




Data Management and Database Design

Week #7 & #8

Northeastern University

The background of the slide features a dark, textured surface with faint, light-colored line art. This art includes a candlestick chart in the upper half and a bar chart in the lower half, both rendered in a minimalist, sketchy style. Overlaid on these charts are several thin, curved lines and dotted lines, suggesting trend lines or data paths. The overall aesthetic is technical and data-oriented.

Sequence

Sequence

- A sequence is an object in Oracle that is used to generate a number sequence
- This is useful if we need a number to be unique
- Sequences are mostly used to fill Primary key fields.
- Syntax to create sequence in oracle is as below.
- Current value of a sequence can be retrieved using **CURRVAL**
- Next value of sequence can be fetched using **NEXTVAL**
- Once sequence reaches Maximum value then it cannot be instantiated (ORA-08004 error)

```
CREATE SEQUENCE <sequence name>
    MINVALUE value
    MAXVALUE value
    START WITH value -- START WITH cannot be less than MINVALUE
    INCREMENT BY value
    CACHE value;
```

Show DEMO on SEQUENCE usage to input primary key column values

Sequence

- How to Update a Sequence

ALTER SEQUENCE <SEQUENCE NAME> ...

Examples –

ALTER SEQUENCE <sequence-name>

- How to Delete/drop a Sequence

DROP SEQUENCE <SEQUENCE NAME>

Note: In relational database systems an OBJECT (Such as Table, View, Sequence, Index, Constraints) can be deleted using a key word **DROP**.

However, Syntax will change based on the OBJECT that you plan to delete.

DML

Insert Data into Table

Insert Data to Table

Syntax –

```
INSERT INTO <table-name> (Column1, column2, ...)  
VALUES (Value1, Value2, ...);
```

Remember below points when inserting data into table –

- Varchar2 (Character data) and Date must be enclosed within Quotes
- NULL values are given as NULL
- If length of input values exceeds data definition length then we get Error
- Example: **INSERT INTO** EMP (empno, ename, sal) **VALUES** (8234, 'Sam', 5400);

Inserting few records is OK with Insert statements, What if you need to insert millions of Records?

Insert Data to Table

- `Insert into emp * from emp where job='MANAGER' ;` (As long as Constraints are satisfied data gets inserted)
- We can write select statements at column level while inserting data. See below

```
SQL> INSERT into EMP
```

```
VALUES (99,
```

```
        'Joe' ,
```

```
        'CLERK' ,
```

```
        NULL ,
```

```
        sysdate ,
```

```
        round(45000/12,0) ,
```

```
        null ,
```

```
(select deptno from dept where dname = 'OPERATIONS' )
```

```
);
```

Insert Data to Table

- **Points to remember for Insert statement –**
 - Specifying column names before VALUES is not mandatory
 - If column names are not specified, then you need to make sure the number of values that we pass are same as number of columns and positionally they should match.
 - If column names are specified then the order in which column names are specified, in the same order the values needs to be passed respectively.
 - When specifying column names, Make sure to include all not null column names

Insert Data via Parameter substitution

- Parameter substitution provides an easier way to enter data into a TABLE.
- The “&” symbol is used as substitution operator
- SQL *Plus or SQL Developer prompts for the value of the variable, accepts it and then substitutes in place of variable.

```
SQL> desc PRODUCTS
Name                                         Null?    Type
-----
PRODUCT_ID                                NOT NULL NUMBER
PRODUCT_NAME                              NOT NULL VARCHAR2(255)
DESCRIPTION                                VARCHAR2(2000)
STANDARD_COST                              NUMBER(9,2)
LIST_PRICE                                NUMBER(9,2)
CATEGORY_ID                                NOT NULL NUMBER

SQL> INSERT INTO PRODUCTS(product_id, product_name, list_price, category_id)
VALUES (&1, '&2', &3, &4);
  2 Enter value for 1: 9798
Enter value for 2: Dunkin Coffee
Enter value for 3: 2.99
Enter value for 4: 1
old  2: VALUES (&1, '&2', &3, &4)
new  2: VALUES (9798, 'Dunkin Coffee', 2.99, 1)

1 row created.

SQL> commit;

Commit complete.
```

Update Table Data

- UPDATE command is used to update the column in a table
- Values of a single column or a group of columns can be updated
- Updating can be carried out of all the rows in a table or selected rows

- Syntax –

```
UPDATE <table-name> SET <column-name> = <value> [,col-name=value,...]  
[WHERE <condition>]
```

- Example –

```
UPDATE EMP SET comm = comm + 10; -- This adds 10$ on comm for all employees
```

```
UPDATE EMP SET sal = sal + (sal *0.01); --Increase salary by 10% for all employees
```

```
UPDATE EMP SET deptno = 40 where ename = 'KING';
```

Update Table Data with Subqueries

- Modify the data for All the Managers who have more than 2 people reporting are to be directly report to PRESIDENT.

Update Table Data with Subqueries

- Modify the data for All the Managers who have more than 2 people reporting are to be directly report to PRESIDENT.

```
UPDATE EMP SET mgr = (select empno from emp where job = 'PRESIDENT')
WHERE mgr != (select empno from emp where job = 'PRESIDENT')
AND empno in (
    SELECT mgr FROM emp
    GROUP BY mgr
    HAVING count(*) > 2
);
```

Exercise

When Updating a table If we don't specify the WHERE clause, how will the query behave?

Delete Table Data

- DELETE command is used to delete rows from table
- Entire tuple is deleted from the table with DELETE statement
- Specific COLUMN values cannot be deleted using DELETE statement
- CASCADE delete rule instructs database to automatically set the foreign key values to null in the child rows if parent row is deleted

Delete from <table-name> [WHERE <condition>]

DELETE from EMP; --Deleted all the rows

DELETE from EMP where JOB='MANAGER' ;

**DELETE from EMP where deptno not in (
 SELECT deptno FROM dept where loc = 'BOSTON'
);**

Truncate

- Use the TRUNCATE TABLE statement to remove all rows from a table
- Removing rows with the TRUNCATE TABLE statement can be more efficient than dropping and re-creating a table.
- Dropping and re-creating a table invalidates dependent objects which adds more work
- Removing rows with the TRUNCATE TABLE statement can be faster than removing all rows with the DELETE statement especially if it has dependencies.
- Truncate is part of DDL and hence it commits automatically. We cannot rollback once a Truncate is issued.

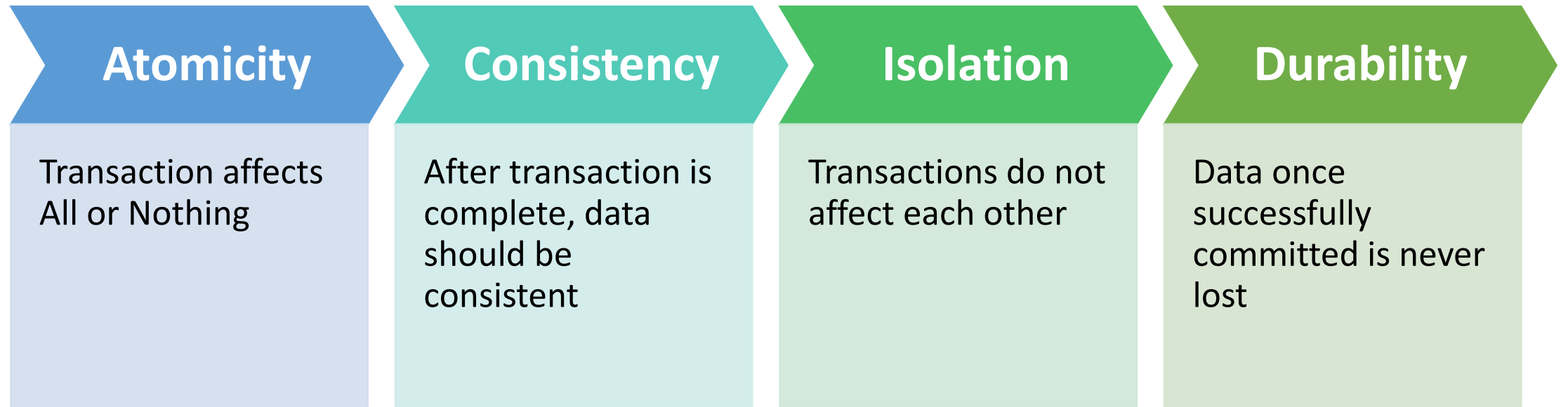
TCL – Transaction control

Commit and Rollback

- A **transaction** is a logical unit of work that contains one or more SQL statements.
- A transaction is an atomic unit.
- The effects of all the SQL statements in a transaction can be either
 - all **committed** (applied to the database) **OR**
 - all **rolled back** (undone from the database)
- A transaction begins with the first executable SQL statement.
- A transaction ends when it is committed or rollback
- DDL statement doesn't fall under TCL as they are implicitly commit in nature

ACID in DBMS

Refers to a standard set of properties
which guarantee that transactions are
processed reliably



Nested Queries

Sub Queries

Nested Queries

- The result of inner query is dynamically substituted in the condition of outer query
- There is no practical limitation to level of nesting of queries
- When using relational operators, ensure that sub query returns a single column output
- Can embed query in
 - FROM clause
 - SELECT clause
 - WHERE clause

Nested Queries Example

- Example – List the employees belonging to the Department of MILLER
To solve this, We need to first identify Department of MILLER and list employees using the returned department number

```
SQL> select deptno from emp where ename = 'MILLER';
```

```
DEPTNO  
-----  
10
```

```
SQL> select ename from emp where deptno = 10;
```

```
ENAME  
-----  
CLARK  
KING  
MILLER
```

```
SQL> select ename from emp where deptno = (select deptno from emp where ename = 'MILLER');
```

```
ENAME  
-----  
CLARK  
KING  
MILLER
```

```
SQL> 
```

Using Aggregate functions in Subqueries

- Aggregate function produces single value of any number of rows.

Lets say we want to see employee details whose salary is greater than average salary of employees whose Hire date is before 1st April, 1981.

Using Aggregate functions in Subqueries

- Aggregate function produces single value of any number of rows.

Lets say we want to see employee details whose salary is greater than average salary of employees whose Hire date is before 1st April, 1981.

```
1 select *
2   from emp
3  where sal > (
4    select avg(sal)
5      from emp
6     where hiredate > '01-APR-1981'
7  )
```

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7566	JONES	MANAGER	7839	02-APR-81	2975		20
7698	BLAKE	MANAGER	7839	01-MAY-81	2850		30
7782	CLARK	MANAGER	7839	09-JUN-81	2450		10
7788	SCOTT	ANALYST	7566	19-APR-87	3000		20
7839	KING	PRESIDENT		17-NOV-81	5000		10
7902	FORD	ANALYST	7566	03-DEC-81	3000		20

Answer the below

List the job with highest average salary (Show the avg salary value as well)

Show details of department whose Manager's employee code is **7698**

Subqueries in Having clause

List the job with highest average salary (Show the avg salary value as well)

```
SQL> r
1  SELECT job,
2      avg(sal) avg_sal
3  FROM emp
4  GROUP BY job
5  HAVING avg(sal) = ( SELECT max(avg(sal)) from emp group by job )
6*
```

Inner SQL

JOB	AVG_SAL
PRESIDENT	5000

The inner query first finds AVG sal for each different job group

MAX function picks highest avg salary (Which is 5000)

Now, That 5000 is used to apply as predicate for HAVING clause

GROUP BY clause in MAIN query is needed because Main query has aggregate and non aggregate column

Distinct clause with Subqueries

Distinct clause is used in certain cases to FORCE a SUBQUERY to generate SINGLE Value.

Show details of department whose Manager's employee code is **7698**

Lets see data in EMP table for manager code **7698**

```
SQL> select * from emp where mgr=7698;
```

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7499	ALLEN	SALESMAN	7698	20-FEB-81	1600	300	30
7521	WARD	SALESMAN	7698	22-FEB-81	1250	500	30
7654	MARTIN	SALESMAN	7698	28-SEP-81	1250	1400	30
7844	TURNER	SALESMAN	7698	08-SEP-81	1500	0	30
7900	JAMES	CLERK	7698	03-DEC-81	950		30

To return only 1 row with department number we would say

```
SELECT distinct deptno FROM emp whee mgr=7698;
```

Final query –

```
SELECT * from dept where deptno = (SELECT distinct deptno FROM emp where mgr=7698);
```

DEPTNO	DNAME	LOC
30	SALES	CHICAGO



Subqueries that return more than ONE row

- Most subqueries produce a relation containing multiple rows
- When a query returns more than one row we need to use **multirow** comparison operator
- Lets create a table called INCR

COLUMN NAME	DATA TYPE	DESCRIPTION
EMPNO	NUMBER(4)	Employee number Foreign key to EMP.EMPNO
INCR_AMT	NUMBER(7,2)	Salary increment amount
INCR_DT	DATE	Date on which Increment started

```
Create table incr(  
    empno      number(4) references emp(empno),  
    incr_amt   number(7,2),  
    incr_dt    date  
);
```

Exercise

- List names of employees who got an Increment

```
SELECT ename  
FROM emp  
WHERE empno IN (SELECT empno from incr);
```

- List names of employees who earn lowest salary in each Department

```
SELECT ename, sal, deptno  
FROM emp  
WHERE sal IN (SELECT min(sal) from emp GROUP BY deptno);
```

Things to remember when writing SUB QUERIES

- INNER QUERY –
 - Enclosed in Parentheses
 - On right hand side of condition
 - ORDER BY to be avoided
- SUBQUERIES widely used in
 - Set-membership tests: **IN**, **NOT IN**
- **IN(...)** and **NOT IN(...)** support subqueries that return multiple columns

Quiz

Question: Find Employee ID with Highest Salary for each Department

Quiz

First, Lets find Highest salary at each department

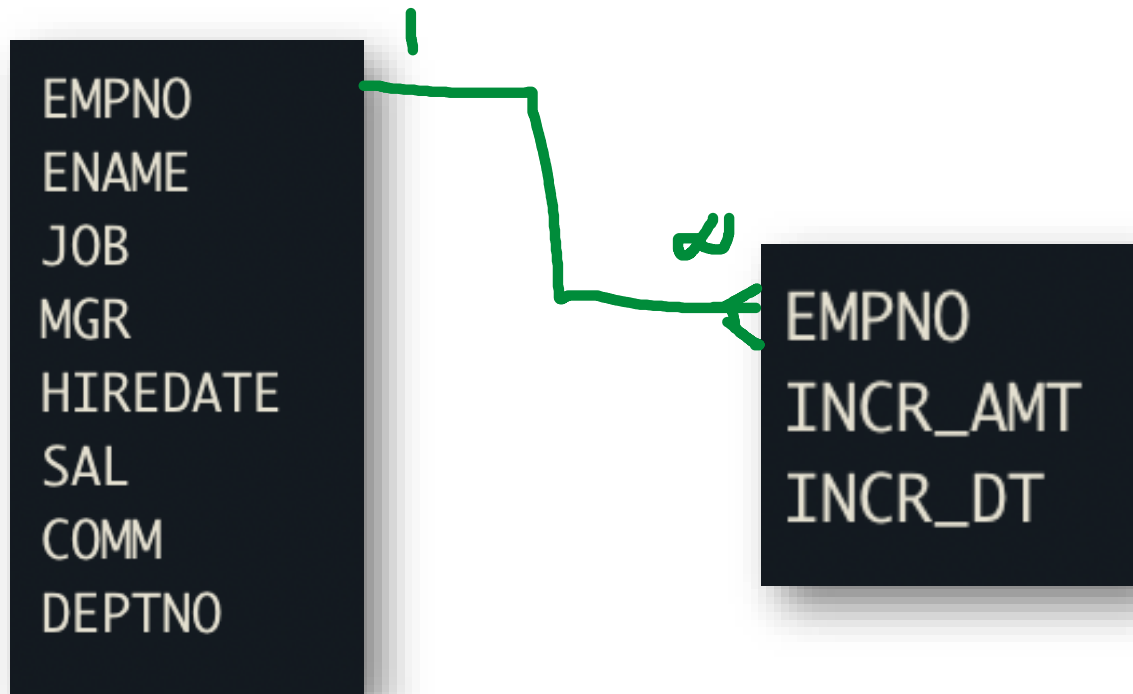
```
SELECT deptno, max(sal)
FROM emp
GROUP BY deptno;
```

Use above result to identify rest of the salary details

```
SELECT *
FROM emp
WHERE (deptno, sal) IN (
    SELECT deptno, max(sal)
    FROM emp
GROUP BY deptno);
```


Exercise

- List employee number, name, Total number of Increments and total increment amount for the employee who has got maximum number of increments



Exercise

```
SELECT  incr.empno,  
        ename,  
        count(*) as incr_counts,  
        sum(incr_amt) as total_incr  
FROM    emp, incr  
WHERE   emp.empno = incr.empno  
GROUP BY incr.empno, ename  
HAVING  count(*) = (  
    SELECT max(count(*))  
    FROM   incr  
    GROUP BY empno  
);
```

Things to remember when writing SUB QUERIES

- So, Subqueries are always executed from most deeply nested to least deeply nested UNLESS they are **Correlated** subqueries.

What are Correlated subqueries?

CORRELATED subqueries

- Correlated subquery is a nested subquery which is executed –
 - once for each “Candidate Row” considered by main query
- In correlated subquery, the column value used in inner sub query refers to –
 - The column value present in outer query forming a correlated subquery
- List the employee name and IDs who got more than ONE increment.

```
SELECT empno,  
       ename
```

```
FROM emp
```

```
WHERE 1 < (
```

```
    SELECT count(*) FROM incr WHERE empno = emp.empno
```

```
);
```

Same column



CORRELATED subqueries

- List employee details who earn salary greater than average salary for their department.

```
SELECT empno,  
       ename,  
       sal,  
       deptno  
FROM emp e  
WHERE sal > (  
    SELECT avg(sal)  
    FROM emp  
    WHERE deptno = e.deptno  
);
```

EMPNO	ENAME	SAL	DEPTNO
7499	ALLEN	1600	30
7566	JONES	2975	20
7698	BLAKE	2850	30
7788	SCOTT	3000	20
7839	KING	5000	10
7902	FORD	3000	20

CORRELATED subqueries

- **Points to remember –**
 - If possible, decorrelate the subquery
 - If not possible, performance testing needs to be performed
 - Database engine will automatically decorrelate

Subqueries special operators

- **Some special operators used in subqueries are –**
 - **EXISTS**
 - Used to check the existence of values
 - Produces Boolean result
 - Takes subquery as argument and evaluates to TRUE to produce output
 - **ANY, SOME and ALL**
 - Used along with relational operators
 - Similar to IN operator – However, used only subqueries

Subqueries special operators

- **EXISTS** operator is frequently used in correlated subqueries
- **NOT EXISTS** operator is more reliable if subquery returns any null values

List all employees who have at least one person reporting to him

```
SELECT empno, job, deptno
FROM emp e
WHERE EXISTS (
    SELECT empno
    FROM emp c
    WHERE c.mgr = e.empno
);
```


Subqueries special operators

- **ANY** operator compares the lowest value from the set

List the employee names whose salary is greater than the lowest salary of any employee belonging to department number 20

```
SELECT  ename
FROM    emp
WHERE   SAL > ANY (
    SELECT sal
    FROM    emp
    WHERE   deptno = 20
);
```

Subqueries special operators

- In case of **ALL** operator, the predicate is true –
 - if every value selected by subquery satisfies condition in the predicate of outer query.

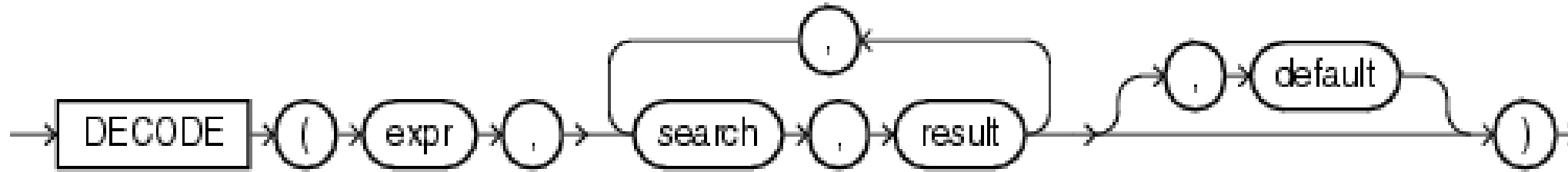
List employee names whose salary is greater than the highest salary of all employees belonging to department 20

```
SELECT  ename
FROM    emp
WHERE   SAL > ALL (
    SELECT sal
    FROM    emp
    WHERE   deptno = 20
);
```

Inner query returns salary of all employees who belong to DEPT 20.

Outer query selects employee name whose salary is greater than all the employees salary who belong to DEPT 20

Few more ...



DECODE compares *expr* to each *search* value one by one.

If *expr* is equal to a *search*,

then Oracle Database returns the corresponding *result*.

If **no match is found**,

then Oracle returns *default*.

If *default* is omitted,

then Oracle returns **null**.

SQL Continues...

1. List employee name, Job and Decode job name as –

Manager to Worker

Clerk to Boss

```
SELECT ename, job,  
       decode(job, 'CLERK', 'BOSS', 'MANAGER', 'WORKER', job) new_job  
FROM scott.EMP;
```

2. Generate amount due report as below using customer table

CODE	30-60 Days	60-90 Days	90-120 Days	>120 Days
101	100			
101		200		
101			399	
102	200			

```
7 create table customer (  
8   code number,  
9   amt number,  
10  duedate date  
11 )  
12 /  
13 select * From customer
```

CODE	AMT	DUE DATE
253	987	21-DEC-18
101	100	21-JAN-19
102	200	21-JAN-19
101	200	21-DEC-18
102	300	20-DEC-18
103	400	21-OCT-18
103	300	18-OCT-18
101	399	18-NOV-18
104	300	28-JAN-19
105	900	23-DEC-18
105	300	18-NOV-18
109	900	29-JAN-19
203	890	20-OCT-18
290	345	18-DEC-18

Amount Due Report solution

```

17 select code,
18        trunc(months_between(sysdate,duedate)) as "Mth diff",
19        decode(trunc(months_between(sysdate,duedate)),1,amt,null) as "30-60 days",
20        decode(trunc(months_between(sysdate,duedate)),2,amt,null) as "60-90 days",
21        decode(trunc(months_between(sysdate,duedate)),3,amt,null) as "90-120 days",
22        decode(trunc(months_between(sysdate,duedate)),0,amt,null) as "<30 days"
23 from customer;
24
25

```

CODE	Mth diff	30-60 days	60-90 days	90-120 days	<30 days
253	1	987	—	—	—
101	0	—	—	—	100
102	0	—	—	—	200
101	1	200	—	—	—
102	1	300	—	—	—
103	3	—	—	400	—
103	3	—	—	300	—
101	2	—	399	—	—
104	0	—	—	—	300
105	1	900	—	—	—
105	2	—	300	—	—
109	0	—	—	—	900
203	3	—	—	890	—
290	1	345	—	—	—

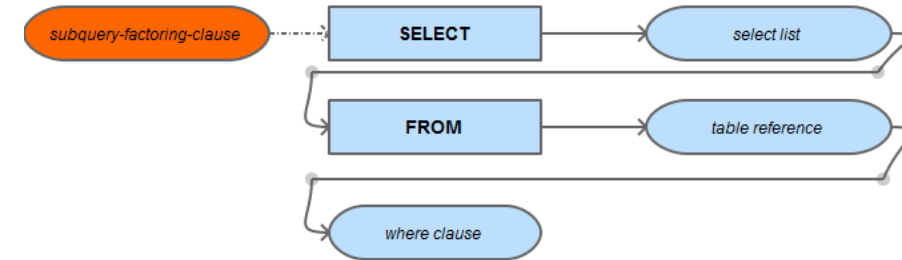
Few more ...

Display the total number of employees in each department as columns instead of rows (Pivot the output)

DEPT_10	DEPT_20	DEPT_30	DEPT_40
3	5	6	—

With clause

- **WITH** clause is a subquery factoring clause which is used
 - to create a named subquery block
- This block acts as a virtual table or an inline view
- you can reference your subquery repeatedly and can also act as Temporary table



```
WITH <alias> AS (subquery-select-sql)
SELECT <column-list> FROM <alias>;
```

This Query creates a subquery called emp_count, Which can be called in the FROM clause as a table

The main advantage with such clause is to keep the main query clean and simple.

```
1  WITH emp_count AS (
2      SELECT COUNT(*) num,
3             deptno
4      FROM scott.emp
5      GROUP BY deptno
6  )
7  SELECT dname department, loc location, num "number of employees"
8  FROM scott.dept, emp_count
9  WHERE dept.deptno = emp_count.deptno;
10
```

DEPARTMENT	LOCATION	number of employees
ACCOUNTING	NEW YORK	3
RESEARCH	DALLAS	5
SALES	CHICAGO	6

With clause

- Below example will illustrate as how we can reference subquery repeatedly (Transpose)

```
32 SELECT round(avg(sal),2) avgsal,  
33         job  
34     FROM scott.emp  
35     GROUP BY job  
36 ;  
37
```

AVGSAL	JOB
3000	ANALYST
1037.5	CLERK
1400	SALESMAN
2758.33	MANAGER
5000	PRESIDENT

```
13 WITH avgsal AS (  
14     SELECT round(avg(sal),2) avgsal,  
15           job  
16     FROM scott.emp  
17     GROUP BY job  
18 )  
19 SELECT a.avgsal "Manager Avg.",  
20        b.avgsal "Salesman Avg.",  
21        c.avgsal "Clerk Avg."  
22     FROM avgsal a,  
23          avgsal b,  
24          avgsal c  
25     WHERE a.job = 'MANAGER'  
26           AND b.job = 'SALESMAN'  
27           AND c.job = 'CLERK'  
28 ;  
29
```

Manager Avg.	Salesman Avg.	Clerk Avg.
2758.33	1400	1037.5

With clause

One more example (Left side using with clause and right side without with clause)

```
40 WITH emp_count AS (  
41     SELECT COUNT(*) cnt,  
42            deptno  
43     FROM scott.emp  
44     GROUP BY deptno  
45 ), avg_sal AS (  
46     SELECT round(AVG(sal),2) avgsal,  
47            deptno  
48     FROM scott.emp  
49     GROUP BY deptno  
50 )  
51 SELECT dname,  
52        loc,  
53        cnt number_of_employees,  
54        avgsal "Avg. salary"  
55 FROM scott.dept d,  
56        emp_count e,  
57        avg_sal a  
58 WHERE d.deptno = e.deptno  
59        AND d.deptno = a.deptno  
60 ;
```

DNAME	LOC	NUMBER_OF_EMPLOYEES	Avg. salary
SALES	CHICAGO	6	1566.67
ACCOUNTING	NEW YORK	3	2916.67
RESEARCH	DALLAS	5	2175

```
63 SELECT d.dname,  
64        d.loc,  
65        count(*) number_of_employees,  
66        round(avg(sal),2) as "Avg. salary"  
67 FROM scott.emp e,  
68        scott.dept d  
69 WHERE e.deptno = d.deptno  
70 GROUP BY d.dname,  
71          d.loc  
72 ;
```

DNAME	LOC	NUMBER_OF_EMPLOYEES	Avg. salary
RESEARCH	DALLAS	5	2175
ACCOUNTING	NEW YORK	3	2916.67
SALES	CHICAGO	6	1566.67

With clause

- Find out employees whose Salary is more than Average of all employee salaries

```
77 WITH temp_avgs AS(  
78     SELECT round(avg(sal),2) avg_sal  
79     FROM scott.emp  
80 )  
81 SELECT a.empno,  
82        a.ename,  
83        a.sal,  
84        b.avg_sal  
85 FROM scott.emp a  
86 JOIN temp_avgs b ON a.sal > b.avg_sal  
87 ;  
88
```

EMPNO	ENAME	SAL	AVG_SAL
7839	KING	5000	2073.21
7698	BLAKE	2850	2073.21
7782	CLARK	2450	2073.21
7566	JONES	2975	2073.21
7788	SCOTT	3000	2073.21
7902	FORD	3000	2073.21

CASE statement usage

For each customer in the customers table, the following statement lists the credit limit as "Low" if it equals \$100, "High" if it equals \$5000, and "Medium" if it equals anything else.

```
1  SELECT cust_last_name,  
2         CASE credit_limit WHEN 100 THEN 'Low'  
3                             WHEN 5000 THEN 'High'  
4                             ELSE 'Medium'  
5         END as grading  
6  FROM oe.customers;  
7  
8  SELECT cust_last_name,  
9         CASE WHEN credit_limit = 100 THEN 'Low'  
10          WHEN credit_limit = 5000 THEN 'High'  
11          ELSE 'Medium'  
12         END as grading  
13  FROM oe.customers;  
14
```

CUST_LAST_NAME	GRADING
Landis	Medium
Pacino	Medium
Fawcett	Medium

Analytical functions

Analytical Functions

Very useful – Analytic functions also operate on subsets of rows, similar to aggregate functions in GROUP BY queries, but they do not reduce the number of rows returned by the query.

1. ROW_NUMBER
2. RANK
3. DENSE_RANK
4. LEAD
5. LAG

All JOINS, WHERE, GROUP BY, and HAVING clauses are completed before the analytic functions are processed.

Analytical Functions Examples

Avg salary on entire table –

```
SQL> SELECT AVG(sal) FROM emp;
```

```
      AVG(SAL)
-----
2073.21429
```

The GROUP BY on Department groups subset of rows reduces rows at DEPT

```
SQL> r
1 SELECT deptno, AVG(sal) FROM emp
2* group by deptno
```

```
      DEPTNO      AVG(SAL)
-----
30 1566.66667
20 2175
10 2916.66667
```

If you want to see AVG salary against each row at their respective department with all rows displayed.

```
SQL> r
1 SELECT empno, deptno, sal,
2        AVG(sal) OVER (PARTITION BY deptno) AS avg_dept_sal
3* FROM emp
```

EMPNO	DEPTNO	SAL	AVG_DEPT_SAL
7782	10	2450	2916.66667
7839	10	5000	2916.66667
7934	10	1300	2916.66667
7566	20	2975	2175
7902	20	3000	2175
7876	20	1100	2175
7369	20	800	2175
7788	20	3000	2175
7521	30	1250	1566.66667
7844	30	1500	1566.66667
7499	30	1600	1566.66667
7900	30	950	1566.66667
7698	30	2850	1566.66667
7654	30	1250	1566.66667

14 rows selected.

Analytical Functions – ROW_NUMBER

The following example finds the three highest paid employees in each department –

```
15 SELECT deptno,  
16        ename,  
17        sal FROM (  
18          SELECT deptno,  
19                 ename,  
20                 sal,  
21                 ROW_NUMBER() OVER (PARTITION BY deptno ORDER BY sal desc) rn  
22          FROM scott.emp  
23        ) WHERE rn <= 3  
24 ORDER BY deptno, sal DESC, ename;
```

DEPTNO	ENAME	SAL	RN
10	KING	5000	1
10	CLARK	2450	2
10	MILLER	1300	3
20	SCOTT	3000	1
20	FORD	3000	2
20	JONES	2975	3
20	ADAMS	1100	4
20	SMITH	800	5
30	BLAKE	2850	1
30	ALLEN	1600	2
30	TURNER	1500	3
30	WARD	1250	4
30	MARTIN	1250	5
30	JAMES	950	6

Analytical Functions – RANK

Ranks the employees in the sample **HR** schema in department **60** based on their salaries –

Identical salary values receive the same rank. Standard Ranking

```
1 SELECT department_id, last_name, salary,  
2     RANK() OVER (PARTITION BY department_id ORDER BY salary) RANK  
3 FROM hr.employees WHERE department_id = 60  
4 ORDER BY RANK, last_name;
```

DEPARTMENT_ID	LAST_NAME	SALARY	RANK
60	Lorentz	4200	1
60	Austin	4800	2
60	Pataballa	4800	2
60	Ernst	6000	4
60	Hunold	9000	5

Analytical Functions – DENSE_RANK

Ranks the employees in the sample **HR** schema in department **60** based on their salaries –

Identical salary values receive the same rank. However, **no rank values are skipped**

```
1 SELECT department_id, last_name, salary,  
2         DENSE_RANK() OVER (PARTITION BY department_id ORDER BY salary) RANK  
3 FROM hr.employees WHERE department_id = 60  
4 ORDER BY RANK, last_name;
```

DEPARTMENT_ID	LAST_NAME	SALARY	RANK
60	Lorentz	4200	1
60	Austin	4800	2
60	Pataballa	4800	2
60	Ernst	6000	3
60	Hunold	9000	4

Analytical Functions – LAG

LAG function is used to access data from a previous row.

The following query returns the salary from the previous row

to calculate the difference between the salary of the current row and that of the previous row.

Note – ORDER BY of the LAG function is used to order the data by salary.

```
1 SELECT empno,  
2         ename,  
3         job,  
4         sal,  
5         LAG(sal, 1, 0) OVER (ORDER BY sal) AS sal_prev,  
6         sal - LAG(sal, 1, 0) OVER (ORDER BY sal) AS sal_diff  
7 FROM scott.emp  
8 ;
```

EMPNO	ENAME	JOB	SAL	SAL_PREV	SAL_DIFF
7369	SMITH	CLERK	800	0	800
7900	JAMES	CLERK	950	800	150
7876	ADAMS	CLERK	1100	950	150
7654	MARTIN	SALESMAN	1250	1100	150
7521	WARD	SALESMAN	1250	1250	0
7934	MILLER	CLERK	1300	1250	50
7844	TURNER	SALESMAN	1500	1300	200
7499	ALLEN	SALESMAN	1600	1500	100
7782	CLARK	MANAGER	2450	1600	850
7698	BLAKE	MANAGER	2850	2450	400
7566	JONES	MANAGER	2975	2850	125
7902	FORD	ANALYST	3000	2975	25
7788	SCOTT	ANALYST	3000	3000	0
7839	KING	PRESIDENT	5000	3000	2000

Analytical Functions – LEAD

LEAD function is used to return data from rows further down the result set.

Example query returns the salary from the next row

to calculate the difference between the salary of the current row and the following row.

```
1 SELECT empno,  
2        ename,  
3        job,  
4        sal,  
5        LEAD(sal, 1, 0) OVER (ORDER BY sal) AS sal_prev,  
6        sal - LEAD(sal, 1, 0) OVER (ORDER BY sal) AS sal_diff  
7 FROM scott.emp;
```

EMPNO	ENAME	JOB	SAL	SAL_PREV	SAL_DIFF
7369	SMITH	CLERK	800	950	-150
7900	JAMES	CLERK	950	1100	-150
7876	ADAMS	CLERK	1100	1250	-150
7654	MARTIN	SALESMAN	1250	1250	0
7521	WARD	SALESMAN	1250	1300	-50
7934	MILLER	CLERK	1300	1500	-200
7844	TURNER	SALESMAN	1500	1600	-100
7499	ALLEN	SALESMAN	1600	2450	-850
7782	CLARK	MANAGER	2450	2850	-400
7698	BLAKE	MANAGER	2850	2975	-125
7566	JONES	MANAGER	2975	3000	-25
7902	FORD	ANALYST	3000	3000	0
7788	SCOTT	ANALYST	3000	5000	-2000
7839	KING	PRESIDENT	5000	0	5000

Analytical Functions – SUM

SUM function on conjunction with partition by clause to calculate cumulative salary of the employees under each department.

```
1 select empno,  
2        ename,  
3        deptno,sal,  
4        sum(sal) over (partition by deptno order by empno)  
5 from scott.emp  
6 ;
```

EMPNO	ENAME	DEPTNO	SAL	SUM(SAL) OVER (PARTITIONBYDEPTNOORDERBYEMPNO)
7782	CLARK	10	2450	2450
7839	KING	10	5000	7450
7934	MILLER	10	1300	8750
7369	SMITH	20	800	800
7566	JONES	20	2975	3775
7788	SCOTT	20	3000	6775
7876	ADAMS	20	1100	7875
7902	FORD	20	3000	10875
7499	ALLEN	30	1600	1600
7521	WARD	30	1250	2850
7654	MARTIN	30	1250	4100
7698	BLAKE	30	2850	6950
7844	TURNER	30	1500	8450
7900	JAMES	30	950	9400

Exercise

- Which one of the following sorts rows in SQL with null values to be displayed on TOP?
 - ORDER BY <column name> WITH NULLS FIRST
 - SORT BY <column name> and IS NULL as ZERO
 - ORDER BY <column name> NULLS FIRST
- Display employee details who is making highest commission?
- searches for employees with the pattern A_B in their name (underscore is part of a name we need to find the employees which contain “_” character in their names)
- Show top 5 highest paid employees
- Display highest salary of the employees reporting to the same manager as the employee (Show the MAX Salary from same reporting manager group)

Exercise

- Display Salary Grades for each employee. Details of SALGRADE table are as below –

COLUMN	DESCRIPTION	DATA TYPE	Constraint
GRADE	Grading number based on Salary	NUMBER	Primary Key
LOSAL	Low Salary range	NUMBER	
HISAL	High Salary range	NUMBER	

- Data contents as below –
 - Use SQL Developer to insert data

GRADE	LOSAL	HISAL
1	700	1200
2	1201	1400
3	1401	2000
4	2001	3000
5	3001	9999

VIEWS

VIEWS

- It is a virtual TABLE
- It has “NO DATA” by itself
- View is created using a Select statement
- View updatable
- Security – Restricts access on data based on business needs
- These are database objects
- “CREATE or REPLACE VIEW” command creates/modifies view
- Changes on base tables reflect automatically

VIEWS

- Restrict access to database by selecting desired columns and data
- No redundancy problem as data is not stored
- Allowing users to apply simple queries on complex queries
- View cannot contain **order by** clause
- We can encapsulate column names using Alias
- View can also be joined to another VIEW and/or TABLE

VIEWS

- INSERT, UPDATE, DELETE can also be used on VIEWS
 - Indirect means of manipulating data
 - To support this –
 - Should not contain aggregate columns
 - No Distinct key word
 - No Group by
 - No subqueries
- WITH CHECK OPTION clause allows integrity constraints and data validation checks to be enforced

VIEWS

- Read only view

```
create view v_emp (empno, tot_sal)
as select empno, sal+nvl(comm,0) tot_sal from emp
```

- With check option clause

```
create view v_emp_dept
as select empno, ename, deptno from emp
Where deptno = 30; -- Rows can be inserted without any restriction
```

```
create view v_emp_dept
as select empno, ename, deptno from emp
where deptno = 30
with check option; -- Rows can be inserted only for dept 30
```

VIEWS

- **Constraint name to With check option clause**

```
create view v_emp_dept
as select empno, ename, deptno from emp
where deptno = 30
with check option constraint DEPTNO_30; -- Rows can be inserted only for dept 30
```

- **Updating a join view**

```
create view v_emp AS
SELECT a.empno, a.ename, a.deptno, b.dname, b.loc
from EMP a, DEPT b
WHERE a.deptno = b.deptno
AND b.loc in ('BOSTON', 'DALLAS');
```

```
UPDATE v_emp set ename = 'JOE' where ename='ADAMS';
```

MAT Views

MAT Views

- A materialized view is a **replica** of a **target master** from a single point in time.
- The master can be either a master table at a master site or a master materialized view at a materialized view site.
- You can use materialized views to achieve one or more of the following goals –
 - Network loads
 - Data sub setting
 - Disconnected computing

MAT Views

- Read-only MAT View

```
CREATE MATERIALIZED VIEW scott.emp  
AS SELECT * FROM hr.employees@<dblink>;
```

- Updatable MAT View

```
CREATE MATERIALIZED VIEW scott.emp FOR UPDATE  
AS SELECT * FROM hr.employees@<dblink>;
```

- MAT View Refresh options

- **COMPLETE** — Refreshes by recalculating the defining query of the materialized view
- **FAST** — Refreshes by incrementally applying changes
- **FORCE** — Attempts a fast refresh. If that is not possible, it does a complete refresh

Views vs MVs

View	Materialized View MV
View query results are not persistent <i>Its just a SQL Query</i>	Result of the query execution is persistent <i>Pre-Answered Query</i>
ROWID's of View and Base tables are same	ROWID's of MAT view and Base tables are different
Result set is always latest	MV needs refresh periodically to get latest result set.
Performance depends on base tables	Performance is high as it has no dependency on Base tables to fetch data
No additional space required	Additional space is required
Indexes are on base table(s)	Can have Indexes on MV

Security Users Roles



Questions?