**PostgreSQL NULL**

--The PostgreSQL NULL is the term used to represent a missing value.

--A field with a NULL value is a field with no value.

--The basic syntax of using NULL while creating a table is as follows

--Here, NOT NULL signifies that column should always accept an explicit value of the given data type.

--There are two columns where we did not use NOT NULL.

ID NOT NULL

NAME NOT NULL

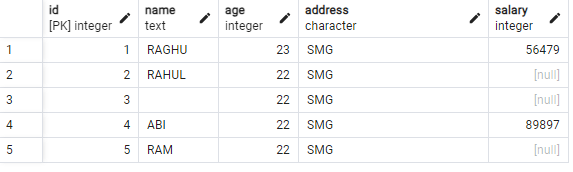
AGE NOT NULL

--these columns could be NULL.

ADDRESS

SALARY

SELECT \* FROM COMPANY



INSERTING VALUES TO THE TABLE COMPANY

INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) VALUES (5,'RAM',22,'SMG',NULL)

--in our example,

--the query would only return the row for “Raghu “, “Abi “ with salary :56479, 89897

--and it would omit "RAHUL" and "RAM", even though they might meet the condition because their salary are unknown (NULL).

To check if a value is NULL or not,

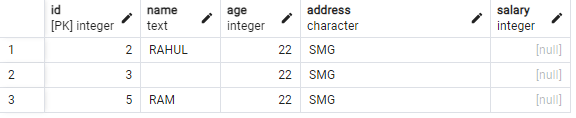
you cannot use the equal to (=) or not equal to (<>) operators. Instead, you use the IS NULL operator.

**value IS NULL**

SELECT ID, NAME, AGE, ADDRESS, SALARY

FROM COMPANY

WHERE SALARY IS NULL;

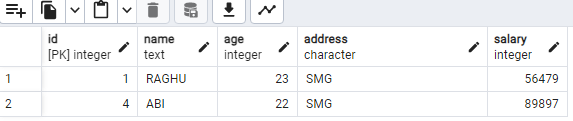


**value IS NOT NULL**

SELECT ID, NAME, AGE, ADDRESS, SALARY

FROM COMPANY

WHERE SALARY IS NOT NULL;



**Behavior of NULL in Comparisons:**

**Equality Comparison (=):**

Comparing a column value with NULL using the equality operator (=) results in UNKNOWN.

Example:

SELECT \* FROM table WHERE column = NULL;



Result: No rows are returned because comparing NULL with anything results in UNKNOWN.

Inequality Comparison (!= or <>):

**Inequality comparisons** (!= or <>) involving NULL also result in UNKNOWN.

Example:

SELECT \* FROM table WHERE column != NULL;

Result: No rows are returned due to the UNKNOWN result.

Comparison Operators (<, >, <=, >=):



**Comparison operators** behave similarly to equality and inequality operators when NULL is involved.

Example:

SELECT \* FROM table WHERE column > NULL;

Result: No rows are returned due to the UNKNOWN result.

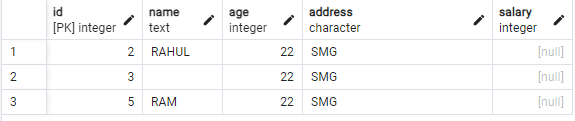


**Handling NULL in Query Logic:**

Use IS NULL/IS NOT NULL operators or COALESCE function to handle NULL values effectively in queries.

Example:

SELECT \* FROM table WHERE column IS NULL;

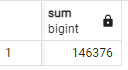


PostgreSQL doesn't have a built-in NULL-safe equality operator like some other databases.

**Aggregate functions** ignore NULL values by default. Use the FILTER clause to include or exclude NULL values explicitly.

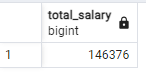
Example:

SELECT SUM(salary) FILTER (WHERE salary IS NOT NULL) FROM COMPANY;

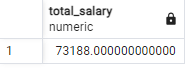


The **COALESCE** function in aggregate functions allows you to handle NULL values effectively when performing calculations.

SELECT **COALESCE**(**SUM**(SALARY), 0) AS total\_salary FROM COMPANY;



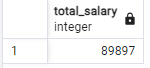
SELECT **COALESCE**(**avg**(SALARY), 0) AS total\_salary FROM COMPANY;



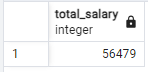
SELECT **COALESCE**(**count**(SALARY), 0) AS total\_salary FROM COMPANY;



SELECT **COALESCE**(**max**(SALARY), 0) AS total\_salary FROM COMPANY;



SELECT **COALESCE**(**min**(SALARY), 0) AS total\_salary FROM COMPANY;



**ORDER BY clause and NULL**

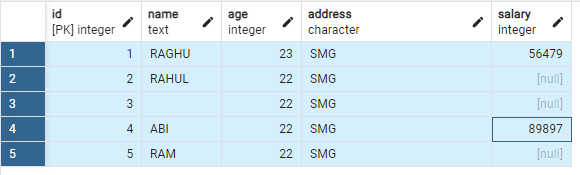
When you sort rows that contain NULL,

you can specify the order of NULL with other non-null values by using the NULLS FIRST or NULLS LAST option of the ORDER BY clause:

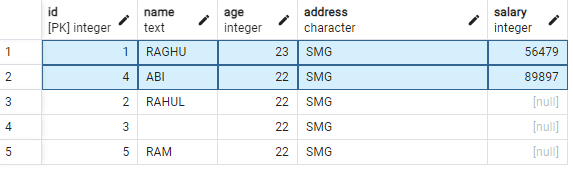
**The NULLS FIRST** option places NULL before other non-null values and the

**NULL LAST** option places NULL after other non-null values.

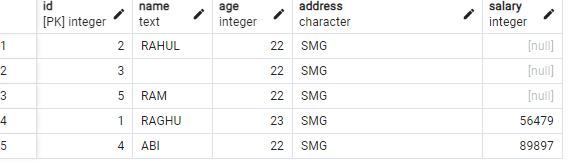
The following query returns data from the COMPANY table: (SALARY)



SELECT \* FROM COMPANY ORDER BY SALARY **NULLS LAST;**



SELECT \* FROM COMPANY ORDER BY SALARY **NULLS FIRST;**



**NULLIF():**

The NULLIF function returns a null value if argument\_1 equals to argument\_2, otherwise, it returns argument\_1

SELECT NULLIF (1, 1); -- return NULL



SELECT NULLIF (1, 0); -- return 1



**COALESCE():**

In PostgreSQL, the **COALESCE** function is used to return the first non-null expression among its arguments. It's particularly useful for handling NULL values in queries.

SELECT

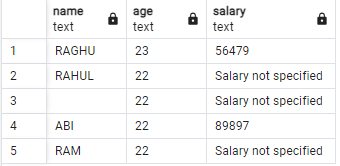
name,

COALESCE(age::TEXT, 'Age not specified') AS age,

COALESCE(salary::text, 'Salary not specified') AS salary

FROM

COMPANY;



****COALESCE Function:**** The **COALESCE** function returns the first non-NULL value among its arguments. It's commonly used to replace NULL values with a default value.

SELECT ID, NAME, AGE, COALESCE(ADDRESS, 'Address not specified') AS ADDRESS, COALESCE(SALARY, 0) AS SALARY FROM COMPANY;

****CASE Expression: null value****

SELECT

ID,

NAME,

AGE,

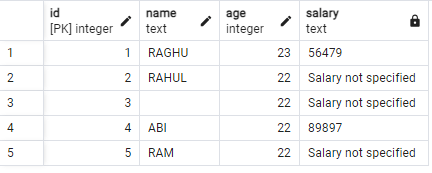
CASE

WHEN SALARY IS NULL THEN 'Salary not specified'

ELSE SALARY::TEXT -- Convert SALARY to text for consistency in output

END AS SALARY

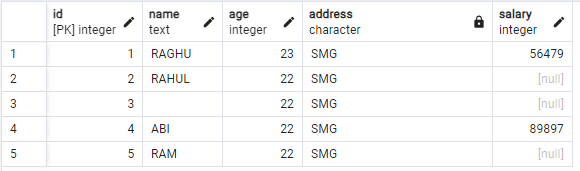
FROM COMPANY;



### **NULLIF Function:**

The **NULLIF** function returns NULL if two expressions are equal, otherwise it returns the first expression. It's useful for replacing specific values with NULL.

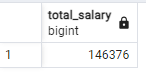
SELECT ID, NAME, AGE, NULLIF(ADDRESS, 'Not Available') AS ADDRESS, SALARY FROM COMPANY;



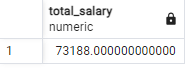
**COALESCE function in aggregate functions**

The **COALESCE** function in aggregate functions allows you to handle NULL values effectively when performing calculations.

SELECT COALESCE(SUM(SALARY), 0) AS total\_salary FROM COMPANY;



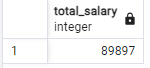
SELECT COALESCE(avg(SALARY), 0) AS total\_salary FROM COMPANY;



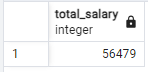
SELECT COALESCE(count(SALARY), 0) AS total\_salary FROM COMPANY;



SELECT COALESCE(max(SALARY), 0) AS total\_salary FROM COMPANY;



SELECT COALESCE(min(SALARY), 0) AS total\_salary FROM COMPANY;



**Handling NULL values in mathematical operations**

SELECT

ID,

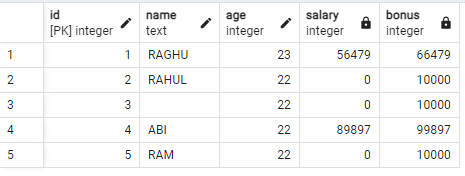
NAME,

AGE,

COALESCE(SALARY, 0) AS SALARY,

COALESCE(SALARY, 0) + 10000 AS BONUS

FROM COMPANY;



**Behavior of NULL in Joins:**

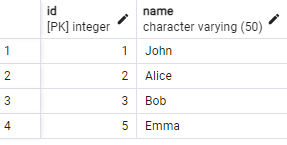


Table 1

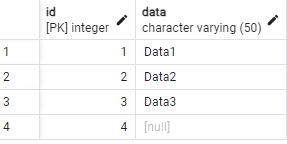


Table 2

**Inner Joins:**

In inner joins, rows with NULL values in the join columns are excluded from the result set.

SELECT \*

FROM table1

INNER JOIN table2 ON table1.id = table2.id;

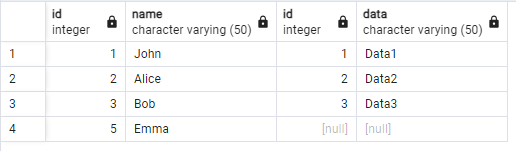


**Left outer joins** include all rows from the left table and matching rows from the right table. If no match is found, NULL values are included for columns from the right table.

SELECT \*

FROM table1

LEFT JOIN table2 ON table1.id = table2.id;

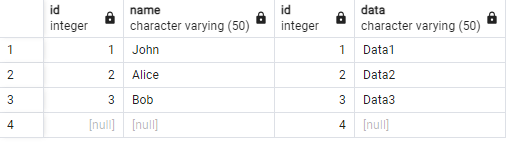


**Right outer joins** include all rows from the right table and matching rows from the left table. If no match is found, NULL values are included for columns from the left table.

SELECT \*

FROM table1

RIGHT JOIN table2 ON table1.id = table2.id;

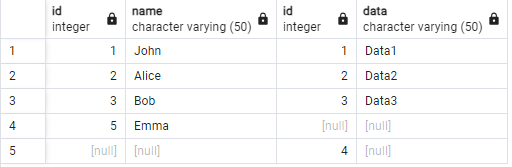


**Full outer joins** include all rows from both tables. If no match is found, NULL values are included for columns from the table without a match.

SELECT \*

FROM table1

FULL OUTER JOIN table2 ON table1.id = table2.id;



CREATE OR REPLACE PROCEDURE uspCaptureError(

ErrorLine BIGINT,

ErrorMessage TEXT,

SchemaName VARCHAR(500),

ErrorProcedure VARCHAR(500),

ErrorSeverity BIGINT,

ErrorState BIGINT,

SPID BIGINT,

ServerName VARCHAR(500),

ServiceName VARCHAR(500),

HostName VARCHAR(500),

ProgramName VARCHAR(1000),

CommandLine TEXT,

LoginUser VARCHAR(50)

)

LANGUAGE plpgsql

AS $$

BEGIN

-- Disable counting of affected rows

SET LOCAL row\_count = 0;

-- Attempt to insert into the SPErrorLog table

BEGIN

INSERT INTO dbo.SPErrorLog(ErrorLine, ErrorMessage, SchemaName, ErrorProcedure, ErrorSeverity, ErrorState,

SPID, ServerName, ServiceName, HostName, ProgramName, CommandLine, LoginUser)

VALUES(ErrorLine, ErrorMessage, SchemaName, ErrorProcedure, ErrorSeverity, ErrorState,

SPID, ServerName, ServiceName, HostName, ProgramName, CommandLine, LoginUser);

EXCEPTION

WHEN raise\_exception THEN

-- Handle general-purpose exceptions

RAISE NOTICE 'General-purpose exception occurred';

WHEN invalid\_cursor\_state THEN

-- Handle exceptions related to cursor operations

RAISE NOTICE 'Invalid cursor state';

WHEN foreign\_key\_violation THEN

-- Handle exceptions related to foreign key constraints

RAISE NOTICE 'Foreign key violation';

WHEN unique\_violation THEN

-- Handle exceptions related to unique constraints

RAISE NOTICE 'Unique violation';

WHEN null\_value\_not\_allowed THEN

-- Handle exceptions related to NOT NULL constraints

RAISE NOTICE 'NULL value not allowed';

WHEN others THEN

-- Handle other types of exceptions

RAISE NOTICE 'An error occurred: %', SQLERRM;

WHEN others THEN

-- If an error occurs during insertion, log the error message and line number

RAISE NOTICE 'Error Message: %, Error Line: %', SQLERRM, SQLSTATE;

END;

END;

$$;

CREATE OR REPLACE FUNCTION uspCaptureError(

ErrorLine BIGINT,

ErrorMessage TEXT,

SchemaName VARCHAR(500),

ErrorProcedure VARCHAR(500),

ErrorSeverity BIGINT,

ErrorState BIGINT,

SPID BIGINT,

ServerName VARCHAR(500),

ServiceName VARCHAR(500),

HostName VARCHAR(500),

ProgramName VARCHAR(1000),

CommandLine TEXT,

LoginUser VARCHAR(50)

)

RETURNS VOID AS

$$

BEGIN

-- Set the error context

RAISE LOG 'Error occurred at line %: %', ErrorLine, ErrorMessage;

-- Insert the error details into the error log table

INSERT INTO dbo.SPErrorLog(ErrorLine, ErrorMessage, SchemaName, ErrorProcedure, ErrorSeverity, ErrorState,

SPID, ServerName, ServiceName, HostName, ProgramName, CommandLine, LoginUser)

VALUES (ErrorLine, ErrorMessage, SchemaName, ErrorProcedure, ErrorSeverity, ErrorState,

SPID, ServerName, ServiceName, HostName, ProgramName, CommandLine, LoginUser);

END;

$$

LANGUAGE plpgsql;