**Anton Slizh’s**

**U1M3.LW.Database Types of Tables, Indexes**

*GitHub: https://github.com/drapejny/DataCamp2022*

**Task 1**

Step 1:

Creating table t

create table t

( a int,

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

c varchar2(4000) default rpad('\*',4000,'\*')

);

Step 2:

Inserting data. Deleting row where a = 2 and inserting one more row to show heap organizing of data.

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);

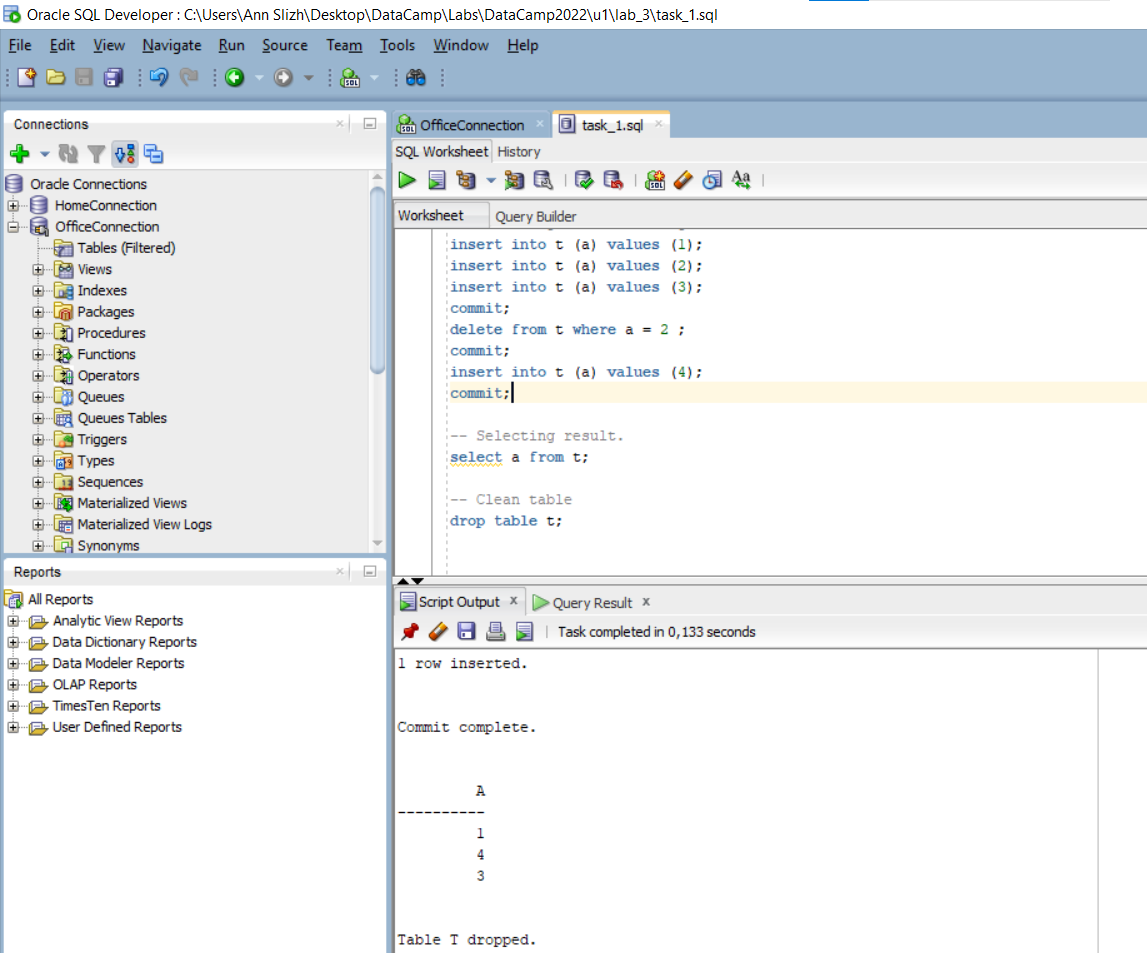
commit;

Step 3:

Selecting result. Results show us that new row was placed into vacant place,

not in the order of inserting.

select a from t;



**Task 2**

Step 1:

Creating table t with deferred segments creation.

create table t

( x int primary key,

y clob,

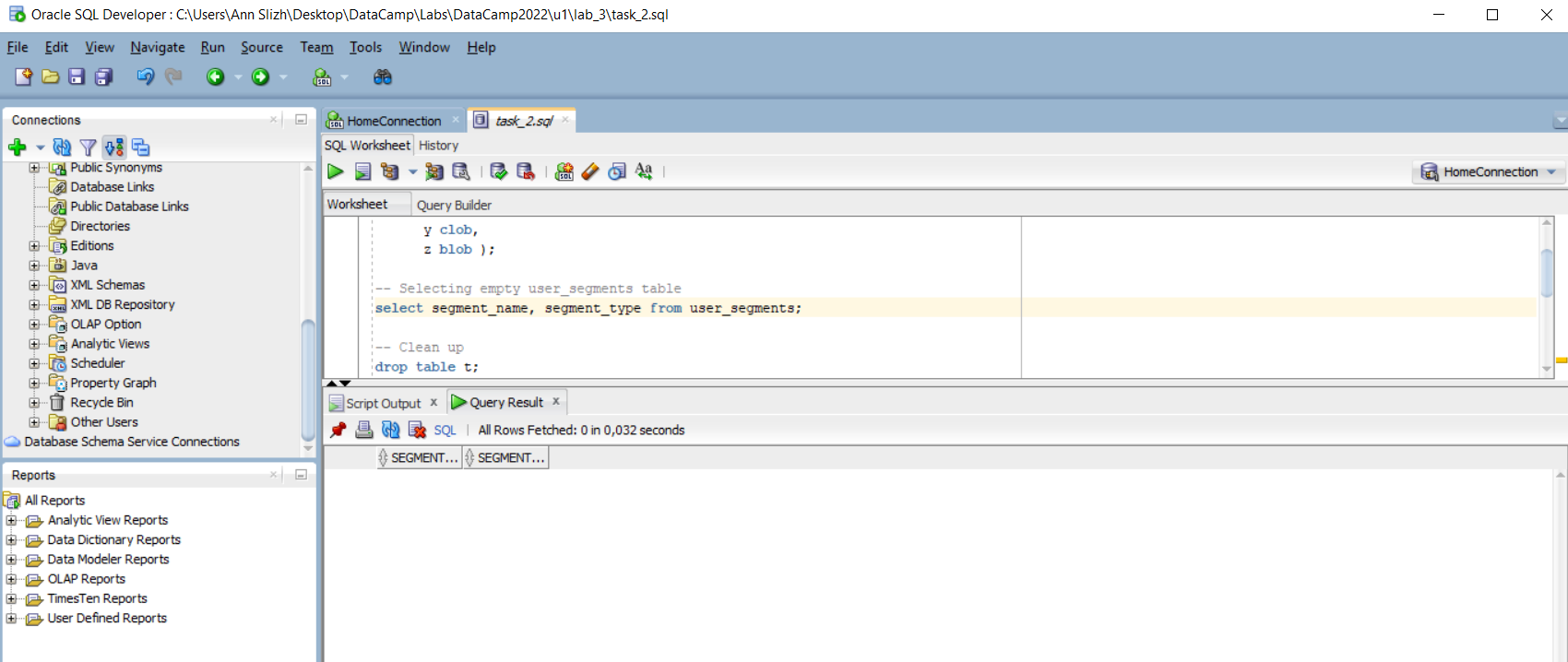
z blob );

Step 2:

Selecting empty user\_segments table

select segment\_name, segment\_type

from user\_segments;



Step 3:

Creating table t with immediate segments creation

create table t

( x int primary key,

y clob,

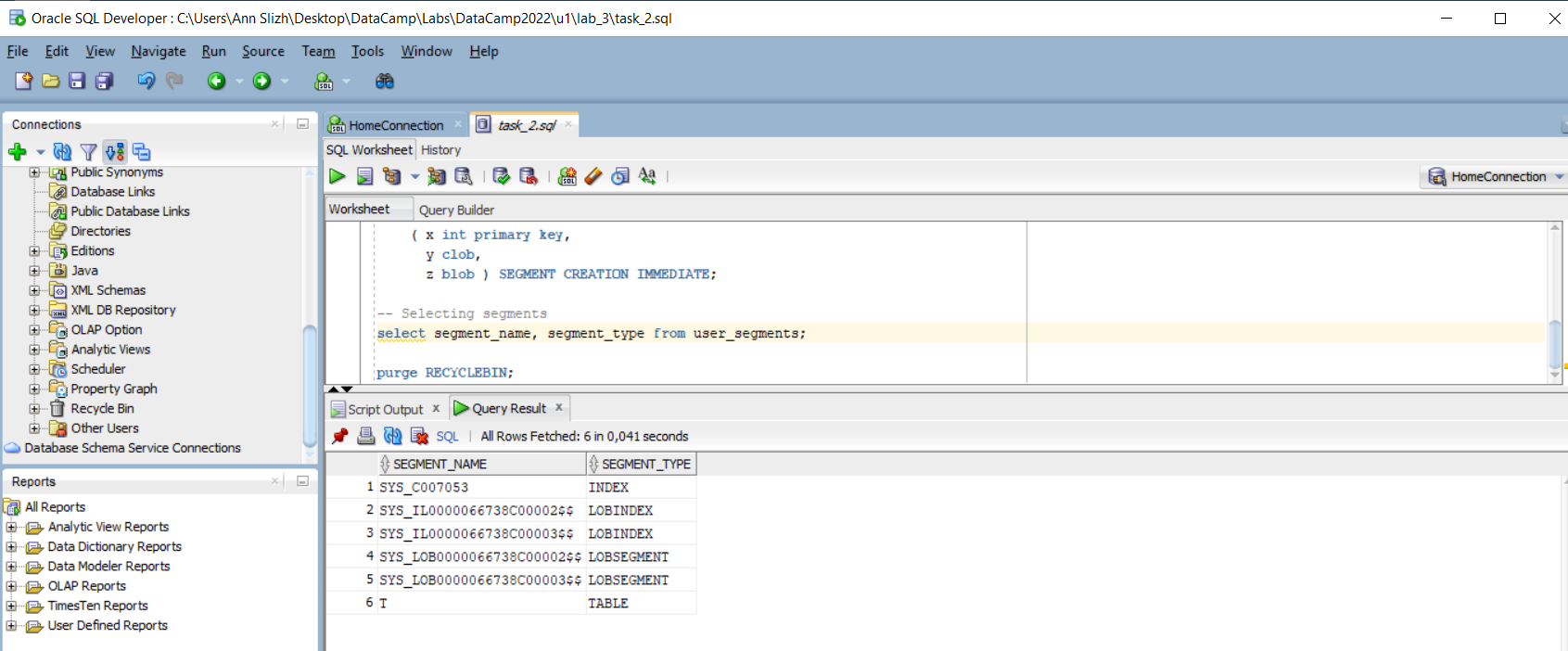
z blob ) SEGMENT CREATION IMMEDIATE;

Step 4:

Selecting user\_segments table (not empty now)

select segment\_name, segment\_type

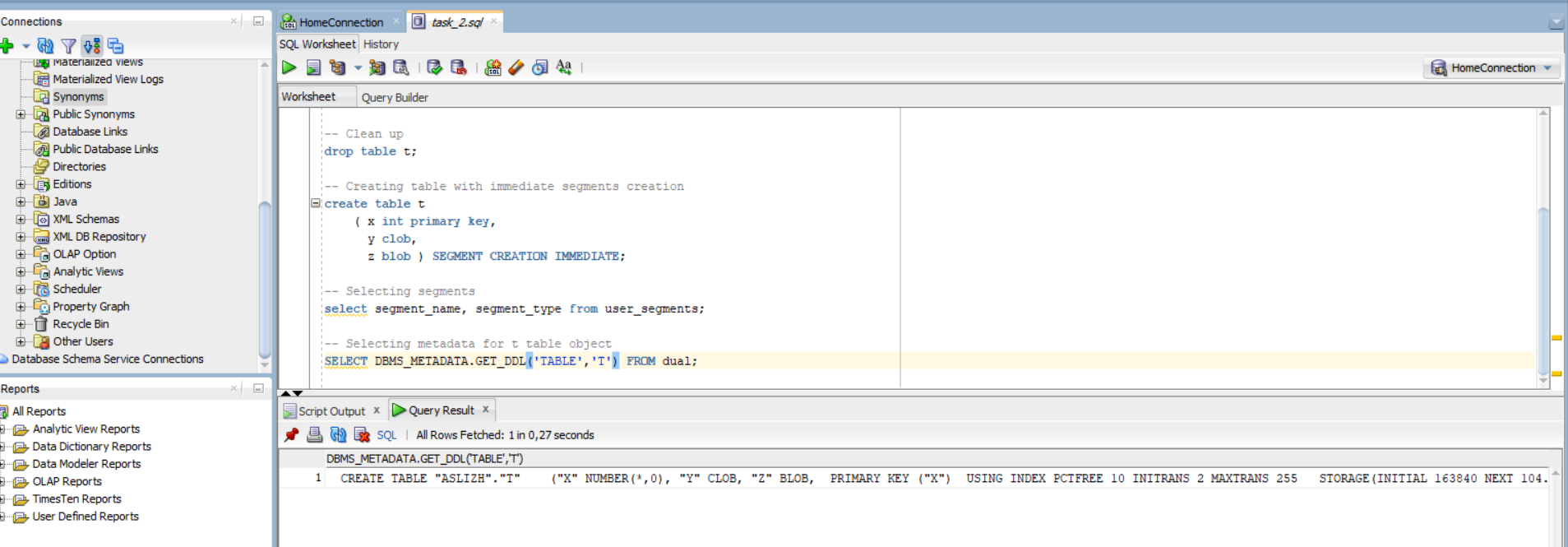
from user\_segments;



Step 5:

Selecting metadata for table t object

SELECT DBMS\_METADATA.GET\_DDL('TABLE','T') FROM dual;



**Task 3**

Step 1:

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)

Calculating statistics

begin

dbms\_stats.gather\_table\_stats( user, 'EMP', cascade=>true );

end;

Step 2:

Creating table *heap\_addresses*

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:

Creating index organized table *iot\_addresses*

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:

Inserting data

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:

Calculating statistics

exec dbms\_stats.gather\_table\_stats( $username$, 'HEAP\_ADDRESSES' );

exec dbms\_stats.gather\_table\_stats( $username$, 'IOT\_ADDRESSES' );

Step 6:

Comparing execution plans of queries on heap and index tables

explain plan for

SELECT \*

FROM emp ,

heap\_addresses

WHERE emp.empno = heap\_addresses.empno

AND emp.empno = 42;

select \* from table(dbms\_xplan.display );

explain plan for

SELECT \*

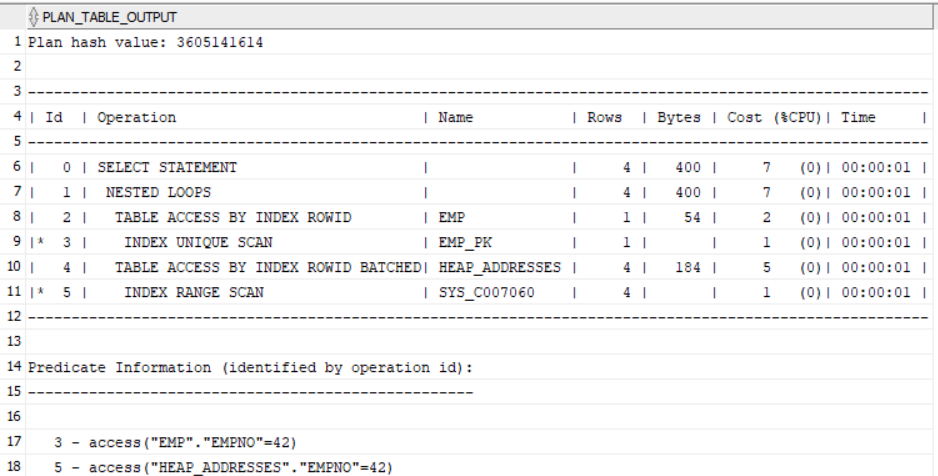
FROM emp ,

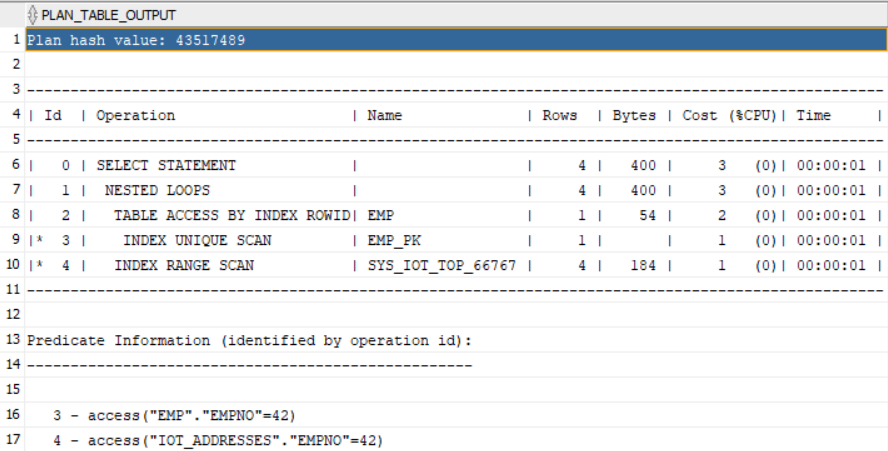
iot\_addresses

WHERE emp.empno = iot\_addresses.empno

AND emp.empno = 42;

select \* from table(dbms\_xplan.display );





We see that in selecting on index organized table is less costly than selecting on heap organized table. The main reason is that the IOT table is fully stored in an index structure. We see, that there is no additional *TABLE ACCESS BY INDEX ROWID* step.

**Task 4**

Step 1:

Creating cluster

CREATE cluster emp\_dept\_cluster( deptno NUMBER( 2 ) )  
 SIZE 1024   
 STORAGE( INITIAL 100K NEXT 50K );

Step 2:

Creating cluster index

CREATE INDEX idxcl\_emp\_dept on cluster emp\_dept\_cluster;

Step 3:

Creating tables in cluster

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 );

Step 4:

Inserting data into tables

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;

Step 5:

Selecting block number for rows in emp and dept tables

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

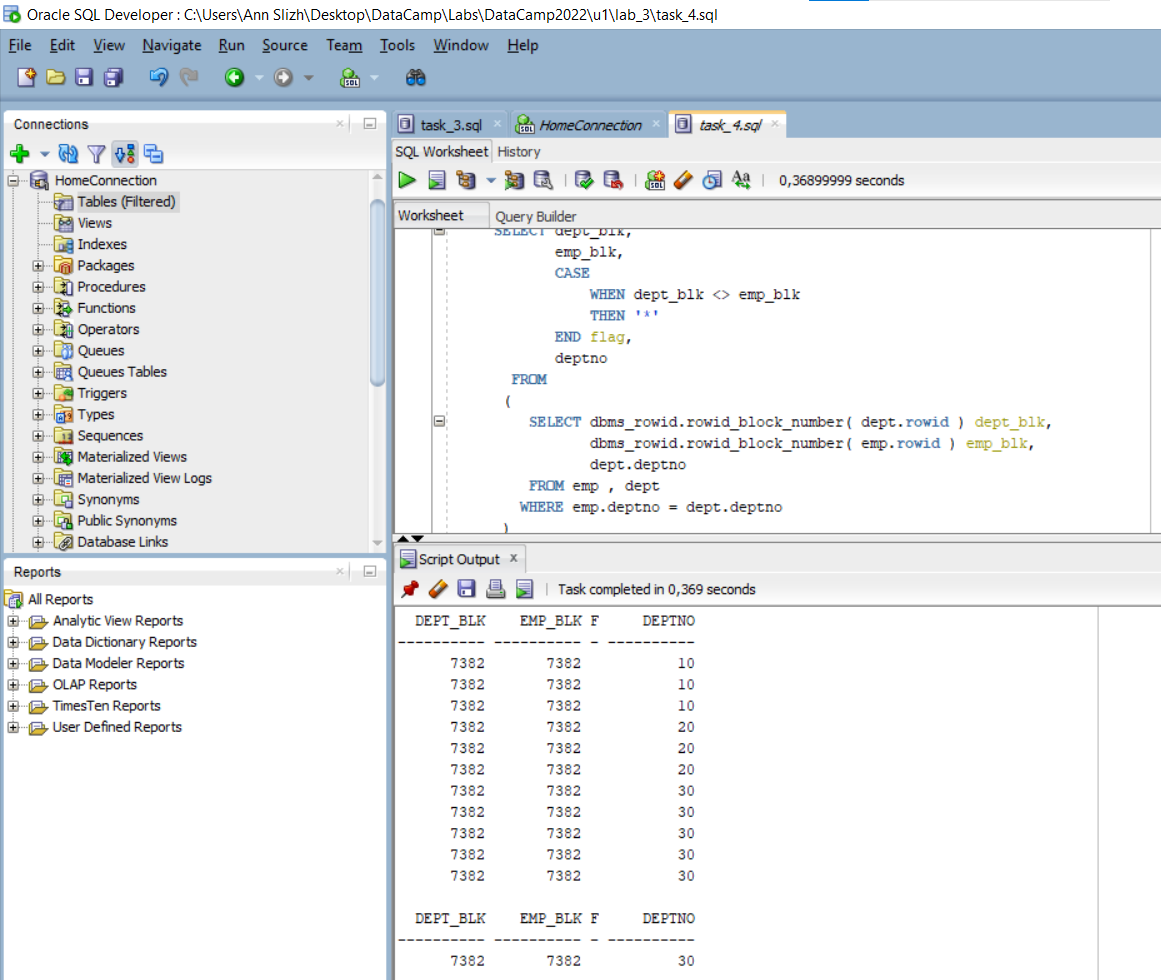
FROM emp , dept

WHERE emp.deptno = dept.deptno

)

)

ORDER BY deptno;



We see the same block number for rows in both tables. All because we use cluster index which store rows physically near each other (at the same block). It allows us to use joins more efficiently.

**Conclusion**

In heap organized tables data inserted into vacant place. I mean, after deleting of row with a = 2, the space became available for further insertions. So, we saw that row with a = 4 inserted there.

When we creating new table segments don’t created immediately. They will be created after first insertion in table. We can change this behavior with SEGMENT CREATION IMMEDIATE statement.

Whereas in a heap organized tables data is stored in random order, in index organized tables data is stored in index structure in sorted order, according to primary key. It allows faster access to data and reduce costs.

Index cluster tables are groups together and stored physically near each other. We can use joins more efficiently. For example, if we know that some tables should be joined every time, we can make cluster index on them and read grouped data by one I/O operation.