

U1M3.LW (Database Types of Tables, Indexes) Report

Viktoria Masyuk <https://github.com/vilo4kaaa/DMDDataCamp22>

1. Prerequisites

Changed password:

```
ALTER USER user_name IDENTIFIED BY new_password;
```

Granted the UNLIMITED TABLESPACE system privilege:

```
GRANT UNLIMITED 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

Checked block size:

```
SELECT DISTINCT BYTES/BLOCKS FROM user_segments;
```

Created table t and inserted three values:

```
CREATE TABLE t
( a INT,
  b VARCHAR2(4000) DEFAULT RPAD(' ', 4000, ' '),
  c VARCHAR2(4000) DEFAULT RPAD(' ', 4000, ' '),
  d VARCHAR2(4000) DEFAULT RPAD(' ', 4000, ' ')
);
```

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

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

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

```
COMMIT;
```

Deleted one value and then inserted a new one:

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

```
COMMIT;
```

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

```
COMMIT;
```

Selected a column from the table t:

```
SELECT a FROM T;
```

Query Result	
SQL	
A	
1	1
2	4
3	3

Script Output	
Task completed in 0.309 seconds	
BYTES/BLOCKS	
32768	
Table T created.	
1 row inserted.	
1 row inserted.	
1 row inserted.	
Commit complete.	
1 row deleted.	
Commit complete.	
1 row inserted.	
Commit complete.	

```
SELECT dbms_rowid.rowid_block_number( t.rowid ) t From t;
```

T	BLOCK_NUMBER
1	1103
2	1103
3	1104

As we can see, the new row fit into the first free block.

Dropped table t:

```
DROP TABLE t;
```

A full scan of the table will be retrieved as it hits it, not in the order of insertion.

It is worth noting that we could get a completely different result because a heap organized tables is a big unordered collection of rows which may come out in a different order with the same query.

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;
```

SEGMENT_NAME	SEGMENT_TYPE
1 BIN\$44SeSIx/0czgVbKcyTjds==\$0	INDEX
2 BIN\$44SeSIx50czgVbKcyTjds==\$0	INDEX
3 BIN\$44SeSIx60czgVbKcyTjds==\$0	TABLE
4 BIN\$44SeSIxo0czgVbKcyTjds==\$0	INDEX
5 BIN\$44SeSIxp0czgVbKcyTjds==\$0	TABLE
6 BIN\$44SeSIxv0czgVbKcyTjds==\$0	INDEX
7 BIN\$44SeSIxw0czgVbKcyTjds==\$0	TABLE
8 BIN\$44SeSIxy0czgVbKcyTjds==\$0	INDEX
9 BIN\$44SeSIyA0czgVbKcyTjds==\$0	TABLE
10 BIN\$44SeSIyC0czgVbKcyTjds==\$0	INDEX

There are a lot of old items which are located in bin, so I purged the recycle bin:

```
PURGE RECYCLEBIN;
```

Created table t and checked user_segments:

```
CREATE TABLE t ( x INT PRIMARY KEY, y CLOB, z BLOB);
SELECT segment_name, segment_type FROM user_segments;
```

SEGMENT_NAME	SEGMENT_TYPE
--------------	--------------

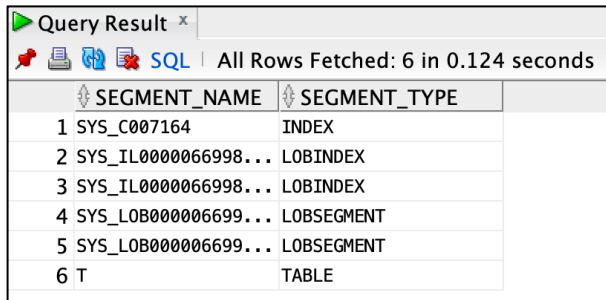
There are no segments. After the release of Oracle 11g segment creation is deferred until the first row is inserted.

Dropped table t:

```
DROP TABLE T;
```

Created table t SEGMENT CREATION IMMEDIATE clause and checked 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;
```



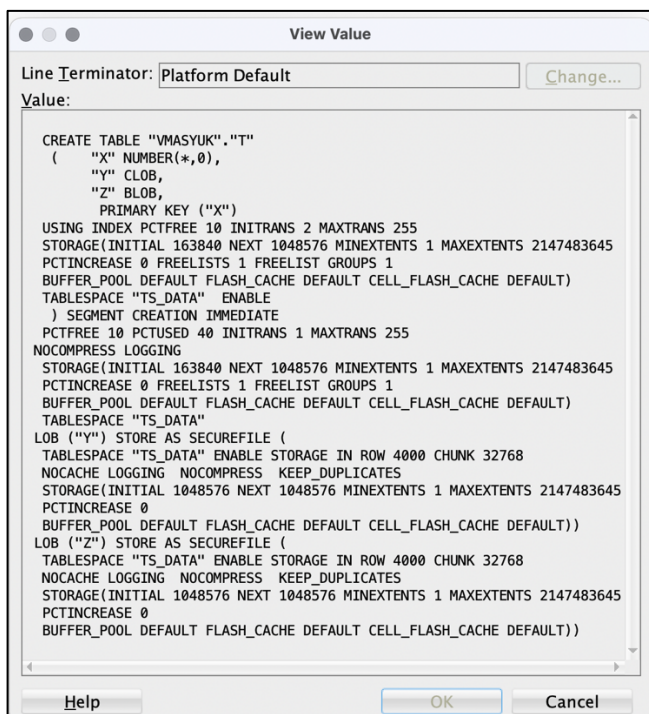
	SEGMENT_NAME	SEGMENT_TYPE
1	SYS_C007164	INDEX
2	SYS_IL0000066998...	LOBINDEX
3	SYS_IL0000066998...	LOBINDEX
4	SYS_LOB000006699...	LOBSEGMENT
5	SYS_LOB000006699...	LOBSEGMENT
6	T	TABLE

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 lobsegment, and the lobindex is used to keep track of where the LOB chunks are and the order in which they should be accessed.

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

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



```
CREATE TABLE "VMSYUK"."T"
(
  "X" NUMBER(*,0),
  "Y" CLOB,
  "Z" BLOB,
  PRIMARY KEY ("X")
  USING INDEX PCTFREE 10 INITRANS 2 MAXTRANS 255
  STORAGE(INITIAL 163840 NEXT 1048576 MINEXTENTS 1 MAXEXTENTS 2147483645
  PCTINCREASE 0 FREELISTS 1 FREELIST GROUPS 1
  BUFFER_POOL DEFAULT FLASH_CACHE DEFAULT CELL_FLASH_CACHE DEFAULT)
  TABLESPACE "TS_DATA" ENABLE
) SEGMENT CREATION IMMEDIATE
PCTFREE 10 PCTUSED 40 INITRANS 1 MAXTRANS 255
NOCOMPRESS LOGGING
STORAGE(INITIAL 163840 NEXT 1048576 MINEXTENTS 1 MAXEXTENTS 2147483645
PCTINCREASE 0 FREELISTS 1 FREELIST GROUPS 1
BUFFER_POOL DEFAULT FLASH_CACHE DEFAULT CELL_FLASH_CACHE DEFAULT)
TABLESPACE "TS_DATA"
LOB ("Y") STORE AS SECUREFILE (
  TABLESPACE "TS_DATA" ENABLE STORAGE IN ROW 4000 CHUNK 32768
  NOCACHE LOGGING NOCOMPRESS KEEP_DUPLICATES
  STORAGE(INITIAL 1048576 NEXT 1048576 MINEXTENTS 1 MAXEXTENTS 2147483645
  PCTINCREASE 0
  BUFFER_POOL DEFAULT FLASH_CACHE DEFAULT CELL_FLASH_CACHE DEFAULT))
LOB ("Z") STORE AS SECUREFILE (
  TABLESPACE "TS_DATA" ENABLE STORAGE IN ROW 4000 CHUNK 32768
  NOCACHE LOGGING NOCOMPRESS KEEP_DUPLICATES
  STORAGE(INITIAL 1048576 NEXT 1048576 MINEXTENTS 1 MAXEXTENTS 2147483645
  PCTINCREASE 0
  BUFFER_POOL DEFAULT FLASH_CACHE DEFAULT CELL_FLASH_CACHE DEFAULT))
```

3. Index Organized Tables

Task 3: Compare performance of using IOT tables

Created and populated an EMP table:

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

Created index:

```
ALTER TABLE emp ADD CONSTRAINT emp_pk PRIMARY KEY (empno);

BEGIN
    dbms_stats.gather_table_stats( user, 'EMP', cascade=>true );
END;
/
```

Implemented the child table as a conventional 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)
)
/
```

Implemented the child table again 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
/
```

Populated these tables by inserting into them a work address for each employee, then a home address, then a previous address, and finally a school address:

```
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;

```

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.

```

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							
1 Plan hash value: 2258490653							
2							
3 -----							
4	Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time ...
5 -----							
6	0	SELECT STATEMENT		4	400	7 (0)	00:00...
7	1	NESTED LOOPS		4	400	7 (0)	00:00...
8	2	TABLE ACCESS BY INDEX ROWID	EMP	1	54	2 (0)	00:00...
9	* 3	INDEX UNIQUE SCAN	EMP_PK	1		1 (0)	00:00...
10	4	TABLE ACCESS BY INDEX ROWID BATCHED	HEAP_ADDRESSES	4	184	5 (0)	00:00...
11	* 5	INDEX RANGE SCAN	SYS_C007173	4		1 (0)	00:00...
12 -----							
13							
14 Predicate Information (identified by operation id):							
15 -----							
16							
17	3	- access("EMP"."EMPNO">=42)					
18	5	- access("HEAP_ADDRESSES"."EMPNO">=42)					

Go to the EMP table by primary key; get the row; then using that EMPNO, go to the address table; and using the index, pick up the child records.

Statistics for heap organized table:

Statistics	
1	CPU used by this session
1	CPU used when call started
1	DB time
5	Requests to/from client
9	consistent gets
4	consistent gets examination
4	consistent gets examination (fastpath)
9	consistent gets from cache
5	consistent gets pin
5	consistent gets pin (fastpath)
6	non-idle wait count
2	opened cursors cumulative
2	opened cursors current
1	pinned cursors current
9	session logical reads
6	user calls

9 consistent gets

```
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									
1	Plan hash value: 2359364176								
2									
3									
4	Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time		
5									
6	0	SELECT STATEMENT		4	400	3 (0)	00:00:01		
7	1	NESTED LOOPS		4	400	3 (0)	00:00:01		
8	2	TABLE ACCESS BY INDEX ROWID	EMP	1	54	2 (0)	00:00:01		
9	* 3	INDEX UNIQUE SCAN	EMP_PK	1		1 (0)	00:00:01		
10	* 4	INDEX RANGE SCAN	SYS_IOT_TOP_67188	4	184	1 (0)	00:00:01		
11									
12									
13	Predicate Information (identified by operation id):								
14									
15									
16	3	access("EMP"."EMPNO"=42)							
17	4	access("IOT_ADDRESSES"."EMPNO"=42)							

Go to the EMP table by primary key; get the row; then using the index, pick up the child records.

Statistics for index organized table:

Statistics	
1	CPU used by this session
1	CPU used when call started
1	DB time
5	Requests to/from client
5	consistent gets
4	consistent gets examination
4	consistent gets examination (fastpath)
5	consistent gets from cache
1	consistent gets pin
1	consistent gets pin (fastpath)
6	non-idle wait count
2	opened cursors cumulative
2	opened cursors current
1	pinned cursors current
5	session logical reads
6	user calls

5 consistent gets

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.

Using IOT we skipped TABLE ACCESS (BY INDEX ROWID BATCHED).
As a result, Heap table cost > IOT table cost.

Dropped all tables:

```
DROP TABLE emp;  
DROP TABLE heap_addresses;  
DROP TABLE iot_addresses;
```

4. Index Clustered Tables

Task 4: Analyses Cluster Storage by Blocks

Created cluster:

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

Created index:

```
CREATE INDEX idxcl_emp_dept ON CLUSTER emp_dept_cluster;
```

Created table dept:

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

Created table emp:

```
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 ) ;

```

Inserted data from scott.dept:

```

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

```

Inserted data from scott.emp:

```

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 rowids 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;

```

DEPT_BLK	EMP_BLK	FLAG	DEPTNO
1	1533	1533 (null)	10
2	1533	1533 (null)	10
3	1533	1533 (null)	10
4	1533	1533 (null)	20
5	1533	1533 (null)	20
6	1533	1533 (null)	20
7	1533	1533 (null)	20
8	1533	1533 (null)	20
9	1533	1533 (null)	30
10	1533	1533 (null)	30
11	1533	1533 (null)	30
12	1533	1533 (null)	30
13	1533	1533 (null)	30
14	1533	1533 (null)	30

If the block numbers are the same, the EMP row and the DEPT row are stored on the same physical database block together.

We observed that all of our data is perfectly stored. We got this result because we use a cluster.

Advantages are described in summary.

Dropped all tables and cluster:

```

DROP TABLE emp;
DROP TABLE dept;
DROP CLUSTER emp_dept_cluster;

```


5. Hash Clustered Tables

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 ) ;
```

Created table emp:

```
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 ) ;
```

Inserted data from scott.dept:

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

Inserted data from scott.emp:

```
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 rowids 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;
```

Query Result x				
SQL All Rows Fetched: 14 in 0.114 seconds				
	DEPT_BLK	EMP_BLK	FLAG	DEPTNO
1	3338	3338 (null)		10
2	3338	3338 (null)		10
3	3338	3338 (null)		10
4	3333	3333 (null)		20
5	3333	3333 (null)		20
6	3333	3333 (null)		20
7	3333	3333 (null)		20
8	3333	3333 (null)		20
9	3363	3363 (null)		30
10	3363	3363 (null)		30
11	3363	3363 (null)		30
12	3363	3363 (null)		30
13	3363	3363 (null)		30
14	3363	3363 (null)		30

If the block numbers are the same, the EMP row and the DEPT row are stored on the same physical database block together.

We observed that 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 to find the data of a specific department.

Dropped all tables and cluster:

```
DROP TABLE emp;
DROP TABLE dept;
DROP CLUSTER hash_cluster;
```

Summary

During this lab, I learned how data is physically organized and stored in tables of various types, when to use these tables, and more.

Heap organized and Index organized tables

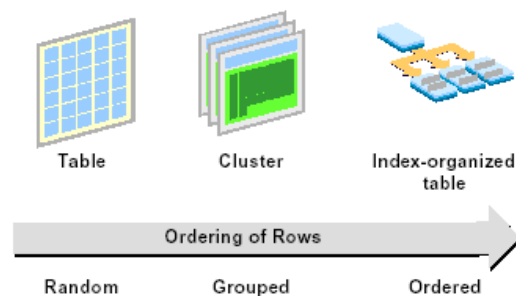
Conventional tables are organized as heap tables because the table rows can be stored in any table block. Fetching a row from a conventional table using a primary key involves primary key index traversal, followed by a table block access using the rowid.

In IOTs, the table itself is organized as an index, all columns are stored in the index tree itself, and the access to a row using a primary key involves index access only. This access using IOTs is better because all columns can be fetched by accessing the index structure, thereby avoiding the table access. This is an efficient access pattern because the number of accesses is minimized.

IOTs are appropriate for tables with the following properties:

- Tables with shorter row length
- Tables accessed mostly through primary key columns

Distribution of Rows Within a Table



Index clustered table and Hash clustered tables

A cluster is a way to store a group of tables that share some common column(s) in the same database blocks and to store related data together on the same block.

Clusters are good to use when we want our tables to be connected all the time.

An index cluster is a table cluster that uses an index to locate data. The cluster index is a B-tree index on the cluster key. A cluster index must be created before any rows can be inserted into clustered tables.

A hash cluster is like an indexed cluster, except the index key is replaced with a hash function. In a hash cluster, the data is the index.