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

Viktoria Masyuk <https://github.com/vilo4kaaa/DMDataCamp22>

**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**,'\*'),

с VARCHAR2(**4000**) **DEFAULT** RPAD('\*',**4000**,'\*'),

d VARCHAR2(**4000**) **DEFAULT** RPAD('\*',**4000**,'\*')

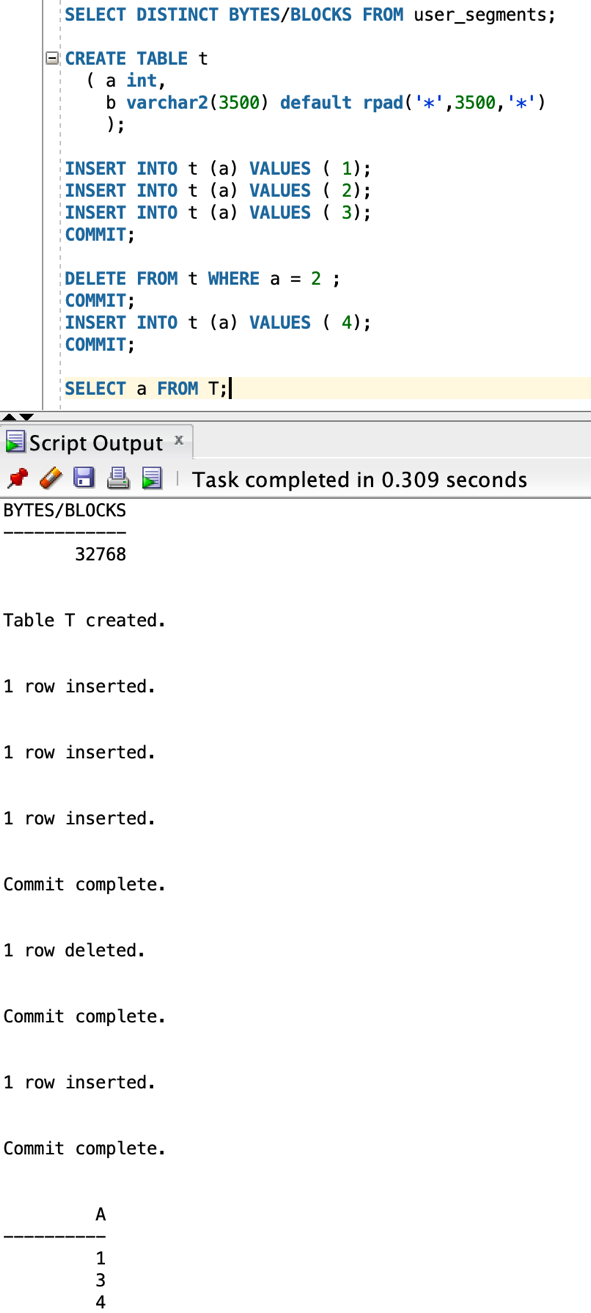
);

**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:

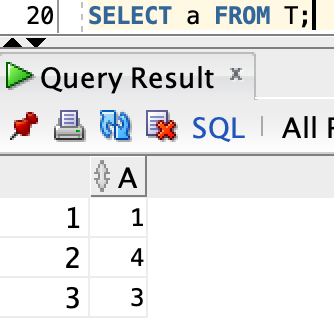
**DELETE** **FROM** t **WHERE** a = **2** ;

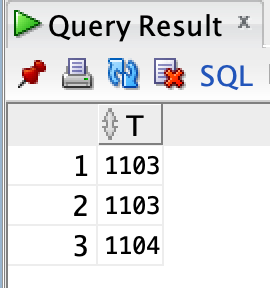
**COMMIT**;

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

**COMMIT**;

Selected a column from the table t:

**SELECT** a **FROM** T;

**SELECT** dbms\_rowid.rowid\_block\_number( t.rowid ) t **From** t;

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

Dropped table t:

**DROP** **TABLE** t;

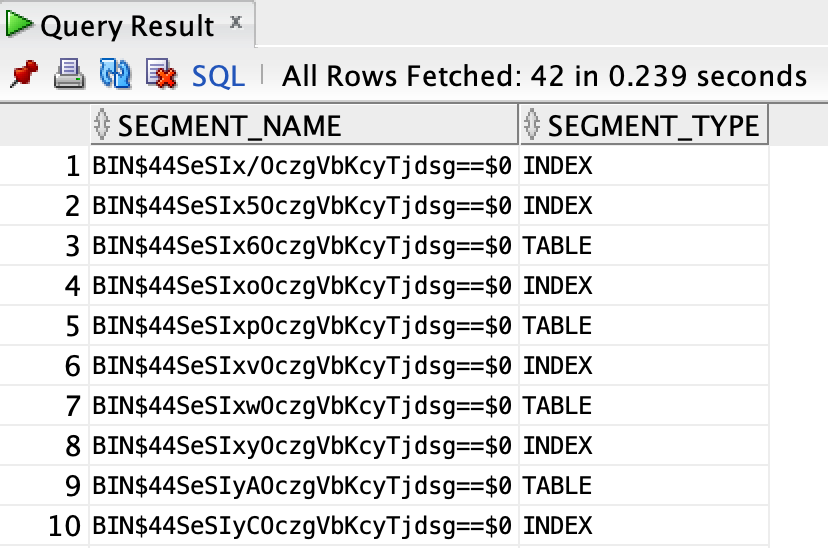
A full scan of the table will be retrieved as it hits it, not in the order of insertion.

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;



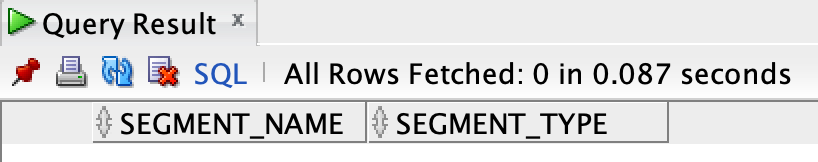
There are a lot of old items which are located in bin, so I purged the recycle bin:

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;



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

Dropped table t:

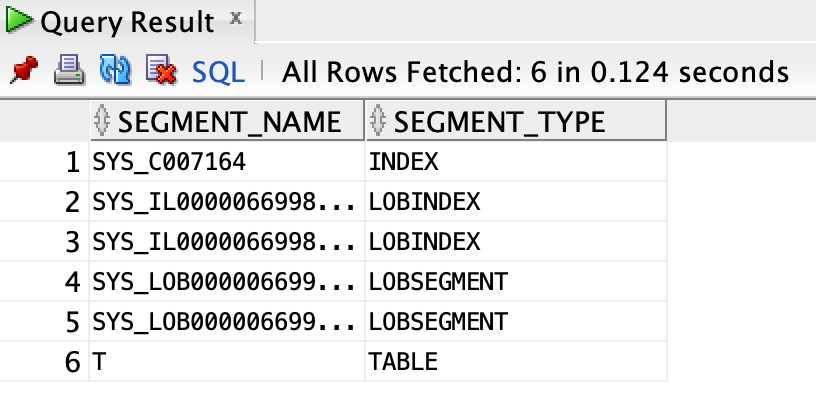
**DROP** **TABLE** T;

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;

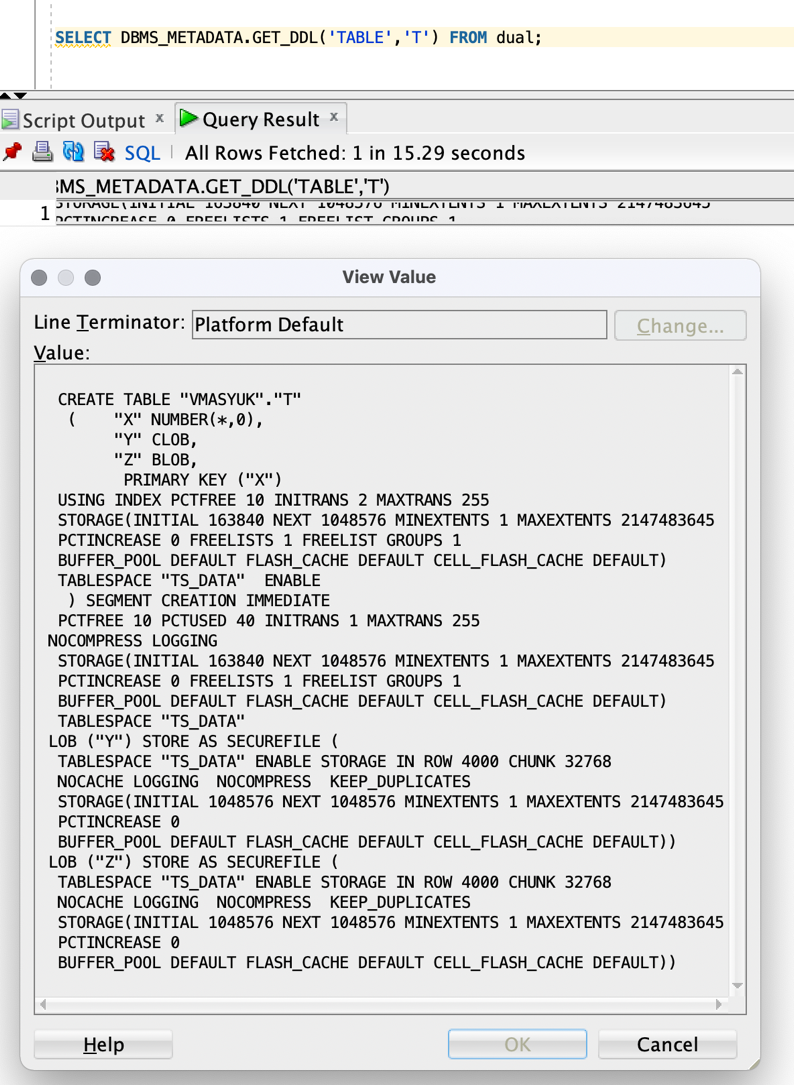


The table itself created a segment and the primary key constraint created an index segment.

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;



**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' );

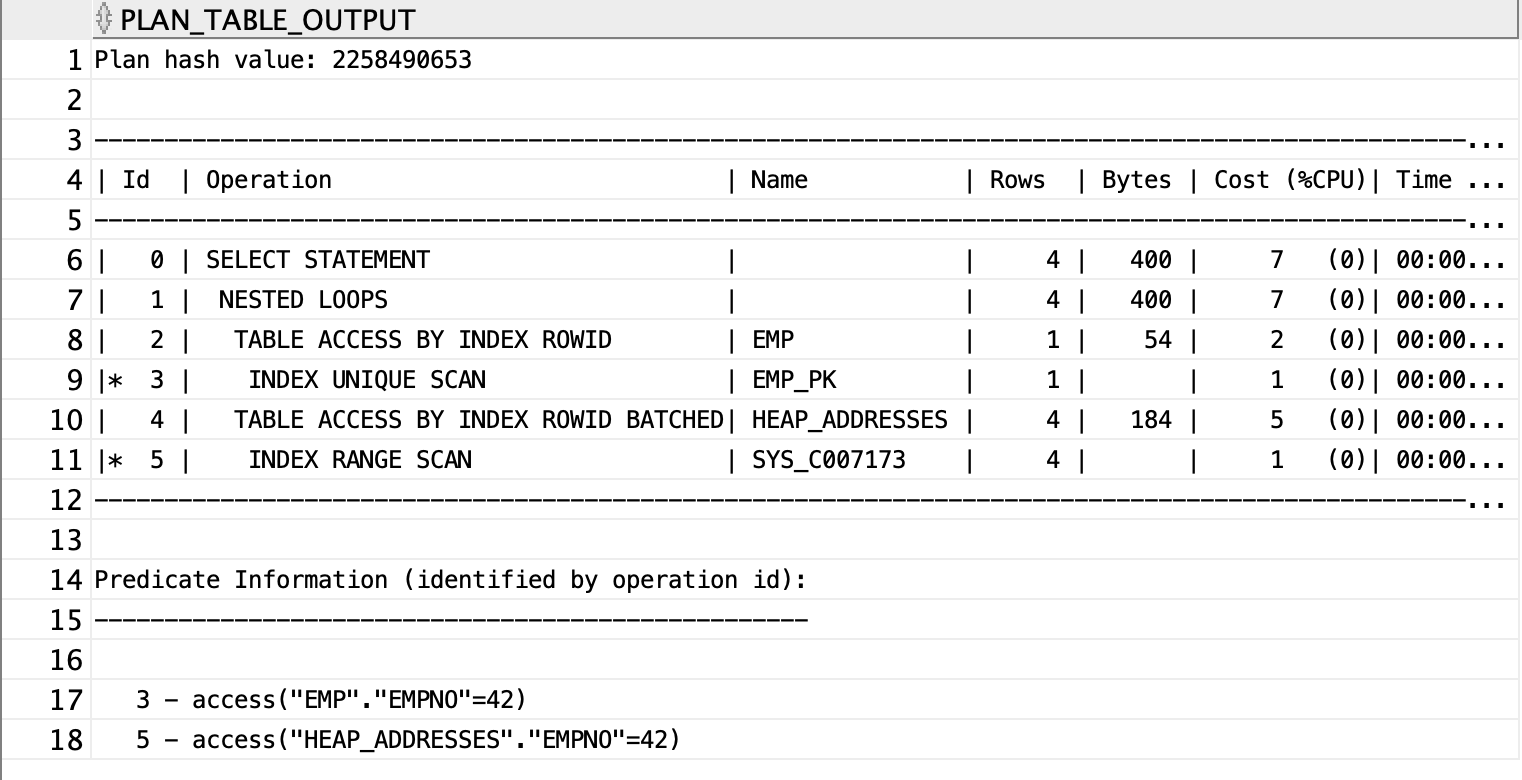
**EXEC** dbms\_stats.gather\_table\_stats( **user**, 'IOT\_ADDRESSES' );

**EXPLAIN** PLAN **FOR**

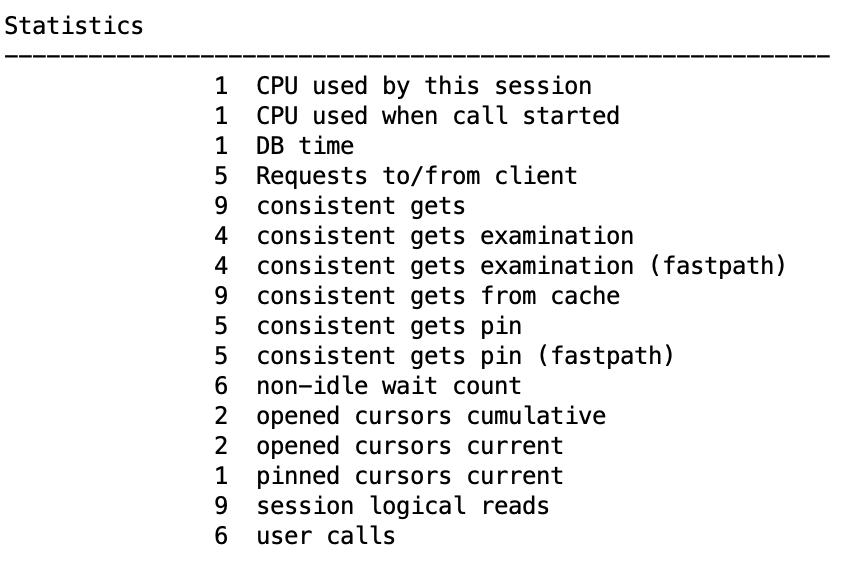
**SELECT** \* **FROM** emp, heap\_addresses

**WHERE** emp.empno = heap\_addresses.empno

**AND** emp.empno = **42**;

**SELECT** \* **FROM** **TABLE**(dbms\_xplan.display );

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:

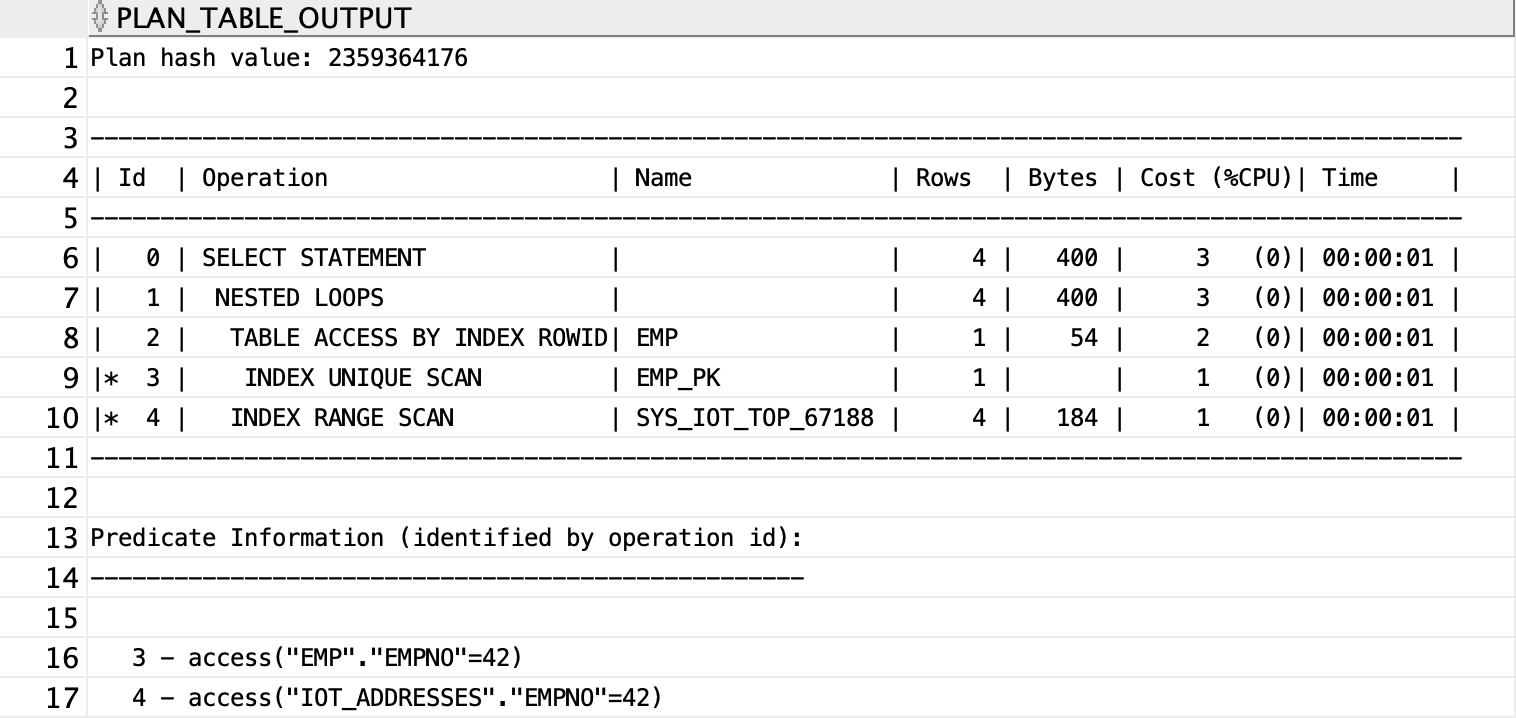
9 consistent gets

**EXPLAIN** PLAN **FOR**

**SELECT** \* **FROM** emp, iot\_addresses

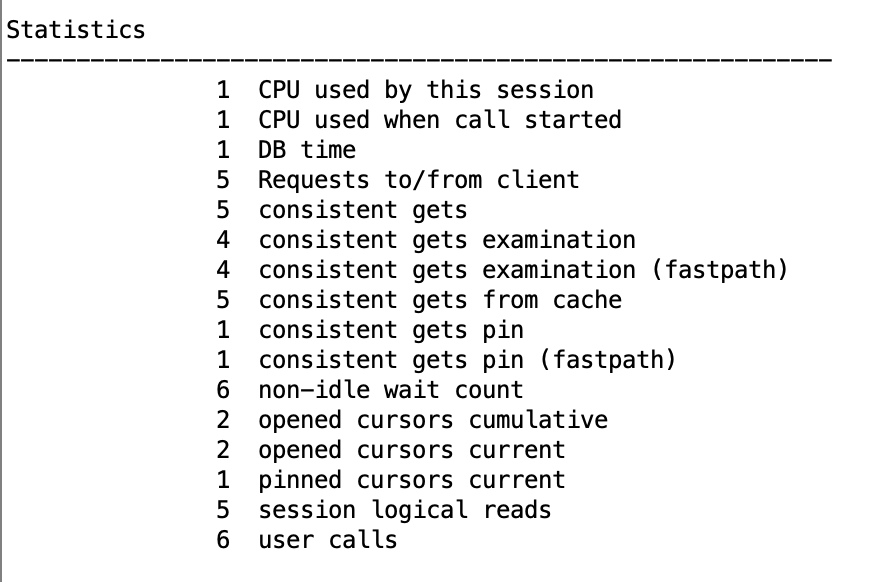
**WHERE** emp.empno = iot\_addresses.empno

**AND** emp.empno = **42**;

**SELECT** \* **FROM** **TABLE**(dbms\_xplan.display );

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:

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 **100**K **NEXT** **50**K );

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:

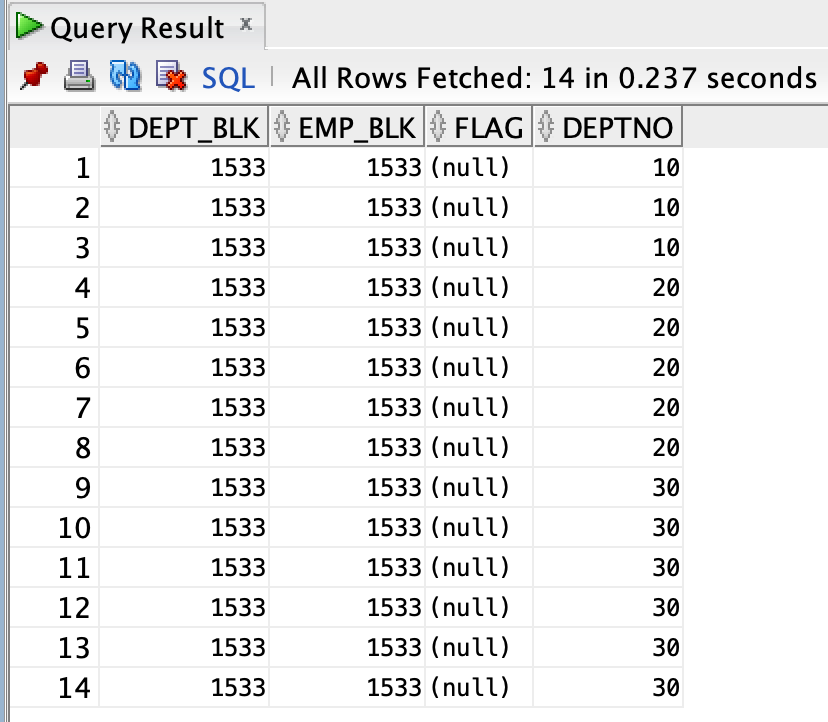
**SELECT** \* **FROM**

(**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;

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 **100**K **NEXT** **50**K );

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**

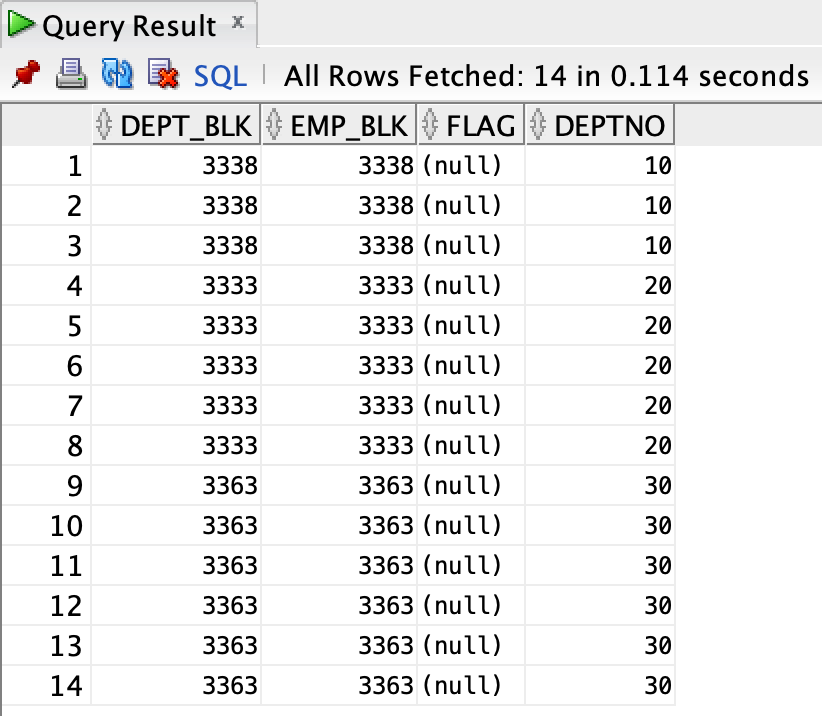
(**SELECT** dept\_blk, emp\_blk, **CASE** **WHEN** dept\_blk <> emp\_blk **THEN** '\*' **END** flag, deptno

**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;



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**

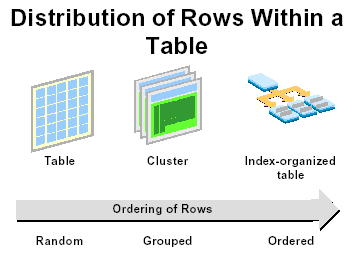
Conventional tables are organized as heap tables because the table rows can be stored in any table block. Fetching a row from a conventional table using a primary key involves primary key index traversal, followed by a table block access using the rowid.

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



**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.