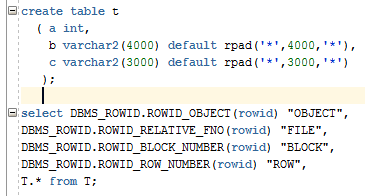
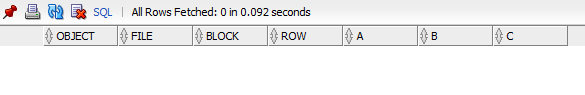
# Heap Organized Tables

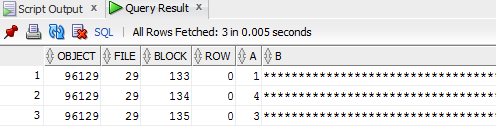
## Task 1 – Heap Understanding

Step 1:





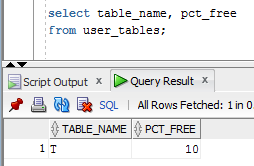
Step 2:



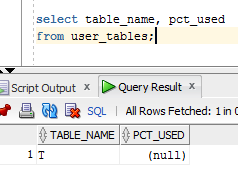
Step 3:



As we can see from the third screenshot a number “4” is in the second row, although we added it after “3”. It’s in the second row, because it’s **heap organized** , so the data is added to the first free space found in the segment.



Pctused is null, because of automatic segment space management (ASSM)( is a simpler and more efficient way of managing space within a segment). It completely eliminates any need to specify and tune the pctused, freelists, and freelist groups storage parameters for schema objects created in the tablespace.



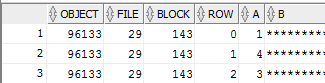
SELECT name,value

FROM v$parameter

WHERE name = 'db\_block\_size';

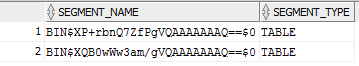


After we set size 1203 for columns b and c:

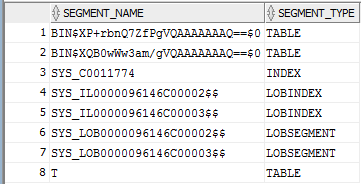


**Task 2 – Understanding Heap Table Segments**

Step 2:



Step 4:



Segments were created for all objects, indexes, table (here they ae duplicated, because I’ve created 2 tables)

Step 5:



CREATE TABLE "SAIDA\_MELIKAVA"."T"

( "X" NUMBER(\*,0),

"Y" CLOB,

"Z" BLOB,

PRIMARY KEY ("X")

USING INDEX PCTFREE 10 INITRANS 2 MAXTRANS 255 NOLOGGING

STORAGE(INITIAL 65536 NEXT 1048576 MINEXTENTS 1 MAXEXTENTS 2147483645

PCTINCREASE 0 FREELISTS 1 FREELIST GROUPS 1

BUFFER\_POOL DEFAULT FLASH\_CACHE DEFAULT CELL\_FLASH\_CACHE DEFAULT)

TABLESPACE "TBS\_PDB\_TEST" ENABLE

) SEGMENT CREATION IMMEDIATE

PCTFREE 10 PCTUSED 40 INITRANS 1 MAXTRANS 255

NOCOMPRESS NOLOGGING

STORAGE(INITIAL 65536 NEXT 1048576 MINEXTENTS 1 MAXEXTENTS 2147483645

PCTINCREASE 0 FREELISTS 1 FREELIST GROUPS 1

BUFFER\_POOL DEFAULT FLASH\_CACHE DEFAULT CELL\_FLASH\_CACHE DEFAULT)

TABLESPACE "TBS\_PDB\_TEST"

LOB ("Y") STORE AS SECUREFILE (

TABLESPACE "TBS\_PDB\_TEST" ENABLE STORAGE IN ROW CHUNK 8192

NOCACHE NOLOGGING NOCOMPRESS KEEP\_DUPLICATES

STORAGE(INITIAL 106496 NEXT 1048576 MINEXTENTS 1 MAXEXTENTS 2147483645

PCTINCREASE 0

BUFFER\_POOL DEFAULT FLASH\_CACHE DEFAULT CELL\_FLASH\_CACHE DEFAULT))

LOB ("Z") STORE AS SECUREFILE (

TABLESPACE "TBS\_PDB\_TEST" ENABLE STORAGE IN ROW CHUNK 8192

NOCACHE NOLOGGING NOCOMPRESS KEEP\_DUPLICATES

STORAGE(INITIAL 106496 NEXT 1048576 MINEXTENTS 1 MAXEXTENTS 2147483645

PCTINCREASE 0

BUFFER\_POOL DEFAULT FLASH\_CACHE DEFAULT CELL\_FLASH\_CACHE DEFAULT))

**A conclusion:**

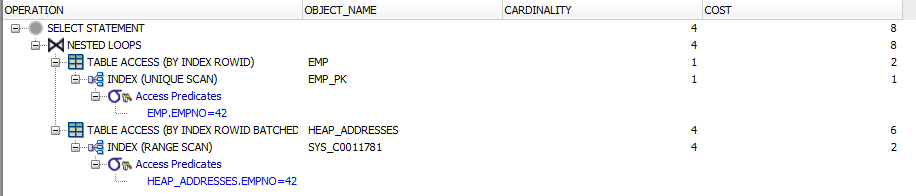
1. In the step 2 we used default Deferred Segment Creation:  When non-partitioned tables are created, none of the associated segments (table, implicit index and LOB segments) are created until rows are inserted into the table. It’s really useful for systems with lots of empty tables (this can represent a large space saving).
2. In the step 4 we used **Immediate Segment Creation**: The table segment is created as part of this CREATE TABLE statement.
3. In the step 5 we retrieved the whole query for our created table (all parameters are mentioned here).

# 

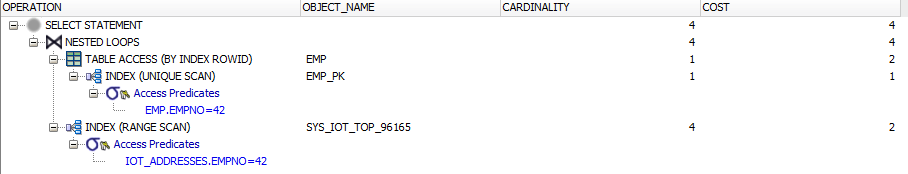
# Index Organized Tables

## Task 3: Compare performance

Select 1:



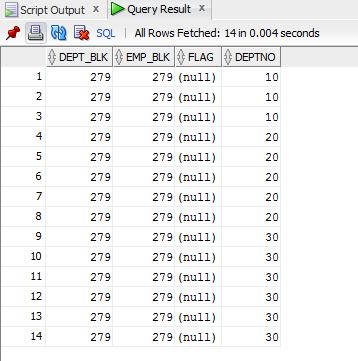
Select 2:



In the select 2 costs are considerably smaller because when we join with the heap table we should firstly get access to rowid and then go to the table, and when we join with index-organized table (the second one) it stores all info within it.

# Index Clustered Tables

## Task 4: Cluster Storage by Blocks



**A conclusion:** Here we can see, that all data that contains the same cluster key value (DEPTNO), is physically stored together (block numbers are the same).

If we don’t use cluster, dept\_blk and emp\_blk will be different.

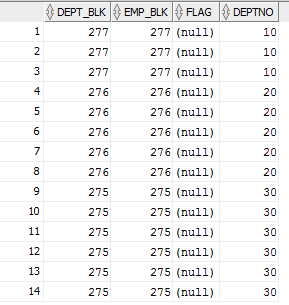
# Hash Clustered Tables

## Task 5: Analyses Cluster Storage by Blocks

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

SIZE 1024 hashkeys 15

STORAGE( INITIAL 100K NEXT 50K );



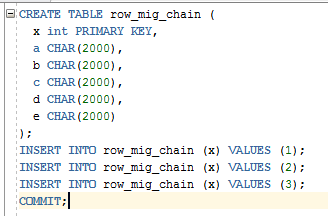
**A conclusion:** This table is similar to the previous one, but instead of using a B\*Tree index to locate the data by cluster key, the hash cluster hashes the key (we created “hashkeys”) to the cluster to arrive at the database block the data should be on (so we can see that each deptno is stored in the different datablock) .

# Row Migration\*

## Task

Create two examples: one with row migration, another with row chaining.

1. Created a table with 6 columns and inserted



Then run the following script:

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';

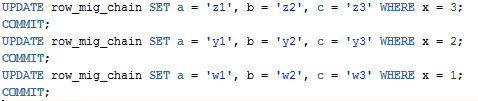
The result is:



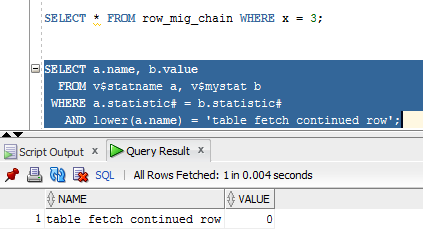
**Table fetch continued row** tells when we didn’t find all that we wanted from the original row piece and had to follow a pointer to the new location of the migrated row (or next row piece of a chained row).

**Row migration:**

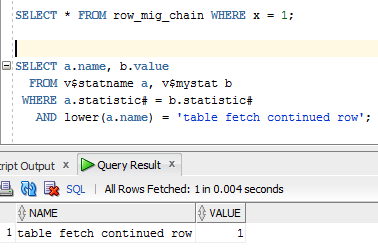
1. Inserted values in columns a,b,c. So now we have **data migration:** a row is inserted small and then updated and there is no space left in the data block, so each row now has it’s own data block.



Proofs:



Here is 0, bcs is was updated first and had enough memory to do so.

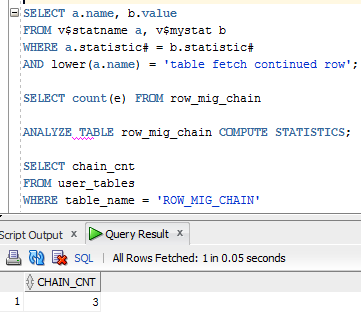


Here is 1, bcs we had to follow a pointer to the new location of the migrated row.

**Row chaining:**



Updated the first row, so now we don’t have enough memory in the first data block and the data should be chained.



I analyzed my table and saw that I had 3 chain counts in table: 2 is for migrated row and 1 for row chaining.