

U1M3.LW.Database Types of Tables,

Indexes

Report

Mastykina Elizaweta

1.Prerequisites

Changed password:

```
ALTERUSER user_name IDENTIFIED BY new_password;
```

Granted the UNLIMITED TABLESPACE system privilege:

```
GRANTUNLIMITED TABLESPACE TO<username>;
```

Created Scott Schema:

<http://www.oracleprofessor.com/cdn/YouTube/Oracle/Database/21c/scott.sql>

2. Heap Organized Tables

2.1. Task 1 – Heap Understanding

```
CREATE TABLE t
(
  a INT,
  b VARCHAR2(4000) DEFAULT RPAD('*',4000,'*'),
  c VARCHAR2(3000) DEFAULT RPAD('*',3000,'*')
)
/
```

```
INSERT INTO t (a) VALUES (1);
```

```
INSERT INTO t (a) values (2);
```

```
INSERT INTO t (a) values (3);
```

```
COMMIT;
```

```
DELETE FROM t WHERE a = 2 ;
```

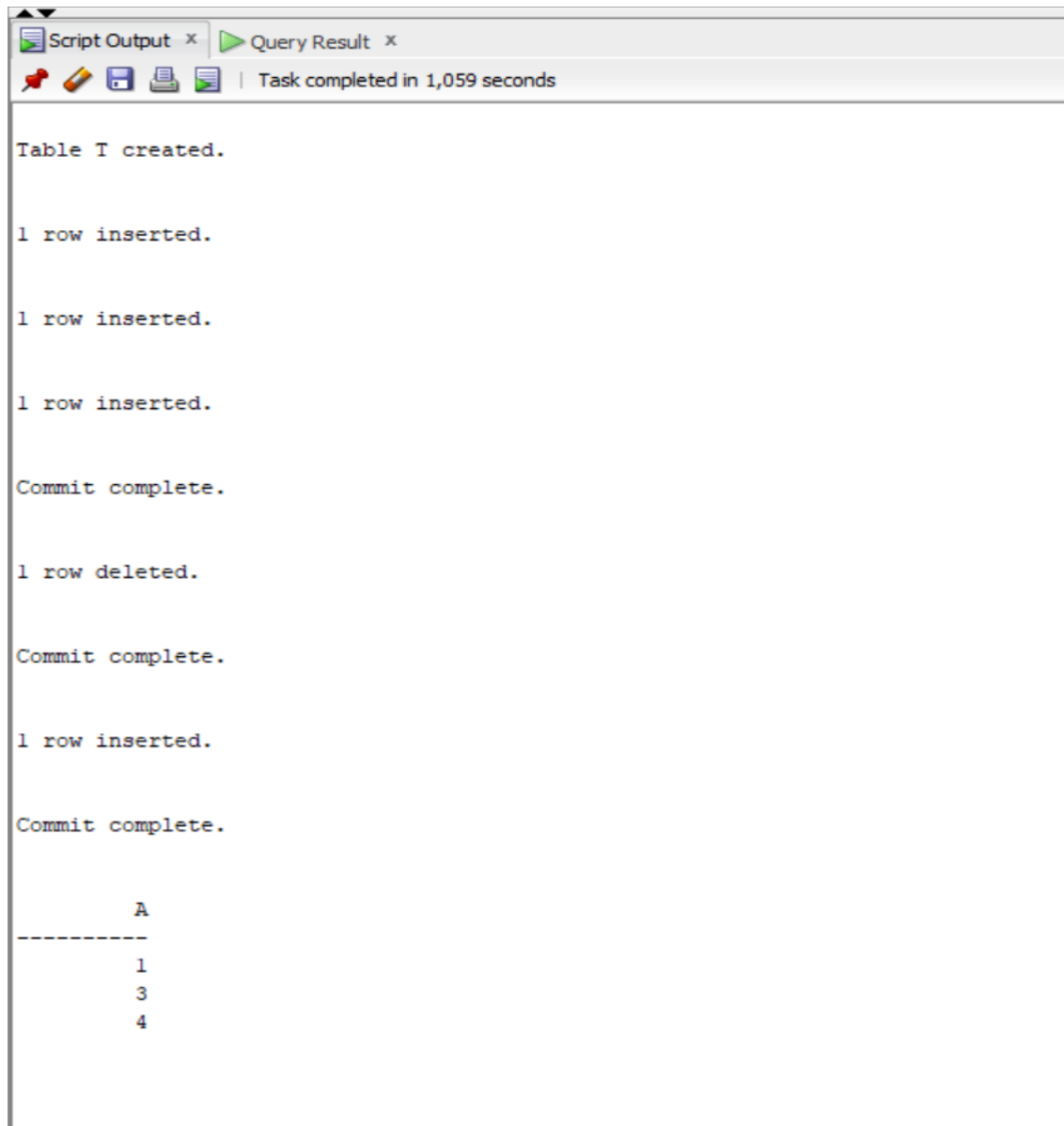
```
COMMIT;
```

```
INSERT INTO t (a) VALUES (4);
```

```
COMMIT;
```

```
SELECT a FROM t;
```

Screenshot of the data results below:



Conclusion:

This code creates table 't' with three columns. First column named 'a' and had type integer. Second column named 'b' and type varchar2 (4000) means that date in this column consists of symbols and its maximum size is 4000 bytes. Third column named 'c' consists of symbols and its maximum size is 4000

bytes. Then we insert values into the column 'a' and apply command delete to the line, specified by value of the column 'a'. We use commit statement to apply changes in database.

In this task we can explore how works blocks conception in Oracle and understand principles of Heap Organized Tables. Management of data in Heap Tables is very similar to management, used in heap: data will be placed where it fits best, rather than in any specific sort of order. This is a key concept to understand about database tables: in general, they are inherently unordered collections of data.

2.2. Task 2 – Understanding Low level of data abstraction: Heap Table Segments

Used USER_SEGMENTS that describe the storage allocated for the segments owned by the current user's objects:

```
SELECT segment_name, segment_type FROM user_segments;
```

As a result we get a lot of old items which are located in bin, so we need purged the recycle bin via command PURGE RECYCLEBIN

```
RECYCLEBIN purged.
```

```
no rows selected
```

Creation of the table and checking segments:

```
CREATE TABLE t ( x INT PRIMARY KEY, y CLOB, z BLOB);
```

```
SELECT segment_name, segment_type FROM user_segments;
```

```
Table T created.
```

```
no rows selected
```

The fact is at the moment we have no segments, because since the release of Oracle 11g segment creation is available only after the first insertion.

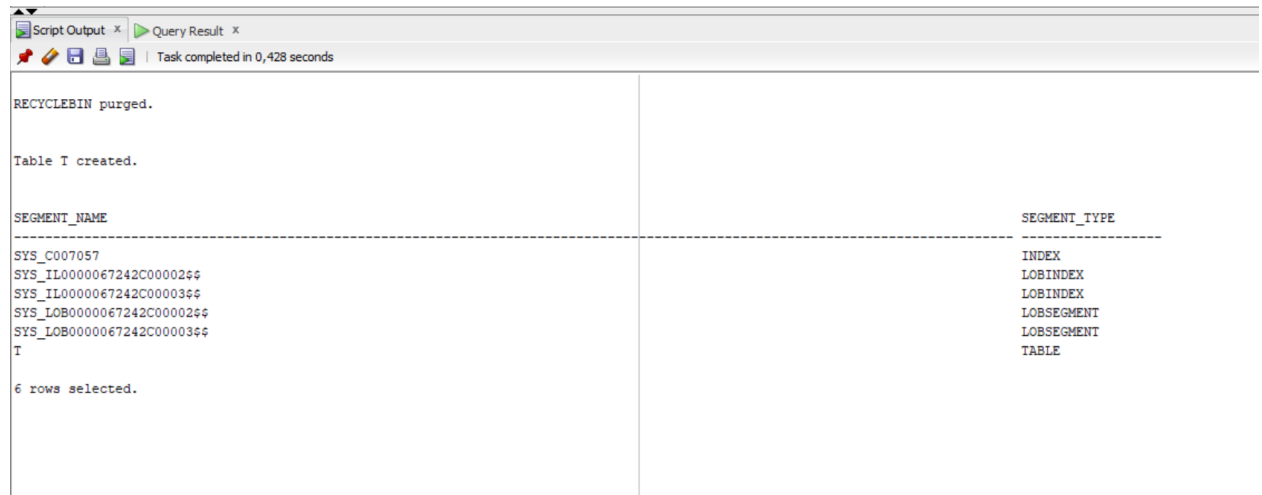
Creating table t SEGMENT CREATION IMMEDIATE clause and checking user_segments again:

```
CREATE TABLE t ( x INT PRIMARY KEY, y CLOB, z BLOB)

SEGMENT CREATION IMMEDIATE;

SELECT segment_name, segment_type FROM user_segments;
```

Result:



The screenshot shows a database query result window with two tabs: 'Script Output' and 'Query Result'. The 'Query Result' tab is active, displaying the output of the SELECT statement. The output is a table with two columns: 'SEGMENT_NAME' and 'SEGMENT_TYPE'. The table contains six rows of data. The first row is 'SYS_C007057' with segment type 'INDEX'. The next two rows are 'SYS_IL0000067242C00002\$\$' and 'SYS_IL0000067242C00003\$\$', both with segment type 'LOBINDEX'. The next two rows are 'SYS_LOB0000067242C00002\$\$' and 'SYS_LOB0000067242C00003\$\$', both with segment type 'LOBSEGMENT'. The final row is 'T' with segment type 'TABLE'. The window also shows '6 rows selected.' and 'Task completed in 0,428 seconds'.

SEGMENT_NAME	SEGMENT_TYPE
SYS_C007057	INDEX
SYS_IL0000067242C00002\$\$	LOBINDEX
SYS_IL0000067242C00003\$\$	LOBINDEX
SYS_LOB0000067242C00002\$\$	LOBSEGMENT
SYS_LOB0000067242C00003\$\$	LOBSEGMENT
T	TABLE

The table itself created a segment and the primary key constraint created an index segment. Additionally, each of the LOB columns created two segments: one segment to store the actual chunks of data pointed to by the character large object (CLOB) or binary large object (BLOB) pointer, and one segment to organize them. LOBs provide support for very large chunks of information, up to many gigabytes in size. They are stored in chunks in the lob segment, and the lob index is used to keep track of where the LOB chunks are and the order in which they should be accessed.

Checking options available in the CREATE TABLE statement for a given table using the standard supplied package DBMS_METADATA:

```
DBMS_METADATA.GET_DDL('TABLE','T')
```

```
CREATE TABLE "EMASTYKINA"."T"  
(  
  "X" NUMBER(*,0),  
  "Y" CLOB,  
  "Z" BLOB,  

```

Conclusion: Using the standard supplied package DBMS_METADATA, we can query the definition of it and see the verbose syntax. It shows many of the options for the CREATE TABLE statement. To allocated segments forcibly before the table creation exists command SEGMENT CREATION IMMEDIATE.

3. Index Organized Tables

Task 3: Compare performance of using IOT tables

Creating table EMP:

```
CREATE TABLE emp AS  
SELECT  
  object_id empno,  
  object_name ename,  
  created hiredate,  
  owner job  
FROM  
  all_objects  
/
```

Creating index:

```
ALTER TABLE emp ADD CONSTRAINT emp_pk PRIMARY KEY(empno);  
  
BEGIN  
  dbms_stats.gather_table_stats( user, 'EMP', cascade=>true );  
END;  
/
```

Implementing the child table (heap 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)
)
/

```

Implementation of the child table as an IOT:

```

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
/

```

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;

```

Calculation statistics:

```

EXEC dbms_stats.gather_table_stats( user, 'HEAP_ADDRESSES' );
EXEC dbms_stats.gather_table_stats( user, 'IOT_ADDRESSES' );
EXPLAIN PLAN FOR
SELECT * FROM emp, heap_addresses
WHERE emp.empno = heap_addresses.empno
AND emp.empno = 42;
SELECT * FROM TABLE(dbms_xplan.display );

```


PLAN_TABLE_OUTPUT									

Plan hash value: 2197251318									

Id	Operation	Name	Rows	Bytes	Cost	(%CPU)	Time		

0	SELECT STATEMENT		4	400	7	(0)	00:00:01		
1	NESTED LOOPS		4	400	7	(0)	00:00:01		
2	TABLE ACCESS BY INDEX ROWID	EMP	1	54	2	(0)	00:00:01		
* 3	INDEX UNIQUE SCAN	EMP_PK	1		1	(0)	00:00:01		
4	TABLE ACCESS BY INDEX ROWID BATCHED	HEAP_ADDRESSES	4	184	5	(0)	00:00:01		
* 5	INDEX RANGE SCAN	SYS_C007063	4		1	(0)	00:00:01		

PLAN_TABLE_OUTPUT									

Predicate Information (identified by operation id):									

3 - access("EMP"."EMPNO">=42)									
5 - access("HEAP_ADDRESSES"."EMPNO">=42)									
18 rows selected.									

```

EXPLAIN PLAN FOR
SELECT * FROM emp, iot_addresses
WHERE emp.empno = iot_addresses.empno
AND emp.empno = 42;
SELECT * FROM TABLE(dbms_xplan.display );

```

PLAN_TABLE_OUTPUT									

Plan hash value: 2457879703									

Id	Operation	Name	Rows	Bytes	Cost	(%CPU)	Time		

0	SELECT STATEMENT		4	400	3	(0)	00:00:01		
1	NESTED LOOPS		4	400	3	(0)	00:00:01		
2	TABLE ACCESS BY INDEX ROWID	EMP	1	54	2	(0)	00:00:01		
* 3	INDEX UNIQUE SCAN	EMP_PK	1		1	(0)	00:00:01		
* 4	INDEX RANGE SCAN	SYS_IOT_TOP_67259	4	184	1	(0)	00:00:01		

PLAN_TABLE_OUTPUT									

Predicate Information (identified by operation id):									

3 - access("EMP"."EMPNO">=42)									
4 - access("IOT_ADDRESSES"."EMPNO">=42)									
17 rows selected.									

Conclusion: Index organized tables (IOTs) are quite simply tables stored in an index structure. Whereas a table stored in a heap is unorganized (i.e., data goes wherever there is available space), data in an IOT is stored and sorted by primary key. The fact is that an index is a complex data structure that requires a lot of work to manage and maintain, and the maintenance requirements increase as the width of the row to store increases. A heap, on the other hand, is trivial to manage by comparison. There are efficiencies in a heap organized table over an IOT. That said, IOTs have some definite advantages over their heap counterparts.

When you want to enforce co-location of data or you want data to be physically stored in a specific order, the IOT is the structure for you.

A heap table would tend to place the data at the end of the table: as the data arrives, the heap table would simply add it to the end, due to the fact that the data is just arriving and no data is being deleted. Over time, if addresses are deleted, the inserts would become more random throughout the table. Suffice it to say, the chance an employee's work address would be on the same block as his home address in the heap table is near zero. For the IOT, however, since the key is on EMPNO, ADDR_TYPE, we'll be pretty sure that all of the addresses for a given EMPNO are located on one or maybe two index blocks together.

So we can see four fewer I/Os. We skipped four TABLE ACCESS (BY INDEX ROWID) steps. The more child records we have, the more I/Os we would anticipate skipping. So, what is four I/Os? Well, in this case it was over one-third of the I/O performed for the query, and if we execute this query repeatedly, it would add up. Each I/O and each consistent get requires an access to the buffer cache, and while it is true that reading data out of the buffer cache is faster than disk, it is also true that the buffer cache gets are not free and not

totally cheap. Each will require many latches of the buffer cache, and latches are serialization devices that will inhibit our ability to scale.

4. Index Clustered Tables

Task 4: Analyses Cluster Storage by Blocks

Cluster creation:

```
CREATE cluster emp_dept_cluster( deptno  NUMBER( 2 ) )
      SIZE                                     1024
      STORAGE( INITIAL 100K NEXT 50K );
```

Index creation:

```
CREATE INDEX idxcl_emp_dept on cluster emp_dept_cluster;
```

Table dept and emp ceration:

```
CREATE TABLE dept
(
    deptno NUMBER( 2 ) PRIMARY KEY
,  dname  VARCHAR2( 14 )
,  loc    VARCHAR2( 13 )
)
cluster emp_dept_cluster ( deptno ) ;

CREATE TABLE emp
(
    empno NUMBER PRIMARY KEY
,  ename  VARCHAR2( 10 )
,  job    VARCHAR2( 9 )
,  mgr    NUMBER
,  hiredate DATE
,  sal    NUMBER
,  comm   NUMBER
,  deptno NUMBER( 2 ) REFERENCES dept( deptno )
)
cluster emp_dept_cluster ( deptno ) ;
```

Loading data from scott:

```
INSERT INTO dept( deptno , dname , loc)
SELECT deptno , dname , loc
FROM scott.dept;
```

```
commit;
```

```
INSERT INTO emp ( empno, ename, job, mgr, hiredate, sal, comm,
deptno )
SELECT rownum, ename, job, mgr, hiredate, sal, comm, deptno
FROM scott.emp
```

```
commit;
```

Checked the row ids of each table and compare the block numbers after joining by DEPTNO:

```
SELECT *
FROM
(
    SELECT dept_blk, emp_blk, CASE WHEN dept_blk <> emp_blk
THEN '*' END flag, deptno
FROM
(
    SELECT dbms_rowid.rowid_block_number( dept.rowid )
dept_blk, dbms_rowid.rowid_block_number( emp.rowid ) emp_blk,
dept.deptno
FROM emp , dept
WHERE emp.deptno = dept.deptno
)
)
ORDER BY deptno
```

Result:

DEPT_BLK	EMP_BLK F	DEPTNO
35	35	20
35	35	20
35	35	20
35	35	20
35	35	20
35	35	30
35	35	30
35	35	30
35	35	30
35	35	30
35	35	30

11 rows selected.

Conclusion: Clustered tables give you the ability to physically prejoin data together. You use clusters to store related data from many tables on the same database block. Clusters can help read-intensive operations that always join data together or access related sets of data (e.g., everyone in department 10). Clustered tables reduce the number of blocks that Oracle must cache. Instead of keeping ten blocks for ten employees in the same department, Oracle will put them in one block and therefore increase the efficiency of your buffer cache. On the downside, unless you can calculate your SIZE parameter setting correctly, clusters may be inefficient with their space utilization and can tend to slow down DML-heavy operations.

5. Hash Clustered Tables

5.1. Task 5: Analyses Cluster Storage by Blocks

Created cluster:

```
CREATE CLUSTER hash_cluster ( deptno NUMBER(2))

  HASHKEYS 1000

  SIZE 1024

  STORAGE( INITIAL 100K NEXT 50K );
```

Created table dept:

```

CREATE TABLE dept(
    deptno NUMBER(2) PRIMARY KEY,
    dname VARCHAR2( 14 ),
    loc VARCHAR2( 13 )
) CLUSTER hash_cluster ( deptno ) ;

```

```

CREATE TABLE emp (
    empno NUMBER PRIMARY KEY,
    ename VARCHAR2( 10 ),
    job VARCHAR2( 9 ),
    mgr NUMBER,
    hiredate DATE,
    sal NUMBER,
    comm NUMBER,
    deptno NUMBER( 2 ) REFERENCES dept( deptno )
) CLUSTER hash_cluster ( deptno ) ;

```

```

INSERT INTO dept( deptno , dname , loc)
SELECT deptno , dname , loc
FROM scott.dept;
COMMIT;

```

```

INSERT INTO emp ( empno, ename, job, mgr, hiredate, sal, comm,
deptno )
SELECT rownum, ename, job, mgr, hiredate, sal, comm, deptno
FROM scott.emp ;
COMMIT;

```

```

SELECT * FROM

  (SELECT dept_blk, emp_blk, CASE WHEN dept_blk <> emp_blk THEN '*'
END flag, deptno

  FROM    (SELECT    dbms_rowid.rowid_block_number(    dept.rowid    )
dept_blk,

dbms_rowid.rowid_block_number( emp.rowid ) emp_blk, dept.deptno

  FROM emp , dept

  WHERE emp.deptno = dept.deptno)

)

ORDER BY deptno;

```

	DEPT_BLK	EMP_BLK	FLAG	DEPTNO
1	1192	1192	(null)	20
2	1192	1192	(null)	20
3	1192	1192	(null)	20
4	1192	1192	(null)	20
5	1192	1192	(null)	20
6	1354	1354	(null)	30
7	1354	1354	(null)	30
8	1354	1354	(null)	30
9	1354	1354	(null)	30
10	1354	1354	(null)	30
11	1354	1354	(null)	30

Conclusion: Hash clusters are similar in concept to index clusters, except a cluster index is not used. The data is the index itself. The cluster key is hashed into a block address and the data is expected to be there. If the block numbers are the same, the EMP row and the DEPT row are stored on the same physical database block together. Data from one department is stored in one block, data from another department is stored in another block. By the value of the hash function, it will be easy and quick to find the data of a specific department.

