combined with itself and with every other row of the table.
- The self join can be viewed as a join of two copies of the same table.

Join (cont..)

Self join:

Example - a table EMPLOYEE, that has one-to-many relationship.

Code to create the table EMPLOYEE

SQL Code:

```
CREATE TABLE employee
(emp_id varchar(5) NOT NULL,
emp_name varchar(20) NULL,
dt_of_join date NULL,
emp_supv varchar(5) NULL,
CONSTRAINT emp_id PRIMARY KEY(emp_id) ,
CONSTRAINT emp_supv FOREIGN KEY(emp_supv) REFERENCES
```

Join (cont..)

Self join:

Column Name	Data Type	Nullable	Default	Primary Key
EMP_ID	VARCHAR2(5)	No	-	1
EMP_NAME	VARCHAR2(20)	Yes	-	-
DT_OF_JOIN	DATE	Yes	-	-
EMP_SUPV	VARCHAR2(5)	Yes	-	-

1 - 4

Constraint	Type	Table
SYS_C004074	C	EMPLOYEE
EMP_ID	P	EMPLOYEE
EMP_SUPV	R	EMPLOYEE

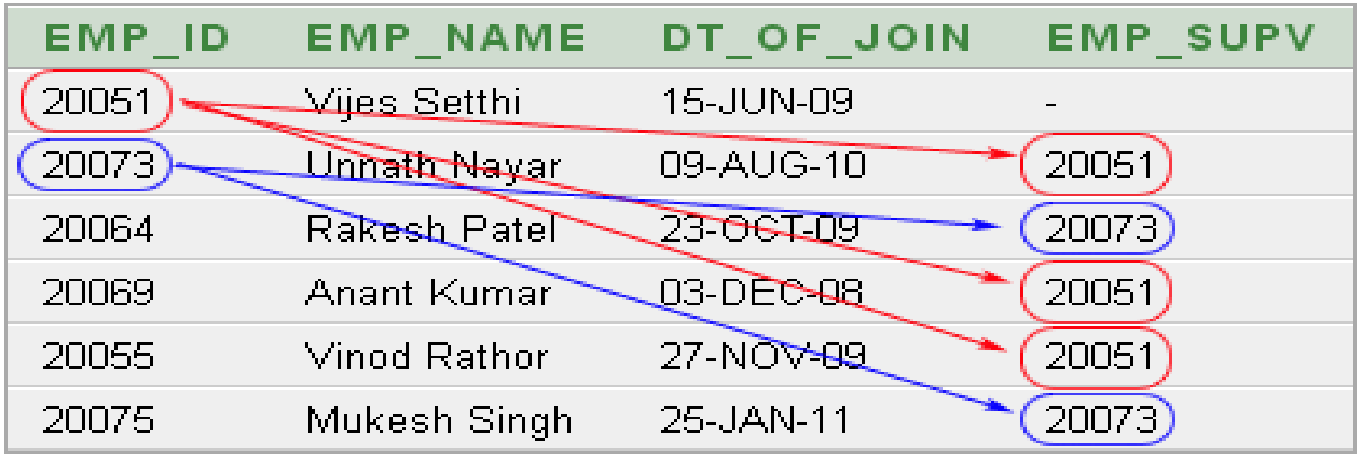
Primary key

Foreign key

Referencing EMP_ID of this table

Join (cont..)

Self join:



EMP_ID	EMP_NAME	DT_OF_JOIN	EMP_SUPV
20051	Vijes Setthi	15-JUN-09	-
20073	Unnath Nayar	09-AUG-10	20051
20064	Rakesh Patel	23-OCT-09	20073
20069	Anant Kumar	03-DEC-08	20051
20055	Vinod Rathor	27-NOV-09	20051
20075	Mukesh Singh	25-JAN-11	20073

SQL Code:

```
SELECT a.emp_id AS "Emp_ID",a.emp_name AS "Employee Name",
b.emp_id AS "Supervisor ID",b.emp_name AS "Supervisor Name"
FROM employee a, employee b WHERE a.emp_supv = b.emp_id;
```

Join (cont..)

Self join:

o/p: shown below for the previous query

Emp_ID	Employee Name	Supervisor ID	Supervisor Name
20055	Vinod Rathor	20051	Vijes Setthi
20069	Anant Kumar	20051	Vijes Setthi
20073	Unnath Nayar	20051	Vijes Setthi
20075	Mukesh Singh	20073	Unnath Nayar
20064	Rakesh Patel	20073	Unnath Nayar

Views

- In some cases, it is not desirable for all users to see the entire logical model (that is, all the actual relations stored in the database.)
- Consider a person who needs to know an instructors name and department, but not the salary. This person should see a relation described, in SQL, by

  select *ID*, *name*, *dept_name*
  from *instructor*

- A **view** provides a mechanism to hide certain data from the view of certain users.
- Any relation that is not of the conceptual model but is made visible to a user as a "virtual relation" is called a **view**.

Views (cont..)

- A VIEW (**virtual table**) is actually a query and the output of the query becomes the content of the view.
- The VIEW can be treated as a base table and it can be QUERIED, UPDATED, INSERTED INTO, DELETED FROM and JOINED with other tables and views.
- **A VIEW is a data object which does not contain any data. Its contents are the resultant of a base table.**
- A view can be accessed with the use of SQL SELECT statement like a table.

CREATE VIEW - create view *v* as < query expression >

Views (cont..)

Syntax:

CREATE VIEW view_name AS

SELECT column_list

FROM table_name

WHERE condition;

For Example: To create a view on the product table the sql query would be like

CREATE VIEW view_product
AS
SELECT product_id, product_name
FROM product;

Query: To access view

Select * from view_product;

View Examples

- A view of instructors without their salary

  create view *faculty* **as**

  select *ID*, *name*, *dept_name*

  from *instructor*

- Find all instructors in the Biology department

  select *name*

  from *faculty* ……. *Accessing view*

  where *dept_name* = 'Biology'

- Create a view of department salary totals

  create view *departments_total_salary*(*dept_name*, *total_salary*) **as**

  select *dept_name*, **sum** (*salary*)

  from *instructor*

  group by *dept_name*;

View Examples

- **Example of view with more than one table :**

List employees associated with design department

```
CREATE VIEW vw_Design_Emp AS

SELECT e.EMP_ID, e.EMP_FIRST_NAME, e.EMP_LAST_NAME, d.DEPT_ID,

d.DEPT_NAME

FROM EMPLOYEE e, DEPARTMENT d

WHERE e.DEPT_ID = d.DEPT_ID  AND d.DEPT_NAME = 'DESIGN';
```

Query:

Select * from **vw_Design_Emp**;

Views Defined Using Other Views

- **create view** *physics_fall_2009* **as**
  select *course.course_id, sec_id, building, room_number*
  from *course, section*
  where *course.course_id = section.course_id*
  and *course.dept_name* = 'Physics'
  and *section.semester* = 'Fall'
  and *section.year* = '2009';

For a physics_fall_2009 course, get course ID and room_number only in building 'watson'

- **create view** *physics_fall_2009_watson* **as**
  select *course_id, room_number*
  from *physics_fall_2009*
  where *building*= 'Watson';

Views

UPDATE VIEW:

- Modify the definition of a SQL VIEW without dropping it by using the SQL CREATE OR REPLACE VIEW Statement.
- Syntax:

CREATE OR REPLACE VIEW view_name

AS SELECT columns FROM table

[WHERE conditions];

Views

- **Example of view with more than one table :**

List employees associated with design department

```
CREATE VIEW vw_Design_Emp AS
SELECT e.EMP_ID, e.EMP_FIRST_NAME, e.EMP_LAST_NAME, d.DEPT_ID,
d.DEPT_NAME
FROM EMPLOYEE e, DEPARTMENT d
WHERE e.DEPT_ID = d.DEPT_ID  AND d.DEPT_NAME = 'DESIGN';
```

UPDATE VIEW:

```
CREATE or REPLACE VIEW vw_Design_Emp AS
SELECT e.EMP_ID, e.EMP_FIRST_NAME, e.EMP_LAST_NAME,
d.DEPT_ID, d.DEPT_NAME
FROM EMPLOYEE e, DEPARTMENT d
WHERE e.DEPT_ID = d.DEPT_ID  AND d.DEPT_NAME = 'ENGINEER';
```

Views

- **WITH CHECK OPTION**

WITH CHECK OPTION is a CREATE VIEW statement option. The purpose is to ensure that all UPDATE and INSERTs satisfy the condition(s) in the view definition.

- **Example:**

CREATE VIEW CUSTOMERS_VIEW AS

SELECT name, age FROM CUSTOMERS

WHERE age IS NOT NULL

WITH CHECK OPTION;

(Explanation: SELECT name, age FROM CUSTOMERS

WHERE age IS NOT NULL

-> select all records where age tuple is not null and

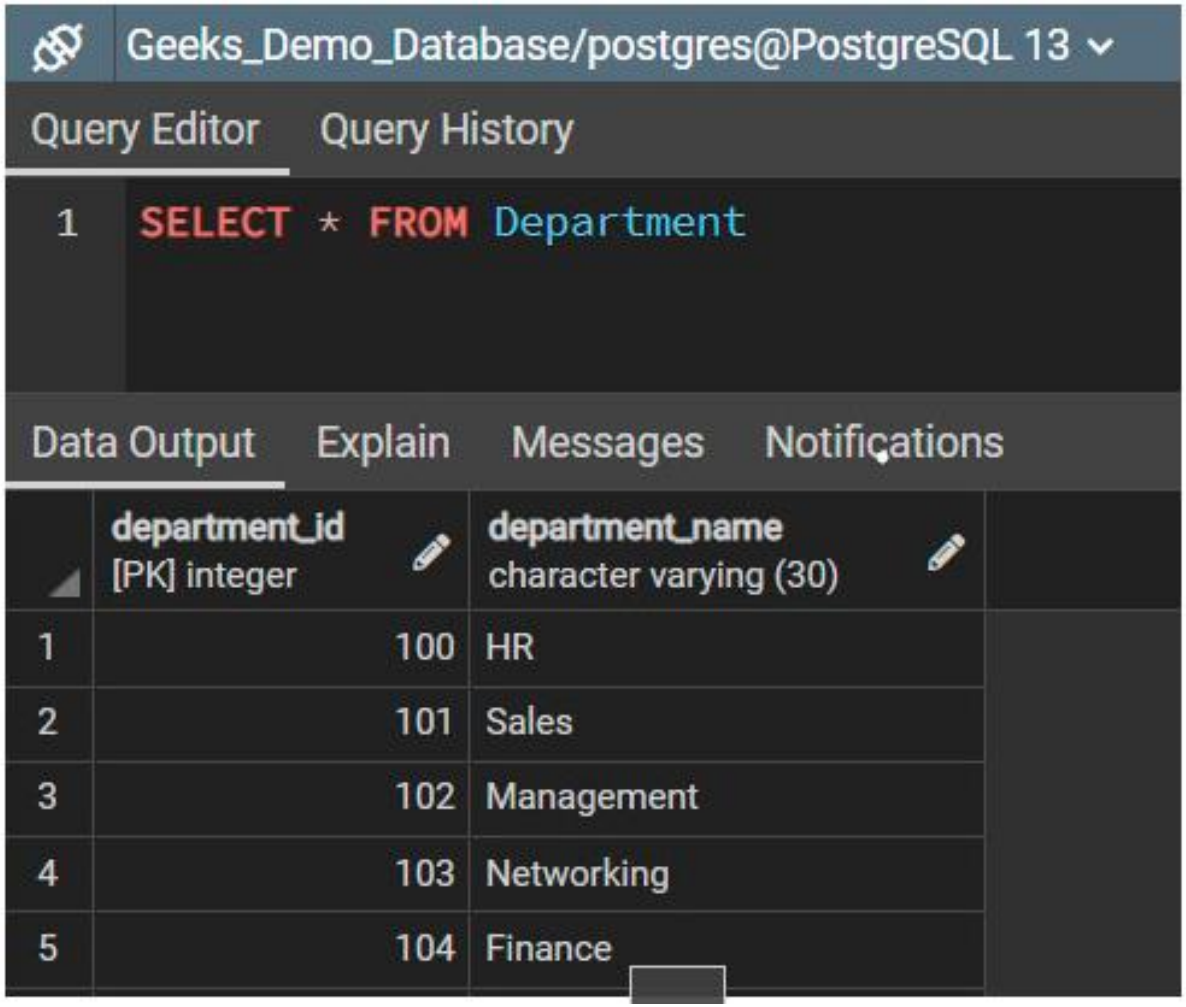
WITH CHECK OPTION;

-> it won't allow further in insert and update statement to enter null value)

Insert, delete , update possible with view

But…….

1. Department Details



2. Employee Details

Geeks_Demo_Database/postgres@PostgreSQL 13

Query Editor | Query History

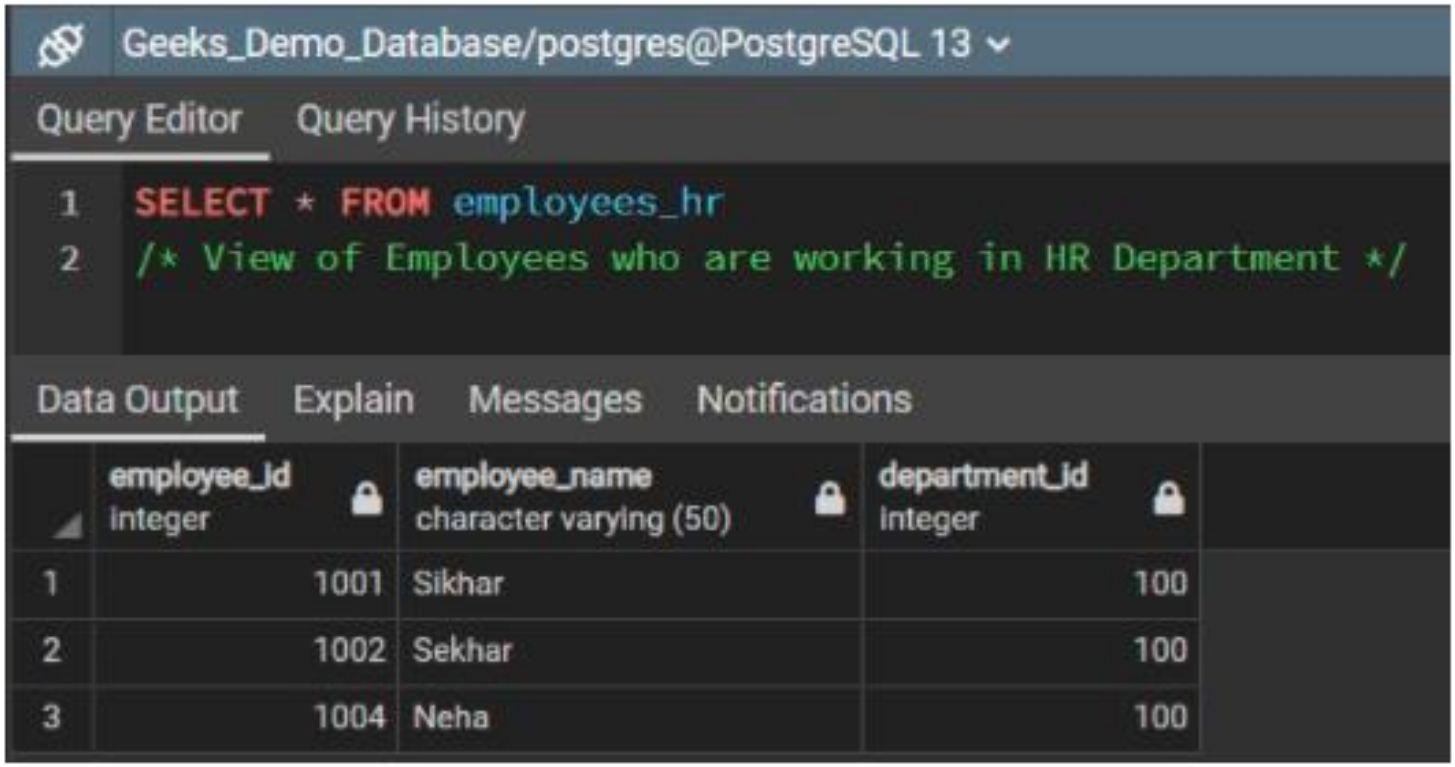
```
1  SELECT * FROM Employee
```

Data Output | Explain | Messages | Notifications

	employee_id [PK] integer	employee_name character varying (50)	department_id integer
1	1001	Sikhar	100
2	1002	Sekhar	100
3	1003	Rohit	101
4	1004	Neha	100
5	1005	Susmita	101
6	1006	Suman	102
7	1007	Raju	102
8	1008	Vishal	102
9	1009	Rimli	102
10	1030	Rishabh	102
11	1010	Ramesh	101
12	1020	Srishti	102

Now, let's create an updatable view for the Employee Table. Let's make a view for all the employees who are working in the "HR Department" of the company.

```
CREATE VIEW employees_hr
AS SELECT employee_id,employee_name,department_id
FROM Employee
WHERE department_id=100
```



Adding any data (not as per view) to actual table through view

Since it is an updatable view, we can insert values and it will reflect back in the table Employee. If the "Department_ID" of the newly inserted value **is equal to 100**, then it will be added in both the Employee Table and Employees_HR view table else it won't be added in this view HR because the WHERE condition becomes FALSE.

Query Editor | Query History

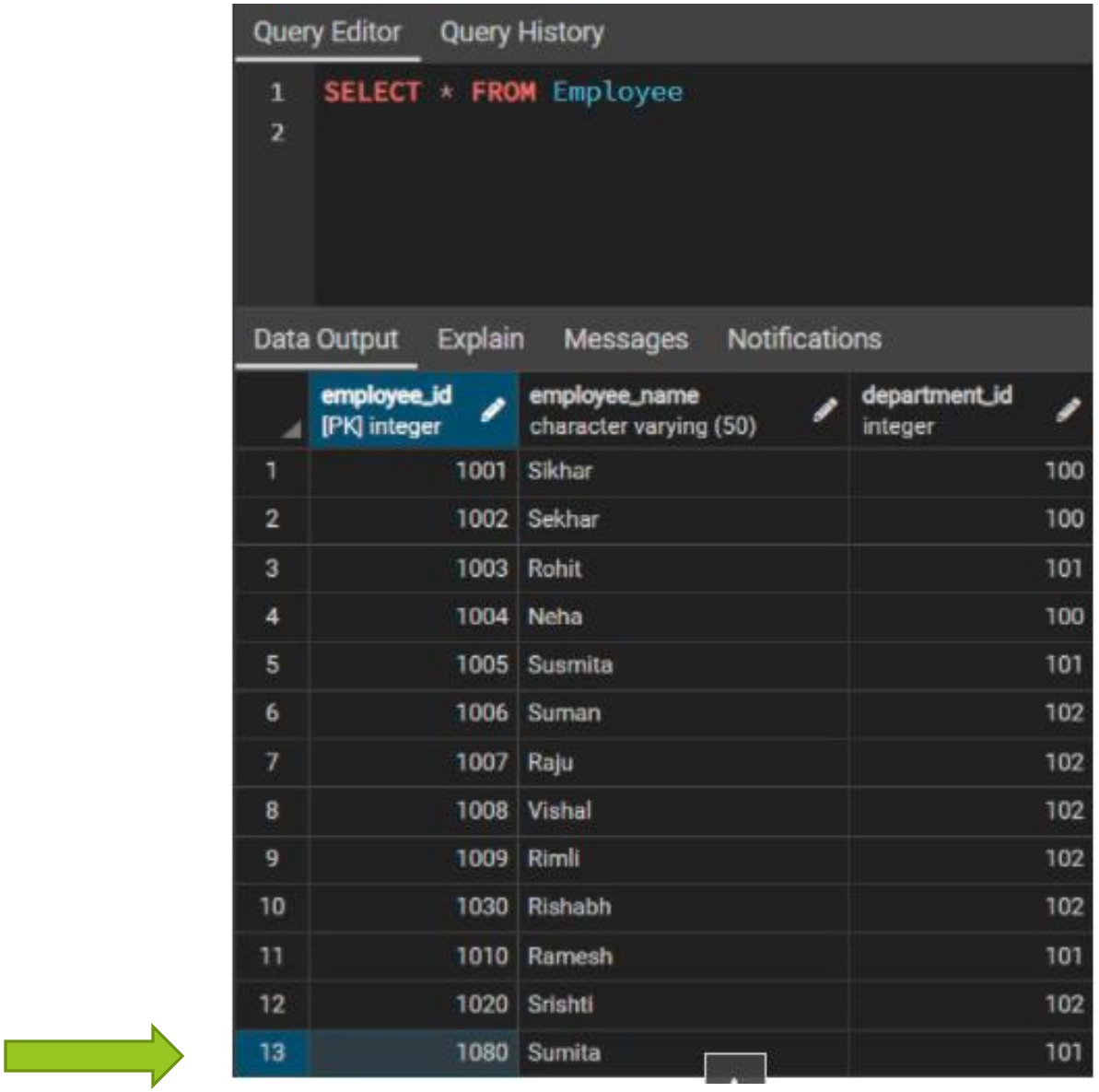
```
1  INSERT INTO employees_hr(employee_id,employee_name,department_id)
2  VALUES(1080,'Sumita',101)
3
4  /* It is updatble view, we can insert any values which will also
5  add in the parent table Employee */
```

Data Output | Explain | Messages | Notifications

```
INSERT 0 1

Query returned successfully in 51 msec.
```

New Data insertion in Updatble View

Adding any data (not as per view) to actual table through view



Query Editor Query History

```
1  SELECT * FROM Employee
2
```

Data Output Explain Messages Notifications

	employee_id [PK] integer	employee_name character varying (50)	department_id integer
1	1001	Sikhar	100
2	1002	Sekhar	100
3	1003	Rohit	101
4	1004	Neha	100
5	1005	Susmita	101
6	1006	Suman	102
7	1007	Raju	102
8	1008	Vishal	102
9	1009	Rimli	102
10	1030	Rishabh	102
11	1010	Ramesh	101
12	1020	Srishti	102
13	1080	Sumita	101

Update data through view

Now, we can also remove a view and make it invisible by making an UPDATE query which is shown below. Suppose we remove Neha from the "HR view" and set to some other department and also change her name to "Nehaa".

```
UPDATE employees_hr
SET
    employee_name='Nehaa',
    department_id=104
WHERE
employee_id=1004

/* Removing view Neha from HR by updating her name and department */
```

Update data through view - example

Geeks_Demo_Database/postgres@PostgreSQL 13

Query Editor | Query History

```
1  SELECT * FROM employees_hr
2
3  /* Neha is no more in this view */
```

Data Output | Explain | Messages | Notifications

	employee_id integer	employee_name character varying (50)	department_id integer
1	1001	Sikhar	100
2	1002	Sekhar	100

Neha is Invisible now in this view

Update data through view - example

Geeks_Demo_Database/postgres@PostgreSQL 13

Query Editor | Query History

```
1  SELECT * FROM Employee
2
3  /* Neha is updated to Nehaa and now is in Department 104 */
```

Data Output | Explain | Messages | Notifications

	employee_id [PK] integer	employee_name character varying (50)	department_id integer
1	1001	Sikhar	100
2	1002	Sekhar	100
3	1003	Rohit	101
4	1005	Susmita	101
5	1006	Suman	102
6	1007	Raju	102
7	1008	Vishal	102
8	1009	Rimli	102
9	1030	Rishabh	102
10	1010	Ramesh	101
11	1020	Srishti	102
12	1080	Sumita	101
13	1004	Nehaa	104

WITH CHECK OPTION

This is the problem in an updatable view without WITH CHECK OPTION. The user who can access the "view HR" can easily modify the data and a change in it will reflect back into the parent table. This is not a good sign for any organization because they are giving access to the users to change the data. People can put wrong data and also change the designation of an employee. Hence, the data is no longer safe in the database.

So, to overcome this we can use the WITH CHECK OPTION clause. Let's make another view for the employees who are working in the Sales Department having Department ID = 101.

```
CREATE VIEW employees_Sales AS
SELECT * FROM Employee
WHERE department_id=101 WITH CHECK OPTION
```

WITH CHECK OPTION – ex – view with sales department

Query Editor | Query History

```
1  SELECT * FROM employees_Sales
2
3  /* VIEW of employees working in Sales Department */
```

Data Output | Explain | Messages | Notifications

	employee_id integer	employee_name character varying (50)	department_id integer
1	1003	Rohit	101
2	1005	Susmita	101
3	1010	Ramesh	101
4	1080	Sumita	101

View of Sales Department

INSERT to emp_sales view

Query Editor | Query History

```
1  INSERT INTO employees_Sales(employee_id,employee_name,department_id)
2  VALUES(1220,'Mohan',103)
3
4  /* Beauty of WITH CHECK OPTION */
```

Data Output | Explain | Messages | Notifications

```
ERROR:  new row violates check option for view "employees_sales"
DETAIL:  Failing row contains (1220, Mohan, 103).
SQL state: 44000
```

INSERT

UPDATE to emp_sales view

Query Editor | Query History

```
1  UPDATE employees_Sales
2  SET
3      employee_name='Roomesh',
4      department_id=103
5  WHERE
6      employee_id=1010
7
8  /* Beauty of WITH CHECK OPTION */
```

Data Output | Explain | Messages | Notifications

```
ERROR:  new row violates check option for view "employees_sales"
DETAIL:  Failing row contains (1010, Roomesh, 103).
SQL state: 44000
```

UPDATE

View – Reference

https://www.geeksforgeeks.org/postgresql-creating-updatable-views-using-with-check-option-clause/

Views (cont..)

- **Updating/Inserting values in a View:**

A view can be updated under certain conditions which are given below –

- The SELECT clause may not contain the keyword DISTINCT.
- The SELECT clause may not contain summary functions.
- The SELECT clause may not contain set operators.
- The SELECT clause may not contain an ORDER BY clause.
- The FROM clause may not contain multiple tables.
- The WHERE clause may not contain subqueries.
- The query may not contain GROUP BY or HAVING.
- Calculated columns may not be updated.
- All NOT NULL columns from the base table must be included in the view in order for the INSERT query to function.

Views (cont..)

Inserting values in a View:

INSERT INTO vw_dept VALUES (6,'hss')

Deleting values in a View:

delete from vw_dept where deptno = 6

DROP VIEW:

Syntax:

DROP VIEW view_name;

Views (cont..)

Example:

-create view vw_dept_without_check

as select deptno, dname, loc from dept

where loc is not null

-select * from vw_dept_without_check

-insert into vw_dept_without_check values(7,'arts','')

(it will add data in base table too. If with check option is provided while creating a view then it won't allow to add null value

)

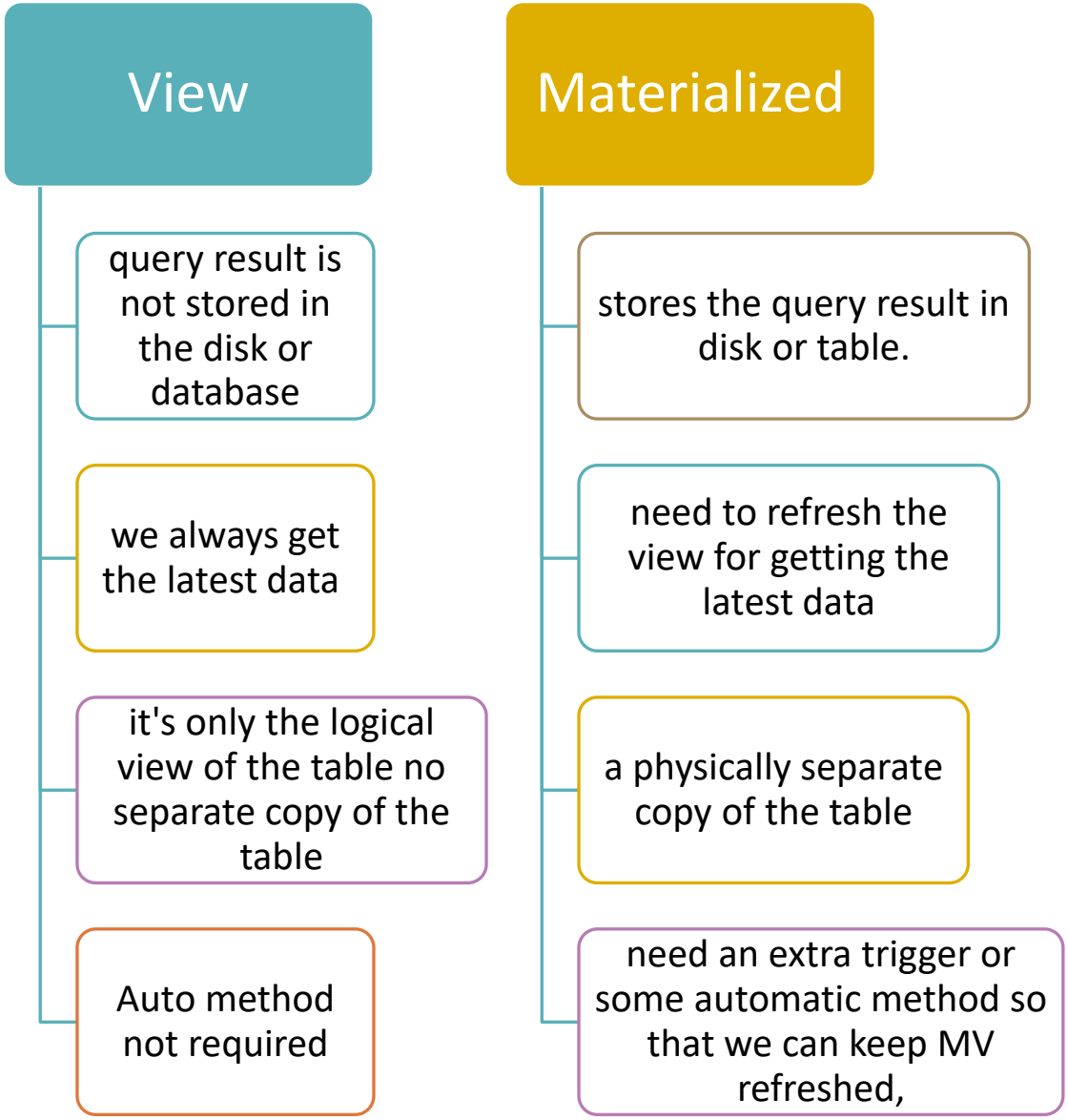
-select * from vw_dept_without_check

(it will show only not null records as per view created)

Materialized Views

- Materialized views are also the logical view of our data-driven by the select query but the result of the query will get stored in the table or disk, also the definition of the query will also store in the database.
- The performance of Materialized view it is better than normal View because
  - the data of materialized view will be stored in table and table may be indexed so faster for joining also joining is done at the time of materialized views refresh time so no need to every time fire joins statement as in case of view.

View vs Materialized Views



Integrity Constraints

- Database integrity refers to the validity and consistency of stored data.
- Integrity is usually expressed in terms of constraints, which are consistency rules that the database is not permitted to violate.
- Constraints may apply to each attribute or they may apply to relationships between tables.
- Integrity constraints ensure that changes (update deletion, insertion) made to the database by authorized users do not result in a loss of data consistency. Thus, integrity constraints guard against accidental damage to the database.

Integrity Constraints (cont..)

TYPES OF INTEGRITY CONSTRAINTS

- Domain Integrity
- Entity Integrity Constraint
- Referential Integrity Constraint

Integrity Constraints (cont..)

Domain Integrity

- Domain integrity means the definition of a valid set of values for an attribute.
- Data type, length or size, is null value allowed , is the value unique or not for an attribute ,the default value, the range (values in between) and/or specific values for the attribute.
- Example

create a table “student_info” with “stu_id” field having value greater than 100,

Query:

```
create domain id_value int constraint id_test check(value > 100);


create table student_info ( stu_id id_value PRIMARY KEY, stu_name varchar(30), stu_age int );
```

Integrity Constraints (cont..)

Domain Integrity – postgres example

Example 1

CREATE DOMAIN statement allows you to create an alias for a built-in data type, and assign range and value constraints:

```
CREATE DOMAIN addr VARCHAR(90) NOT NULL DEFAULT 'N/A';

CREATE DOMAIN idx INT CHECK (VALUE > 100 AND VALUE < 999);
```

Let's create a sample table using the created *domain* types:

```
CREATE TABLE location
(
  address addr,
  index idx
);
```

Note that *idx* domain contains *VALUE* keyword in the CHECK constraint that will be replaced by *index* column name when the constraint is checked.

Integrity Constraints (cont..)

Domain Integrity – postgres example

Example 2

-- create domain

Query: CREATE DOMAIN baseknowledge VARCHAR(90) NOT NULL DEFAULT 'N/A';

-- add new column in existing table

Query: alter table course add column course_base baseknowledge

-- check the result (select * from course)

-- drop column first

Query: alter table course drop column course_base

-- then domain can be dropped

Query: drop domain baseknowledge

Integrity Constraints (cont..)

Domain Integrity – postgres example

Reference:

https://www.sqlines.com/postgresql/statements/create_domain

Integrity Constraints (cont..)

- **Entity Integrity Constraint**

This integrity ensures that each record in the table is unique and has primary key which is not NULL.

STUDENT

STUDENT_ID	STUDENT_NAME	ADDRESS
100	Joseph	Alaiedon Township
101	Allen	Fraser Township
102	Chris	Clinton Township
103	Patty	Troy

Integrity Constraints (cont..)

Referential Integrity Constraint

- Let *r*1 and *r*2 be relations whose set of attributes are *R*1 and *R*2, respectively, with primary keys *K*1 and *K*2. We say that a subset of *R*2 is a **foreign key** referencing *K*1 in relation *r*1 if it is required that, for every tuple *t*2 in *r*2, there must be a tuple *t*1 in *r*1 such that *t*1.*K*1 = *t*2..
- Requirements of this form are called **referential-integrity constraints**, or **subset dependencies**.

Trigger

- Triggers are stored programs, which are automatically executed or fired when some events occur.
- Triggers are written to be executed in response to any of the following events −
  - A **database manipulation (DML)** statement (DELETE, INSERT, or UPDATE)
  - A **database definition (DDL)** statement (CREATE, ALTER, or DROP).
- Triggers can be defined on the table, view, schema, or database with which the event is associated.

Trigger (cont..)

A typical trigger has three components:

1. **Event**: insert/delete/update operation
2. **Condition**: Test whether trigger should run or not.

   Once the triggering event has occurred, an *optional* condition may be evaluated.
3. **Action**: The action is usually a sequence of SQL statements, but it could also be a database transaction or an external program that will be automatically executed.

Trigger (cont..)

Creating Triggers

- The syntax for creating a trigger is –

```
CREATE [OR REPLACE ] TRIGGER trigger_name
{BEFORE | AFTER | INSTEAD OF }
{INSERT [OR] | UPDATE [OR] | DELETE}
[OF col_name] ON table_name
[REFERENCING OLD AS o NEW AS n]
[FOR EACH ROW]
WHEN (condition)
DECLARE Declaration-statements
 BEGIN Executable-statements
 EXCEPTION Exception-handling-statements
 END;
/
```

Trigger (cont..)

Where,

- CREATE [OR REPLACE] TRIGGER trigger_name − Creates or replaces an existing trigger with the *trigger_name*.
- {BEFORE | AFTER | INSTEAD OF} − This specifies when the trigger will be executed. The INSTEAD OF clause is used for creating trigger on a view.
- {INSERT [OR] | UPDATE [OR] | DELETE} − This specifies the DML operation.
- [OF col_name] − This specifies the column name that will be updated.
- [ON table_name] − This specifies the name of the table associated with the trigger.
- [REFERENCING OLD AS o NEW AS n] − This allows you to refer new and old values for various DML statements, such as INSERT, UPDATE, and DELETE.
- [FOR EACH ROW] − This specifies a row-level trigger, i.e., the trigger will be executed for each row being affected. Otherwise the trigger will execute just once when the SQL statement is executed, which is called a table level trigger.
- WHEN (condition) − This provides a condition for rows for which the trigger would fire. This clause is valid only for row-level triggers.

Row vs. Statement Triggers

- A trigger can be invoked (by the DBMS) before or after the triggering event.
- There are two types of triggers:
  - a row level trigger *and*
  - a statement level trigger.

  - A **row level trigger** is defined using the clause **`FOR EACH ROW`**. If this clause is not given, the trigger is assumed to be a **statement trigger**.
  - A row trigger executes once for each row after (or before) the event.
  - In contrast, a **statement trigger** is executed once after (or before) the event, independent of how many rows are affected by the event.

Row Triggers

- Only with a row trigger it is possible to access the attribute values of a tuple before and after the modification
  - This is because the trigger is executed once for each tuple.

- For an update trigger, the old attribute value can be accessed using **:old.<column>**
- and the new attribute value can be accessed using

  :new.<column>

- For an insert trigger, only **:new.<column>** can be used.
- For a delete trigger only **:old.<column>** can be used.

- In a row trigger thus it is possible to specify comparisons between old and new attribute values in e.g.,

```
if :old.sal != :new.sal then . . .
```

Example

- Let's create the following two tables:

```
CREATE TABLE T1 (
  a INTEGER,
  b CHAR(10)
);
CREATE TABLE T2 (
  c CHAR(10),
  d INTEGER
);
```

- Problem:
  - We create a trigger that may insert a tuple into T2 when a tuple is inserted into T1.
  - Specifically, the trigger checks whether the new tuple has a first component 10 or less, and if so inserts the reverse tuple into T2:

Example

```
CREATE OR REPLACE TRIGGER trig1
     AFTER INSERT ON T1
     FOR EACH ROW
     WHEN (new.a <= 10)
BEGIN
          INSERT INTO T2 VALUES(:new.b, :new.a);
END trig1;
/
```

- Notice that we end the CREATE TRIGGER statement with a /, as for all PL/SQL statements in general.
  - This only creates the trigger; it does not execute the trigger. Only a triggering event, such as an insertion into T1 in this example, causes the trigger to execute.

Try now:
`insert into T1 values(8, 'Uvic');`
The tuple `('Uvic',8)` is automatically inserted into T2.

Aborting Triggers with Errors

- Triggers can often be used to enforce constraints.
- The **`WHEN`** clause or body of the trigger can check for the violation of certain conditions and signal an error accordingly using the Oracle built-in function **`RAISE_APPLICATION_ERROR`**.
- The action that activated the trigger (insert, update, or delete) would be aborted.

Aborting Triggers with Errors (cont..)

- For example, the following trigger enforces the constraint Person.age >= 0:

```
CREATE TABLE Person (age INT);

CREATE TRIGGER PersonCheckAge
      BEFORE INSERT OR UPDATE OF age ON Person
      FOR EACH ROW
BEGIN
   IF (:new.age < 0) THEN
        RAISE_APPLICATION_ERROR(-20000, 'no negative age
allowed');
   END IF;
END;
/
```

Aborting Triggers with Errors (Cont'd)

- If we attempted to execute the insertion:

```
INSERT INTO Person VALUES (-3);
```

we would get the error message:

```
ERROR at line 1:
ORA-20000: no negative age allowed
ORA-06512: at "MYNAME.PERSONCHECKAGE", line 3
ORA-04088: error during execution of trigger
'MYNAME.PERSONCHECKAGE'
```

and nothing would be inserted.

In general, the effects of both the trigger and the triggering statement are rolled back.

Trigger Examples

- Example 1: - to keep previous and current records

```
CREATE OR REPLACE TRIGGER tr_marks_log
        AFTER UPDATE ON MARKS
        FOR EACH ROW
BEGIN
        INSERT INTO marks_log (ROLLNO, OLD_MARK, NEW_MARK)
         VALUES (:old.rollno,:old.marks, :new.marks);
END;
/
```

- Marks table = columns roll_no and marks with values (1513101, 75)

Queries:

- update marks set marks = 90 where rollno = 1513101
- select * from marks_log

output:

ROLLNO	OLD_MARK	NEW_MARK
1513101	75	90

Trigger Examples

- Example 2:

If the employee salary increased by more than 10%, make sure the 'rank' field is not empty and its value has changed, otherwise reject the update (rank should be increased with increase in salary)

Query:

```
Create or Replace Trigger EmpSal
Before Update On emp
For Each Row
Begin
 IF (:new.sal > (:old.sal * 1.1)) Then
 IF (:new.rank is null or :new.rank = :old.rank) Then
 RAISE_APPLICATION_ERROR(-20004, 'rank field not correct');
 End IF;
 End IF;
End;
/
```

Trigger Examples

Queries:

1. update emp set sal= 7000, rank = 1 where empno = 7782

o/p:

ORA-20004: rank field not correct

ORA-06512: at "COLLEGE.EMPSAL", line 4

ORA-04088: error during execution of trigger 'COLLEGE.EMPSAL‘

2. update emp set sal= 7000, rank = 2 where empno = 7698

o/p:

1 row updated

Trigger Examples

Example 3:

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

```
Create Trigger EmpDate
Before Insert On Employee
For Each Row
Declare
        temp date;
Begin
        Select sysdate into temp from dual;
        IF (:new.hiredate is null) Then
                :new.hiredate := temp;
        End IF;
End;
/
```

Trigger Examples

Example 3:

Queries:

insert into emp(empno, ename) values(123, 'abc')

o/p;

123 abc 2/25/2020

Trigger Examples

Example 4:

To calculate derived attribute

```
Create Trigger studentage
Before Insert Or Update On student
For Each Row
Begin
 :new.age := SYSDATE - :new.dob ;
End;
/
```

Queries:

Sysdate: ‘3/2/2020’

update student set dob = '3/1/2020' where rollno = 102

o/p: age value in days

102	xyz	-	-	03/02/2020	2

Before Statement-Level Trigger

```
CREATE OR REPLACE TRIGGER emp_alert_trig
    BEFORE INSERT ON emp
BEGIN
    DBMS_OUTPUT.PUT_LINE(
       'New employees are about to be added');
END;
/
```

Don’t forget:

SET SERVEROUTPUT ON

Inserting

- The following INSERT is constructed so that several new rows are inserted upon a single execution of the command.
- For each row that has an employee id between 7900 and 7999, a new row is inserted with an employee id incremented by 1000.

```
INSERT INTO emp (empno, ename, deptno)
  SELECT empno + 1000, ename, 40
  FROM emp
  WHERE empno BETWEEN 7900 AND 7999;
```

```
Trigger fired - New employees are about to
  be added
```

Result

```
SELECT empno, ename, deptno
FROM emp
WHERE empno BETWEEN 8900 AND 8999;
```

EMPNO	ENAME	DEPTNO
8900	JAMES	40
8902	FORD	40
8934	MILLER	40

After Statement-Level Trigger

- Whenever an insert, update, or delete operation occurs on the **emp** table, a row is added to the **empauditlog** table recording the date, user, and action.

- First let's create the **empauditlog** table:

```
CREATE TABLE empauditlog (
    audit_date      DATE,
    audit_user      VARCHAR2(20),
    audit_desc      VARCHAR2(20)
);
```

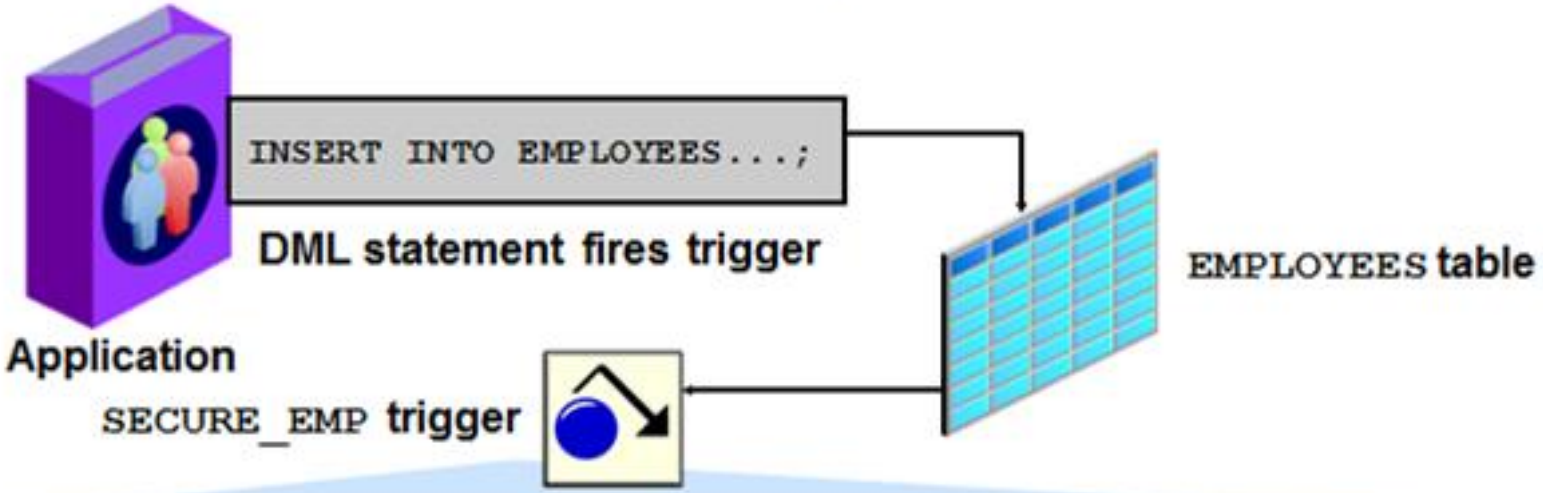
Now the trigger

```
CREATE OR REPLACE TRIGGER emp_audit_trig
    AFTER INSERT OR UPDATE OR DELETE ON emp
DECLARE
    v_action          VARCHAR2(20);
BEGIN
    IF INSERTING THEN
        v_action := 'Added employee(s)';
    ELSIF UPDATING THEN
        v_action := 'Updated employee(s)';
    ELSIF DELETING THEN
        v_action := 'Deleted employee(s)';
    END IF;
    INSERT INTO empauditlog VALUES (SYSDATE, USER,
        v_action);
END;
/
```

Let's trigger it…

```
INSERT INTO emp VALUES (9001,'SMITH',50);
INSERT INTO emp VALUES (9002,'JONES',50);

UPDATE emp
SET ename = 'SMITH BROWN'
WHERE empno=9001;

DELETE FROM emp WHERE empno IN (9001, 9002);

SELECT TO_CHAR(AUDIT_DATE,'DD-MON-YY HH24:MI:SS') AS "AUDIT DATE",
    audit_user, audit_desc
FROM empauditlog
ORDER BY 1 ASC;

AUDIT DATE          AUDIT_USER           AUDIT_DESC
------------------ -------------------- --------------------
31-OCT-06 11:11:30 THOMO                Added employee(s)
31-OCT-06 11:11:30 THOMO                Deleted employee(s)
31-OCT-06 11:11:30 THOMO                Updated employee(s)
31-OCT-06 11:11:30 THOMO                Added employee(s)
```

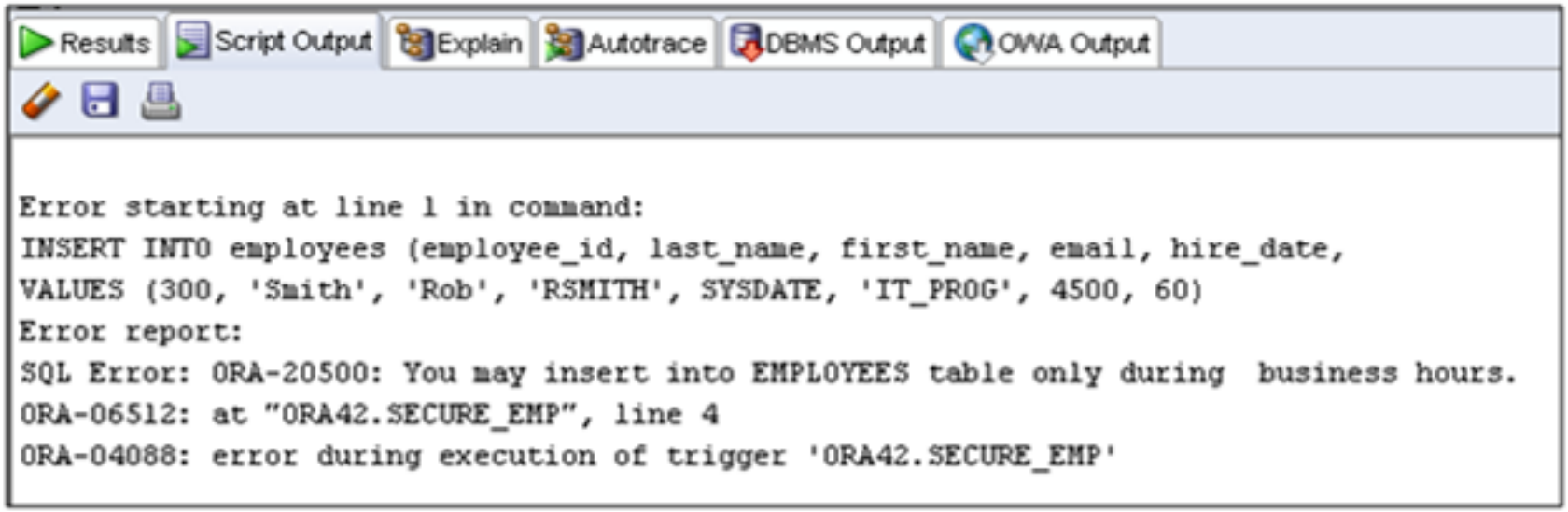
Creating a DML Statement Trigger Example: SECURE_EMP




```
CREATE OR REPLACE TRIGGER secure_emp
BEFORE INSERT ON employees
  BEGIN
   IF (TO_CHAR(SYSDATE,'DY') IN ('SAT','SUN')) OR
      (TO_CHAR(SYSDATE,'HH24:MI')
         NOT BETWEEN '08:00' AND '18:00') THEN
   RAISE_APPLICATION_ERROR(-20500, 'You may insert'
     ||' into EMPLOYEES table only during '
     ||' normal business hours.');
   END IF;
  END;
```

Testing Trigger SECURE_EMP

```
INSERT INTO employees (employee_id, last_name,
    first_name, email, hire_date, job_id, salary,
    department_id)
VALUES (300, 'Smith', 'Rob', 'RSMITH', SYSDATE,
      'IT_PROG', 4500, 60);
```



Statement-Level trigger

Statement-Level Triggers
Is the default when creating a trigger
Fires once for the triggering event
Fires once even if no rows are affected

Ex: security check on (user, time,…)

ROW-level Triggers

Row-Level Triggers
Use the `FOR EACH ROW` clause when creating a trigger.
Fires once for each row affected by the triggering event
Does not fire if the triggering event does not affect any rows

Ex: log the transactions

Trigger

- Benefits of Triggers
  - Generating some derived column values automatically
  - Enforcing referential integrity
  - Event logging and storing information on table access
  - Auditing
  - Synchronous replication of tables
  - Imposing security authorizations
  - Preventing invalid transactions

Assertions

- **Specifying General Constraints as Assertions**
- **Declarative assertions**, using the **CREATE ASSERTION** statement of the DDL.
- Each assertion is given a constraint name and is specified via a condition similar to the WHERE clause of an SQL query
- For example, to specify the constraint that *the salary of an employee must not be greater than the salary of the manager of the department that the employee works for* in SQL, write the following assertion:

```
CREATE ASSERTION SALARY_CONSTRAINT
CHECK ( NOT EXISTS
( SELECT *
FROM EMPLOYEE E, EMPLOYEE M,
DEPARTMENT D
WHERE E.Salary>M.Salary
AND E.Dno=D.Dnumber
AND D.Mgr_ssn=M.Ssn ) );
```


- If the result of the query is not empty, the assertion is violated.

Data Control Language

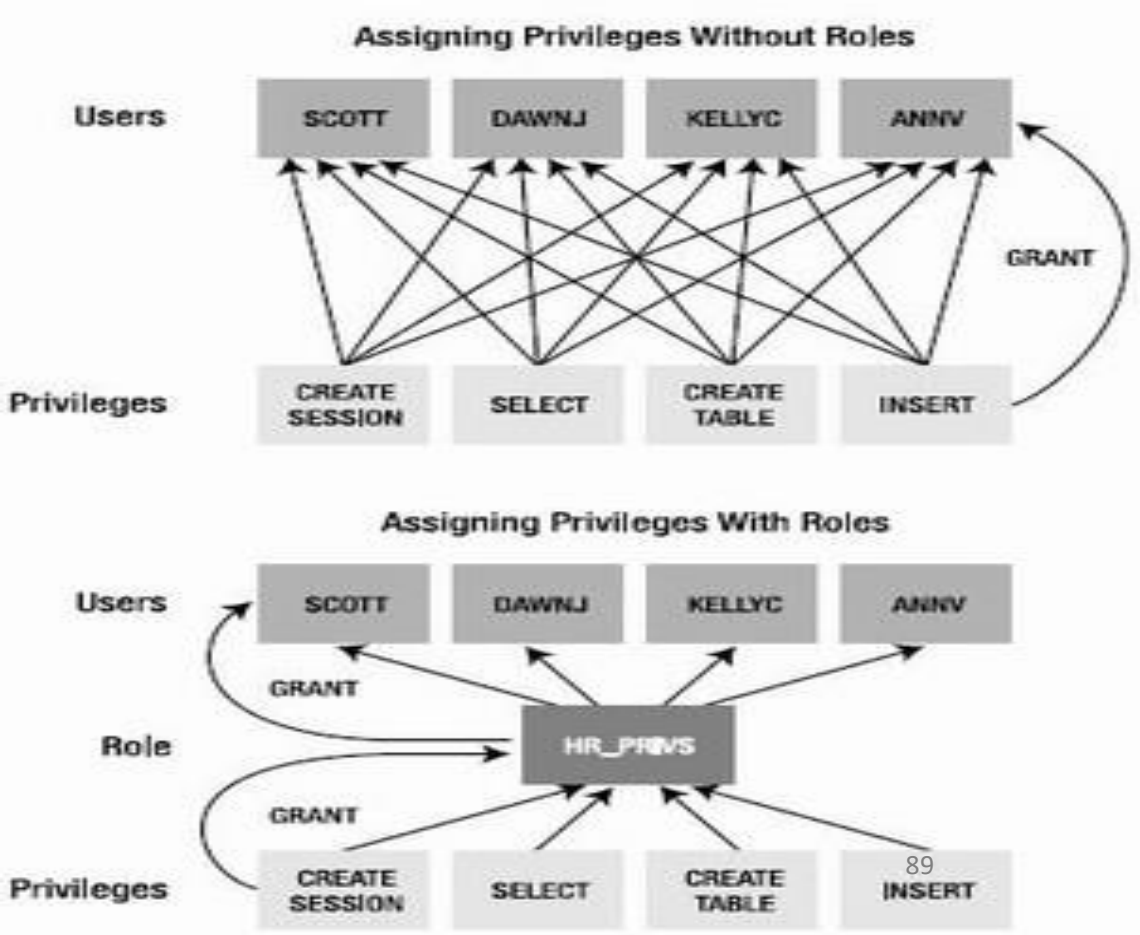
- DCL languages are used to control the user access to the database, tables, views, procedures, functions and packages.
- It gives different levels of access to the objects in the database.
- Privileges and Roles:
  - Permits only certain users to access, process or alter data.
  - Applying varying limitations on user accounts/actions.
- Privilege:
  - **System**: for performing admin task. It can create tablespace, user, drop tablespace etc. It should be restricted.
  - **Object**: Managing of privileges for different database objects. Select, update, execute etc.

All privileges are granted and revoked by GRANT and REVOKE Commands.

Data Control Language

- Roles:
  - Created by users (DBA) to group together privileges or other roles.
  - Used for quick and easy grant permissions to users.
  - Command:

    Create role **Manager** identified by **manager123;**



Data Control Language

GRANT Command:

- GRANT provides the privileges to the users on the database objects.
- The privileges could be select, delete, update and insert on the tables and views.
- On the procedures, functions and packages it gives select and execute privileges.
- We can either give all the privileges or any one or more privileges to the objects.

The syntax of GRANT is as below:

GRANT privilege_name ON object_name TO {user_name |role|public}

[IDENTIFIED BY password] [WITH GRANT OPTION]

Data Control Language

Where,

- ***Privilege_name*** is the level of access given to the users. Some of the access rights are ALL, DELETE, UPDATE, INSERT, EXECUTE and SELECT.
- ***Object_name*** is the name of a database object like TABLE, VIEW, PROCEDURE, FUNCTION, PACKAGE and SEQUENCE.
- ***User_name*** is the name of the user to whom an access is being granted.
- ***With Grant option***: that user can grant permissions to some other user
- Examples:

1. SELECT and INSERT grants are given to Mathew on STUDENT table.

```
GRANT SELECT, INSERT ON STUDENT TO Mathew;
```

2. grant gives the execution rights to Joseph on the stored procedure sp_getStudentNames.

```
GRANT EXECUTE on sp_getStudentNames to Joseph;
```

Data Control Language

- Examples:

3. Grant create table right to user

GRANT CREATE TABLE TO Mathew;

4. grant all privileges to user

sysdba is a set of priviliges which has all the permissions in it. So if we want to provide all the privileges to any user, we can simply grant them the sysdba permission.

GRANT sysdba TO username

5. Grant permission to create any table

GRANT CREATE ANY TABLE TO username

6. Grant permission to drop any table

GRANT DROP ANY TABLE TO username

Data Control Language

REVOKE Command:

REVOKE removes the privileges given on the database objects.

The syntax of REVOKE is as below:

REVOKE privilege_name ON object_name TO {user_name }

Examples:

1. Removes the INSERT grant **from** Mathew on STUDENT Table

REVOKE INSERT ON STUDENT FROM Mathew;

2. Removes the CREATE TABLE grant **from** Mathew

REVOKE CREATE TABLE FROM Mathew

(Note: Refer document for queries and o/p)

Authorization in SQL

- Authorization in SQL is achieved by Data Control Language Commands (DCL) are
  - Refer upcoming slides to learn more about authorization

Authorization

Forms of authorization on parts of the database:

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

Forms of authorization to modify the database schema

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

Authorization Specification in SQL

- 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 *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 name or view name> **from** <user list>

- Example:

  revoke select on *branch* **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.
- All privileges that depend on the privilege being revoked are also revoked.

Roles

- **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;

Authorization on Views

- **create view** *geo_instructor* **as**
  (**select** *
  from *instructor*
  where *dept_name* = 'Geology');
- **grant select on** *geo_instructor* **to** *geo_staff*
- Suppose that a *geo_staff* member issues
  - **select** *
    from *geo_instructor*;
- What if
  - *geo_staff* does not have permissions on *instructor?*
  - creator of view did not have some permissions on *instructor?*

Other Authorization Features

- **references** privilege to create foreign key
  - **grant reference** (*dept_name*) **on** *department* **to** Mariano;
  - why is this required?
- transfer of privileges
  - **grant select on** *department* **to** Amit **with grant option**;
  - **revoke select on** *department* **from** Amit, Satoshi **cascade**;
  - **revoke select on** *department* **from** Amit, Satoshi **restrict**;

Security in SQL

- A DBMS system always has a separate system for security which is responsible for protecting database against accidental or intentional loss, destruction or misuse.

- **Security Levels:**
  - Database level:- DBMS system should ensure that the authorization restriction needs to be there on users.
  - Operating system Level:- Operating system should not allow unauthorized users to enter in system.
  - Network Level:- Database is at some remote place and it is accessed by users through the network so security is required.

Security Mechanisms

- **Access Control(Authorization)**
  - Which identifies valid users who may have any access to the valid data in the Database and which may restrict the operations that the user may perform?
  - For Example The movie database might designate two roles: “users” (query the data only) and “designers”(add new data)user must be assigned to a role to have the access privileges given to that role.
  - Each applications is associated with a specified role. Each role has a list of authorized users who may execute/Design/administers the application.

Security Mechanisms

- **Cryptographic control/Data Encryption:**
  - Encode data in a cryptic form(Coded)so that although data is captured by unintentional user still he can't decode the data.
  - Used for sensitive data, usually when transmitted over communications links but also may be used to prevent by passing the system to gain access to the data.

Security Mechanisms

- **Authenticate the User:**
  - Which identify valid users who may have any access to the data in the Database?
  - Restrict each user's view of the data in the database
  - This may be done with help of concept of views in Relational databases.
- **Inference control:**
  - Ensure that confidential information can't be retrieved even by deduction.
  - Prevent disclosure of data through statistical summaries of confidential data.

Security Mechanisms

- **Flow control or Physical Protection:**
  - Prevents the copying of information by unauthorized person.
  - Computer systems must be physically secured against any unauthorized entry.
- **Virus control:**
  - At user level authorization should be done to avoid intruder attacks through humans.
  - There should be mechanism for providing protection against data virus.

Security Mechanisms

- **User defined control:**
  - Define additional constraints or limitations on the use of database.
  - These allow developers or programmers to incorporate their own security procedures in addition to above security mechanism.

References

- Navathe
- Korth