JOINS

Null Data type

- String ‘null’
- System Default ‘null’

JOINS

```
select * from dept
```

cript Output x | Query Result x

SQL | All Rows Fetched: 14 in 0.002 second

	ID
1	1
2	null
3	3
4	null
5	null
6	null
7	3
8	3
9	(null)
10	(null)
11	(null)
12	(null)
13	(null)
14	(null)

ID
1
null
3
null
null
null
3
3

String Null

System Default Null

string null	18	46	20	18	19	18	19	18
System default null	18	46	20	18	22	24	28	18
	EQUI JOIN	NON-EQUI	SELF JOIN	NATURAL JOIN	LEFT OUTER JOIN	RIGHT OUTER JOIN	FULL OUTER JOIN	INNER JOIN
	1	1	1	1	1	1	1	1
	1	1	1	1	1	1	1	1
	1	1	1	1	1	null	1	1
	null	1	1	null	null	null	null	null
	null	1	1	null	null	null	null	null
	null	1	1	null	null	null	null	null
	3	1	2	3	3	null	3	3
	null	1	null	null	null	null	null	null
	null	1	null	null	null	null	null	null
	null	1	null	null	null	null	null	null
	null	1	null	null	null	1	null	null
	null	1	null	null	null	3	null	null
	null	1	null	null	null	3	null	null
	null	1	1	null	null	3	null	null
	null	2	1	null	null	null	null	null
	null	2	1	null	null	null	null	null
	3	2	3	3	3	null	3	3
	3	2	null	3	3	null	3	3
		2	null		2		2	
		2						
		2						
		2						

GROUP BY AND HAVING

GROUP BY

Using group by, we can create groups of related information.

Columns used in select must be used with group by, otherwise it was not a group by expression.

Ex:

SQL> select deptno, sum(sal) from emp group by deptno;

SQL> select deptno,job,sum(sal) from emp group by deptno,job

GROUP BY AND HAVING

HAVING

This will work as where clause which can be used only with group by because of absence of where clause in group by.

SQL> select deptno,job,sum(sal) tsal from emp group by deptno,job having sum(sal) > 3000;

SQL> select deptno,job,sum(sal) tsal from emp group by deptno,job having sum(sal) > 3000 order by job;

GROUP BY AND HAVING

ORDER OF EXECUTION

- Group the rows together based on group by clause.
- Calculate the group functions for each group.
- Choose and eliminate the groups based on the having clause.
- Order the groups based on the specified column.

SUBQUERIES

SUBQUERIES

Nesting of queries, one within the other is termed as a subquery.

A statement containing a subquery is called a parent query.

Subqueries are used to retrieve data from tables that depend on the values in the table itself.

SUBQUERIES

Single row subqueries

Multi row subqueries

Multiple subqueries

Correlated subqueries

SUBQUERIES

SINGLE ROW SUBQUERIES

In single row subquery, it will return one value.

Ex:

```
SQL> select * from emp where sal > (select sal from emp where empno = 7566);
```

SUBQUERIES

MULTI ROW SUBQUERIES

In multi row subquery, it will return more than one value. In such cases we should include operators like any, all, in or not in between the comparision operator and the subquery.

```
select * from emp where sal > any (select sal from emp where sal between 2500 and 4000);
select * from emp where sal > all (select sal from emp where sal between 2500 and 4000);
```

SUBQUERIES

MULTIPLE SUBQUERIES

There is no limit on the number of subqueries included in a where clause. It allows nesting of a query within a subquery.

Ex:

```
SQL> select * from emp where sal = (select max(sal) from emp where sal < (select
max(sal) from emp));
```

SUBQUERIES

CORRELATED SUBQUERIES

A subquery is evaluated once for the entire parent statement where as a correlated subquery is evaluated once for every row processed by the parent statement.

Ex:

```
SQL> select distinct deptno from emp e where 5 <= (select count(ename) from emp where e.deptno = deptno);
```

SUBQUERIES

EXISTS

Exists function is a test for existence. This is a logical test for the return of rows from a query.

Ex:

Suppose we want to display the department numbers which has more than 4 employees.

```
SQL> select deptno,count(*) from emp group by deptno having count(*) > 4;
```

SUBQUERIES

NOT EXISTS

```
SQL> select deptno,ename from emp e1 where not exists (select * from emp e2
where e1.deptno=e2.deptno group by e2.deptno having count(e2.ename) > 4) order by
deptno,ename;
```

Window Functions

A **window function** performs a calculation **across a set of rows** related to the current row,
without grouping or reducing rows.
It uses the OVER() clause.

```
function_name() OVER (
  [PARTITION BY column]
  [ORDER BY column]
  [ROWS or RANGE clause]
)
```

Window Functions: Common Window Functions

Function	Purpose
ROW_NUMBER()	Gives unique sequential number per row
RANK()	Ranks with gaps
DENSE_RANK()	Ranks without gaps
NTILE(n)	Divides rows into n equal buckets
LAG()	Fetches previous row’s value
LEAD()	Fetches next row’s value
FIRST_VALUE()	Gets first value in window
LAST_VALUE()	Gets last value in window
SUM(), AVG()	Performs aggregation over window

Window Functions

```
CREATE TABLE employees (
  emp_id   NUMBER PRIMARY KEY,
  name     VARCHAR2(50),
  dept     VARCHAR2(50),
  salary   NUMBER);

INSERT INTO employees (emp_id, name, dept, salary) VALUES (1, 'Alice', 'HR', 50000);
INSERT INTO employees (emp_id, name, dept, salary) VALUES (2, 'Bob', 'HR', 55000);
INSERT INTO employees (emp_id, name, dept, salary) VALUES (3, 'Charlie', 'Sales', 60000);
INSERT INTO employees (emp_id, name, dept, salary) VALUES (4, 'Diana', 'Sales', 62000);
INSERT INTO employees (emp_id, name, dept, salary) VALUES (5, 'Edward', 'HR', 50000);
INSERT INTO employees (emp_id, name, dept, salary) VALUES (6, 'Frank', 'Sales', 65000);
```

Window Functions

RANK() – Rank with gaps

sql

```
SELECT emp_id, name, dept, salary, RANK() OVER (PARTITION BY dept ORDER BY salary DESC) AS rank FROM employees;
```

DENSE_RANK() – Rank without gaps

sql

```
SELECT emp_id, name, dept, salary, DENSE_RANK() OVER (PARTITION BY dept ORDER BY salary DESC) AS dense_rank
FROM employees;
```

NTILE(n) – Divides into n equal buckets

sql

```
SELECT emp_id, name, salary, NTILE(3) OVER (ORDER BY salary DESC) AS salary_group FROM employees;
```

Window Functions

LAG() – Previous row's value

sql

```
SELECT emp_id, name, salary, LAG(salary) OVER (ORDER BY emp_id) AS prev_salary FROM employees;
```

LEAD() – Next row's value

sql

```
SELECT emp_id, name, salary, LEAD(salary) OVER (ORDER BY emp_id) AS next_salary FROM employees;
```

FIRST_VALUE() – First value in the window

sql

```
SELECT emp_id, name, salary, FIRST_VALUE(salary) OVER (PARTITION BY dept ORDER BY salary DESC) AS top_salary FROM employ
```

Window Functions

LAST_VALUE() – Last value in the window

sql

```
SELECT emp_id, name, salary, LAST_VALUE(salary) OVER (PARTITION BY dept ORDER BY salary DESC ROWS
BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS bottom_salary FROM employees;
```

SUM() as Window Function – Running total

sql

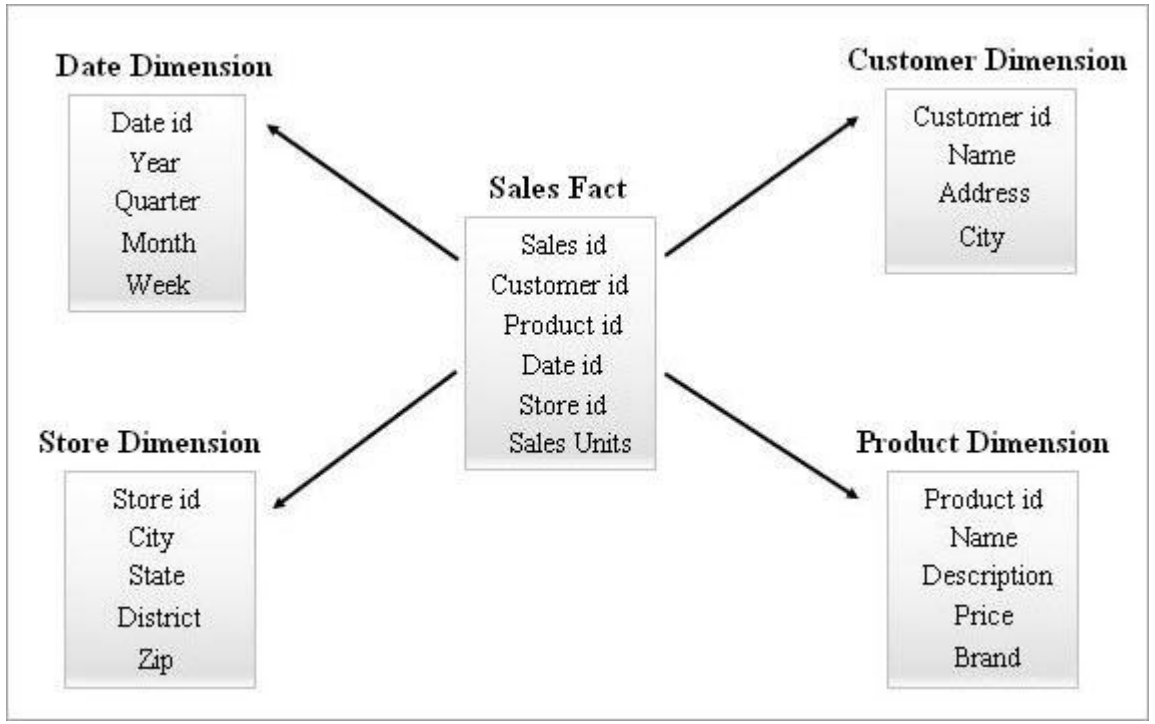
```
SELECT emp_id, name, salary, SUM(salary) OVER (ORDER BY emp_id) AS running_salary FROM
employees;
```

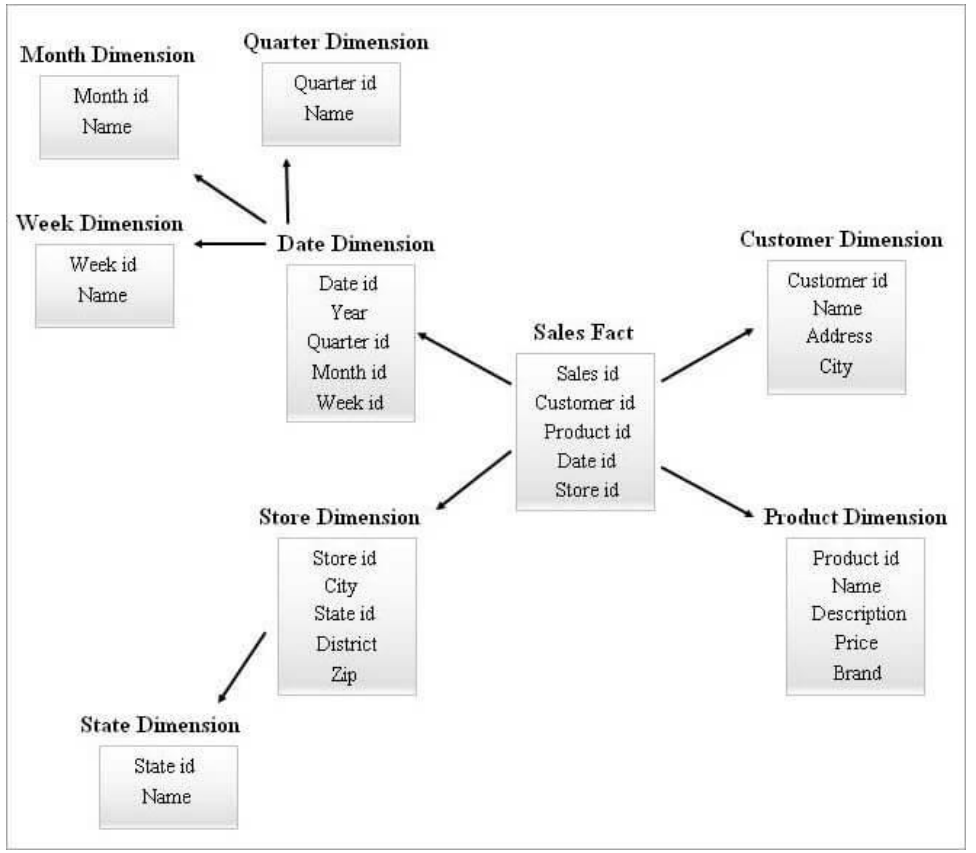
AVG() as Window Function – Moving average

sql

```
SELECT emp_id, name, salary, AVG(salary) OVER (ORDER BY emp_id ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING)
AS moving_avg FROM employees;
```

Star schema vs snowflake schema





Star schema vs snowflake schema

Feature	Star Schema	Snowflake Schema	Galaxy Schema
Normalization	Denormalized	Normalized	Mixed
Query Performance	Fastest	Slower (more joins)	Moderate
Storage Efficiency	Lower	Higher	Depends
Complexity	Low (simple joins)	Medium (more joins)	High (multiple facts)
Use Case	Simple BI reports	Complex hierarchies	Advanced analytics

Denormalized vs Normalized Star Schema

Feature	Denormalized Star Schema	Normalized Star (Snowflake)
Query Performance	Fast	Slower (more joins)
Storage Space	Larger	Smaller
Design Simplicity	Simple	Complex
Ideal For	BI reporting, OLAP	Complex hierarchies, normalized source systems
Maintenance	Harder (more updates)	Easier (changes in one place)