

---

# Introduction to Database

---

Lecture Note

---

**American International  
University-Bangladesh**

---



## Lecture -11, 12(Sub-query)

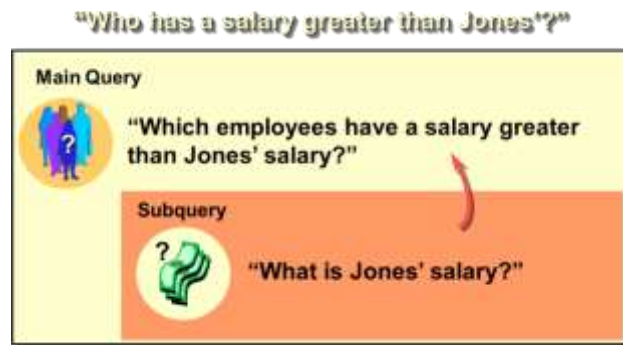
### Objective:

- Describe the types of problems that subqueries can solve
- Define subqueries
- List the types of subqueries
- Write single-row and multiple-row subqueries

In this lesson, you will learn about more advanced features of the SELECT statement. You can write subqueries in the WHERE clause of another SQL statement to obtain values based on an unknown conditional value. This lesson covers single-row subqueries and multiple-row subqueries.

### Using a Subquery to Solve a Problem

Suppose you want to write a query to find out who earns a salary greater than Jones' salary. To solve this problem, you need *two* queries: one query to find what Jones earns and a second query to find who earns more than that amount. You can solve this problem by combining the two queries, placing one query *inside* the other query. The inner query or the *subquery* returns a value that is used by the outer query or the main query. Using a subquery is equivalent to performing two sequential queries and using the result of the first query as the search value in the second query.



### Subqueries

A subquery is a SELECT statement that is embedded in a clause of another SELECT statement. You can build powerful statements out of simple ones by using subqueries. They can be very useful when you need to select rows from a table with a condition that depends on the data in the table itself.

You can place the subquery in a number of SQL clauses:

- WHERE clause
- HAVING clause
- FROM clause

In the syntax:

*operator* includes a comparison operator such as >, =, or IN

**Note:** Comparison operators fall into two classes: single-row operators (>, =, >=, <, <>, <=) and multiple-row operators (IN, ANY, ALL). The subquery is often referred to as a nested SELECT, sub-SELECT, or inner SELECT statement. The subquery generally executes first, and its output is used to complete the query condition for the main or outer query.

```
SQL> SELECT ename
2 FROM emp
3 WHERE sal >
4         (SELECT sal
5          FROM emp
6          WHERE empno=7566);
```

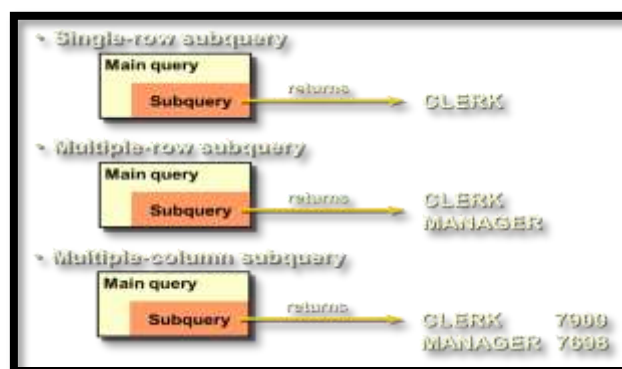
-----  
 KING  
 FORD  
 SCOTT

### Guidelines for Using Subqueries

- A subquery must be enclosed in parentheses.
- A subquery must appear on the right side of the comparison operator.
- Subqueries cannot contain an ORDER BY clause. You can have only one ORDER BY clause for a SELECT statement, and if specified it must be the last clause in the main SELECT statement.
- Two classes of comparison operators are used in subqueries: single-row operators and multiple-row operators.

### Types of Subqueries

- Single-row subqueries: Queries that return only one row from the inner SELECT statement
- Multiple-row subqueries: Queries that return more than one row from the inner SELECT statement
- Multiple-column subqueries: Queries that return more than one column from the inner SELECT statement



### Single-Row Subqueries

A *single-row subquery* is one that returns one row from the inner SELECT statement. This type of subquery uses a single-row operator.

### Example

Display the employees whose job title is the same as that of employee 7369.

### Executing Single-Row Subqueries

A SELECT statement can be considered as a query block. The example on the displays employees whose job title is the same as that of employee 7369 and whose salary is greater than that of employee 7876.

The example consists of three query blocks: the outer query and two inner queries. The inner query blocks are executed first, producing the query results: CLERK and 1100, respectively. The outer query block is then processed and uses the values returned by the inner queries to complete its search conditions.

Both inner queries return single values (CLERK and 1100, respectively), so this SQL statement is called a single-row subquery.

SQL> SELECT	ename, job
2 FROM	emp
3 WHERE	job =
4	(SELECT job
5	FROM emp
6	WHERE empno = 7369)
7 AND	sal >
8	(SELECT sal
9	FROM emp
10	WHERE empno = 7876) ;

ENAME	JOB
-----	-----
MILLER	CLERK

### Using Group Functions in a Subquery

You can display data from a main query by using a group function in a subquery to return a single row. The subquery is in parentheses and is placed after the comparison operator.

The example displays the employee name, job title, and salary of all employees whose salary is equal to the minimum salary. The MIN group function returns a single value (800) to the outer query.

SQL> SELECT	ename, job, sal
2 FROM	emp
3 WHERE	sal =
4	(SELECT MIN(sal)
5	FROM emp) ;

ENAME	JOB	SAL
-----	-----	-----
SMITH	CLERK	800

### HAVING Clause with Subqueries

You can use subqueries not only in the WHERE clause, but also in the HAVING clause. The Oracle Server executes the subquery, and the results are returned into the HAVING clause of the main query. The SQL statement stated below displays all the departments that have a minimum salary greater than that of department 20.

```
SQL> SELECT      deptno, MIN(sal)
  2 FROM          emp
  3 GROUP BY      deptno
  4 HAVING MIN(sal) > (SELECT MIN(sal)
  5                      FROM emp
  6                      WHERE deptno = 20);
  7
```

### Errors with Subqueries

One common error with subqueries is more than one row returned for a single-row subquery. In the SQL statement shown below, the subquery contains a GROUP BY (deptno) clause, which implies that the subquery will return multiple rows, one for each group it finds. In this case, the result of the subquery will be 800, 1300, and 950. The outer query takes the results of the subquery (800, 950, 1300) and uses these results in its WHERE clause. The WHERE clause contains an equal (=) operator, a single-row comparison operator expecting only one value. The = operator cannot accept more than one value from the subquery and hence generates the error. To correct this error, change the = operator to IN.

```
SQL> SELECT empno, ename
  2 FROM emp
  3 WHERE sal = (SELECT MIN(sal)
  4                FROM emp
  5                GROUP BY deptno);
  6
```

*Single-row operator with multiple-row subquery*

```
ERROR:
ORA-01427: single-row subquery returns more than
one row
no rows selected
```

### Problems with Subqueries

A common problem with subqueries is no rows being returned by the inner query. In the SQL statement given below, the subquery contains a WHERE (ename='SMYTHE') clause. Presumably, the intention is to find the employee whose name is Smythe. The statement seems to be correct but selects no rows when executed.

```
SQL> SELECT ename, job
  2 FROM emp
  3 WHERE job = (SELECT job
  4                FROM emp
  5                WHERE ename='SMYTHE');
  6
```

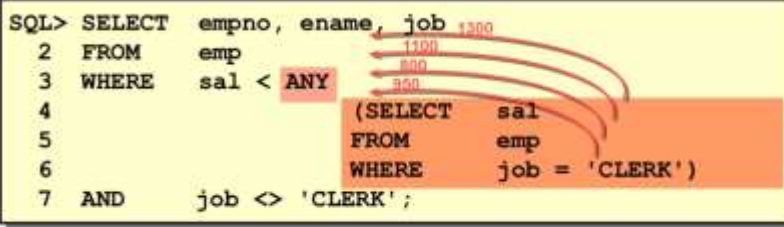
```
no rows selected
```

The problem is that Smythe is misspelled. There is no employee named Smythe. So the subquery returns no rows. The outer query takes the results of the subquery (null) and uses these results in its WHERE clause. The outer query finds no employee with a job title equal to null and so returns no rows.

### Multiple-Row Subqueries

Subqueries that return more than one row are called *multiple-row subqueries*. You use a multiple-row operator, instead of a single-row operator, with a multiple-row subquery. The multiple-row operator expects one or more values.

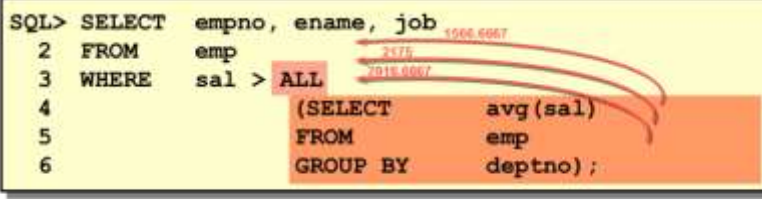
The ANY operator (and its synonym SOME operator) compares a value to *each* value returned by a subquery. The example displays employees whose salary is less than any clerk and who are not clerks. The maximum salary that a clerk earns is \$1300. The SQL statement displays all the employees who are not clerks but earn less than \$1300. <ANY means less than the maximum. >ANY means more than the minimum. =ANY is equivalent to IN.



```
SQL> SELECT empno, ename, job
2 FROM emp
3 WHERE sal < ANY
4 (SELECT sal
5 FROM emp
6 WHERE job = 'CLERK')
7 AND job <> 'CLERK';
```

EMPNO	ENAME	JOB
7654	MARTIN	SALESMAN
7521	WARD	SALESMAN

The ALL operator compares a value to *every* value returned by a subquery. The example displays employees whose salary is greater than the average salaries of all the departments. The highest average salary of a department is \$2916.66, so the query returns those employees whose salary is greater than \$2916.66. >ALL means more than the maximum and <ALL means less than the minimum. The NOT operator can be used with IN, ANY, and ALL operators.



```
SQL> SELECT empno, ename, job
2 FROM emp
3 WHERE sal > ALL
4 (SELECT avg(sal)
5 FROM emp
6 GROUP BY deptno);
```

EMPNO	ENAME	JOB
7839	KING	PRESIDENT
7566	JONES	MANAGER
7902	FORD	ANALYST
7788	SCOTT	ANALYST

## Multiple-Column Subqueries

So far you have written single-row subqueries and multiple-row subqueries where only one column was compared in the WHERE clause or HAVING clause of the SELECT statement. If you want to compare two or more columns, you must write a compound WHERE clause using logical operators. Multiple-column subqueries enable you to combine duplicate WHERE conditions into a single WHERE clause.

The example below shows a multiple-column subquery because the subquery returns more than one column. It compares the values in the PRODID column and the QTY column of each candidate row in the ITEM table to the values in the PRODID column and QTY column for items in order 605. First, execute the subquery to see the PRODID and QTY values for each item in order 605.

```
SQL> SELECT  ordid, prodid, qty
2  FROM      item
3  WHERE     (prodid, qty) IN
4             (SELECT prodid, qty
5              FROM      item
6              WHERE     ordid = 605)
7  AND       ordid <> 605;
```

When the above SQL statement is executed, the Oracle server compares the values in both the PRODID and QTY columns and returns those orders where the product number and quantity for *that* product match *both* the product number and quantity for an item in order 605.

The output of the SQL statement is:

ORDID	PRODID	QTY
617	100861	100
617	100870	500
616	102130	10

The output shows that there are three items in other orders that contain the same product number and quantity as an item in order 605. For example, order 617 has ordered a quantity 500 of product 100870. Order 605 has also ordered a quantity 500 of product 100870. Therefore, that candidate row is part of the output.

## Pairwise Versus Nonpairwise Comparisons

Column comparisons in a multiple-column subquery can be pairwise comparisons or nonpairwise comparisons. The example shows the product numbers and quantities of the items in order 605. In the example, a pairwise comparison was executed in the WHERE clause. Each candidate row in the SELECT statement must have *both* the same product number and same quantity as an item in order 605. This is illustrated on the left side of the example stated above. The arrows indicate that both the product number and quantity in a candidate row match a product number and quantity of an item in order 605.

A multiple-column subquery can also be a nonpairwise comparison. If you want a nonpairwise comparison (a cross product), you must use a WHERE clause with multiple conditions. A candidate row

must match the multiple conditions in the WHERE clause but the values are compared individually. A candidate row must match some product number in order 605 as well as some quantity in order 605, but these values do not need to be in the same row. This is illustrated on the right side of the example. For example, product 102130 appears in other orders, one order matching the quantity in order 605 (10), and another order having a quantity of 500. The arrows show a sampling of the various quantities ordered for a particular product.

```

SQL> SELECT  ordid, prodid, qty
2  FROM      item
3  WHERE     prodid IN (SELECT  prodid
4                        FROM    item
5                        WHERE    ordid = 605)
6  AND       qty  IN  (SELECT  qty
7                        FROM    item
8                        WHERE    ordid = 605)
9  AND       ordid <> 605;

```

### **Exercise:**

1. Display all the employees who are earning more than all the managers.
2. Display all the employees who are earning more than any of the managers.
3. Select employee number, job & salaries of all the Analysts who are earning more than any of the managers.
4. Select all the employees who work in DALLAS.
5. Select department name & location of all the employees working for CLARK.
6. Select all the departmental information for all the managers
7. Display the first maximum salary.
8. Display the second maximum salary.
9. Display the third maximum salary.
10. Display all the managers & clerks who work in Accounts and Marketing departments.
11. Display all the salesmen who are not located at DALLAS.
12. Get all the employees who work in the same departments as of SCOTT.
13. Select all the employees who are earning same as SMITH.
14. Display all the employees who are getting some commission in marketing department where the employees have joined only on weekdays.
15. Display all the employees who are getting more than the average salaries of all the employees.



## Lecture 14, 16 (Normalization)

### **Objective:**

- Introductions
- The Normal Forms
- Primary Key
- Relationships and Referential Integrity
- When NOT to Normalize
- Real World Exercise

### **Normalization Definition:**

- In relational database design, the process of organizing data to minimize duplication.
- *Normalization* usually involves dividing a database into two or more tables and defining relationships between the tables.

The objective is to isolate data so that additions, deletions, and modifications of a field can be made in just one table and then propagated through the rest of the database via the defined relationships.”

"Normalization" refers to the process of creating an efficient, reliable, flexible, and appropriate "relational" structure for storing information. Normalized data must be in a "relational" data structure.

### **Anomaly**

An error or inconsistency that may result when a user attempts to update a table that contains redundant data.

There are three types of Anomaly - **Insertion Anomaly, Deletion Anomaly, Modification Anomaly**

### **Well Structure Relation**

A relation that contains minimal redundancy and allows users to insert, modify and delete the rows without error or inconsistencies.

EMPLOYEE2

Emp_ID	Name	Dept_Name	Salary	Course_Title	Date_Completed
100	Margaret Simpson	Marketing	48,000	SPSS	6/19/200X
100	Margaret Simpson	Marketing	48,000	Surveys	10/7/200X
140	Alan Beeton	Accounting	52,000	Tax Acc	12/8/200X
110	Chris Lucero	Info Systems	43,000	SPSS	1/12/200X
110	Chris Lucero	Info Systems	43,000	C++	4/22/200X
190	Lorenzo Davis	Finance	55,000		
150	Susan Martin	Marketing	42,000	SPSS	6/19/200X
150	Susan Martin	Marketing	42,000	Java	8/12/200X

**Anomalies in the above table:**

- **Insertion** – can't enter a new employee without having the employee take a class
- **Deletion** – if we remove employee 140, we lose information about the existence of a Tax Acc class
- **Modification** – giving a salary increase to employee 100 forces us to update multiple records

**The Normal Forms:**

A series of logical steps to take to normalize data tables

- First Normal Form (1NF)
- Second Normal Form (2NF)
- Third Normal Form (3NF)

**First Normal Form Rule:**

A relation that contains no multivalued Attributes.

All columns (fields) must be atomic. Means no repeating items in columns

OrderDate	Customer	Items
11/30/1998	Joe Smith	Hammer, Saw, Nails

OrderDate	Customer	Item1	Item2	Item3
11/30/1998	Joe Smith	Hammer	Saw	Nails

- **Before 1NF applied**

Employees					
<u>emp_id</u>	name	dept_name	salary	<u>course_title</u>	date_completed
100	M.S.	MKT	48000	SPSS Survey	8/9/16 10/7/16
140	A.B.	ACC	52000	Tally ACC	12/8/16
110	C.L.	I.S.	43000	SPSS CTT	1/12/16 4/12/16

– After 1NF

Employees					
<u>emp_id</u>	name	dept_name	salary	<u>course_title</u>	date_completed
100	M.S.	MKT	48000	SPSS	8/9/16
100	M.S.	MKT	48000	Survey	10/7/16
140	A.B.	ACC	52000	Tally ACC	12/8/16
110	C.L.	I.S.	43000	SPSS	1/12/16
110	C.L.	I.S.	43000	CTT	4/12/16

**Second Normal Form Rule:**

A relation in First Normal Form in which every attribute is fully functionally dependent in the primary key or Partial Functional dependency should be removed.

**Partial Functional Dependency**

A functional dependency in which one or more non-key attribute are functionally dependent in part (but not all) of the primary key.

**Functional Dependency**

A constraint between two attribute or two sets of attributes.

**EmpID, CourseTitle → DateCompleted**

**EmpID → Name, DeptName, Salary**

So, the above table is not in 2NF.

- After 2NF

employees			
<u>emp_id</u>	name	dept_name	salary
100	M.S.	MKT	68000
140	A.B.	ACC	53000
120	C.L.	IS	43000

Courses		
<u>emp_id</u>	<u>course_title</u>	<u>date_employed</u>
100	SPSS	2/9/16
100	Survey	10/7/16
140	ACC	12/8/16
110	SPSS	1/12/16
110	CTT	4/12/16

**Third Normal Form Rule:**

A relation in Second Normal Form has no Transitive Dependency present.

**Transitive Dependency: A Functional Dependency between two (or more) non-key attributes.**

**-Before 3NF applied**

Sales			
<u>cust_id</u>	Name	sales_person	region
8023	Anderson	Smith	South
9167	Boston	Hikes	West
7924	Haile	Smith	South
6837	Tuck	David	East
8596	Haile	Hikes	West
7018	Anderson	Forth	North

CustID → Name

CustID → Salesperson

CustID → Region

All this is OK

(2<sup>nd</sup> NF)

BUT

CustID → Salesperson → Region

Transitive dependency

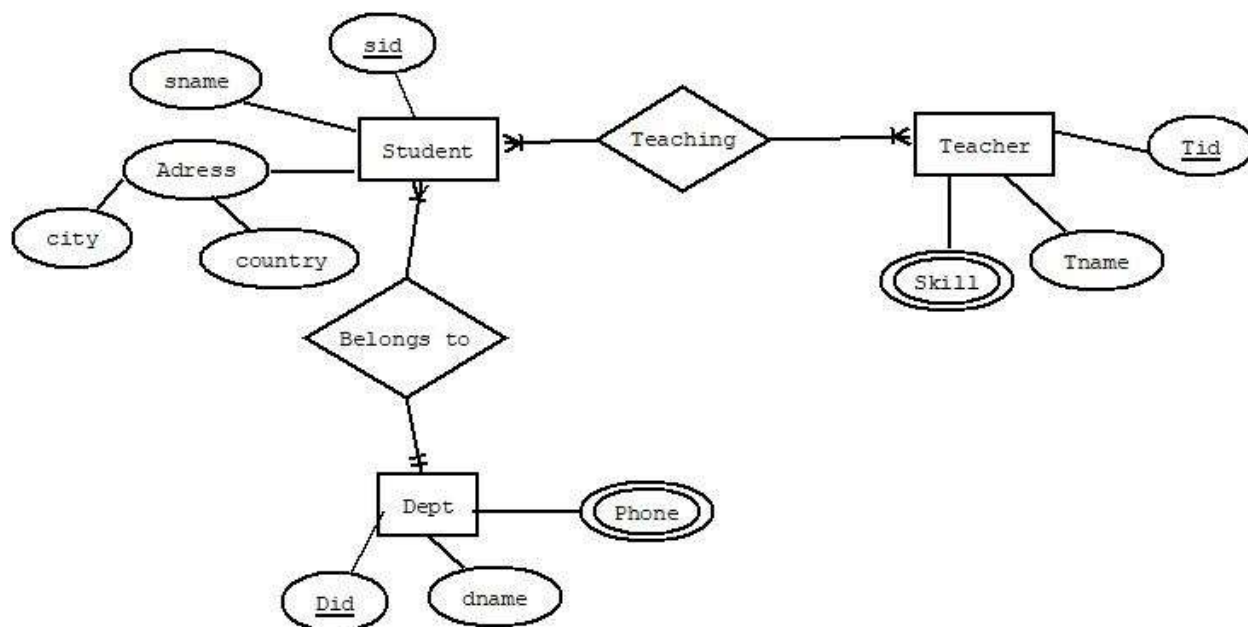
(not 3<sup>rd</sup> NF)

- After 3NF

sales		
<u>cust_id</u>	name	sale_person
8023	Anderson	Smith
9167	Boston	Hikes
7924	Haile	Smith
6837	Tuck	David
8596	Haile	Hikes
7018	Anderson	Forth

salesman	
<u>sale_person</u>	region
Smith	South
Hikes	West
David	East
Forth	North

### Example with ER Diagram



**Teaching:**

UNF: 1<sup>st</sup>: Sid, Sname, City, Country, Tid, Tname, Skill

1NF: 1<sup>st</sup>: Sid, Skill, Tid, Sname, City, Country, Tname

2NF: 1<sup>st</sup>: Skill, Tid

2<sup>nd</sup>: Sid, Sname, City, Country

3<sup>rd</sup>: Tid, Tname

4<sup>th</sup>: Tid, Sid

3NF: 1<sup>st</sup>: Skill, Tid

2<sup>nd</sup>: Sid, Sname, tid, city

3<sup>rd</sup>: City, Country

4<sup>th</sup>: Tid, Tname

5<sup>th</sup>: Tid, Sid

**Belongs to:**

UNF: 1<sup>st</sup>: Sid, Sname, City, Country, Did, Dname, Phone

1NF: 1<sup>st</sup>: Sid, Did, Phone, Sname, City, Country, Dname

2NF: 1<sup>st</sup>: Sid, Sname, City, Country, Did

2<sup>nd</sup>: Did, Phone, Dname

3NF: 1<sup>st</sup>: Sid, Sname, City, Did

2<sup>nd</sup>: City, Country

3<sup>rd</sup>: Did, Phone, Dname

**Final Table:**

1<sup>st</sup>: Skill, Tid

2<sup>nd</sup>: Sid, Sname, city

3<sup>rd</sup>: City, Country

4<sup>th</sup>: Tid, Tname

5<sup>th</sup>: Tid, Sid

6<sup>th</sup>: Sid, Sname, City, Did

7<sup>th</sup>: Did, Phone, Dname

## Lecture-13, 15(Joining: Displaying Data from Multiple Tables)

### Objective:

- Write SELECT statements to access data from more than one table using equality and nonequality joins.
- View data that generally does not meet a join condition by using outer joins.
- Join a table to itself.

### Data from Multiple Tables

Sometimes you need to use data from more than one table. In the example stated below, the report displays data from two separate tables.

- EMPNO exists in the EMP table.
- DEPTNO exists in both the EMP and DEPT the tables.
- LOC exists in the DEPT table.

EMPNO	ENAME	DEPTNO
7839	KING	10
7698	BLAKE	30
7934	MILLER	10

DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON

EMPNO	DEPTNO	LOC
7839	10	NEW YORK
7698	30	CHICAGO
7782	10	NEW YORK
7966	20	DALLAS
7654	30	CHICAGO
7499	30	CHICAGO

14 rows selected.

To produce the report, you need to link EMP and DEPT tables and access data from both of them.

### Defining Joins

When data from more than one table in the database is required, a *join* condition is used. Rows in one table can be joined to rows in another table according to common values existing in corresponding columns, that is, usually primary and foreign key columns. To display data from two or more related tables, write a simple join condition in the WHERE clause.

```
SELECT table1.column, table2.column
FROM table1, table2
WHERE table1.column1 = table2.column2;
```

In the syntax:

*table1.column* denotes the table and column from which data is retrieved

*table1.column1* = is the condition that joins (or relates) the tables together *table2.column2*

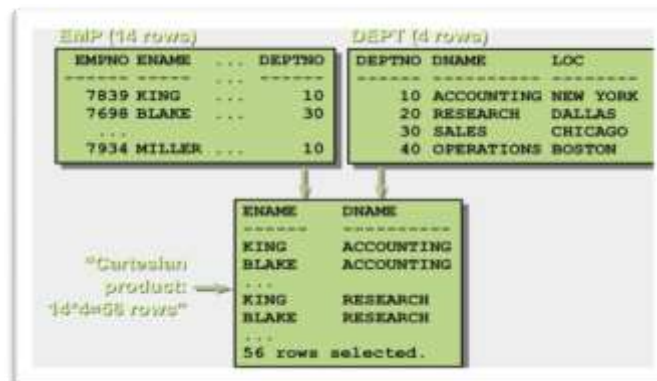
### Guidelines

- When writing a SELECT statement that joins tables, precede the column name with the table name for clarity and to enhance database access.

- If the same column name appears in more than one table, the column name must be prefixed with the table name.
- To join  $n$  tables together, you need a minimum of  $(n-1)$  join conditions. Therefore, to join four tables, a minimum of three joins are required. This rule may not apply if your table has a concatenated primary key, in which case more than one column is required to uniquely identify each row.

### Cartesian Product

When a join condition is invalid or omitted completely, the result is a *Cartesian product* in which all combinations of rows will be displayed. All rows in the first table are joined to all rows in the second table. A Cartesian product tends to generate a large number of rows, and its result is rarely useful. You should always include a valid join condition in a WHERE clause, unless you have a specific need to combine all rows from all tables. A Cartesian product is generated if a join condition is omitted. The example displays employee name and department name from EMP and DEPT tables. Because no WHERE clause has been specified, all rows (14 rows) from the EMP table are joined with all rows (4 rows) in the DEPT table, thereby generating 56 rows in the output.



### Types of Joins

There are two main types of join conditions:

- Equijoins
- Non-equijoins

Additional join methods include the following:

- Outer joins
- Self joins

### Equijoins

To determine the name of an employee's department, you compare the value in the DEPTNO column in the EMP table with the DEPTNO values in the DEPT table. The relationship between the EMP and DEPT tables is an *equijoin*—that is, values in the DEPTNO column on both tables must be equal. Frequently, this type of join involves primary and foreign key complements. Equijoins are also called *simple joins* or *inner joins*.

### Retrieving Records with Equijoins

In the below example:

- The SELECT clause specifies the column names to retrieve:



- employee name, employee number, and department number, which are columns in the EMP table
- department number, department name, and location, which are columns in the DEPT table
- The FROM clause specifies the two tables that the database must access:
  - EMP table
  - DEPT table
- The WHERE clause specifies how the tables are to be joined:
  - EMP.DEPTNO=DEPT.DEPTNO

Because the DEPTNO column is common to both tables, it must be prefixed by the table name to avoid ambiguity.

SQL>	SELECT	emp.empno, emp.ename, emp.deptno,
2		dept.deptno, dept.loc
3	FROM	emp, dept
4	WHERE	emp.deptno=dept.deptno;

EMPNO	ENAME	DEPTNO	DEPTNO	LOC
7839	KING	10	10	NEW YORK
7698	BLAKE	30	30	CHICAGO
7782	CLARK	10	10	NEW YORK
7566	JONES	20	20	DALLAS
...				
14 rows selected.				

### Qualifying Ambiguous Column Names

You need to qualify the names of the columns in the WHERE clause with the table name to avoid ambiguity. Without the table prefixes, the DEPTNO column could be from either the DEPT table or the EMP table. It is necessary to add the table prefix to execute your query. If there are no common column names between the two tables, there is no need to qualify the columns. However, you will gain improved performance by using the table prefix because you tell the Oracle Server exactly where to go to find columns. The requirement to qualify ambiguous column names is also applicable to columns that may be ambiguous in other clauses, such as the SELECT clause or the ORDER BY clause.

### Table Aliases

Qualifying column names with table names can be very time consuming, particularly if table names are lengthy. You can use table *aliases* instead of table names. Just as a column alias gives a column another name, a table alias gives a table another name. Table aliases help to keep SQL code smaller, therefore using less memory. Notice how table aliases are identified in the FROM clause in the example. The table name is specified in full, followed by a space and then the table alias. The EMP table has been given an alias of E, whereas the DEPT table has an alias of D.

### Guidelines

- Table aliases can be up to 30 characters in length, but the shorter they are the better.
- If a table alias is used for a particular table name in the FROM clause, then that table alias must be substituted for the table name throughout the SELECT statement.
- Table aliases should be meaningful.
- The table alias is valid only for the current SELECT statement.

```
SQL> SELECT emp.empno, emp.ename, emp.deptno,
2      dept.deptno, dept.loc
3 FROM   emp, dept
4 WHERE  emp.deptno=dept.deptno;
```

```
SQL> SELECT e.empno, e.ename, e.deptno,
2      d.deptno, d.loc
3 FROM   emp e, dept d
4 WHERE  e.deptno=d.deptno;
```

### Non-Equijoins

The relationship between the EMP table and the SALGRADE table is a non-equijoin, meaning that no column in the EMP table corresponds directly to a column in the SALGRADE table. The relationship between the two tables is that the SAL column in the EMP table is between the LOSAL and HISAL column of the SALGRADE table. The relationship is obtained using an operator other than equal (=). The below example creates a non-equijoin to evaluate an employee's salary grade. The salary must be *between* any pair of the low and high salary ranges. It is important to note that all employees appear exactly once when this query is executed. No employee is repeated in the list. There are two reasons for this:

- None of the rows in the salary grade table contain grades that overlap. That is, the salary value for an employee can only lie between the low salary and high salary values of one of the rows in the salary grade table.
- All of the employees' salaries lie within the limits provided by the salary grade table. That is, no employee earns less than the lowest value contained in the LOSAL column or more than the highest value contained in the HISAL column.

**Note:** Other operators such as <= and >= could be used, but BETWEEN is the simplest. Remember to specify the low value first and the high value last when using BETWEEN. Table aliases have been specified for performance reasons, not because of possible ambiguity.

```
SQL> SELECT e.ename, e.sal, s.grade
2 FROM   emp e, salgrade s
3 WHERE  e.sal
4 BETWEEN s.losal AND s.hisal;
```

ENAME	SAL	GRADE
JAMES	950	1
SMITH	800	1
ADAMS	1100	1
...		

14 rows selected.

### Returning Records with No Direct Match with Outer Joins

If a row does not satisfy a join condition, the row will not appear in the query result. For example, in the equijoin condition of EMP and DEPT tables, department OPERATIONS does not appear because no one works in that department. The missing row(s) can be returned if an *outer join* operator is used in the join condition. The operator is a plus sign enclosed in parentheses (+), and it is *placed on the "side" of the join that is deficient in information*. This operator has the effect of creating one or more null rows, to which one or more rows from the no deficient table can be joined.

```
SQL> SELECT  e.ename, d.deptno, d.dname
2 FROM      emp e, dept d
3 WHERE     e.deptno(+) = d.deptno
4 ORDER BY e.deptno;
```

```
ENAME      DEPTNO DNAME
-----
KING              10 ACCOUNTING
CLARK              10 ACCOUNTING
...
40 OPERATIONS
15 rows selected.
```

In the syntax:

**table1.column** = is the condition that joins (or relates) the tables together.

**table2.column (+)** is the outer join symbol, which can be placed on either side of the WHERE clause condition, but not on both sides (Place the outer join symbol following the name of the column in the table without the matching rows.)

### Joining a Table to Itself

Sometimes you need to join a table to itself. To find the name of each employee's manager, you need to join the EMP table to itself, or perform a self-join. For example, to find the name of Blake's manager, you need to:

- Find Blake in the EMP table by looking at the ENAME column.
- Find the manager number for Blake by looking at the MGR column. Blake's manager number is 7839.
- Find the name of the manager with EMPNO 7839 by looking at the ENAME column. King's employee number is 7839, so King is Blake's manager.

In this process, you look in the table twice. The first time you look in the table to find Blake in the ENAME column and MGR value of 7839. The second time you look in the EMPNO column to find 7839 and the ENAME column to find King.

```
SQL> SELECT worker.ename || ' works for ' || manager.ename
2 FROM      emp worker, emp manager
3 WHERE     worker.mgr = manager.empno;
```

```
WORKER. ENAME || 'WORKSFOR' || MANAG
-----
BLAKE works for KING
CLARK works for KING
JONES works for KING
MARTIN works for BLAKE
...
13 rows selected.
```

The example stated above joins the EMP table to itself. To simulate two tables in the FROM clause, there are two aliases, namely WORKER and MANAGER, for the same table, EMP. In this example the WHERE clause contains the join condition that means "where a worker's manager number matches the employee number for the manager."

**Exercise:**

**ASSIGNMENTS ON EQUI-JOINS**

1. Display all the managers & clerks who work in Accounts and Marketing departments.
2. Display all the salesmen who are not located at DALLAS.
3. Select department name & location of all the employees working for CLARK.
4. Select all the departmental information for all the managers
5. Select all the employees who work in DALLAS.
6. Delete the records from the DEPT table that don't have matching records in EMP

**ASSIGNMENTS ON OUTER-JOINS**

7. Display all the departmental information for all the existing employees and if a department has no employees display it as "No employees".
8. Get all the matching & non-matching records from both the tables.
9. Get only the non-matching records from DEPT table (matching records shouldn't be selected).
10. Select all the employees name along with their manager names, and if an employee does not have a manager, display him as "CEO".

**ASSIGNMENTS ON SELF-JOINS**

11. Get all the employees who work in the same departments as of SCOTT
12. Display all the employees who have joined before their managers.
13. List all the employees who are earning more than their managers.
14. Fetch all the employees who are earning same salaries.
15. Select all the employees who are earning same as SMITH. Display employee name , his date of joining, his manager name & his manager's date of joining.

## Lecture-17 (View)

### View:

You can present logical subsets or combinations of data by creating views of tables. A view is a logical table based on a table or another view. A view contains no data of its own but is like a window through which data from tables can be viewed or changed. The tables on which view is based are called *base tables*. The view is stored as a SELECT statement in the data dictionary.

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7839	KING	PRESIDENT		17-NOV-81	5000		10
7782	CLARK	MANAGER	7839	09-JUN-81	1500	300	10
7934	MILLER	CLERK	7782	23-JAN-82	1300		10
7566	JONES	MANAGER	7839	02-APR-81	2975		20
7566	JONES	ANALYST	7566	08-DEC-82	3000		20
7900	JAMES	CLERK	7566	03-DEC-81	950		30
7521	WARD	SALESMAN	7566	22-FEB-81	1250	500	30

EMPNO	ENAME	JOB
7839	KING	PRESIDENT
7782	CLARK	MANAGER
7934	MILLER	CLERK

### Simple Syntax:

```
CREATE VIEW <view_name> AS
SELECT <col>,<col> FROM <table_name> WHERE <condition>;
```

### Complex Syntax:

```
CREATE [OR REPLACE] [FORCE|NOFORCE] VIEW <view> [(alias[, alias]...)]
AS <subquery>
[WITH CHECK OPTION [CONSTRAINT constraint]] [WITH READ ONLY];
```

In the syntax:

<b>OR REPLACE</b>	re-creates the view if it already exists
<b>FORCE</b>	creates the view regardless of whether or not the base tables exist
<b>NOFORCE</b>	creates the view only if the base tables exist (This is the default.)
<b>View</b>	is the name of the view
<b>Alias</b>	specifies names for the expressions selected by the view's query (The number of aliases must match the number of expressions selected by the view.)
<b>subquery</b>	is a complete SELECT statement (You can use aliases for the columns in the SELECT list.)
<b>WITH CHECK OPTION</b>	specifies that only rows accessible to the view can be inserted or updated

<b>constraint</b>	is the name assigned to the CHECK OPTION constraint
<b>WITH READ ONLY</b>	ensures that no DML operations can be performed on this view

### Retrieving Data from a View

You can retrieve data from a view as you would from any table. You can either display the contents of the entire view or just view specific rows and columns.

Syntax: *SELECT \* FROM <view\_name>;*

### Modifying a View

The OR REPLACE option allows a view to be created even if one exists with this name already, thus replacing the old version of the view for its owner. This means that the view can be altered without dropping, re-creating, and regranteeing object privileges.

Note: When assigning column aliases in the CREATE VIEW clause, remember that the aliases are listed in the same order as the columns in the subquery.

### Performing DML Operations on a View

You can perform DML operations on data through a view if those operations follow certain rules.

- You can remove a row from a view unless it contains any of the following:
- Group functions
- A GROUP BY clause
- The DISTINCT keyword

You can modify data in a view unless it contains any of the conditions mentioned in the previous section and any of the following:

- Columns defined by expressions—for example, SALARY \* 12
- The ROWNUM pseudocolumn

You can add data through a view unless it contains any of the above and there are NOT NULL columns, without a default value, in the base table that are not selected by the view. All required values must be present in the view. Remember that you are adding values directly into the underlying table through the view.

### Removing a View

You use the DROP VIEW statement to remove a view. The statement removes the view definition from the database. Dropping views has no effect on the tables on which the view was based. Views or other applications based on deleted views become invalid. Only the creator or a user with the DROP ANY VIEW privilege can remove a view.

**Syntax:** *DROP VIEW <view\_name>;*

### Inline Views

- An inline view is a sub query with an alias (correlation name) that you can use within a SQL statement.
- An inline view is similar to using a named sub query in the FROM clause of the main query.
- An inline view is not a schema object.

### Exercise

1. Create a view called **EMP\_VU** based on the employee number, employee name, and department number from the EMP table. Change the heading for the employee name to EMPLOYEE.

2. Display the contents of the **EMP\_VU** view.

EMPNO	EMPLOYEE	DEPTNO
7839	KING	10
7698	BLAKE	30
7782	CLARK	10
7566	JONES	20
7654	MARTIN	30
7499	ALLEN	30
7844	TURNER	30
7900	JAMES	30
7521	WARD	30
7902	FORD	20
7369	SMITH	20
7788	SCOTT	20
7876	ADAMS	20
7934	MILLER	10

3. using your view EMP\_VU, enter a query to display all employee names and department numbers.

EMPLOYEE	DEPTNO
KING	10
BLAKE	30
CLARK	10

JONES	20
-------	----

MARTIN	30
--------	----

4. Create a view named **DEPT20** that contains the employee number, employee name, and department number for all employees in department 20. Label the view column EMPLOYEE\_ID, EMPLOYEE, and DEPARTMENT\_ID. Do not allow an employee to be reassigned to another department through the view.

5. Create a view called SALARY\_VU based on the employee name, department name, salary, and salary grade for all employees. Label the columns Employee, Department, Salary, and Grade, respectively.

**\*\* Please save the SQL commands in a text file for further use.**



## Lecture-18, 20 (Relational Algebra)

### Objectives:

Basic operators

Join

Notations

Example

### Relational Algebra:

Relational algebra is a procedural query language. It gives a step by step process to obtain the result of the query. It uses operators to perform queries.

#### ➤ Six basic operators

- select:  $\sigma$
- project:  $\Pi$
- union:  $\cup$
- set difference:  $-$
- Cartesian product:  $\times$
- rename:  $\rho$

➤ The operators take one or two relations as inputs and produce a new relation as a result.

### Banking Example

*branch* (*branch\_name*, *branch\_city*, *assets*)

*customer* (*customer\_name*, *customer\_street*, *customer\_city*)

*account* (*account\_number*, *branch\_name*, *balance*)

*loan* (*loan\_number*, *branch\_name*, *amount*)

*depositor* (*customer\_name*, *account\_number*)

*borrower* (*customer\_name*, *loan\_number*)

### Select Operation:

- Notation:  $\sigma_p(r)$
- $p$  is called the **selection predicate**
- Defined as:

$$\sigma_p(r) = \{t \mid t \in r \text{ and } p(t)\}$$

Where  $p$  is a formula in propositional calculus consisting of **terms** connected by  $\wedge$  (**and**),  $\vee$  (**or**),  $\neg$  (**not**)

Each **term** is one of:

<attribute>  $op$  <attribute> or <constant>

where  $op$  is one of:  $=, \neq, >, \geq, <, \leq$

- Example of selection:

$$\sigma_{\text{branch\_name} = \text{"Perryridge"}}(\text{account})$$

### Select Operation Example:

Relation  $r$

$A$	$B$	$C$	$D$
$\alpha$	$\alpha$	1	7
$\alpha$	$\beta$	5	7
$\beta$	$\beta$	12	3
$\beta$	$\beta$	23	10

$$\sigma_{A=B \wedge D > 5}(r)$$

$A$	$B$	$C$	$D$
$\alpha$	$\alpha$	1	7
$\beta$	$\beta$	23	10

### Project Operation:

– Notation:

- where  $A_1, A_2$  are attribute names and  $r$  is a relation name.
- The result is defined as the relation of  $k$  columns obtained by erasing the columns that are not listed
- Duplicate rows removed from result, since relations are sets
- Example: To eliminate the *branch\_name* attribute of *account*

$$\Pi_{\text{account\_number, balance}}(\text{account})$$

### Project Operation Example:

Relation  $r$ :

$A$	$B$	$C$
$\alpha$	10	1
$\alpha$	20	1
$\beta$	30	1
$\beta$	40	2

What is Sequence

$\Pi_{A,C}(r)$

$A$	$C$		$A$	$C$
$\alpha$	1		$\alpha$	1
$\alpha$	1	=	$\beta$	1
$\beta$	1		$\beta$	2
$\beta$	2			

### Composition of Relational Operations

➤ Find the customer who live in Harrison

➤  $\Pi_{customer\_name}(\sigma_{customer\_city="Harrison"}(customer))$

➤ Notice that instead of giving the name of a relation as the argument of the projection operation, we give an expression that evaluates to a relation

### Union Operation:

- Notation:  $r \cup s$
- Defined as:
  - $r \cup s = \{t \mid t \in r \text{ or } t \in s\}$
- For  $r \cup s$  to be valid.
- $r, s$  must have the *same* arity (same number of attributes)
  - 2. The attribute domains must be compatible (example: 2<sup>nd</sup> column of  $r$  deals with the same type of values as does the 2<sup>nd</sup> column of  $s$ )
- Example: to find all customers with either an account or a loan  
 $\Pi_{customer\_name}(depositor) \cup \Pi_{customer\_name}(borrower)$

### Union Operation Example:

Relations  $r, s$ :

A	B
$\alpha$	1
$\alpha$	2
$\beta$	1

$r$

A	B
$\alpha$	2
$\beta$	3

$s$

$\sigma(r \cup s)$ :

A	B
$\alpha$	1
$\alpha$	2
$\beta$	1
$\beta$	3

### Set Difference Operation:

- Notation  $r - s$
- Defined as:
  - $r - s = \{t \mid t \in r \text{ and } t \notin s\}$
- Set differences must be taken between compatible relations.
  - $r$  and  $s$  must have the same arity
  - attribute domains of  $r$  and  $s$  must be compatible

### Set Difference Operation Example:

Relations  $r, s$ :

$A$	$B$
$\alpha$	1
$\alpha$	2
$\beta$	1

$r$

$A$	$B$
$\alpha$	2
$\beta$	3

$s$

$\sigma(r - s)$ :

$A$	$B$
$\alpha$	1
$\beta$	1

### Cartesian-Product Operation:

- Notation  $r \times s$
- Defined as:
  - $r \times s = \{t \mid t \in r \text{ and } t \in s\}$
- Assume that attributes of  $r(R)$  and  $s(S)$  are disjoint. (That is,  $R \cap S = \emptyset$ ).
- If attributes of  $r(R)$  and  $s(S)$  are not disjoint, then renaming must be used.

### Cartesian Product Operation Example:

Relations  $r, s$ :

A	B	C	D	E
$\alpha$	1	$\alpha$	10	a
$\beta$	2	$\beta$	10	a
		$\beta$	20	b
		$\gamma$	10	b

$r$

$s$

$\sigma(r \times s)$ :

A	B	C	D	E
$\alpha$	1	$\alpha$	10	a
$\alpha$	1	$\beta$	10	a
$\alpha$	1	$\beta$	20	b
$\alpha$	1	$\gamma$	10	b
$\beta$	2	$\alpha$	10	a
$\beta$	2	$\beta$	10	a
$\beta$	2	$\beta$	20	b
$\beta$	2	$\beta$	10	b

## Joining Operations:

$\sigma(r \times s)$

<i>A</i>	<i>B</i>	<i>C</i>	<i>D</i>	<i>E</i>
$\alpha$	1	$\alpha$	10	<i>a</i>
$\alpha$	1	$\beta$	10	<i>a</i>
$\alpha$	1	$\beta$	20	<i>b</i>
$\alpha$	1	$\gamma$	10	<i>b</i>
$\beta$	2	$\alpha$	10	<i>a</i>
$\beta$	2	$\beta$	10	<i>a</i>
$\beta$	2	$\beta$	20	<i>b</i>
$\beta$	2	$\beta$	10	<i>b</i>

$\sigma_{A=C}(r \times s)$

<i>A</i>	<i>B</i>	<i>C</i>	<i>D</i>	<i>E</i>
$\alpha$	1	$\alpha$	10	<i>a</i>
$\beta$	2	$\beta$	10	<i>a</i>
$\beta$	2	$\beta$	20	<i>b</i>

## Rename Operation:

- Allows us to name, and therefore to refer to, the results of relational-algebra expressions.
- Allows us to refer to a relation by more than one name.
- Example:

$\rho_x(E)$

returns the expression *E* under the name *X*

- If a relational-algebra expression *E* has arity *n*, then

$\rho_{x(A_1, A_2, \dots, A_n)}(E)$

returns the result of expression *E* under the name *X*, and with the attributes renamed to *A*<sub>1</sub>, *A*<sub>2</sub>, ..., *A*<sub>*n*</sub>.

## Example Queries:

Q. Find the names of all customers who have a loan at the Perryridge branch.

Ans:  $\Pi_{\text{customer\_name}} (\sigma_{\text{branch\_name} = \text{"Perryridge"}} (\sigma_{\text{borrower\_loan\_number} = \text{loan\_loan\_number}} (\text{borrower} \times \text{loan})))$

## Natural-Join Operation:

Notation:  $r \bowtie s$

- Let  $r$  and  $s$  be relations on schemas  $R$  and  $S$  respectively.

Then,  $r \bowtie s$  is a relation on schema  $R \cup S$  obtained as follows:

- Consider each pair of tuples  $t_r$  from  $r$  and  $t_s$  from  $s$ .
- If  $t_r$  and  $t_s$  have the same value on each of the attributes in  $R \cap S$ , add a tuple  $t$  to the result, where
  - $t$  has the same value as  $t_r$  on  $r$
  - $t$  has the same value as  $t_s$  on  $s$

Example:

$R = (A, B, C, D)$

$S = (E, B, D)$

- Result schema =  $(A, B, C, D, E)$
- $\sigma(r \bowtie s)$  is defined as:

$$\Pi_{r.A, r.B, r.C, r.D, s.E} (\sigma_{r.B=s.B \wedge r.D=s.D} (r \times s))$$

## Natural Join Operation – Example:

Relations  $r$ ,  $s$ :

$\sigma(r \bowtie s)$	$r$				$s$			
	$A$	$B$	$C$	$D$		$B$	$D$	$E$
	$\alpha$	1	$\alpha$	a		1	a	$\alpha$
	$\beta$	2	$\gamma$	a		3	a	$\beta$
	$\gamma$	4	$\beta$	b		1	a	$\gamma$
	$\alpha$	1	$\gamma$	a		2	b	$\delta$
	$\delta$	2	$\beta$	b		3	b	$\epsilon$



### Extended Relational-Algebra-Operations:

- Generalized Projection
- Aggregate Functions
- Outer Join

### Generalized Projection

Extends the projection operation by allowing arithmetic functions to be used in the projection list.

$$\Pi_{F_1, F_2, \dots, F_n}(E)$$

$E$  is any relational-algebra expression

Each of  $F_1, F_2, \dots, F_n$  are arithmetic expressions involving constants and attributes in the schema of  $E$ .

Given relation *credit\_info(customer\_name, limit, credit\_balance)*, find how much more each person can spend:

$$\Pi_{customer\_name, limit - credit\_balance}(credit\_info)$$

### Aggregate Functions and Operations

Aggregation function takes a collection of values and returns a single value as a result.

- avg: average value
- min: minimum value
- max: maximum value
- sum: sum of values
- count: number of values

Aggregate operation in relational algebra

$$G_1, G_2, \dots, G_n \mathcal{G}_{F_1(A_1), F_2(A_2), \dots, F_n(A_n)}(E)$$

$E$  is any relational-algebra expression

$G_1, G_2, \dots, G_n$  is a list of attributes on which to group (can be empty)

Each  $F_i$  is an aggregate function

Each  $A_i$  is an attribute name

### Aggregate Operation – Example:

Relation  $r$ :

A	B	C
$\alpha$	$\alpha$	7
$\alpha$	$\beta$	7
$\beta$	$\beta$	3

$g_{\text{sum}(c)}(r)$

**sum(c)**

27

Aggregate Operation – Example:

Relation *account* grouped by *branch-name*:

<i>branch name</i>	<i>account number</i>	<i>balance</i>
Perryridge	A-102	400
Perryridge	A-201	900
Brighton	A-217	750
Brighton	A-215	750
Redwood	A-222	700

*branch\_name*  $g_{\text{sum}(\text{balance})}(\text{account})$

<i>branch name</i>	<b>sum(balance)</b>
Perryridge	1300
Brighton	1500
Redwood	700

## Outer Join:

An extension of the join operation that avoids loss of information.

Computes the join and then adds tuples from one relation that does not match tuples in the other relation to the result of the join.

Uses *null* values:

*null* signifies that the value is unknown or does not exist

All comparisons involving *null* are (roughly speaking) false by definition.

## Outer Join – Example:

Relation *loan*

<i>loan_number</i>	<i>branch_name</i>	<i>amount</i>
L-170	Downtown	300
L-230	Redwood	0
L-260	Perryridge	400

Relation *borrower*

<i>customer_name</i>	<i>loan_number</i>
Jones	L-170
Smith	L-230
	L-155

Join

$\sigma ( loan \bowtie borrower )$

<i>loan_number</i>	<i>branch_name</i>	<i>amount</i>	<i>customer_name</i>
L-170	Downtown	300	Jones
L-230	Redwood	0	Smith

## Left Outer Join

$\sigma (loan \bowtie borrows)$

<i>loan_numbe</i>	<i>branch_name</i>	<i>amount</i>	<i>customer_nam</i>
L-170	Downtown	300	Jones
L-230	Redwood	0	Smith
L-260	Perryridge	400	<i>null</i>

## Right Outer Join

$\sigma (loan \bowtie borrows)$

<i>loan_numbe</i>	<i>branch_name</i>	<i>amount</i>	<i>customer_nam</i>
L-170	Downtown	300	Jones
L-230	Redwood	0	Smith
L-155	<i>null</i>	400	Haye

## Full Outer Join

$\sigma (loan \bowtie borrows)$

<i>loan_numbe</i>	<i>branch_name</i>	<i>amount</i>	<i>customer_nam</i>
L-170	Downtown	300	Jones
L-230	Redwood	0	Smith
L-260	Perryridge	400	<i>null</i>
L-155	<i>null</i>	0	Haye

## Lecture-19 (Sequence)

### What is Sequence:

A sequence generator can be used to automatically generate sequence numbers for rows in tables. A sequence is a database object created by a user and can be shared by multiple users. A typical usage for sequences is to create a primary key value, which must be unique for each row. The sequence is generated and incremented (or decremented) by an internal Oracle8 routine. This can be a time-saving object because it can reduce the amount of application code needed to write a sequence-generating routine. Sequence numbers are stored and generated independently of tables. Therefore, the same sequence can be used for multiple tables.

```
CREATE SEQUENCE sequence
  [INCREMENT BY n]
  [START WITH n]
  [(MAXVALUE n | NOMAXVALUE)]
  [(MINVALUE n | NOMINVALUE)]
  [(CYCLE | NOCYCLE)]
  [(CACHE n | NOCACHE)];
```

### Creating a Sequence

Automatically generate sequential numbers by using the CREATE SEQUENCE statement. In the syntax:

<b>sequence</b>	is the name of the sequence generator
<b>INCREMENT BY n</b>	specifies the interval between sequence numbers where <i>n</i> is an integer (If this clause is omitted, the sequence will increment by 1.)
<b>START WITH n</b>	specifies the first sequence number to be generated (If this clause is omitted, the sequence will start with 1.)
<b>MAXVALUE n</b>	specifies the maximum value the sequence can generate
<b>NOMAXVALUE</b>	specifies a maximum value of $10^{27}$ for an ascending sequence and $-1$ for a descending sequence (This is the default option.)
<b>MINVALUE n</b>	specifies the minimum sequence value
<b>NOMINVALUE</b>	specifies a minimum value of 1 for an ascending sequence and $-(10^{26})$ for a descending sequence (This is the default option.)
<b>CYCLE   NOCYCLE</b>	specifies that the sequence continues to generate values after reaching either its maximum or minimum value or does not generate additional values (NOCYCLE is the default option.)
<b>CACHE n   NOCACHE</b>	specifies how many values the Oracle Server will pre allocate and keep in memory (By default, the Oracle Server will cache 20 values.)

```
SQL> CREATE SEQUENCE dept_deptno
2      INCREMENT BY 1
3      START WITH 91
4      MAXVALUE 100
5      NOCACHE
6      NOCYCLE;
Sequence created.
```

The above example creates a sequence named DEPT\_DEPTNO to be used for the DEPTNO column of the DEPT table. The sequence starts at 91, does not allow caching, and does not allow the sequence to cycle. Do not use the CYCLE option if the sequence is used to generate primary key values unless you have a reliable mechanism that purges old rows faster than the sequence cycles.

## Confirming Sequences

Once you have created your sequence, it is documented in the data dictionary. Since a sequence is a database object, you can identify it in the USER\_OBJECTS data dictionary table. You can also confirm the settings of the sequence by selecting from the data dictionary USER\_SEQUENCES table.

```
SQL> SELECT  sequence_name, min_value, max_value,
2          increment_by, last_number
3 FROM      user_sequences;
```

## Using a Sequence

Once you create your sequence, you can use the sequence to generate sequential numbers for use in your tables. Reference the sequence values by using the NEXTVAL and CURRVAL pseudocolumns.

### NEXTVAL and CURRVAL Pseudocolumns

The NEXTVAL pseudocolumn is used to extract successive sequence numbers from a specified sequence. You must qualify NEXTVAL with the sequence name. When you reference *sequence*.NEXTVAL, a new sequence number is generated and the current sequence number is placed in CURRVAL. The CURRVAL pseudocolumn is used to refer to a sequence number that the current user has just generated. NEXTVAL must be used to generate a sequence number in the current user's session before CURRVAL can be referenced. You must qualify CURRVAL with the sequence name. When *sequence*.CURRVAL is referenced, the last value returned to that user's process is displayed.

### Rules for Using NEXTVAL and CURRVAL

You can use NEXTVAL and CURRVAL in the following:

- The SELECT list of a SELECT statement that is not part of a subquery
- The SELECT list of a subquery in an INSERT statement
- The VALUES clause of an INSERT statement
- The SET clause of an UPDATE statement

You cannot use NEXTVAL and CURRVAL in the following:

- A SELECT list of a view
- A SELECT statement with the DISTINCT keyword
- A SELECT statement with the GROUP BY, HAVING, or ORDER BY clauses
- A subquery in a SELECT, DELETE, or UPDATE statement
- A DEFAULT expression in a CREATE TABLE or ALTER TABLE statement

## Using a Sequence

The example inserts a new department in the DEPT table. It uses the DEPT\_DEPTNO sequence for generating a new department number.

```
SQL> INSERT INTO    dept(deptno, dname, loc)
2 VALUES          (dept_deptno.NEXTVAL,
3                  'MARKETING', 'SAN DIEGO');
1 row created.
```

You can view the current value of the sequence:

```
SQL> SELECT  dept_deptno.CURRVAL
2 FROM      dual;
```

### Caching Sequence Values

Cache sequences in the memory to allow faster access to those sequence values. The cache is populated at the first reference to the sequence. Each request for the next sequence value is retrieved from the cached sequence. After the last sequence is used, the next request for the sequence pulls another cache of sequences into memory.

### Beware of Gaps in Your Sequence

Although sequence generators issue sequential numbers without gaps, this action occurs independent of a commit or rollback. Therefore, if you roll back a statement containing a sequence, the number is lost. Another event that can cause gaps in the sequence is a system crash. If the sequence caches values in the memory, then those values are lost if the system crashes. Because sequences are not tied directly to tables, the same sequence can be used for multiple tables. If this occurs, each table can contain gaps in the sequential numbers.

### Viewing the Next Available Sequence Value without Incrementing It

If the sequence was created with NOCACHE, it is possible to view the next available sequence value without incrementing it by querying the USER\_SEQUENCES table.

### Altering a Sequence

If you reach the MAXVALUE limit for your sequence, no additional values from the sequence will be allocated and you will receive an error indicating that the sequence exceeds the MAXVALUE. To continue to use the sequence, you can modify it by using the ALTER SEQUENCE statement.

#### Syntax

```
ALTER SEQUENCE sequence
[INCREMENT BY n]
[{MAXVALUE n | NOMAXVALUE}]
[{MINVALUE n | NOMINVALUE}]
[{CYCLE | NOCYCLE}]
[{CACHE n | NOCACHE}];
```

**Where:** *sequence* is the name of the sequence generator

#### Guidelines

- You must be the owner have the ALTER privilege for the sequence in order to modify it.
- Only future sequence numbers are affected by the ALTER SEQUENCE statement.
- The START WITH option cannot be changed using ALTER SEQUENCE. The sequence must be dropped and re-created in order to restart the sequence at a different number.
- Some validation is performed. For example, a new MAXVALUE cannot be imposed that is less than the current sequence number.

### Removing a Sequence

To remove a sequence from the data dictionary, use the DROP SEQUENCE statement. You must be the owner of the sequence or have the DROP ANY SEQUENCE privilege to remove it.

#### Syntax

```
DROP SEQUENCE sequence;
```

**Where:** *sequence* is the name of the sequence generator

***Exercise:***

1. Create a sequence to be used with the primary key column of the DEPARTMENT table. The sequence should start at 60 and have a maximum value of 200. Have your sequence increment by ten numbers. Name the sequence DEPT\_ID\_SEQ.
2. Write a script to display the following information about your sequences: sequence name, maximum value, increment size, and last number.
3. Write an interactive script to insert a row into the DEPARTMENT table. Be sure to use the sequence that you created for the ID column. Create a customized prompt to enter the department name. Execute your script. Add two departments named Education and Administration. Confirm your additions.



## Lecture-21 (User Control Access)

### **Objective:**

- Create users
- Create roles to ease setup and maintenance of the security model
- Use the GRANT and REVOKE statements to grant and revoke object privileges
- Create and access database links

### **User Privileges**

- Database security:
  - System security
  - Data security
- System privileges: Gaining access to the database
- Object privileges: Manipulating the content of the database objects
- Schemas: Collections of objects, such as tables, views, and sequences

### **System Privileges**

- More than 100 privileges are available.
- The database administrator has high-level system privileges for tasks such as:
  - Creating new users
  - Removing users
  - Removing tables
  - Backing up tables

### **Creating Users**

The DBA creates users by using the CREATE USER statement.

General form: *CREATE USER userIDENTIFIED BY password;*

Example: *CREATE USER scottIDENTIFIED BY tiger;*

### **User System Privileges**

- Once a user is created, the DBA can grant specific system privileges to a user.

*GRANT privilege [, privilege...] TO user [, user] role, PUBLIC...;*

- An application developer, for example, may have the following system privileges:
  - CREATE SESSION
  - CREATE TABLE
  - CREATE SEQUENCE
  - CREATE VIEW
  - CREATE PROCEDURE

### **Granting System Privileges**

The DBA can grant a user specific system privileges.

*GRANT create session, create table, create sequence, create view TO scott;*

### ***Creating and granting privillages to ROLE***

*CREATE ROLE manager;*

*GRANT create table, create view TO manager;*

*GRANT manager TO DEHAAN, KOCHHAR;*

### ***Changing Your Password***

- The DBA creates your user account and initializes your password.
- You can change your password by using the ALTER USER statement.

*ALTER USER scott IDENTIFIED BY lion;*

### **Object Privileges**

<b>Object Privilege</b>	<i>Table</i>	<i>View</i>	<i>Sequence</i>	<i>Procedure</i>
<b>Alter</b>	√		√	
<b>Delete</b>	√	√		
<b>Execute</b>				√
<b>Index</b>	√			
<b>Insert</b>	√	√		
<b>References</b>	√	√		
<b>Select</b>	√	√	√	
<b>Update</b>	√	√		

- Object privileges vary from object to object.
- An owner has all the privileges on the object.
- An owner can give specific privileges on that owner's object.

*GRANT object\_priv[(columns)]ONobjectTO {user|role|PUBLIC}  
[WITH GRANT OPTION];*

### **Granting Object Privileges**

- Grant query privileges on the EMPLOYEES table.
- Grant privileges to update specific columns to users and roles.

*GRANT select ON employees TO sue, rich;*

*GRANT update (department\_name, location\_id) ON departments TO scott, manager;*

### Using the WITH GRANT OPTION and PUBLIC Keywords

- Give a user authority to pass along privileges.
- Allow all users on the system to query data from Alice's DEPARTMENTS table.

*GRANT select, insert ON departments TO scott WITH GRANT OPTION;*

*GRANT select ON alice.departments TO PUBLIC;*

### Confirming Privilege Granted

Data Dictionary View	Description
ROLE_SYS_PRIVS	System privileges granted to roles
ROLE_TAB_PRIVS	Table privileges granted to roles
USER_ROLE_PRIVS	Roles accessible by the user
USER_TAB_PRIVS_MADE	Object privileges granted on the user's objects
USER_TAB_PRIVS_RECD	Object privileges granted to the User
USER_COL_PRIVS_MADE	Object privileges granted on the columns of the user's objects
USER_COL_PRIVS_RECD	Object privileges granted to the user on specific columns
USER_SYS_PRIVS	Lists system privileges granted to the user

### How to Revoke Object Privileges

- You use the REVOKE statement to revoke privileges granted to other users.
- Privileges granted to others through the WITH GRANT OPTION clause are also revoked.

*REVOKE {privilege [, privilege...]} [ALL] ON object FROM {user[, user...]} [role] PUBLIC} [CASCADE CONSTRAINTS];*

As user Alice, revoke the SELECT and INSERT privileges given to user Scott on the DEPARTMENTS table.

*REVOKE select, insert ON departments FROM scott;*

### Exercise:

Suppose you are the DBA for the following schemas. Complete the following task with appropriate sql command.

**Employee**

<u>Eid</u>	EName	Job	Supervisor	Sal	Did
E001	Asif	Manager	E009	20000.00	10
E002	Arif	Manager	E009	30000.00	10
E004	Abul	Salesman	E001	15000.00	20
E005	Kuddus	Salesman	E001	15000.00	20
E006	Maruf	Salesman	E003	15000.00	20
E009	Hasan	President		40000.00	10

**Departments**

<u>Did</u>	Name	Manager
10	Admin	E009
20	Sales	E002

**Products**

<u>OrderID</u>	<u>ProductID</u>	Quantity
O001	P001	10
O002	P001	10
O002	P003	10
O003	P002	10

**OrderDetails**

<u>ProductID</u>	PName	Price
P001	Machinery	50000.00
P002	Hardware	55000.00
P003	Software	65000.00

- Create a user **Rahul** with the password **ret23erz**.
- Create a new role **Accounts**.
- Grant system privileges create table, view and sequence to role Accounts.
- Assign role Accounts to Rahul.
- Change password of **Rahul** with the new password **rec34tg**
- Grant query privilege to Asif and Arif on Departments table.
- Grant privilege update to column Price on OrderDetails table to role Manager and user Hasan.
- Give Asif the authority to pass along update and insert privilege on Departments table.
- Revoke the update and delete privileges given to user kuddus on Product table.

## **Last Week Lecture -(Project Presentation and Discussion)**