**CSE2007 DBMS LAB**

**SLOT: L39+L40**

**NAME - AMAN SAHU**

**REG. NO – 22BCE7224**

**EXPERIMENT NO.-7**

# 1.



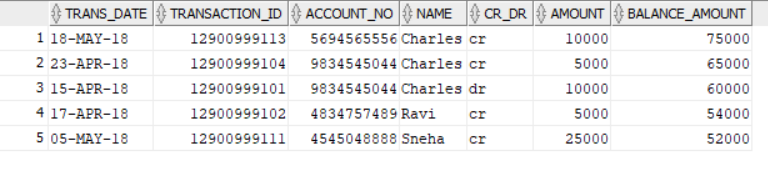
1. Select Top 5 Balanced accounts

SELECT \* FROM (

SELECT a.\*, ROW\_NUMBER() OVER (PARTITION BY account\_no ORDER BY balance\_amount DESC) rn

FROM transactions a

) WHERE rn <= 5;



1. Select latest 5 transactions for each account

SELECT account\_no, name, trans\_date, transaction\_id, cr\_dr, amount, balance\_amount

FROM (

SELECT account\_no, name, trans\_date, transaction\_id, cr\_dr, amount, balance\_amount,

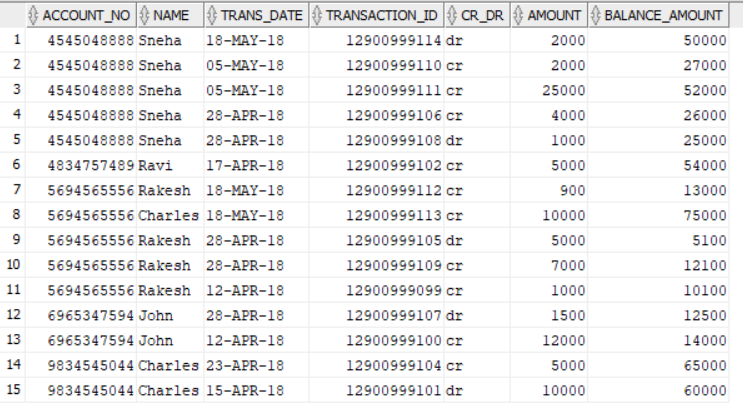
ROW\_NUMBER() OVER (PARTITION BY account\_no ORDER BY trans\_date DESC) rn

FROM transactions

)

WHERE rn <= 5

ORDER BY account\_no, trans\_date DESC;

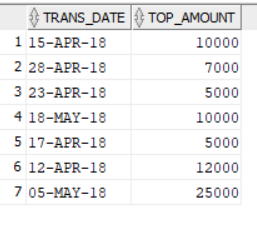


1. Which is the top amount transaction in each day

SELECT Trans\_date, MAX(amount) AS top\_amount

FROM transactions

GROUP BY Trans\_date;

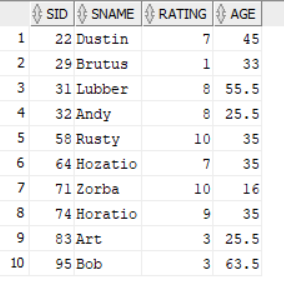


2.

1. Create view on sailors table with all the attributes as sailors\_v and perform DML operations. Note down your observations in base table.

CREATE VIEW sailors\_u AS SELECT \* FROM sailors;

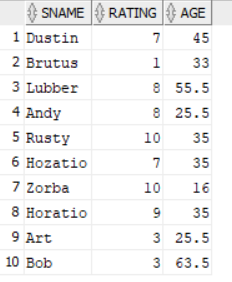
select \* from sailors\_u;



1. Create view on sailors table with sname, rating, age attributes as sailors\_v1 and perform DML operations. Justify your answer with proper observation on base table.

CREATE VIEW sailors\_v1 AS SELECT sname, rating, age FROM sailors;

select \* from sailors\_v1;



1. Create a view ‘sailor\_boat’ to display the all details of sailors and boats who have reserved boats 103 or 104. Justify your answer about whether we able to apply DML operations on this view.

CREATE VIEW sailor\_boats AS

SELECT s.\*, b.\*

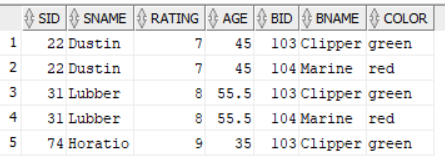
FROM sailors s

JOIN reserves r ON s.sid = r.sid

JOIN boats b ON r.bid = b.bid

WHERE b.bid IN (103, 104);

select \* from sailor\_boats;



1. Drop all non-updatable views created by you.

DROP VIEW sailor\_boats;

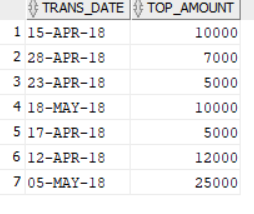
DROP VIEW sailors\_v1;

DROP VIEW sailors\_v;

3

a.Create a non-unique index on color attribute of Boats.

CREATE INDEX idx\_color ON boats(color);



b.Create a function-based index on name column of employee table. Function to be used is

*substr* on the 3rd and 4th characters.

CREATE INDEX idx\_emp\_name ON emp (SUBSTR(ename, 3, 4));

SELECT index\_name, table\_name, uniqueness

FROM user\_indexes

WHERE table\_name = 'EMP' AND index\_name = 'IDX\_EMP\_NAME';

c. Create a composite index on department and salary attributes of employee.

CREATE INDEX idx\_emp\_dept\_sal ON emp (deptno, sal);

SELECT index\_name, table\_name, uniqueness

FROM user\_indexes

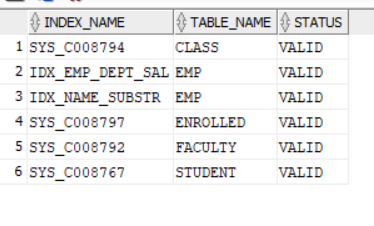
WHERE table\_name = 'EMP' AND index\_name = 'IDX\_EMP\_DEPT\_SAL';



d. Refer the system table to check the status of created indexes.

SELECT index\_name, table\_name, status

FROM user\_indexes;



4.

1. Copy the structure of employee table in to emp\_v2 table.

CREATE TABLE emp\_v2 AS SELECT \* FROM emp WHERE 1=0;

SELECT \* FROM EMP\_V2;



1. Use the **merge** statement to copy the contents of employee table into emp\_v2.

MERGE INTO emp\_v2 e

USING (SELECT \* FROM emp) s

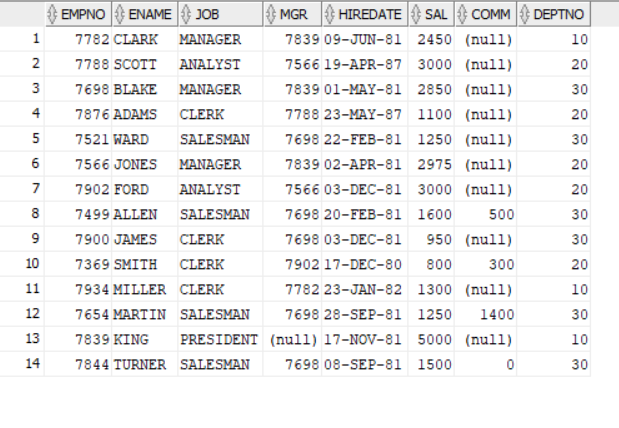
ON (e.empno = s.empno)

WHEN NOT MATCHED THEN

INSERT (e.empno, e.ename, e.job, e.mgr, e.hiredate, e.sal, e.comm, e.deptno)

VALUES (s.empno, s.ename, s.job, s.mgr, s.hiredate, s.sal, s.comm, s.deptno);

SELECT \* FROM emp\_v2;



1. Change the commission value of an employee in employee table and update it in emp\_v2 using merge statement

UPDATE emp SET comm = ... WHERE empno = ...;

MERGE INTO emp\_v2 e

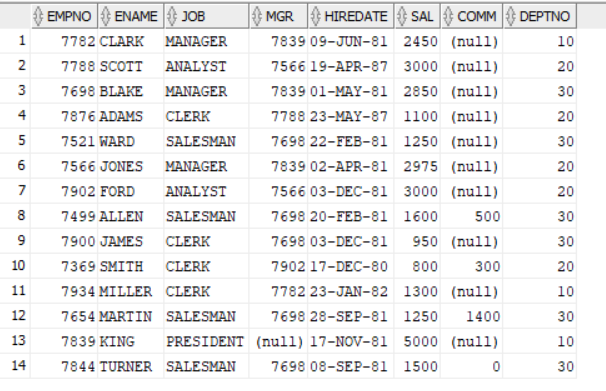
USING (SELECT \* FROM emp) s

ON (e.empno = s.empno)

WHEN MATCHED THEN

UPDATE SET e.comm = s.comm;

SELECT \* FROM emp\_v2;



1. Delete a employee in source table and update it in emp\_v2 using merge statement.

DELETE FROM emp WHERE empno = ...;

MERGE INTO emp\_v2 e

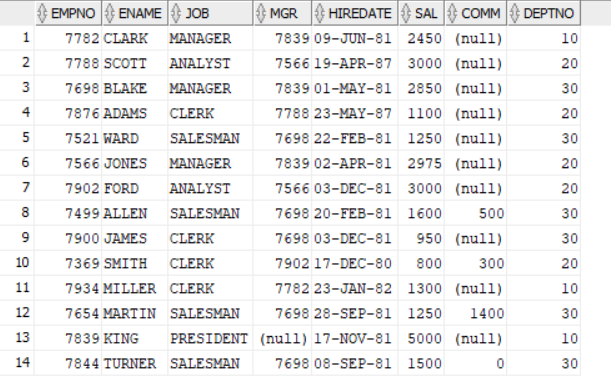
USING (SELECT \* FROM emp) s

ON (e.empno = s.empno)

WHEN NOT MATCHED THEN

DELETE;

SELECT \* FROM emp\_v2;



5.

1. Do self-study on ‘explain plan for’ command and write your observations about the output. Identify how indexing can improve the performance of query with query optimizer.

Ans. The EXPLAIN PLAN FOR command in Oracle SQL is a powerful tool for understanding the execution plan that the Oracle optimizer would use to execute a query. It provides a detailed breakdown of the operations the database would perform, such as table access or sort, the options describing the operation, the name of the database object acted upon by the step, and an estimated cost proportional to the expected resource use needed to execute the operation.

Indexing can significantly enhance the performance of a query. When an index is created on a column, the database can use the index to quickly locate the data without having to scan the entire table. This can result in a substantial performance boost, especially for large tables. The Oracle optimizer can leverage these indexes to create more efficient execution plans, reducing the time and resources required to execute queries.

1. Do self-study on how partitioning of real time databases helps in query processing? What are the different partitioning types? Write your observations with the examples.

Ans. Partitioning is a technique used in databases to divide a large table into smaller, more manageable pieces called partitions. Each partition is stored separately, leading to improved query performance because a query can scan a single partition instead of the entire table. This is particularly beneficial in real-time databases where performance and speed are critical.

There are several types of partitioning:

Range Partitioning: The data is partitioned according to a specified range. For example, a table that stores sales data could be partitioned by date, with each partition containing a month of data.

List Partitioning: The data is partitioned according to a list of values. This is useful when the partition key has a discrete set of known values.

Hash Partitioning: The data is partitioned using a hash function on the partition key. This ensures a more even distribution of data among partitions, which can lead to improved performance.

Composite Partitioning: This is a combination of other partitioning types, allowing for more complex partitioning schemes.

By partitioning the data, queries that filter by the partition key can run faster because they need to scan less data. This can significantly improve the performance of real-time databases. For example, a query that retrieves sales data for a specific month would only need to scan the partition for that month, rather than the entire sales table. This can lead to faster query execution and more efficient use of resources.