DAY-3

Study and Revision Notes on Type Conversion Functions

In SQL, type conversion refers to converting one data type to another. Oracle SQL supports both implicit and explicit conversions. Here's a breakdown of the provided code snippets.

Implicit Conversion

Implicit conversion occurs automatically by Oracle when you mix different data types in an expression. Oracle implicitly converts the data type of one or more operands to a common type that is compatible with the expression.

1. **Adding a Number and a String**

```
sql
CopyEdit
SELECT 100 + '200' FROM DUAL;
```

   a. Oracle implicitly converts the string `'200'` to a number. The result of `100 + 200` is `300`.

2. **Implicit Conversion of String to Date**

```
sql
CopyEdit
SELECT '01-JAN-2025' FROM DUAL;
```

   a. When you select a date in a string format, Oracle implicitly converts the string `'01-JAN-2025'` into a date type.

3. **Adding Months to a Date Using `ADD_MONTHS()`**

```
sql
CopyEdit
SELECT ADD_MONTHS('01-JAN-2025', 3) FROM DUAL;
```

a. Here, `'01-JAN-2025'` (a string) is implicitly converted to a DATE type. The function `ADD_MONTHS()` adds 3 months to the date, returning `01-APR-2025`.

4. **Length of a Number**

```
sql
CopyEdit
SELECT LENGTH(1000) FROM DUAL;
```

a. The number `1000` is implicitly converted to a string to calculate its length. The result is 4, as the number `1000` contains 4 characters.

Explicit Conversion

Explicit conversion involves using built-in functions to manually convert one data type to another.

1. **Using `TO_NUMBER()` for String to Number Conversion**

```
sql
CopyEdit
SELECT 100 + TO_NUMBER('100') FROM DUAL;
```

a. The string `'100'` is explicitly converted to a number using the `TO_NUMBER()` function. This results in `100 + 100`, which equals `200`.

2. **Handling Invalid String Conversion (Error Handling)**

```
sql
CopyEdit
SELECT 100 + TO_NUMBER('100ABC') FROM DUAL; --ERROR
```

a. The string `'100ABC'` cannot be implicitly converted to a number. The `TO_NUMBER()` function raises an error because the string contains non-numeric characters.

3. **Using `TO_DATE()` for String to Date Conversion**

```
sql
CopyEdit
SELECT TO_DATE('10-JANUARY-2025') FROM DUAL;
```

a. The string `'10-JANUARY-2025'` is explicitly converted to a date using the `TO_DATE()` function. Oracle recognizes the date format, so the conversion is successful.

4. **Error Handling with `TO_DATE()` Function**

```
sql
CopyEdit
SELECT TO_DATE('10-10-2025') FROM DUAL; --ERROR
```

a. This results in an error because Oracle cannot determine the date format based on the string `'10-10-2025'` (the format is ambiguous). The format should be provided explicitly.

5. **Using `TO_DATE()` with Custom Format**

```
sql
CopyEdit
SELECT TO_DATE('10-10-2025', 'DD-MM-YY') FROM DUAL;
```

a. Here, the string `'10-10-2025'` is converted into a date, and the format `DD-MM-YY` specifies the day, month, and year format. This allows Oracle to correctly interpret the date.

6. **Using `TO_DATE()` with an Alternative Format**

```
sql
CopyEdit
SELECT TO_DATE('12-23-25') FROM DUAL;
```

a. Oracle interprets this date string as `12-23-25` in the default date format (depending on the session settings). It could be seen as `MM-DD-YY`.

7. **Custom Date Format Conversion**

```
sql
CopyEdit
SELECT TO_DATE('122325', 'MMDDYYYY') FROM DUAL;
```

a. This converts the string `'122325'` into a date using the format `MMDDYYYY`. The result would be `12/23/2025`.

8. **Converting Number to String Using `TO_CHAR()`**

```
sql
CopyEdit
SELECT CONCAT(TO_CHAR(12345), 'HELLO') FROM DUAL;
```

a. `TO_CHAR(12345)` converts the number 12345 into a string. The `CONCAT()` function combines the string `'12345'` with `'HELLO'`, resulting in `'12345HELLO'`.

9. Using SYSDATE

```
sql
CopyEdit
SELECT SYSDATE FROM DUAL;
```

a. `SYSDATE` returns the current system date and time in the default date format.

10. Formatting the Current Date

```
sql
CopyEdit
SELECT TO_CHAR(SYSDATE, 'DDTH MONTH YYYY') FROM DUAL;
```

a. This uses `TO_CHAR()` to format the current date as `DDTH MONTH YYYY`. The `TH` adds the ordinal suffix (like `1st`, `2nd`, etc.) to the day.

11. Basic Date Formatting

```
sql
CopyEdit
SELECT TO_CHAR(SYSDATE, 'DD MONTH YYYY') FROM DUAL;
```

a. This formats the current date to show the day, full month name, and year (e.g., `06 FEBRUARY 2025`).

12. Date Formatting with YEAR

```
sql
CopyEdit
SELECT TO_CHAR(SYSDATE, 'DD MONTH YEAR') FROM DUAL;
```

a. This formats the current date similarly but uses the word YEAR instead of YYYY, which still returns the year in four digits.

Summary of Type Conversion Functions:

1. **Implicit Conversion**:
   a. Occurs automatically by Oracle when different data types are used in expressions.

    b. Examples: Adding a number to a string, converting string to date, calculating the length of a number.
2. **Explicit Conversion**:
    a. Performed using conversion functions like `TO_NUMBER()`, `TO_DATE()`, and `TO_CHAR()`.
    b. These functions explicitly convert one data type into another and handle errors when the conversion cannot be performed.
3. **Common Conversion Functions**:
    a. **`TO_NUMBER()`**: Converts a string or other types to a number.
    b. **`TO_DATE()`**: Converts a string to a date, with optional date format masks.
    c. **`TO_CHAR()`**: Converts a number or date to a string, with optional formatting.
    d. **`SYSDATE`**: Returns the current system date and time.

Date Functions in SQL

Date functions in SQL are used to manipulate and extract information from date and time values.

1. Finding the Difference Between Two Dates in Months

Query:

```
sql
CopyEdit
SELECT ROUND(MONTHS_BETWEEN(SYSDATE, '01-Jan-25'), 2) FROM DUAL;
```

Explanation:

- `MONTHS_BETWEEN(SYSDATE, '01-Jan-25')` calculates the number of months between the current date (SYSDATE) and **January 1, 2025**.
- `ROUND(..., 2)` rounds the result to 2 decimal places.
- If SYSDATE is **before** `01-Jan-25`, the result is negative.

2. Finding the Difference Between Two Dates in Reverse Order

Query:

```
sql
CopyEdit
SELECT ROUND(MONTHS_BETWEEN('01-Jan-25', SYSDATE), 2) FROM DUAL;
```

Explanation:

- The order of parameters is reversed.
- Now, `01-Jan-25` is the first argument, and SYSDATE is the second.
- The result will be positive if `01-Jan-25` is in the future.

3. Adding Months to a Date

Query:

```
sql
CopyEdit
SELECT ADD_MONTHS(SYSDATE, 3) FROM DUAL;
```

Explanation:

- `ADD_MONTHS(SYSDATE, 3)` adds **3 months** to the current date.

4. Adding Months to a Specific Date (String Format)

Query:

```
sql
CopyEdit
SELECT ADD_MONTHS('02/05/2025', 3) FROM DUAL;
```

Explanation:

- This query **will produce an error** because `'02/05/2025'` is treated as a string.
- Date functions require proper date formats.

✅ **Corrected Query**:

```
sql
CopyEdit
SELECT ADD_MONTHS(TO_DATE('02/05/2025','MM/DD/YYYY'),3) FROM DUAL;
```

- `TO_DATE('02/05/2025','MM/DD/YYYY')` converts the string into a date format.
- `ADD_MONTHS(..., 3)` adds 3 months.

5. Finding the Next Occurrence of a Specific Day

Query:

```
sql
CopyEdit
SELECT NEXT_DAY(SYSDATE, 'SUNDAY') FROM DUAL;
```

Explanation:

- `NEXT_DAY(SYSDATE, 'SUNDAY')` returns the next **Sunday** after the current date (SYSDATE).
- Useful for scheduling tasks.

6. Finding the Last Day of the Current Month

Query:

```
sql
CopyEdit
SELECT LAST_DAY(SYSDATE) FROM DUAL;
```

Explanation:

- `LAST_DAY(SYSDATE)` returns the last date of the current month.

7. Finding the Difference Between Two Dates in Months

Query:

```
sql
CopyEdit
SELECT ROUND (MONTHS_BETWEEN(SYSDATE, '01-Jan-24'), 2) FROM DUAL;
```

Explanation:

- `MONTHS_BETWEEN(SYSDATE, '01-Jan-24')` calculates the difference in months between today and **January 1, 2024**.
- `ROUND(..., 2)` rounds the result to two decimal places.

8. Finding the Difference Between Two Dates in Days

Query:

```
sql
CopyEdit
SELECT SYSDATE - TO_DATE('01-JAN-25') FROM DUAL;
```

Explanation:

- In Oracle, subtracting two dates returns the difference in **days**.

9. Truncating Dates

Query:

```
sql
CopyEdit
```

```
SELECT TRUNC(TO_DATE('31-DEC-24')) FROM DUAL;
```

Explanation:

- `TRUNC(TO_DATE('31-DEC-24'))` removes the **time portion**, keeping only the date.

✅ **Truncating to the Start of the Month:**

```
sql
CopyEdit
SELECT TRUNC(TO_DATE('31-DEC-24'),'MONTH') FROM DUAL;
```

- This returns **01-DEC-24**, truncating to the first day of the month.

❌ **Incorrect Query (Will Fail):**

```
sql
CopyEdit
SELECT TRUNC('31-DEC-24','DD-MON-YY') FROM DUAL;
```

- `'31-DEC-24'` is not converted into a **proper date format**.

✅ **Truncating to the Quarter Start:**

```
sql
CopyEdit
SELECT TRUNC(TO_DATE('31-DEC-24'),'Q') FROM DUAL;
```

- This returns **01-OCT-24**, the first day of the quarter.

10. Extracting Date Components

Extracting the Day

✅ **Using EXTRACT:**

```
sql
CopyEdit
SELECT EXTRACT(DAY FROM SYSDATE) FROM DUAL;
```

✅ **Alternative Using TO_CHAR:**

```
sql
CopyEdit
SELECT TO_NUMBER(TO_CHAR(SYSDATE,'DD')) FROM DUAL;
```

Extracting the Month

```
sql
CopyEdit
SELECT EXTRACT(MONTH FROM SYSDATE) FROM DUAL;
```

Extracting the Year

```
sql
CopyEdit
SELECT EXTRACT(YEAR FROM SYSDATE) FROM DUAL;
```

11. Extracting Time Components

Query:

```
sql
CopyEdit
SELECT CURRENT_TIMESTAMP FROM DUAL;
```

Explanation:

- CURRENT_TIMESTAMP returns the current date and time, including **timezone**.

✅ **Extracting the Hour:**

```
sql
CopyEdit
SELECT EXTRACT(HOUR FROM CURRENT_TIMESTAMP) AS current_hour FROM
DUAL;
```

✅ **Extracting the Minute:**

```
sql
CopyEdit
SELECT EXTRACT(MINUTE FROM CURRENT_TIMESTAMP) AS current_minute FROM
DUAL;
```

12. Truncating SYSDATE

Query:

```
sql
CopyEdit
SELECT TRUNC(SYSDATE) FROM DUAL;
```

Explanation:

- `TRUNC(SYSDATE)` removes the **time** part and keeps only the date.

Summary of Date Functions

Function	Description	Example
`MONTHS_BETWEEN(d1, d2)`	Finds the number of months between two dates	`MONTHS_BETWEEN(SYSDATE, '01-Jan-25')`
`ADD_MONTHS(d, n)`	Adds n months to a date	`ADD_MONTHS(SYSDATE, 3)`
`NEXT_DAY(d, 'DAY')`	Finds the next occurrence of a specific weekday	`NEXT_DAY(SYSDATE, 'SUNDAY')`
`LAST_DAY(d)`	Returns the last day of the month for a given date	`LAST_DAY(SYSDATE)`
`SYSDATE - d`	Finds the number of days between two dates	`SYSDATE - TO_DATE('01-JAN-25')`
`TRUNC(d, 'MONTH')`	Returns the first day of the month	`TRUNC(SYSDATE, 'MONTH')`
`TRUNC(d, 'Q')`	Returns the first day of the quarter	`TRUNC(SYSDATE, 'Q')`

`EXTRACT(field FROM d)`	Extracts a specific part of the date	`EXTRACT(DAY FROM SYSDATE)`
`CURRENT_TIMESTAMP`	Returns the current date and time	`CURRENT_TIMESTAMP`
`EXTRACT(HOUR FROM CURRENT_TIMESTAMP)`	Extracts the hour from the timestamp	`EXTRACT(HOUR FROM CURRENT_TIMESTAMP)`

Conclusion

- Date functions are powerful for handling **date arithmetic, formatting, and extraction**.
- Always use `TO_DATE` when working with string dates.
- Use `EXTRACT` for retrieving specific parts of a date.
- `TRUNC` helps in rounding down to the start of **months, quarters, and years**.

📌 SQL Joins - Study and Revision Notes

Joins in SQL are used to **retrieve data from multiple tables** based on a **common column**.

🛠 1. Creating and Populating Tables

Creating the DEPT (Department) Table

```
sql
CopyEdit
CREATE TABLE DEPT(DEPTNO NUMBER, DNAME VARCHAR2(20), LOC
VARCHAR2(20));
```

- `DEPTNO`: **Department Number** (Primary Key)
- `DNAME`: **Department Name**
- `LOC`: **Department Location**

Creating the EMP (Employee) Table

```
sql
CopyEdit
```

```
CREATE TABLE EMP(EMPNO NUMBER, ENAME VARCHAR2(20), DEPTNO NUMBER);
```

- EMPNO: **Employee Number** (Primary Key)
- ENAME: **Employee Name**
- DEPTNO: **Department Number (Foreign Key)**

Inserting Data into DEPT Table

```
sql
CopyEdit
INSERT INTO DEPT VALUES(10, 'ACCOUNT', 'T1 10F');
INSERT INTO DEPT VALUES(20, 'AUDIT', 'T2 3F');
INSERT INTO DEPT VALUES(30, 'IT', 'T1 5F');
```

Inserting Data into EMP Table

```
sql
CopyEdit
INSERT INTO EMP VALUES(101, 'AMIT', 10);
INSERT INTO EMP VALUES(102, 'RAGHU', 10);
INSERT INTO EMP VALUES(103, 'SHEETAL', 20);
INSERT INTO EMP VALUES(104, 'RAJU', 40); -- No matching department
```

Viewing the Tables

```
sql
CopyEdit
SELECT * FROM DEPT;
SELECT * FROM EMP;
```

🛠 2. CROSS JOIN (Cartesian Product)

- Returns **all possible combinations** of rows from both tables.
- Produces **m × n** rows.

Old Style Cross Join

```
sql
CopyEdit
SELECT * FROM DEPT, EMP;
```

New Style Cross Join

```
sql
CopyEdit
SELECT * FROM DEPT CROSS JOIN EMP;
```

- **Example Output (Partial)**:

```
markdown
CopyEdit
DEPTNO | DNAME    | LOC    | EMPNO | ENAME    | DEPTNO
-----------------------------------------------
10     | ACCOUNT | T1 10F | 101   | AMIT     | 10
10     | ACCOUNT | T1 10F | 102   | RAGHU    | 10
...
```

🛠 3. INNER JOIN (Only Matching Records)

- Returns **only rows** that have **matching values** in both tables.

Old Style Inner Join

```
sql
CopyEdit
SELECT * FROM DEPT, EMP WHERE DEPT.DEPTNO = EMP.DEPTNO;
```

Using NATURAL JOIN (Automatic Match on Same Column Name)

```
sql
CopyEdit
SELECT * FROM DEPT NATURAL JOIN EMP;
```

- Works **only if both tables have a column with the exact same name**.

Using ON Clause (Recommended)

```
sql
CopyEdit
SELECT * FROM DEPT JOIN EMP ON DEPT.DEPTNO = EMP.DEPTNO;
```

Using USING Clause (When Column Names Are the Same)

```
sql
CopyEdit
SELECT * FROM DEPT JOIN EMP USING(DEPTNO);
```

🛠 4. Handling Column Name Changes

Renaming Column in EMP Table

```
sql
CopyEdit
ALTER TABLE EMP RENAME COLUMN DEPTNO TO DEPTID;
```

- DEPTNO is now DEPTID, meaning:
  - NATURAL JOIN **won't work** anymore.
  - USING clause **won't work** anymore.

Updated Join with New Column Name

```
sql
CopyEdit
SELECT * FROM DEPT, EMP WHERE DEPT.DEPTNO = EMP.DEPTID;
```

Using INNER JOIN with Explicit Condition

```
sql
CopyEdit
```

```
SELECT * FROM DEPT INNER JOIN EMP ON DEPT.DEPTNO = EMP.DEPTID;
```

Selecting Specific Columns Using Aliases

```
sql
CopyEdit
SELECT D.DEPTNO, D.DNAME, E.EMPNO, E.ENAME
FROM DEPT D INNER JOIN EMP E ON D.DEPTNO = E.DEPTID;
```

🛠️ 5. OUTER JOINS (Includes Non-Matching Records)

Outer Joins return records **even if there is no match**.

LEFT OUTER JOIN (All from Left Table, Matches from Right)

```
sql
CopyEdit
SELECT * FROM DEPT LEFT JOIN EMP ON DEPT.DEPTNO = EMP.DEPTID;
SELECT * FROM DEPT LEFT OUTER JOIN EMP ON DEPT.DEPTNO = EMP.DEPTID;
```

- Includes **all departments**, even those **without employees**.

RIGHT OUTER JOIN (All from Right Table, Matches from Left)

```
sql
CopyEdit
SELECT * FROM DEPT RIGHT JOIN EMP ON DEPT.DEPTNO = EMP.DEPTID;
SELECT * FROM DEPT RIGHT OUTER JOIN EMP ON DEPT.DEPTNO = EMP.DEPTID;
```

- Includes **all employees**, even those **without departments**.

FULL OUTER JOIN (Includes All Records from Both Tables)

```
sql
CopyEdit
SELECT * FROM DEPT FULL JOIN EMP ON DEPT.DEPTNO = EMP.DEPTID;
SELECT * FROM DEPT FULL OUTER JOIN EMP ON DEPT.DEPTNO = EMP.DEPTID;
```

- Returns **all departments** and **all employees**, even if they don’t match.

🛠 6. SELF JOIN (Joining a Table with Itself)

A **self join** is used to compare **rows within the same table**.

Viewing Employee Table

```
sql
CopyEdit
SELECT * FROM SCOTT.EMP;
```

Finding Which Employee Works for Which Manager

```
sql
CopyEdit
SELECT WORKER.ENAME || ' WORKS FOR ' || MANAGER.ENAME
FROM SCOTT.EMP WORKER JOIN SCOTT.EMP MANAGER
ON WORKER.MGR = MANAGER.EMPNO;
```

- Each employee (`WORKER`) has a **manager (`MGR`)**, who is also an employee.

Finding Employees Who Earn More Than Their Managers

```
sql
CopyEdit
SELECT WORKER.ENAME || ' EARNS MORE THAN ' || MANAGER.ENAME
FROM SCOTT.EMP WORKER JOIN SCOTT.EMP MANAGER
ON WORKER.MGR = MANAGER.EMPNO
AND WORKER.SAL > MANAGER.SAL;
```

- Filters only employees who **earn more than their managers**.

📌 Summary of SQL Joins

Join Type	Description
CROSS JOIN	Returns all possible combinations (Cartesian product)
INNER JOIN	Returns only matching rows

NATURAL JOIN	Automatically matches columns with the same name
JOIN ON	Explicitly joins using a condition
JOIN USING	Joins on a specific column name (must be the same in both tables)
LEFT JOIN	Returns all rows from the left table, even if no match exists
RIGHT JOIN	Returns all rows from the right table, even if no match exists
FULL JOIN	Returns all rows from both tables, with NULLs if no match
SELF JOIN	Joins a table with itself

🚀 Conclusion

- **INNER JOIN** is the most commonly used join.
- **OUTER JOINs** ensure **no data is lost** when relationships are incomplete.
- **SELF JOIN** is useful for hierarchical relationships (e.g., employees and managers).

📌 SQL Analytical Functions - Study & Revision Notes

Analytical functions in SQL **perform calculations across a set of rows** related to the current row. They allow **ranking, aggregations, and window operations** without affecting the overall dataset structure.

🛠 1. Row Numbering & Ranking Functions

These functions **assign a rank or number** to each row based on a specified order.

```
sql
CopyEdit
SELECT EMPNO, DEPTNO, SAL,
    ROW_NUMBER() OVER(ORDER BY SAL) AS RN,
    RANK() OVER(ORDER BY SAL) AS RNK,
    DENSE_RANK() OVER(ORDER BY SAL) AS DNS_RNK
FROM SCOTT.EMP;
```

◇ Explanation:

Function	Description
ROW_NUMBER()	Assigns a **unique** sequential number to each row. No duplicates.
RANK()	Assigns a rank, but **skips numbers** for ties (gaps in ranking).
DENSE_RANK()	Assigns a rank, but **does not skip numbers** for ties.

📝 Example Output

EMPNO	DEPTNO	SAL	ROW_NUMBER	RANK	DENSE_RANK
103	20	2000	1	1	1
105	10	2000	2	1	1
101	30	2500	3	3	2
102	10	3000	4	4	3

🛠 2. Ranking in Descending Order

This ranks **salaries in descending order**.

```
sql
CopyEdit
SELECT EMPNO, DEPTNO, SAL,
    ROW_NUMBER() OVER(ORDER BY SAL DESC) AS RN,
    RANK() OVER(ORDER BY SAL DESC) AS RNK,
    DENSE_RANK() OVER(ORDER BY SAL DESC) AS DNS_RNK
FROM SCOTT.EMP;
```

- **Changes Order:** Highest salary gets 1, second highest 2, etc.

🛠 3. Partitioning Data with ROW_NUMBER()

This resets the row number **for each department**.

```
sql
CopyEdit
SELECT EMPNO, DEPTNO, SAL,
    ROW_NUMBER() OVER(PARTITION BY DEPTNO ORDER BY SAL DESC) AS RN
```

```
FROM SCOTT.EMP;
```

◇ Explanation:

- `PARTITION BY DEPTNO`: **Groups employees by department**.
- `ORDER BY SAL DESC`: **Orders by salary within each department**.
- Each **department gets a separate ranking**.

📝 Example Output

EMPNO	DEPTNO	SAL	ROW_NUMBER
101	10	4000	1
102	10	3500	2
105	20	4500	1
106	20	3000	2

🛠 4. Aggregating Values into a List (`LISTAGG`)

Concatenates **employee names into a single string** per department.

```
sql
CopyEdit
SELECT DEPTNO, LISTAGG(ENAME, ',') WITHIN GROUP (ORDER BY ENAME)
FROM SCOTT.EMP
GROUP BY DEPTNO;
```

◇ Explanation:

- `LISTAGG(ENAME, ',')`: Combines employee names **separated by commas**.
- `WITHIN GROUP (ORDER BY ENAME)`: **Orders names alphabetically**.

📝 Example Output

DEPTNO	Employees
10	Amit, Rahul, Suresh
20	John, Mark, Steve
30	Alice, Bob, Charlie

🛠 5. Finding First & Last Values in a Set

◇ First Value in a Set

```
sql
CopyEdit
SELECT ENAME, SAL, DEPTNO,
    FIRST_VALUE(ENAME) OVER(ORDER BY SAL) AS PERSON_WITH_LOWEST_SAL
FROM SCOTT.EMP;
```

- Finds the **employee with the lowest salary** in the entire table.
- Uses **`ORDER BY SAL ASC`**.

◇ First Value Per Department

```
sql
CopyEdit
SELECT ENAME, SAL, DEPTNO,
    FIRST_VALUE(ENAME) OVER(PARTITION BY DEPTNO ORDER BY SAL)
    AS PERSON_WITH_LOWEST_SAL
FROM SCOTT.EMP;
```

- **Partitions by department**.
- Finds **lowest salary in each department**.

📝 Example Output

ENAME	SAL	DEPTNO	PERSON_WITH_LOWEST_SAL
Steve	2000	10	Steve
John	2500	10	Steve
Mark	3000	20	Mark

🛠 6. Finding Last Value in a Set

```
sql
CopyEdit
SELECT ENAME, SAL, DEPTNO,
    LAST_VALUE(ENAME) OVER(ORDER BY SAL
    RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING)
    AS PERSON_WITH_HIGHEST_SAL
```

```
FROM SCOTT.EMP;
```

◇ Explanation:

- **Finds the last row (highest salary)**.
- `RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING`
    - Extends the window to **include all rows**.

📝 Example Output

ENAME	SAL	DEPTNO	PERSON_WITH_HIGHEST_SAL
Steve	2000	10	Rahul
John	2500	10	Rahul
Mark	3000	20	Rahul

📌 Summary Table

Function	Description
ROW_NUMBER()	Assigns unique row numbers.
RANK()	Assigns ranks, skips numbers for ties.
DENSE_RANK()	Assigns ranks without skipping numbers for ties.
PARTITION BY	Divides data into **groups** before applying a function.
LISTAGG()	Concatenates values **into a string**.
FIRST_VALUE()	Gets the **first value** in an ordered set.
LAST_VALUE()	Gets the **last value** in an ordered set.

🚀 Conclusion

- **Analytical functions allow ranking, ordering, and partitioning.**
- **They do NOT collapse rows** (unlike aggregate functions like SUM).
- **Use `PARTITION BY` carefully** when working with groups.

📌 SQL LEAD & LAG Functions - Study & Revision Notes

LEAD and LAG functions in SQL **allow us to access the next or previous row's data** within the same result set. They are part of **window functions** and are useful for analyzing trends over time.

🛠 1. Creating & Populating the SALES Table

```
sql
CopyEdit
CREATE TABLE SALES(SALES_ID NUMBER, SALES_YEAR NUMBER, SALES_AMOUNT
NUMBER);
```

- **Creates a SALES table** with:
  - SALES_ID: Unique identifier.
  - SALES_YEAR: Year of sales.
  - SALES_AMOUNT: Total sales in that year.

```
sql
CopyEdit
INSERT INTO SALES VALUES(1001, 2015, 10500);
INSERT INTO SALES VALUES(1002, 2016, 11500);
INSERT INTO SALES VALUES(1003, 2017, 12000);
INSERT INTO SALES VALUES(1004, 2018, 14000);
INSERT INTO SALES VALUES(1005, 2019, 16000);
INSERT INTO SALES VALUES(1006, 2020, 15000);
INSERT INTO SALES VALUES(1007, 2021, 14500);
INSERT INTO SALES VALUES(1008, 2022, 17000);
INSERT INTO SALES VALUES(1009, 2023, 18000);
INSERT INTO SALES VALUES(1010, 2024, 20000);
```

- **Populates sales data** from 2015 to 2024.

🛠 2. Using LEAD( ) to Get Next Year's Sales

```
sql
```

```
CopyEdit
SELECT SALES.*,
    LEAD(SALES_AMOUNT) OVER(ORDER BY SALES_YEAR) AS NEXT_YEAR_SALES
FROM SALES;
```

◇ Explanation

- **`LEAD(SALES_AMOUNT)`**: Gets the **next row's `SALES_AMOUNT`**.
- **`OVER(ORDER BY SALES_YEAR)`**: Ensures sales are sorted **by year**.

📝 Example Output

SALES_ID	SALES_YEAR	SALES_AMOUNT	NEXT_YEAR_SALES
1001	2015	10500	11500
1002	2016	11500	12000
1003	2017	12000	14000
1004	2018	14000	16000
1005	2019	16000	15000
1006	2020	15000	14500
1007	2021	14500	17000
1008	2022	17000	18000
1009	2023	18000	20000
1010	2024	20000	NULL

- The last row has **NULL** because there is no "next year."

🛠 3. Incorrect Use of `LEAD()` for Previous Year

```
sql
CopyEdit
SELECT SALES.*, LEAD(SALES_AMOUNT) OVER(ORDER BY SALES_YEAR) AS
PREV_YEAR_SALES FROM SALES;
```

❌ Issue

- **`LEAD()` moves forward**, not backward, so **this does not fetch previous year sales.**
- **Use `LAG()` instead!** (Fixed in Section 6)

🛠 4. Getting Sales for the Next 2 Years

```
sql
CopyEdit
SELECT SALES.*,
    LEAD(SALES_AMOUNT) OVER(ORDER BY SALES_YEAR) AS NEXT_YEAR_SALES,
    LEAD(SALES_AMOUNT, 2) OVER(ORDER BY SALES_YEAR) AS
NEXT_TO_NEXT_YEAR_SALES
FROM SALES;
```

◇ Explanation

- `LEAD(SALES_AMOUNT, **1**)`: Fetches **sales of next year**.
- `LEAD(SALES_AMOUNT, **2**)`: Fetches **sales of two years ahead**.

📝 Example Output

SALES_ID	SALES_YEAR	SALES_AMOUNT	NEXT_YEAR_SALES	NEXT_TO_NEXT_YEAR_SALES
1001	2015	10500	11500	12000
1002	2016	11500	12000	14000
1003	2017	12000	14000	16000

🛠 5. Using a Default Value in LEAD( )

```
sql
CopyEdit
SELECT SALES.*,
    LEAD(SALES_AMOUNT, 1, 0) OVER(ORDER BY SALES_YEAR) AS
NEXT_YEAR_SALES,
    LEAD(SALES_AMOUNT, 2, 0) OVER(ORDER BY SALES_YEAR) AS
NEXT_TO_NEXT_YEAR_SALES
FROM SALES;
```

◇ Explanation

- `LEAD(SALES_AMOUNT, 1, 0)`: If **no next row exists**, return **0 instead of NULL**.

📝 Example Output

SALES_ID	SALES_YEAR	SALES_AMOUNT	NEXT_YEAR_SALES	NEXT_TO_NEXT_YEAR_SALES
1009	2023	18000	20000	0
1010	2024	20000	0	0

🛠️ 6. Using LAG( ) for Previous Year's Sales

```
sql
CopyEdit
SELECT SALES.*,
    LAG(SALES_AMOUNT, 1, 0) OVER(ORDER BY SALES_YEAR) AS
PREV_YEAR_SALES,
    LAG(SALES_AMOUNT, 2, 0) OVER(ORDER BY SALES_YEAR) AS
PREV_TO_PREV_YEAR_SALES
FROM SALES;
```

🔹 Explanation

- `LAG(SALES_AMOUNT, 1, 0)`: Gets **sales of the previous year**.
- `LAG(SALES_AMOUNT, 2, 0)`: Gets **sales two years back**.
- Default value **0 is used** if there's no previous row.

📝 Example Output

SALES_ID	SALES_YEAR	SALES_AMOUNT	PREV_YEAR_SALES	PREV_TO_PREV_YEAR_SALES
1001	2015	10500	0	0
1002	2016	11500	10500	0
1003	2017	12000	11500	10500
1004	2018	14000	12000	11500

📌 Summary Table

Function	Description
LEAD(column)	Gets **next row's** value.
LEAD(column, n)	Gets value **n rows ahead**.

LEAD(column, n, default_value)	If no row exists, return **default_value** instead of NULL.
LAG(column)	Gets **previous row's** value.
LAG(column, n)	Gets value **n rows before**.
LAG(column, n, default_value)	If no row exists, return **default_value**.

🚀 Conclusion

- **LEAD()** is used to **look forward** at future rows.
- **LAG()** is used to **look back** at previous rows.
- **Partitioning and ordering are crucial** for accurate results.
- **Default values help prevent NULL issues.**

📌 SQL GROUP FUNCTIONS & GROUPING CONCEPTS - Study & Revision Notes

◇ Introduction

Group functions (also known as aggregate functions) **perform calculations on multiple rows and return a single result**. They are commonly used with the `GROUP BY` clause to summarize data.

🛠 1. SQL Aggregate (Group) Functions

☑ Basic Aggregate Functions

Function	Description
`SUM(column)`	Returns the total sum of values in a column.
`MIN(column)`	Returns the smallest value in a column.
`MAX(column)`	Returns the largest value in a column.

`AVG(column)`	Returns the average (mean) value.
`COUNT(column)`	Returns the number of rows in a column (excluding NULLs).
`STDDEV(column)`	Returns the standard deviation.
`VARIANCE(column)`	Returns the variance.

🛠 2. Using Aggregate Functions

◇ Calculating SUM, MIN, MAX, AVG, COUNT, VARIANCE, and STDDEV

```
sql
CopyEdit
SELECT SUM(SAL) FROM SCOTT.EMP;
SELECT MIN(SAL) FROM SCOTT.EMP;
SELECT MAX(SAL) FROM SCOTT.EMP;
SELECT ROUND(AVG(SAL), 2) FROM SCOTT.EMP;
SELECT COUNT(SAL) FROM SCOTT.EMP;
SELECT ROUND(VARIANCE(SAL), 2) FROM SCOTT.EMP;
SELECT ROUND(STDDEV(SAL), 2) FROM SCOTT.EMP;
```

✅ **Explanation:**

- `SUM(SAL)`: Returns the total salary.
- `MIN(SAL)`: Returns the minimum salary.
- `MAX(SAL)`: Returns the highest salary.
- `AVG(SAL)`: Returns the average salary (rounded to 2 decimal places).
- `COUNT(SAL)`: Returns the number of employees with non-null salaries.
- `VARIANCE(SAL)`: Measures how salaries vary from the mean.
- `STDDEV(SAL)`: Measures the spread of salaries.

🛠 3. Using GROUP BY Clause

The `GROUP BY` clause **groups rows with the same values** in a specified column and applies aggregate functions to each group.

```
sql
CopyEdit
```

```
SELECT DEPTNO, SUM(SAL) FROM SCOTT.EMP GROUP BY DEPTNO;
```

✅ **Explanation:**

- Groups employees **by DEPTNO**.
- Returns **total salary per department**.

❌ Incorrect Grouping:

```
sql
CopyEdit
SELECT ENAME, DEPTNO, SUM(SAL) FROM SCOTT.EMP GROUP BY DEPTNO;
```

❌ **Error:**

- ENAME is **not part of GROUP BY** and **not used in an aggregate function**.
- **Rule:** Columns in SELECT **must be either**:
  - Used in GROUP BY OR
  - Wrapped in an aggregate function.

✅ **Correct Query:**

```
sql
CopyEdit
SELECT DEPTNO, JOB, SUM(SAL), COUNT(*) FROM SCOTT.EMP GROUP BY
DEPTNO, JOB;
```

- Groups by **Department (DEPTNO) & Job (JOB)**.
- Returns **total salary and count of employees per job per department**.

🛠 4. HAVING Clause (Filtering Groups)

The HAVING clause filters groups **after grouping is done**.

❌ Incorrect Query

```
sql
CopyEdit
```

```
SELECT DEPTNO, AVG(SAL) FROM SCOTT.EMP WHERE AVG(SAL) > 2000;
```

❌ **Error:** `AVG(SAL)` is an **aggregate function** and cannot be used in `WHERE`.

✅ **Correct Query**

```
sql
CopyEdit
SELECT DEPTNO, AVG(SAL)
FROM SCOTT.EMP
GROUP BY DEPTNO
HAVING AVG(SAL) > 2000;
```

✅ **Explanation:**

- `GROUP BY DEPTNO`: Groups employees by department.
- `HAVING AVG(SAL) > 2000`: **Filters** departments where the **average salary exceeds 2000**.

🛠 5. ORDER BY Clause (Sorting)

```
sql
CopyEdit
SELECT SUM(SAL) AS TOTAL_SAL FROM SCOTT.EMP GROUP BY DEPTNO ORDER BY
TOTAL_SAL;
```

✅ **Explanation:**

- `ORDER BY TOTAL_SAL`: Sorts results by **total salary** in **ascending order**.

◇ Sorting in Descending Order

```
sql
CopyEdit
SELECT SUM(SAL) AS TOTAL_SAL FROM SCOTT.EMP GROUP BY DEPTNO ORDER BY
TOTAL_SAL DESC;
```

- `ORDER BY TOTAL_SAL DESC`: Sorts **in descending order**.

🛠️ 6. Filtering Employees by Hire Date

```
sql
CopyEdit
SELECT DEPTNO, SUM(SAL), COUNT(*)
FROM SCOTT.EMP
WHERE HIREDATE < '01-JAN-82'
GROUP BY DEPTNO;
```

✅ **Explanation:**

- Filters employees hired **before** `01-JAN-82`.
- Groups by **Department** and calculates **total salary** and **count**.

🛠️ 7. Fetching Top Rows Using `ROWNUM`

```
sql
CopyEdit
SELECT * FROM SCOTT.EMP WHERE ROWNUM < 5;
SELECT * FROM SCOTT.EMP WHERE ROWNUM <= 6;
```

✅ **Explanation:**

- `ROWNUM < 5`: Returns the **first 4 rows**.
- `ROWNUM <= 6`: Returns the **first 6 rows**.

◇ Display Top 3 Salaries

```
sql
CopyEdit
SELECT * FROM SCOTT.EMP ORDER BY SAL DESC FETCH FIRST 3 ROWS ONLY;
```

✅ **Explanation:**

- `ORDER BY SAL DESC`: Sorts salaries in **descending order**.
- `FETCH FIRST 3 ROWS ONLY`: Limits output to **top 3 earners**.

🛠 8. String Manipulation

◇ Find Employees with 'A' in Their Name

```
sql
CopyEdit
SELECT ENAME FROM SCOTT.EMP WHERE INSTR(UPPER(ENAME), 'A') > 0;
```

- `INSTR(UPPER(ENAME), 'A') > 0`: Finds employees **with 'A' (case insensitive) in their name**.

◇ Concatenating Values

```
sql
CopyEdit
SELECT TO_CHAR(DEPTNO) || '_' || TO_CHAR(EMPNO) AS deptid_empid FROM
SCOTT.EMP;
```

- Combines **Department No. (DEPTNO)** and **Employee No. (EMPNO)** into a **single string**.

◇ Extract Last 3 Characters of Employee Names in Department 20

```
sql
CopyEdit
SELECT SUBSTR(ENAME, -3) AS last_3_chars FROM SCOTT.EMP WHERE DEPTNO
= 20;
```

- `SUBSTR(ENAME, -3)`: Extracts **last 3 characters** of `ENAME`.

🛠 9. Using NVL2( ) for Conditional Output

```
sql
CopyEdit
SELECT ENAME, NVL2(MGR, 'Has a Manager', 'Does not have a Manager')
AS Manager_Status
```

```
FROM SCOTT.EMP;
```

✅ **Explanation:**

- `NVL2(MGR, 'Has a Manager', 'Does not have a Manager')`:
  - If `MGR` (Manager ID) **is NOT NULL**, returns **"Has a Manager"**.
  - If `MGR` **is NULL**, returns **"Does not have a Manager"**.

🛠 10. Using `DECODE()` for Categorization

```
sql
CopyEdit
SELECT
    empno,
    ename,
    sal,
    NVL(
        DECODE(
            TRUE,
            sal < 1000, 'Poor Salary',
            sal < 2000, 'Just Okay',
            sal < 3000, 'Good Salary',
            'Big Shots'
        ), 'Unknown'
    ) AS sal_comment
FROM
    scott.emp;
```

✅ **Explanation:**

- `DECODE(TRUE, condition1, result1, condition2, result2, ...)`
  - If `SAL < 1000`, returns **"Poor Salary"**.
  - If `SAL < 2000`, returns **"Just Okay"**.
  - If `SAL < 3000`, returns **"Good Salary"**.
  - Otherwise, returns **"Big Shots"**.
- **`NVL(..., 'Unknown')`** ensures non-null output.

🚀 Summary

- `GROUP BY` groups data; `HAVING` filters groups.
- Aggregate functions summarize data.
- `ORDER BY` sorts output.
- `ROWNUM` and `FETCH` limit results.
- `NVL2()` and `DECODE()` provide conditional formatting.