TASK 2 REPORT

1. PREREQUISITE TASK

Step 1: create new tablespace:

alter session set "_oracle_script"=true;

create tablespace tbs_lab datafile 'db_lab_001.dat' size 5M autoextend ON next 5M

MAXSIZE 100M;

Step 2: create new user:

create user OR_DEF identified by admin default tablespace tbs_lab;

Step 3: gtant privileges:

grant connect to OR_DEF;

grant resource to OR_DEF;

grant select on scott.dept to OR_DEF;

grant select on scott.emp to OR_DEF;

ALTER USER OR_DEF QUOTA 100M ON tbs_lab;

2. HEAP ORGANIZED TABLES

Step 1:show block size

SELECT name,value

FROM v$parameter

WHERE name = 'db_block_size';

```
NAME
---------------------------
VALUE
---------------------------
db_block_size
8192
```

Step 2: create table

create table t (

a int,

b varchar2(4000) default rpad('',4000,'*'),*

c varchar2(3000) default rpad('',3000,'*'))*

Step 3: insert and delete values

insert into t (a) values (1);

insert into t (a) values (2);

insert into t (a) values (3);

	A
1	1
2	2
3	3

commit; delete from t where a = 2 ;

commit;

	A
1	1
2	3

insert into t (a) values (4);

commit;

Step 4: show result

select a from t;

	A
1	1
2	4
3	3

We see an example of a Heap organized table.

For a record, it looks not for a sequence in a block, but for the first free block. So when writing the last value 4, it occupied the block, the freed block after deleting the value 2

TASK 2 – UNDERSTANDING HEAP TABLE SEGMENTS

Step 1: create table

create table t2 (x int primary key, y clob, z blob);

Step 2: show segments

select segment_name, segment_type from user_segments;

	SEGMEN...	SEGMEN...

Step 3: create table

Create table t (x int primary key, y clob, z blob)

SEGMENT CREATION IMMEDIATE

Step 4: show segments

select segment_name, segment_type from user_segments;

	SEGMENT_NAME	SEGMENT_TYPE
1	SYS_C0010676	INDEX
2	SYS_IL0000078323C00002$$	LOBINDEX
3	SYS_IL0000078323C00003$$	LOBINDEX
4	SYS_LOB0000078323C00002$$	LOBSEGMENT
5	SYS_LOB0000078323C00003$$	LOBSEGMENT
6	T	TABLE

Step 5: insert data to t2

insert into t2 (x) values (1);

commit;

Step 6: show segments

select segment_name, segment_type from user_segments;

	SEGMENT_NAME	SEGMENT_TYPE
1	SYS_C0010675	INDEX
2	SYS_C0010676	INDEX
3	SYS_IL0000078317C00002$$	LOBINDEX
4	SYS_IL0000078317C00003$$	LOBINDEX
5	SYS_IL0000078323C00002$$	LOBINDEX
6	SYS_IL0000078323C00003$$	LOBINDEX
7	SYS_LOB0000078317C00002$$	LOBSEGMENT
8	SYS_LOB0000078317C00003$$	LOBSEGMENT
9	SYS_LOB0000078323C00002$$	LOBSEGMENT
10	SYS_LOB0000078323C00003$$	LOBSEGMENT
11	T	TABLE
12	T2	TABLE

Step 5: show metadata

SELECT DBMS_METADATA.GET_DDL('TABLE','T') FROM dual

	DBMS_METADATA.GET_DDL('TABLE','T')
1	CREATE TABLE "OR_DEF"."T" ("X" NUMBER(*,0), "Y" CLOB, "Z" BLOB, PRIMARY KEY ("X") USING INDEX PCTFREE 10 INITRANS 2 MAXTRANS 255 ST...

Oracle has a feature - creating segments on demand. This means that when we define any segment (table, index, etc.), none of the associated segments are created until rows are inserted.

By default, the DEFERRED_SEGMENT_CREATION system value is enabled. This behavior is shown in the table t2 as an example. If we call the user_segments table, we can tell that none of the visible segments have been created. After inserting data into the table - segments are created.

By specifying SEGMENT CREATION IMMEDIATE as in table t, the segment will be created when the table is created.

3.INDEX ORGANIZED TABLES. COMPARE PERFORMANCE

Step 1: create table and index

CREATE TABLE emp

AS SELECT object_id empno , object_name ename , created hiredate , owner job

FROM all_objects

alter table emp add constraint emp_pk primary key(empno)

Calculate Statistic:

begin

dbms_stats.gather_table_stats(user, 'EMP', cascade=>true);

end;

Step 2: create table

CREATE TABLE heap_addresses (

empno REFERENCES emp(empno) ON DELETE CASCADE ,

addr_type VARCHAR2(10) ,

street VARCHAR2(20) ,

city VARCHAR2(20) ,

state VARCHAR2(2) ,

zip NUMBER ,

PRIMARY KEY (empno,addr_type))

Step 3: create table

CREATE TABLE iot_addresses (

empno REFERENCES emp(empno) ON DELETE CASCADE ,

addr_type VARCHAR2(10) ,

street VARCHAR2(20) ,

city VARCHAR2(20) ,

state VARCHAR2(2) ,

zip NUMBER, PRIMARY KEY (empno,addr_type))

ORGANIZATION INDEX

Step 4 : Initial inserts:

INSERT INTO heap_addresses

SELECT empno, 'WORK' , '123 main street' , 'Washington' , 'DC' , 20123 FROM emp; INSERT INTO iot_addresses SELECT empno , 'WORK' , '123 main street' , 'Washington' , 'DC' , 20123

FROM emp;

INSERT INTO heap_addresses SELECT empno, 'HOME' , '123 main street' , 'Washington' , 'DC' , 20123

FROM emp;

INSERT INTO iot_addresses SELECT empno, 'HOME' , '123 main street' , 'Washington' , 'DC' , 20123

FROM emp;

INSERT INTO heap_addresses SELECT empno, 'PREV' , '123 main street' , 'Washington' , 'DC' , 20123

FROM emp;

INSERT INTO iot_addresses SELECT empno, 'PREV' , '123 main street' , 'Washington' , 'DC' , 20123

FROM emp;

INSERT INTO heap_addresses SELECT empno, 'SCHOOL' , '123 main street' , 'Washington' , 'DC' , 20123

FROM emp;

INSERT INTO iot_addresses SELECT empno, 'SCHOOL' , '123 main street' , 'Washington' , 'DC' , 20123

FROM emp;

Commit;

Step 5: Calculate statistic:

exec dbms_stats.gather_table_stats($username$, 'HEAP_ADDRESSES');

exec dbms_stats.gather_table_stats($username$, 'IOT_ADDRESSES');

Step 6: Compare Trace and Performance:

Explain 1:

*SELECT **

FROM emp , heap_addresses

WHERE emp.empno = heap_addresses.empno

AND emp.empno = 42;

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			4	8
NESTED LOOPS			4	8
TABLE ACCESS	EMP	BY INDEX ROWID	1	2
INDEX	EMP_PK	UNIQUE SCAN	1	1
Access Predicates				
EMP.EMPNO=42				
TABLE ACCESS	HEAP_ADDRESSES	BY INDEX ROWID BATCHED	4	6
INDEX	SYS_C0010682	RANGE SCAN	4	2
Access Predicates				

Explain 2:

*SELECT **

FROM emp , iot_addresses

WHERE emp.empno = iot_addresses.empno

AND emp.empno = 42;

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			4	4
NESTED LOOPS			4	4
TABLE ACCESS	EMP	BY INDEX ROWID	1	2
INDEX	EMP_PK	UNIQUE SCAN	1	1
Access Predicates				
EMP.EMPNO=42				
INDEX	SYS_IOT_TOP_78336	RANGE SCAN	4	2
Access Predicates				
IOT_ADDRESSES.EMPNO=42				

The first query with no heap_addresses index has a COST of 8. The second query with an iot_addresses index has a COST of 4.

The only difference in the plans of these queries is that in the first case we use table access to read addresses (cost - 6), and in the second case we use an index to get information (cost - 2).

4. *** ROW MIGRATION

my block size:

SELECT name,value

FROM v$parameter

WHERE name = 'db_block_size';

	NAME	VALUE
1	db_block_size	8192

Create example table:

CREATE TABLE row_mig_chain_demo(

x int PRIMARY KEY,

a CHAR(2000),

b CHAR(2000),

c CHAR(2000),

d CHAR(2000),

e CHAR(2000)

);

Number of database chained rows:

SELECT a.name, b.value

FROM v$statname a, v$mystat b

WHERE a.statistic# = b.statistic#

AND lower(a.name) = 'table fetch continued row';

Script Output × | Autotrace × | Explain Plan × | Query Result × | Query Result 1 ×

SQL | All Rows Fetched: 1 in 0.003 seconds

	NAME	VALUE
1	table fetch continued row	220

Insert values and check again:

INSERT INTO row_mig_chain_demo (x) VALUES (1);

INSERT INTO row_mig_chain_demo (x) VALUES (2);

INSERT INTO row_mig_chain_demo (x) VALUES (3);

Script Output × | Autotrace × | Explain Plan × | Query Result × | Query Result 1 ×

SQL | All Rows Fetched: 1 in 0.003 seconds

	NAME	VALUE
1	table fetch continued row	220

The number hasn't changed. This data is so small right now, all three rows fit on a single block.

Update rows:

UPDATE row_mig_chain_demo SET a = 'z1', b = 'z2', c = 'z3' WHERE x = 3;

UPDATE row_mig_chain_demo SET a = 'y1', b = 'y2', c = 'y3' WHERE x = 2;

UPDATE row_mig_chain_demo SET a = 'w1', b = 'w2', c = 'w3' WHERE x = 1;

Script Output × | Autotrace × | Explain Plan × | Query Result × | Query Result 1 ×

SQL | All Rows Fetched: 1 in 0.003 seconds

	NAME	VALUE
1	table fetch continued row	220

SELECT * FROM row_mig_chain_demo WHERE x = 1;

Script Output × | Autotrace × | Explain Plan × | Query Result × | Query Result 1 ×

SQL | All Rows Fetched: 1 in 0.004 seconds

	NAME	VALUE
1	table fetch continued row	221

SELECT * FROM row_mig_chain_demo WHERE x = 2;

Script Output | Autotrace | Explain Plan | Query Result | Query Result 1

SQL | All Rows Fetched: 1 in 0.003 seconds

	NAME	VALUE
1	table fetch continued row	222

This means rows with id 2 and 1 migrated.

5. TYPES OF INDEXES

Test query:

*SELECT **

FROM bl_3nf.ce_products

WHERE unit_price >400;

Before creation of INDEX returns Explain plan:

SQL | 0.04 seconds

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			1	68
TABLE ACCESS	BL_3NF.CE_PRODUCTS	FULL	1	68
Filter Predicates				
PRODUCT='Metallic Eye Shadow'				
Other XML				
{info}				
info type="has_user_tab"				
yes				
info type="db_version"				
21.0.0.0				
info type="parse_schema"				
"SYSTEM"				
info type="plan_hash_full"				
3779575837				
info type="plan_hash"				
3136137573				

Create index:

CREATE INDEX idx_btree ON bl_3nf.ce_products (product);

SQL | 0.035 seconds

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			1	2
TABLE ACCESS	BL_3NF.CE_PRODUCTS	BY INDEX ROWID BATCHED	1	2
INDEX	IDX_BTREE	RANGE SCAN	1	1
Access Predicates				
PRODUCT='Metallic Eye Shadow'				
Other XML				
{info}				
info type="has_user_tab"				
yes				
info type="db_version"				
21.0.0.0				
info type="parse_schema"				
"SYSTEM"				
info type="plan_hash_full"				
4102950606				
info type="plan_hash"				

We should use b-tree index When columns are unique or near-unique.

Create index:

CREATE INDEX idx_desc ON bl_3nf.ce_products (product desc);

SQL | 0.034 seconds

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			1	34
TABLE ACCESS	BL_3NF.CE_PRODUCTS	BY INDEX ROWID BATCHED	1	34
INDEX	IDX_DESC	RANGE SCAN	43	1

```
Access Predicates
    SYS_OP_DESCEND(PRODUCT)=HEXTORAW('B29A8B9E9393969CDFBA869ADFAC979E9B9088FF')
Filter Predicates
    SYS_OP_UNDESCEND(SYS_OP_DESCEND(PRODUCT))='Metallic Eye Shadow'
Other XML
  {info}
    info type="has_user_tab"
      yes
    info type="db_version"
      21.0.0.0
    info type="parse_schema"
    "SYSTEM"
    info type="plan_hash_full"
```

We should use descending indexes in cases, when our B-tree can be unbalanced by an ever-increasing value for an index.

Create index:

CREATE INDEX idx_reverse ON bl_3nf.ce_products (product) REVERSE;

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			1	2
TABLE ACCESS	BL_3NF.CE_PRODUCTS	BY INDEX ROWID BATCHED	1	2
INDEX	IDX_REVERSE	RANGE SCAN	1	1

```
Access Predicates
    PRODUCT='Metallic Eye Shadow'
Other XML
  {info}
    info type="has_user_tab"
      yes
    info type="db_version"
      21.0.0.0
    info type="parse_schema"
    "SYSTEM"
    info type="plan_hash_full"
      1511718748
    info type="plan_hash"
```

We should use reverse index for monotonically increasing values (e.g. auto-increment identifier)

For bitmap index I used different query:

Test query:

Select gender from bl_3nf.ce_customers Where gender='M';

Before creation of INDEX returns Explain plan:

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			15398	309
TABLE ACCESS	BL_3NF.CE_CUSTOMERS	FULL	15398	309

```
Filter Predicates
    GENDER='M'
Other XML
  {info}
    info type="has_user_tab"
      yes
    info type="db_version"
      21.0.0.0
    info type="parse_schema"
    "SYSTEM"
    info type="plan_hash_full"
      3751562893
    info type="plan_hash"
      879858611
```

Create index:

CREATE BITMAP INDEX idx_bitmap ON bl_3nf.ce_customers (gender);

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			15398	1
BITMAP CONVERSION		TO ROWIDS	15398	1
BITMAP INDEX	IDX_BITMAP	SINGLE VALUE		
Access Predicates				
GENDER='M'				
Other XML				
{info}				
info type="has_user_tab"				
yes				
info type="db_version"				
21.0.0.0				
info type="parse_schema"				
"SYSTEM"				
info type="plan_hash_full"				
2766206081				
info type="plan_hash"				
3005354435				
info type="plan_hash_2"				
2766206081				
stats type="compilation"				
stat name="bg"				

We should use bitmap index for columns having low distinct values

Create index:

CREATE INDEX idx_funct_baced ON bl_3nf.ce_products (lower(product));

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			107	34
TABLE ACCESS	BL_3NF.CE_PRODUCTS	BY INDEX ROWID BATCHED	107	34
INDEX	IDX_FUNCT_BACED	RANGE SCAN	43	1
Access Predicates				
LOWER(PRODUCT)='metallic eye shadow'				
Other XML				
{info}				
info type="has_user_tab"				
yes				
info type="db_version"				
21.0.0.0				
info type="parse_schema"				
"SYSTEM"				
info type="plan_hash_full"				
1643142193				
info type="plan_hash"				
1904479728				
info type="plan_hash_2"				
1643142193				
stats type="compilation"				

We should use function based index If we have a query that consists of expression and use this query many times