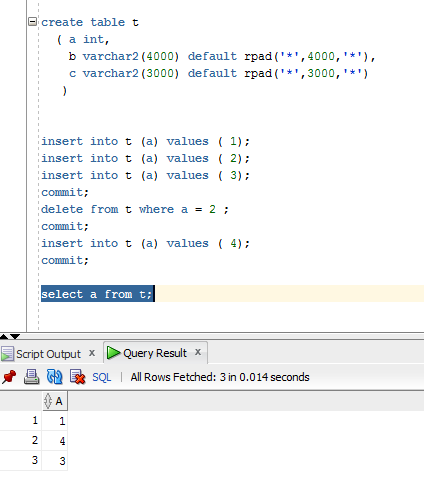
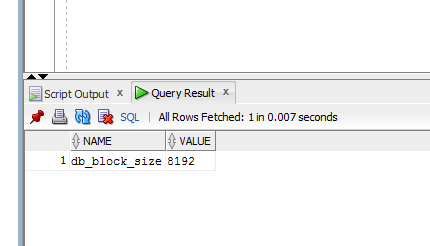
# Heap Organized Tables

## Task 1 – Heap Understanding



## Task Results:



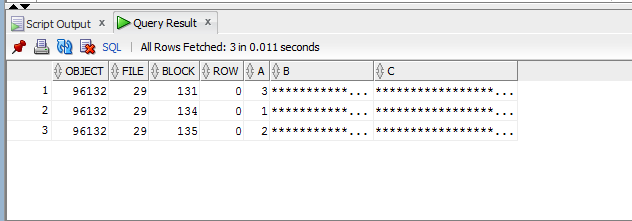
Column B and C in the total sum equal to 7000 symbols or 7000 B, our block-size is 8192 B, so one block will store one row. Evidence of it is shown bellow.

Once several rows have been inserted, they have been placed into different blocks in non-ordered way

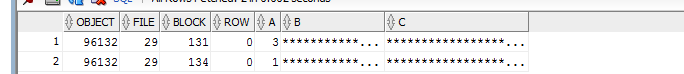
insert into t (a) values ( 1);

insert into t (a) values ( 2);

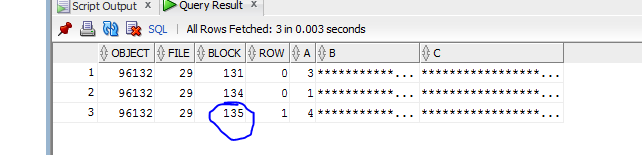
insert into t (a) values ( 3);



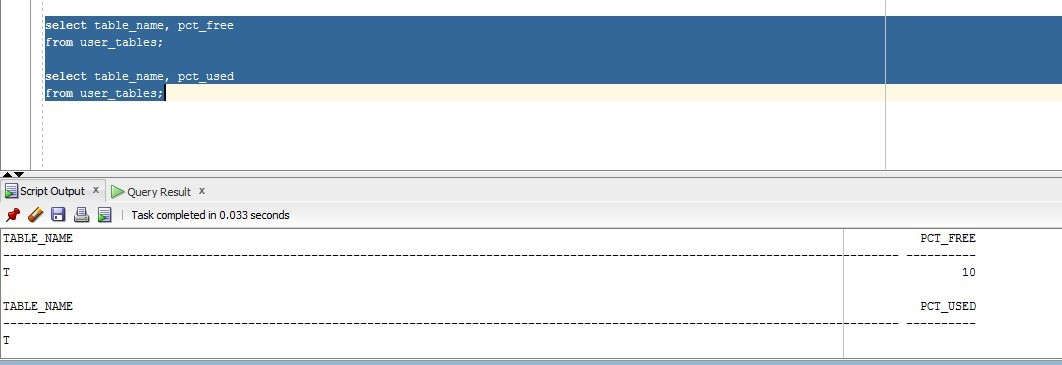
delete from t where a = 2 ;



insert into t (a) values ( 4);



One block should store one row. On the screenshots is shown how one block can be reused after deleting row from it and inserting another one.



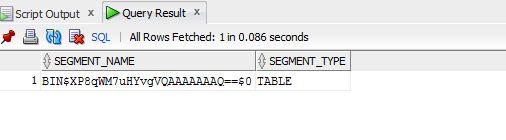
PCTUsed isn’t displayed, because it’s simply ignored.

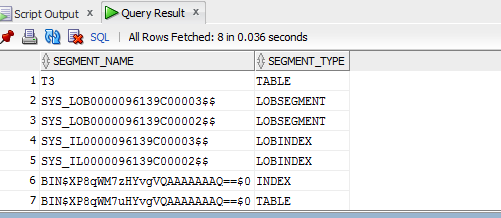
## Task 2 – Understanding Heap Table Segments

Step 1:

create table t3 ( x int primary key, y clob, z blob );

Step 2:





Step 3:

Create table t (

x int primary key,

y clob,

z blob

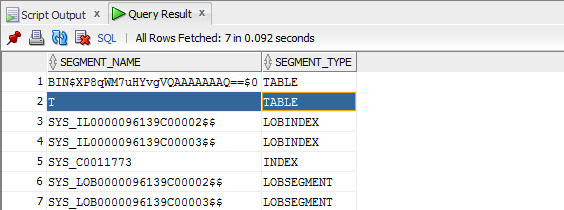
)

SEGMENT CREATION IMMEDIATE

/

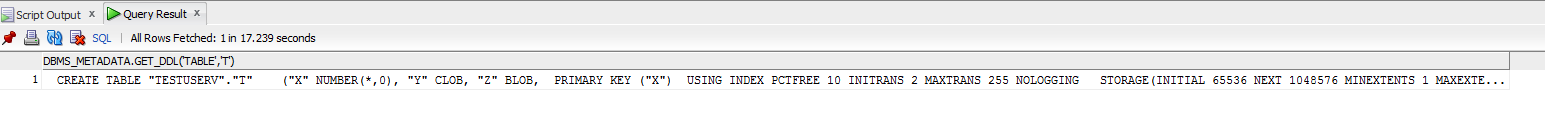
Step 4:

select segment\_name, segment\_type FROM user\_segments;



Step 5:

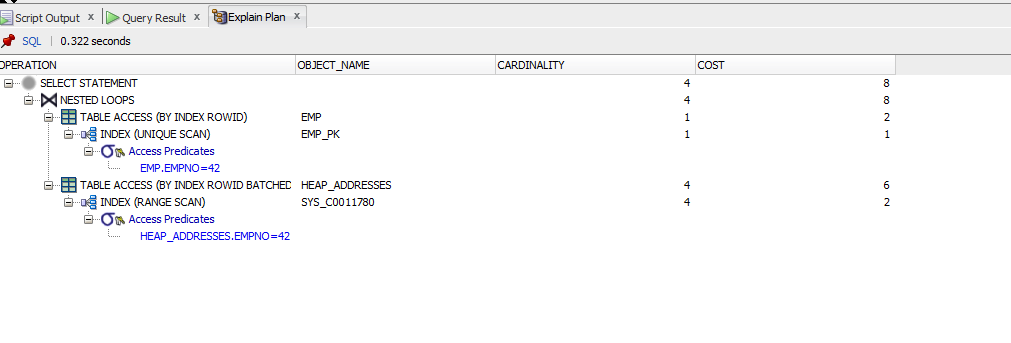
# SELECT DBMS\_METADATA.GET\_DDL('TABLE','T') FROM dual



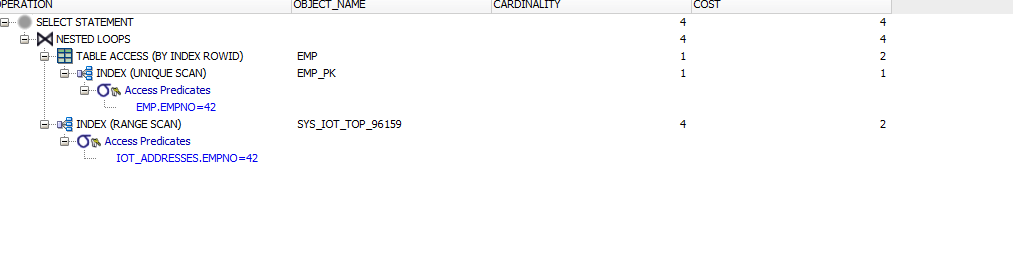
Once table has been created no segment has been created with is (first screenshot in step2. But if some rows is inserted into table, then a segment will be created (second screnshot in the step2). Also segment will be created and represented (step 4-5) in the user\_segment table with SEGMENT CREATION IMMEDIATE As a default behaviour of Database.

# Index Organized Tables

Explain 1:



Explain 2:

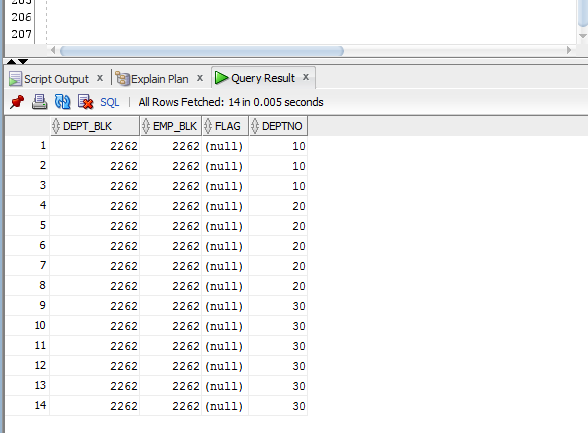


In the first case table has no special index so it stores data in unordered collection (heap), as a result performance in the first case shows bigger cost than in the second case. Data for an index-organized table is stored in a B-tree index structure in a primary key sorted manner. Each leaf block in the index structure stores both the key and nonkey columns.The structure of an index-organized table provides the following benefits:

* Fast random access on the primary key because an index-only scan is sufficient. And, because there is no separate table storage area, changes to the table data (such as adding new rows, updating rows, or deleting rows) result only in updating the index structure.
* Fast range access on the primary key because the rows are clustered in primary key order.
* Lower storage requirements because duplication of primary keys is avoided. They are not stored both in the index and underlying table, as is true with heap-organized tables.

# Index Clustered Tables

## Task 3: Cluster Storage by Blocks

.

Two tables are stored on the same block in order to make data transportation is much efficient and reliable. We can see that they have the same clustered key so they are physically stored together.

# Hash Clustered Tables

CREATE cluster emp\_dept\_cluster1( deptno NUMBER( 2 ) )

SIZE 1024 hashkeys 15

STORAGE( INITIAL 100K NEXT 50K );

CREATE TABLE dept

(

deptno NUMBER( 2 ) PRIMARY KEY

, dname VARCHAR2( 14 )

, loc VARCHAR2( 13 )

)

cluster emp\_dept\_cluster1 ( 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\_cluster1 ( 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

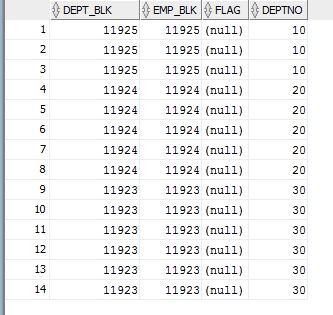
WHERE emp.deptno = dept.deptno

)

)

ORDER BY deptno

Indexes can not being created with the hash-clustered tables because of the fact that “In a hash cluster, the data is the index (metaphorically speaking)”.



With the hash clustered key data with the same cluster key is stored in the one block, because they have the same value of the hashed clustered key in comparison with first case when the data is clustered basing on B-Tree index, and data with different clustered keys may be stored in one block.

# Row Migration\*

Demonstration of the Row Migration

CREATE TABLE row\_mig\_chain\_demo (

x int PRIMARY KEY,

a CHAR(1000),

b CHAR(1000),

c CHAR(1000),

d CHAR(1000),

e CHAR(1000)

);

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);

COMMIT;

UPDATE row\_mig\_chain\_demo SET a = 'z1', b = 'z2', c = 'z3' WHERE x = 3;

COMMIT;

UPDATE row\_mig\_chain\_demo SET a = 'y1', b = 'y2', c = 'y3' WHERE x = 2;

COMMIT;

UPDATE row\_mig\_chain\_demo SET a = 'w1', b = 'w2', c = 'w3' WHERE x = 1;

COMMIT;

SELECT \* FROM row\_mig\_chain\_demo;

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';

SELECT \* FROM row\_mig\_chain\_demo WHERE x = 3;

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';

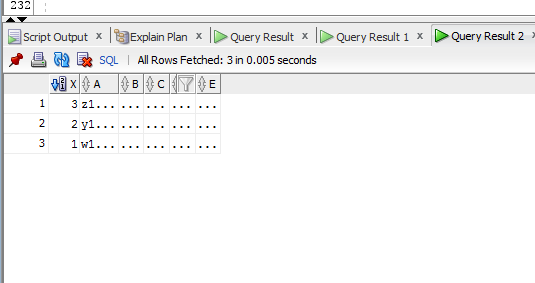
SELECT \* FROM row\_mig\_chain\_demo WHERE x = 1; //Row 1 is migrated, using the primary key index, we forced a «table fetch continued row».

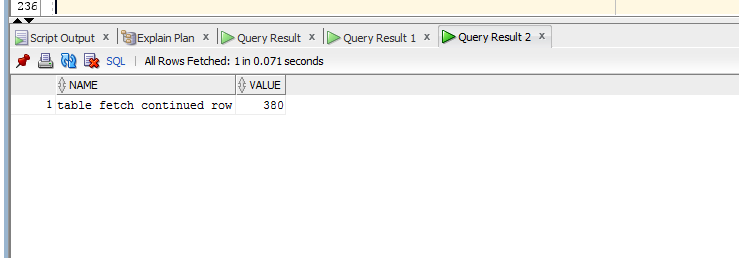
SELECT a.name, b.value

FROM v$statname a, v$mystat b

WHERE a.statistic# = b.statistic#

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





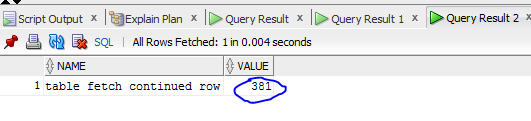
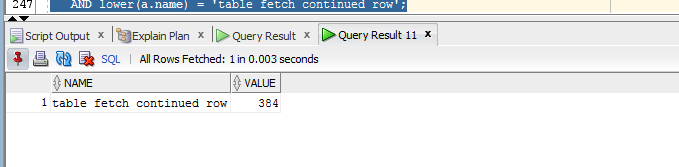
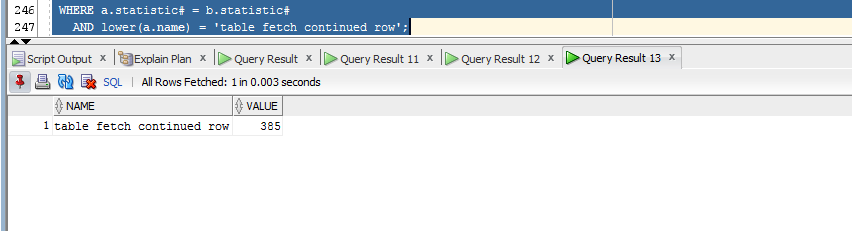


Figure Row 1 is migrated, using the primary key index, we forced a «table fetch continued row».

Demonstration of the Row Chaining





* **Chained rows** - a chained rows is a LOB row (usually a BLOB, CLOB, RAW or LONG RAW) where the row length is large than the data block size.  Many shops will create a 32k blocksize to store large columns without row chaining.
* **Migrated rows** - Migrated rows occur when an UPDATE DML causes the rows to expand onto another data block.  This can be avoided by setting PCTFREE to a large enough value to accommodate row expansion, and existing migrated rows can be fixed by reorganizing the tables with the dbms\_redefinition utility.

Also, note that chained rows and migrated rows only degrade performance for single block fetches, and they will not impact the performance of full table scan operations.

Honestly, I have got no full impression on the data migration and changing from the code point of view, but generally I have understood what it is.