SEQUENCES

Wednesday, May 15, 2024 9:13 AM

SEQUENCES:

- **SEQUENCE is one ORACLE DB OBJECT.**
- **SEQUENCE is used to generate sequential integers.**

EMPID	CUSTID	PRODUCTID
1234	123456	1001
1235	123457	1002
1236	123459	1003
1237		

Syntax:

```
CREATE SEQUENCE <seq_name>
[START WITH <value>]
[INCREMENT BY <value>]
[MINVALUE <value>]
[MAXVALUE <value>]
[CYCLE / NOCYCLE]
[CACHE <size> / NOCACHE];
```

1	10
2	20
3	30
4	40
5	50
6	60
7	70
8	80
9	90
10	

Example:

```
CREATE SEQUENCE s1;
```

CLAUSE [Sequence Option]	DEFAULT VALUE
START WITH	1
INCREMENT BY	1
MINVALUE	1
MAXVALUE	10 power 28

CYLCE	NOCYCLE
CACHE	20

User_Sequences:

- It maintains all sequences info

```
SELECT * FROM user_sequences;
```

Pseudo Columns of SEQUENCE:

SEQUENCE provides 2 pseudo columns. They are:

- NEXTVAL
- CURRVAL

Syntax:

```
<sequence_name>.<pseudo_column>
```

Example:

s1.nextval	it returns next value in sequence
s1.currval	it returns current value in the sequence

Example:

EMPLOYEE

EMPID	ENAME	SAL

generate sequntial emp id. start from 1001:

```
CREATE TABLE employee
(
empid NUMBER(4),
ename VARCHAR2(10),
sal NUMBER(8,2)
);
```

1001

```
sal NUMBER(8,2)
);

CREATE SEQUENCE s2
START WITH 1001
MAXVALUE 1005;
```

1001
1002
1003
1004
1005

```
INSERT INTO employee VALUES(s2.nextval, '&ename', &sal);
```

Output:

```
enter value for ename: A
enter value for sal: 6000

/
```

Output:

```
enter value for ename: B
enter value for sal: 9000

/
```

Output:

```
enter value for ename: C
enter value for sal: 5000

/
```

Output:

```
enter value for ename: D
enter value for sal: 7000

/
```

Output:

```
enter value for ename: E
enter value for sal: 4000

/
```

Output:

```
enter value for ename: F
enter value for sal: 6000
ERROR: sequence reached max value
```

Altering Sequence:

Syntax:

```
ALTER SEQUENCE <seq_name>
```

<SEQUENCE_OPTIONS>;

Example:

ALTER SEQUENCE s2
MAXVALUE 1020 INCREMENT BY 2;

Example:

COURSE

CID	CNAME
10	JAVA
20	ORACLE
30	HTML
..	
..	
90	PYTHON

generate course ids using sequence:

```
CREATE SEQUENCE s3
START WITH 10
INCREMENT BY 10
MAXVALUE 90;

INSERT INTO course VALUES(s3.nextval, '&cname');
```

START WITH	is used to specify starting value in sequence
INCREMENT BY	is used to specify step value
MINVALUE	is used to specify min value in the sequence. it is useful in CYCLE.
MAXVALUE	is used to specify max value in sequence

501
502
.
.
999

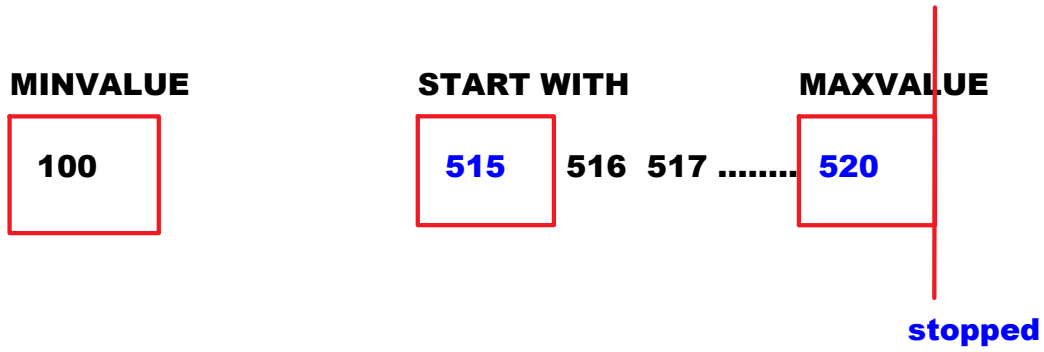
CYCLE / NOCYCLE:

- default one: NOCYCLE [no repetition]

CASE-1: NOCYCLE

```
CREATE SEQUENCE s4
START WITH 515
```

```
MINVALUE 100
MAXVLAUE 520
NOCYCLE;
```

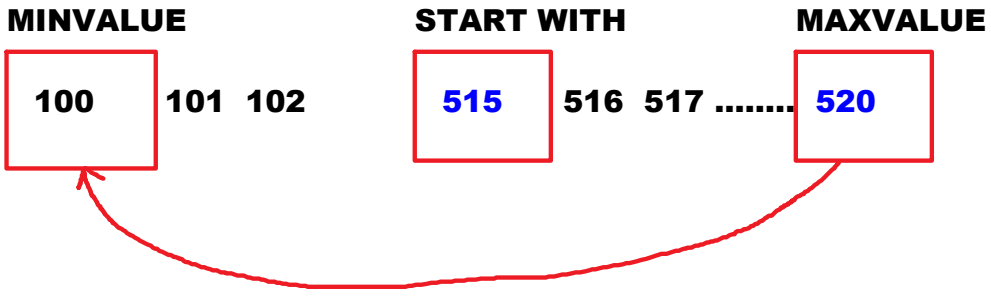


If sequence is created with NOCYCLE,

- **sequence starts from START WITH value**
- **generates next value up to MAXVALUE**
- **after reaching MAXVALUE, sequence will be stopped.**

CASE-2: CYCLE

```
CREATE SEQUENCE s4
START WITH 515
MINVALUE 100
MAXVLAUE 520
CYCLE;
```



If sequence is created with CYCLE,

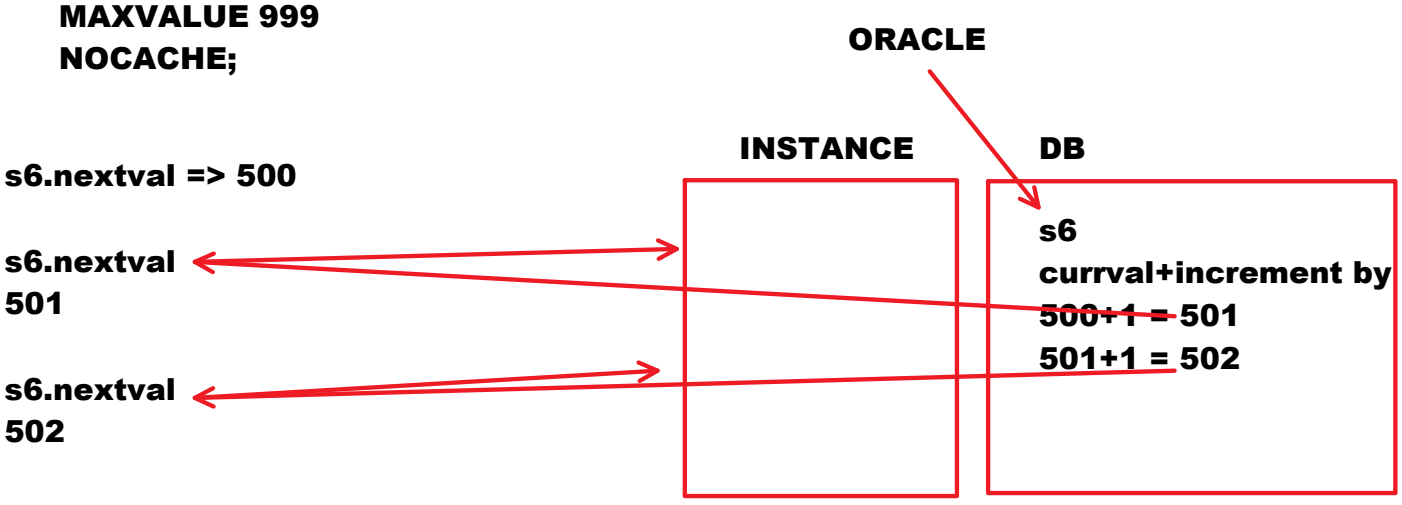
- **sequence starts from START WITH value**
- **generates next value up to MAXVALUE**
- **after reaching MAXVALUE, it will be reset to MINVALUE.**
- **Now, it generates sequential numbers from MINVALUE to MAXVALUE**

CACHE <size> / NOCACHE:

- **Default CACHE size: 20**
- **CACHE is used to improve the performance of generating sequential numbers.**

CASE-1: NOCACHE

```
CREATE SEQUENCE s6
START WITH 500
MAXVALUE 999
NOCACHE;
```



If sequence is created with NOCACHE,

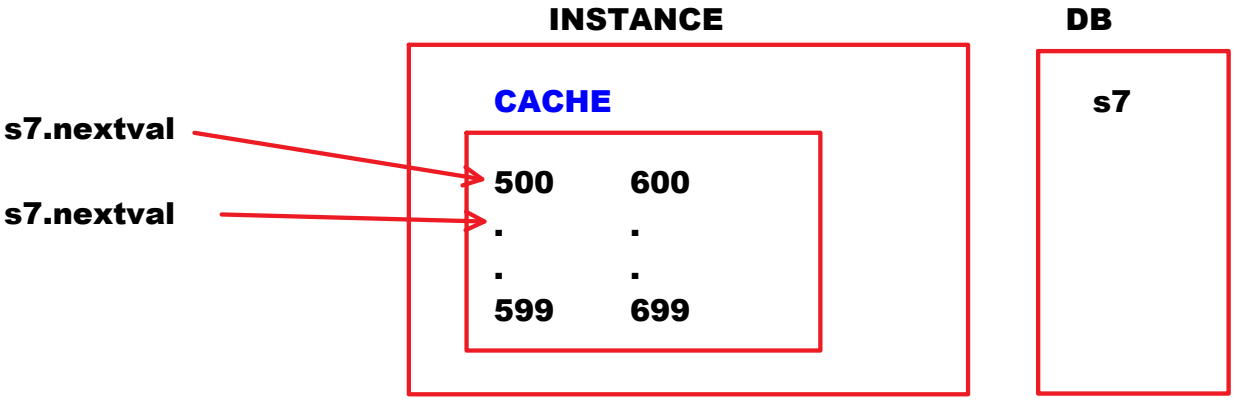
- **For every SEQUENCE CALL,**
  ORACLE goes to DB
  Identifies CURRVAL
  adds INCREMENT BY value
  returns Sequential value to Sequence call

It degrades the performance.
If number of travels to DB are increased then performance will be degraded.

To improve performance we use CACHE

CASE-2: CACHE 100

```
CREATE SEQUENCE s7
START WITH 500
MAXVALUE 999
CACHE 100;
```



If sequence is created with CACHE,

- **For every SEQUENCE CALL,**
  ORACLE will not go to DB.
  Just it collects sequential number from CACHE.

CACHE reduces number of travel to DB.
So, it improves the performance.

Can we call the sequence from CREATE command?
YES.
From ORACLE 12C version onwards we can call the sequence from CREATE command.

Example:

```
CREATE SEQUENCE s8
```

```
START WITH 5001
MAXVALUE 9999;

CREATE TABLE employee1
(
empid NUMBER(4) DEFAULT s8.nextval,
ename VARCHAR2(10),
cname VARCHAR2(10) DEFAULT 'WIPRO'
);

INSERT INTO employee1(ename) VALUES('A');
INSERT INTO employee1(ename) VALUES('B');
INSERT INTO employee1(ename) VALUES('C');
COMMIT;

SELECT * FROM employee1;
Output:
empid
---------
5001
5002
5003
```

Can we call the sequence from UPDATE command?
YES.

Example:
Make all empnos in sequential order in emp table:

```
CREATE SEQUENCE s9
START WITH 5001
MAXVALUE 9999;

UPDATE emp
SET empno=s9.nextval;
```

Generate sequential numbers in descending order from 50 to 1:

```
CREATE SEQUENCE s10
START WITH 50
INCREMENT BY -1
MINVALUE 1
MAXVALUE 50;

SELECT s10.nextval FROM dual;   --50
SELECT s10.nextval FROM dual;   --49
SELECT s10.nextval FROM dual;   --48
```

```
50
49
48
.
.
1
```

Note:
From ORACLE 12c version onwards,
we can generate sequential numbers using 2 ways:

- **using SEQUENCE**
- **using IDENTITY**

Generating sequential values using IDENTITY:

Syntax:

```
CREATE TABLE <name>
(
   <field_name> <data_type> GENERATED ALWAYS AS IDENTITY(sequence_options)
);
```

Example:

```
CREATE TABLE student1
(
sid NUMBER(4) generated always as identity,
sname VARCHAR2(10)
```

```
);

INSERT INTO student1(sname) VALUES('A');
INSERT INTO student1(sname) VALUES('B');
INSERT INTO student1(sname) VALUES('C');
INSERT INTO student1(sname) VALUES('D');

select * from student1;
output:
SID
-------
1
2
3
4
```

Example:

COURSE10

CID	CNAME
10	
20	
..	
90	

```
CREATE TABLE course10
(
cid NUMBER(2) generated always as
identity(START WITH 10 INCREMENT BY 10
MAXVALUE 90),
cname VARCHAR2(10)
);
```

Dropping Sequence:

Syntax:

```
DROP SEQUENCE <seq_name>;
```

Example:

```
DROP SEQUENCE s1;
```

VIEWS

Thursday, May 16, 2024 9:17 AM

VIEWS:

- **VIEW is one ORACLE DB Object.**
- **VIEW is Virtual Table.**
- **Virtual Table means, It does not contain physical data. It does not occupy the memory.**
- **VIEW holds SELECT QUERY.**
- **When we retrieve data through VIEW implicitly ORACLE runs SELECT QUERY which is stored in VIEW.**

Syntax:

```
CREATE [OR REPLACE] VIEW <name>
AS
<SELECT QUERY>;
```

Example:

```
CREATE VIEW v1
AS
SELECT empno,ename,job FROM emp;
```

Output:

```
view created.
```

v1

```
SELECT empno,ename,job
FROM emp
```

```
SELECT * FROM v1;
```

Output:

```
EMPNO      ENAME     JOB
-------------------------------------------
7369       SMITH     CLERK
7499       ALLEN     SALESMAN
```

Note:

SELECT * FROM v1;

above query will be rewritten by ORACLE as following:

SELECT * FROM (SELECT empno,ename,job FROM emp);

Granting permission to create the view:

Login as DBA:

username: system
password: nareshit

```
GRANT create view
TO c##batch9am;
```

Note:
A table on which view is created is called "Base table"

c##batch9am	c##userA
CREATE VIEW v2 **AS** **SELECT empno,ename,job** **FROM emp;** **v2 => created based on emp** **emp => base table** **GRANT all ON v2** **TO c##userA;**	**SELECT * FROM c##batch9am.v2;** **Output:** **EMPNO ENAME JOB** **----------** **.... ....** **INSERT INTO c##batch9am.v2** **VALUES(7001,'AA','CLERK');** **Output:** **1 row created.** **Note:** **When we insert data through view it will be inserted in base table.**
SELECT * FROM emp; **Output:** **does not display 7001 record**	

Output:
does not display 7001 record

```
COMMIT;
```

```
SELECT * FROM emp;
```
Output:
displays 7001 record

```
UPDATE c##batch9am.v2
SET job='MANAGER'
WHERE empno=7001;

COMMIT;
```

```
DELETE FROM c##batch9am.v2
WHERE empno=7001;

COMMIT;
```

Advantages:

- **VIEW provides security for the data.**
- **VIEW reduces complexity and simplifies the queries.**

Security:

To implement database level security we use SCHEMA.
To implement table level security we use GRANT, REVOKE.
To implement data level security we use VIEW.

Data Level Security:

Data Level Security can be implemented at 2 levels.

- **Column Level**
- **Row Level**

Column Level Security:

Example:

EMP

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO

V3

EMPNO	ENAME	JOB

On this VIEW v3 we give permission to others.
Now, others can see 3 columns data only.
For remaining 5 columns we are providing security.

Row Level Security:

EMP

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
							10
							10
							10
							20
							20
							30
							30

create view v4 on 10th dept records and give permission on v4 to 10th dept manager

now 10th dept manager can see only 10th dept records.

for remaining rows we are providing security.

Example on Row Level Security:

c##batch9am	c##userA
CREATE VIEW v4 AS SELECT * FROM emp WHERE deptno=10; GRANT all ON v4 TO c##userA;	
	SELECT * FROM c##batch9am.v4; Output: displays 10th dept records only
	INSERT INTO c##batch9am.v4(empno,ename,deptno) VALUES(1001,'A',30); Output: 1 row created.

WITH CHECK OPTION:

```
CREATE OR REPLACE VIEW v4
AS
SELECT * FROM emp
WHERE deptno=10
WITH CHECK OPTION;
```

WITH CHECK OPTION clause restricts the user from entering WHERE condition violated records.

```
INSERT INTO
c##batch9am.v4(empno,ename,deptno)
VALUES(1002,'B',30);
Output:
ERROR: WITH CHECK OPTION -
WHERE clause violated
```

Types of Views:

2 types:

- Simple View / Updatable View
- Complex View / Read-Only View

Simple View:

- **If view is created based on one table then it is called "Simple View".**

- **we can perform DML operations through SIMPLE VIEW. That's why it can be also called as "Updatable View".**

  Examples:

```
CREATE VIEW v2
AS
SELECT empno,ename,job
FROM emp;


CREATE OR REPLACE VIEW v4
AS
SELECT * FROM emp
WHERE deptno=10
WITH CHECK OPTION;
```

Complex View:

- **If VIEW created based on multiple tables (joins) / group by / having / aggregate functions / sub queries / set operators / expressions then it is called "Complex View".**

- **We cannot perform DML operations through Complex View. That's why it can be also called as "Read-Only View".**

  Examples:

```
CREATE VIEW v5
AS
SELECT e.ename, d.dname
FROM emp e, dept d
WHERE e.deptno=d.deptno;

SELECT * FROM v5;



CREATE VIEW v6
AS
SELECT deptno, sum(sal) AS sum_of_sal
FROM emp
GROUP BY deptno;
```

```
SELECT * FROM v6;
```

user_views:

- **it is a system table / readymade table**
- **it maintains all views information**

```
SELECT view_name, text
FROM user_views;
```

Dropping View:

Syntax:

```
DROP VIEW <view_name>;
```

Example:

```
DROP VIEW v1;
```

VIEW:

- **virtual table => no physical data**
- **view holds select query**
- **when we retrieve data through view it runs select query which is in view.**
- **a table on which view is created is called "base table".**
- **when we perform DML operations through view these will be performed in base table.**

Advantages:

- **security**
- **reduces complexity and simplifies queries**

Types of Views:

2 types:

- **simple view / updatable view => 1 table**
- **complex view / read-only view => multiple tables/group by ...etc**

INDEXES

Friday, May 17, 2024 9:18 AM

INDEX GOAL:
INDEX improves performance of data retrieval.

BOOK INDEX

Chapter	Pg No
DDL commands	10
DML commands	20
JOINS	60
SUB QUERIES	80

INDEXES:

- **INDEX is one ORACLE DB Object.**
- **INDEX is used to improve the performance of data retrieval.**
- **INDEX is created on columns.**
- **The column which we use frequently in WHERE condition, on that we create INDEX to improve the performance of data retrieval.**
- **using BOOK INDEX we can refer the chapter quickly. In the same way, using ORACLE INDEX we can retrieve the records quickly.**
- **BOOK INDEX contains chapter name and pgno. in the same way, ORACLE INDEX contains values and row ids.**

Syntax:

```
CREATE INDEX <index_name>
ON <table_name>(<columns_list>);
```

When we submit SELECT command, ORACLE may perform any 1 of 2 scans. They are:

- **Table Scan**
- **Index Scan**

- **If INDEX is not created ORACLE performs TABLE SCAN**
- **If INDEX is created ORACLE performs INDEX SCAN**

- **INDEX SCAN reduces number of comparisons. So, it improves the performance.**

- **INDEX SCAN is faster than TABLE SCAN.**

Example on Creating Index:

to see execution plan:
SET AUTOTRACE ON EXPLAIN

SELECT ename, sal FROM emp WHERE sal>13000; --Table Scan

CREATE INDEX i1 ON emp(sal);
Output:
Index created.

SELECT ename, sal FROM emp WHERE sal>13000; --Index Scan

Note:
- **When we create the INDEX, implicitly B-Tree will be created using some algorithm [program]**
- **B-Tree => Balanced tree**
- **Tree is a collection of nodes**

SELECT ename, sal FROM emp WHERE sal>4000;

In TABLE SCAN, WHERE condition will be applied on every row

SAL

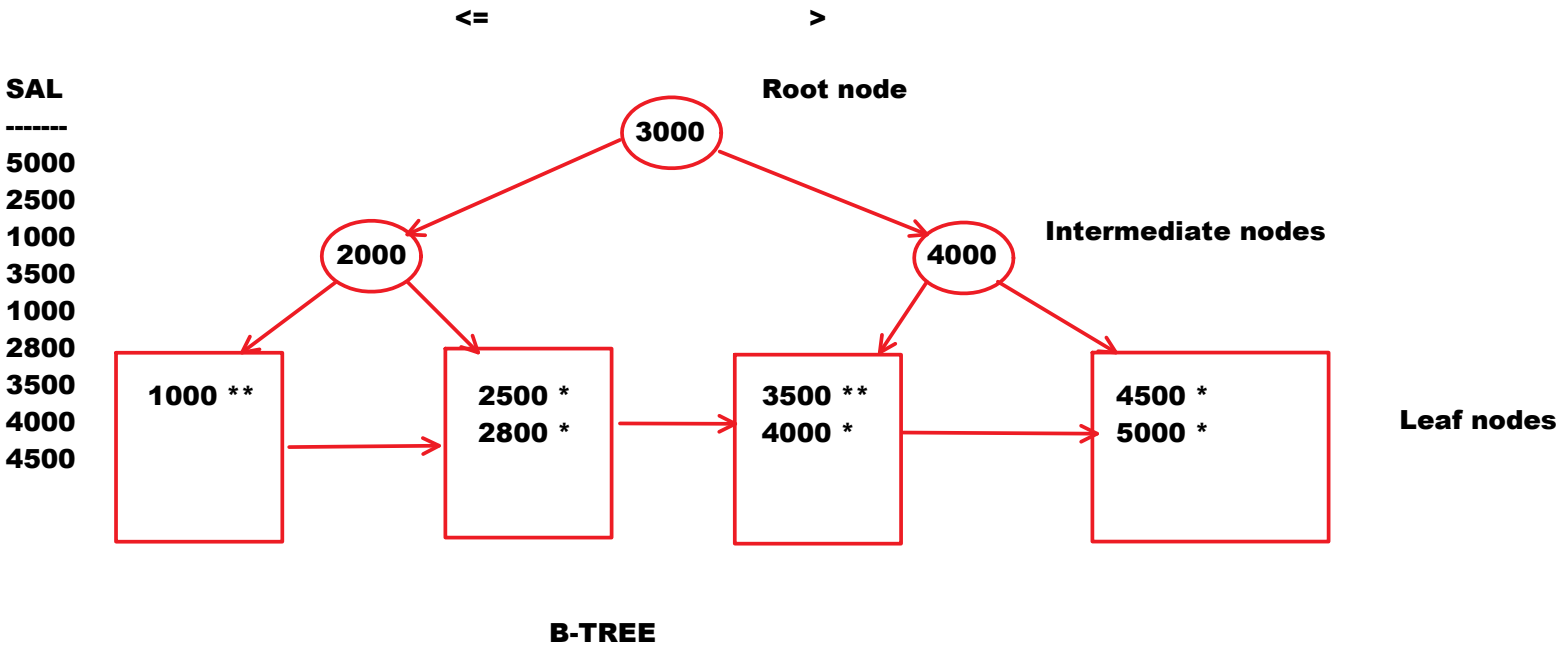
5000>4000 T
2500>4000 F
1000>4000 F
3500>4000 F
1000>4000 F
2800>4000 F
3500>4000 F

9 comparisons

In Table Scan,
no of comparisons = no of records

4000>4000 F
4500>4000 T

SELECT ename, sal FROM emp WHERE sal>4000;



B-TREE

INDEX SCAN reduces no of comparisons. So, performance will be improved.

Types of Indexes:

2 types:

- **B-Tree Index / Normal Index**
  - **Simple Index**
  - **Composite Index**
  - **Function-based index**
  - **Unique index**
- **Bitmap Index**

Simple index:

- **If index is created on 1 column then it is called "Simple index".**

Example:

```
SELECT ename, sal FROM emp WHERE sal>13000;   --Table Scan

CREATE INDEX i2 ON emp(sal);

SELECT ename, sal FROM emp WHERE sal>13000;   --Index Scan
```

Composite Index:

- **If INDEX is created on multiple columns then it is called "Composite Index".**

- **We can create COMPOSITE INDEX on max of 32 columns.**

Example:

```
SELECT ename, deptno, job, sal
FROM emp
WHERE deptno=30 AND job='SALESMAN';     --Table Scan

CREATE INDEX i3 ON emp(deptno,job);

SELECT ename, deptno, job, sal
FROM emp
WHERE deptno=30 AND job='SALESMAN';      --Index Scan


SELECT ename, deptno, job, sal
FROM emp
WHERE deptno=30;                                  --Index Scan


SELECT ename, deptno, job, sal
FROM emp
WHERE job='CLERK';                                --Index Scan
```

Function-Based Index:

- **if index is created based on function or expression then it is called "Function-based Index".**

Example:

```
SELECT * FROM emp WHERE ename='BLAKE';   --Table Scan

CREATE INDEX i4 ON emp(ename);    --Simple index

SELECT * FROM emp WHERE ename='BLAKE';   --Index Scan


SELECT * FROM emp WHERE lower(ename)='blake';  --Table Scan

CREATE INDEX i5 On emp(lower(ename));   --function based index
```

```
SELECT * FROM emp WHERE lower(ename)='blake';  --Index Scan


SELECT ename, sal FROM emp WHERE sal*12>80000;   --table Scan

CREATE INDEX i6 ON emp(sal*12);     ----function based index

SELECT ename, sal FROM emp WHERE sal*12>80000;   --Index Scan
```

user_indexes:

- **it maintains all indexes information**

```
SELECT index_name, index_type
FROM user_indexes;
```

Unique Index:

- **UNIQUE INDEX will be created on the column which has unique values.**

Syntax:

```
CREATE UNIQUE INDEX <name>
ON <table_name>(<column>);
```

Example:

```
SELECT * FROM dept WHERE dname='SALES';  -- Table Scan

CREATE UNIQUE INDEX i7 ON dept(dname);     --unique index

SELECT * FROM dept WHERE dname='SALES';  -- Index Scan

INSERT INTO dept VALUES(50,'SALES','HYD');
--ERROR: unique index created on dname column.
--so, it does not accept duplicates
```

Note:
When we create a table with PK,
on PK column unique index will be created implicitly.

When we create a table with UNIQUE constraint,
on UNIQUE column unique index will be created implicitly.

```
CREATE TABLE t100
(
f1 NUMBER(4) CONSTRAINT c100 PRIMARY KEY,
f2 VARCHAR2(10) CONSTRAINT c101 UNIQUE,
f3 date
);
```

When above table is created,
implicitly 2 unique indexes will be created with constraint names as index names.

Note:

B-tree Index:

- when we create the index if B-tree is created then it is called "B-Tree Index".
- Leaf node contains values and row ids.

Bitmap Index:

- Bitmap Index contains bits [0s and 1s].
- These bits are associated with row ids.
- These bits will be converted to row ids and selects the records using row ids.
- It will be created on low cardinality columns.

Low cardinality column:

A column which has less distinct values is called "Low Cardinality Column".

Examples:

GENDER
M
F
F
M
M
M
F
F

M
F

DEPTNO
10
20
20
30
30
30
10
10
20
30

10
20
30

Note:
On low cardinality columns create BITMAP INDEX.
On high cardinality columns create B-TREE INDEX.

Syntax to create Bitmap Index:

```
CREATE BITMAP INDEX <name>
ON <table_name>(<column>);
```

Example:

STUDENT

SID	SNAME	GENDER

```
create table student
(
sid number(4),
sname varchar2(10),
gender char
);

insert into student values(1,'A','M');
insert into student values(2,'B','M');
insert into student values(3,'C','F');
insert into student values(4,'D','F');
insert into student values(5,'E','M');
insert into student values(6,'F','F');
commit;

SELECT * FROM student WHERE gender='F';      --Table Scan

CREATE BITMAP INDEX bi1 ON student(gender);

SELECT * FROM student WHERE gender='F';      --Index Scan
```

STUDENT

SID	SNAME	GENDER
1	A	M
2	B	M
3	C	F
4	D	F
5	E	M
6	F	F

row id

row id

row id

BI1

M	F
1	0
1	0
0	1
0	1
1	0
0	1

0=1 FALSE
0=1 FALSE
1=1 TRUE
1=1 TRUE
0=1 FALSE
1=1 TRUE

gender='F' table scan
M=F

F=1 Bitmap Index scan
0=1

gender='F' table scan
M=F
value comparison

F=1 Bitmap Index scan
0=1
Bit comparison

No of comparisons in TABLE SCAN and in BITMAP INDEX SCAN are same. then how Bitmap Index improves the performance?

Bit comparison is faster than value comparison.

Differences b/w B-Tree Index and Bitmap Index:

B-Tree Index	Bitmap Index
• in this, B-Tree will be created.	• in this, B-Tree will not be created.
• it contains values and row ids.	• it contains bits [0s and 1s]
• it will be created on high cardinality columns. Examples: empno, ename	• it will be created on low cardinality columns. Examples: gender, deptno, job

Dropping Indexes:

Syntax:
DROP INDEX <index_name>;

Example:
DROP INDEX i2;

When we drop the table does it drop the Indexes?
YES.

When we drop the base table does it drop the VIEWS?
NO.
These views will not work until base table is created.

When we drop the table does it drop the triggers?
YES

Table
- **Rows and Columns**
- **Constraints**
- **Indexes**
- **Triggers**

When we drop the table,
all rows and columns will be dropped
constraints will be dropped
Indexes will be dropped
Triggers will be dropped

Materialized Views

Tuesday, January 23, 2024 2:20 PM

VIEW Disadvantage:
Less performance

PERSON

PID	PNAME	STATE	AADHAR

Finding state wise population:

```
CREATE VIEW v20
AS
SELECT state, count(*) as no_of_people
FROM person
GROUP BY state;

SELECT * FROM v20;  --calculates

SELECT * FROM v20;  --calculates

SELECT * FROM v20;  --calculates
```

V20

```
SELECT state, count(*) as
no_of_people
FROM person
GROUP BY state
```

```
CREATE MATERIALIZED VIEW mv20
AS
SELECT state, count(*)
FROM person
GROUP BY state;

SELECT * FROM mv20;   --retrieves
SELECT * FROM mv20;   --retrieves
SELECT * FROM mv20;   --retrieves
```

mv20

state	count(*)
TS	..
AP	..
MH	..

Materialized Views:

- **M.View is a Db Object.**
- **M.View is not a virtual table.**
- **It contains physical data. It occupies memory.**
- **It holds result of SELECT query.**
- **It holds precomputed result.**
- **To maintain summarized tables physically we use M.VIEW. It is mainly used in DataWare Housing.**
- **It improves the performance.**
- **Using it, we can maintain physical copy of remote database.**

Syntax:

```
CREATE MATERIALIZED VIEW <name>
AS
<SELECT QUERY>;
```

granting permission to create M.VIEW:

login as DBA:

username: system
password: nareshit

```
GRANT create materialized view
TO c##batch2pm;
```

Login as user: c##batch2pm

VIEW	M.VIEW

```
CREATE VIEW v20
AS
SELECT deptno, sum(sal) as sum_of_sal
FROM emp
GROUP BY deptno;
```

```
CREATE MATERIALIZED VIEW mv1
AS
SELECT deptno, sum(sal)
FROM emp
GROUP BY deptno;
```

v20

```
SELECT deptno, sum(sal) as sum_of_sal
FROM emp
GROUP BY deptno
```

mv1

deptno	sum(sal)
10	..
20	..
30	..

```
SELECT * FROM v20; --calculates

SELECT * FROM v20; --calculates

SELECT * FROM v20; --calculates
```

```
SELECT * FROM mv1; --retrieves

SELECT * FROM mv1; --retrieves

SELECT * FROM mv1; --retrieves
```

Note:
In above example, VIEW calculates dept wise sum of salaries every time. But, M.VIEW will not calculate every time. It holds precomputed result. So, It improves the performance.

Note:
VIEW always gives recent data.
M.VIEW does not give recent data.
That is why we need to refresh materialized view.

Refreshing M.VIEW:

- **Applying Base table changes to M.VIEW is called "Refreshing".**
- **M.VIEW can be refreshed in 3 ways:**
  - **ON DEMAND [default]**
  - **ON COMMIT**
  - **ON regular interval of time**

- **ON DEMAND [default]:**

- **We call refresh procedure to refresh the M.VIEW.**
- **refresh procedure defined in package "DBMS_MVIEW"**

Syntax:

```
EXEC dbms_mview.refresh(<m.view_name>);
```

Example:

```
CREATE MATERIALIZED VIEW mv2
REFRESH ON DEMAND
AS
SELECT deptno, sum(Sal)
FROM emp
GROUP BY deptno;

SELECT * FROM mv2;
```

Output:

deptno	sum(sal)
20	15875

```
UPDATE emp SET sal=sal+1000;

COMMIT;

SELECT * FROM mv2;
```

Output:

deptno	sum(sal)
20	15875

Note:
20 dept has 5 emps

```
EXEC dbms_mview.refresh('mv2');

SELECT * FROM mv2;
```

Output:

deptno	sum(sal)
20	20875

ON COMMIT:

- **When COMMIT command is executed implicitly M.VIEW will be refreshed.**

Example:

```
CREATE MATERIALIZED VIEW mv3
REFRESH ON COMMIT
AS
SELECT deptno, sum(Sal)
FROM emp
GROUP BY deptno;

SELECT * FROM mv3;
```

Output:

deptno	sum(sal)
20	20875

```
UPDATE emp SET sal=sal+1000;

SELECT * FROM mv3;
```

Output:

deptno	sum(sal)

20	20875

COMMIT; --m.view will be refreshed

SELECT * FROM mv3;

Output:

deptno	sum(sal)
20	25875

ON regular interval of time:

weekly sales, monthly sales, yearly sales

In this way, materialized view will be refreshed in a regular interval of time.
Example:
every 24 Hrs
every 1 week
every 1 month

Example:

```
CREATE MATERIALIZED VIEW mv4
REFRESH                                    hour
START WITH sysdate                         minute
NEXT sysdate+Interval '2' minute           day
AS                                         month
SELECT deptno, sum(sal)                    year
FROM emp
GROUP BY deptno;
```

SELECT * FROM mv4;

Output:

deptno	sum(sal)

20	25875

```
UPDATE emp SET sal=sal+1000;
COMMIT;

SELECT * FROM mv4;
```

Output:

deptno	sum(sal)
20	25875

After 2 minutes:

```
SELECT * FROM mv4;
```

Output:

deptno	sum(sal)
20	30875

user_mviews:

- it is a system table.
- it maintains all m.views information

to see m.view list:

```
SELECT MVIEW_NAME, QUERY
FROM USER_MVIEWS;
```

Dropping m.view:

Syntax:

```
DROP MATERIALIZED VIEW <name>;
```

Example:

```
DROP MATERIALIZED VIEW mv1;
```

SYNONYMS

Tuesday, January 23, 2024 3:28 PM

SYNONYMS:

- **SYNONYM is a DB Object.**
- **It is used to give permanent alias name to DB Objects like tables.**
- **Table alias is temporary. SYNONYM is permanent.**

Advantage:

- **SYNONYM is used to make lengthy name short.**

Example:

HYD_BRANCH_EMPLOYEE_DETAILS e
Table name Synonym

```
SELECT * FROM e;

UPDATE e SET sal=sal+1000;
```

Syntax:

```
CREATE SYNONYM <name> FOR <DB_Object_Name>;
```

Granting permission to create synonym:

login as DBA:

GRANT create synonym TO c##batch2pm;

Example:
login as c##batch2pm:

CREATE SYNONYM e FOR emp;

SELECT * FROM e;

UPDATE e SET sal=sal+1000;

Example:
c##batch2pm

HYD_BRANCH_EMPLOYEE_DETAILS

GRANT all ON HYD_BRANCH_EMPLOYEE_DETAILS
TO c##userA;

c##userA:

SELECT *
FROM c##batch2pm.HYD_BRANCH_EMPLOYEE_DETAILS;

CREATE SYNONYM h
FOR c##batch2pm.HYD_BRANCH_EMPLOYEE_DETAILS;

SELECT * FROM h;

Types of Synonyms:

2 Types:

- **Private Synonym => created by DEVELOPER**

- **Public Synonym => DBA**

Syntax to create public synonym:

```
CREATE PUBLIC SYNONYM <name>
FOR <db_object>;
```

Login as DBA:

```
CREATE PUBLIC SYNONYM z
FOR c##batch2pm.emp;

GRANT all ON z
TO c##userA, c##userB, c##userC;
```

```
c##userA:
select * from z;

c##userB:
select * from z;

c##userC:
select * from z;
```

user_synonyms:

- **it maintains all synonyms information**

```
SELECT synonym_name, table_name
FROM user_synonyms;
```

Dropping private synonym:

DROP SYNONYM e;

Dropping public synonym:

DROP PUBLIC SYNONYM z;