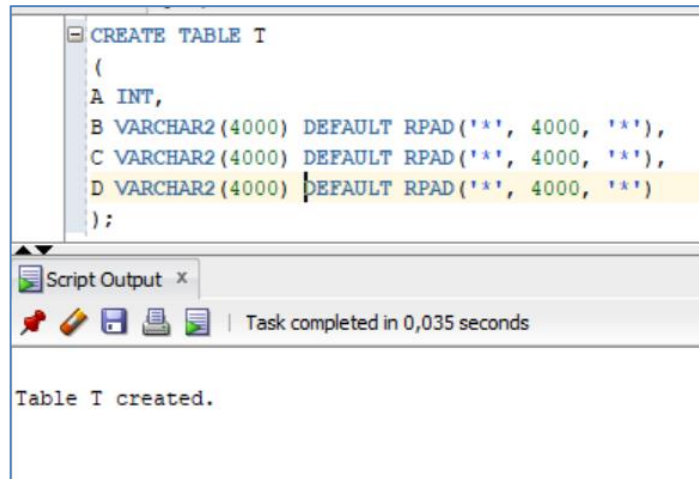


Lab Report #3

Sadovskaya Veronika

Task 1 –Heap Understanding

Step 1: create table T with 4 columns (figure 1.1).



```
CREATE TABLE T
(
  A INT,
  B VARCHAR2(4000) DEFAULT RPAD(' ', 4000, ' '),
  C VARCHAR2(4000) DEFAULT RPAD(' ', 4000, ' '),
  D VARCHAR2(4000) DEFAULT RPAD(' ', 4000, ' ')
);
```

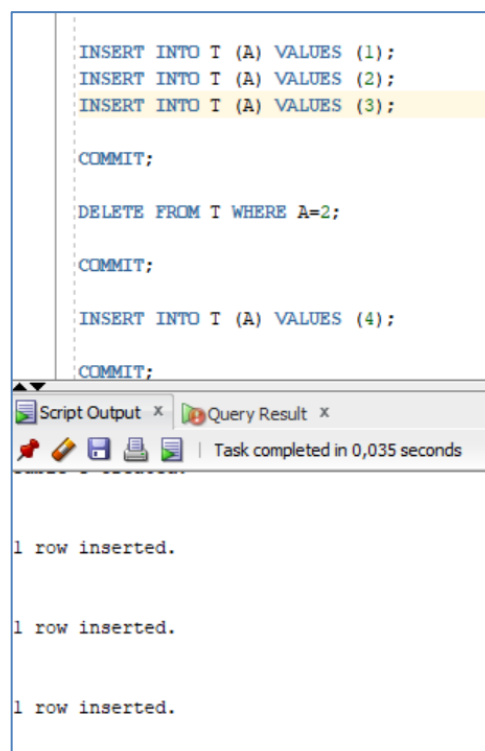
Script Output x

Task completed in 0,035 seconds

Table T created.

Figure 1.1

Step 2: insert into table T in column A values 1, 2, 3 (figure 1.2). Then we make a commit, delete the row in the table T where column A has a value of 2 and make a commit (figure 1.3). We insert into table T in column A value 4 and make commit (figure 1.4).



```
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;
```

Script Output x Query Result x

Task completed in 0,035 seconds

1 row inserted.

1 row inserted.

1 row inserted.

Figure 1.2

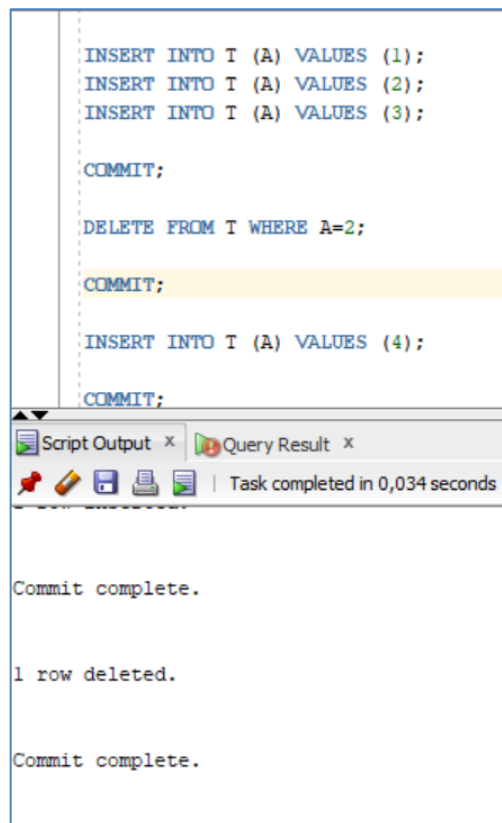


Figure 1.3

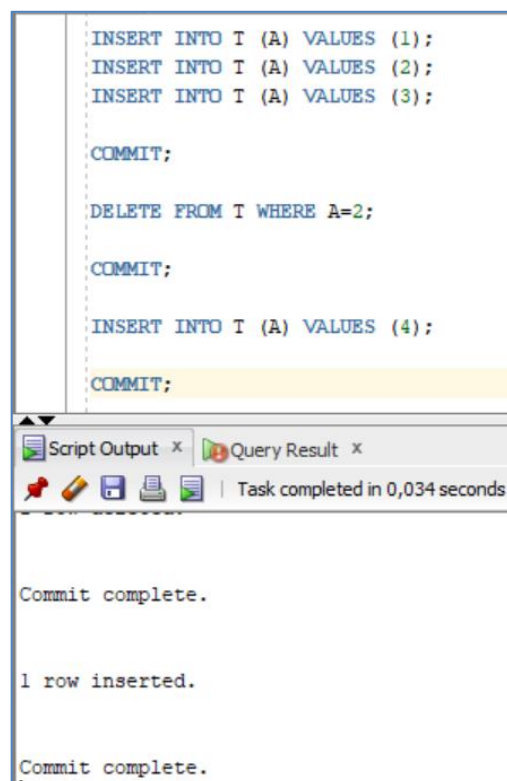


Figure 1.4

Step 3: make a selection from table T by column A (figure 1.5). Clear the table T (figure 1.6).

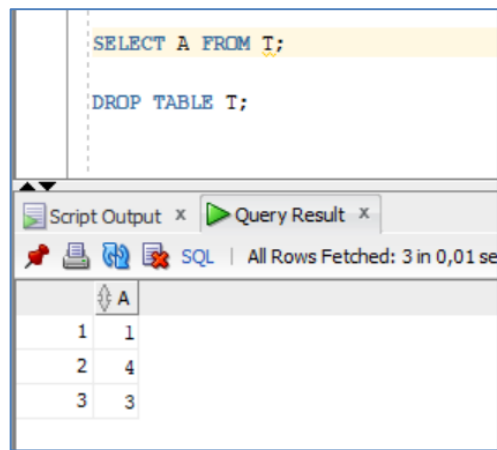


Figure 1.5

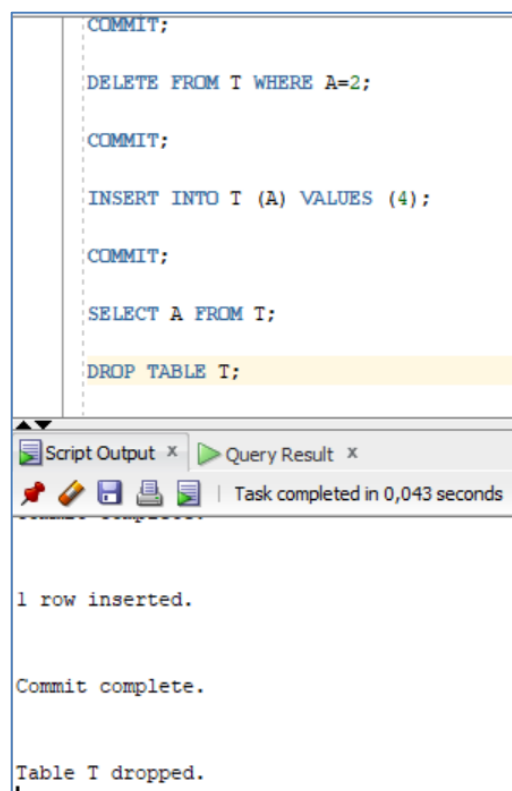


Figure 1.6

Heap organized tables: обычные, стандартные таблицы базы данных. Управление данными в этой таблице похоже на управление в heap. При добавлении данных используется первое найденное в сегменте свободное пространство, которое может уместить эти данные. При удалении данных из такой таблицы пространство, которое они занимали, становится доступным для повторного применения при вставке или обновлении.

Данные помещаются в таблицу в туда, где они лучше умещаются, а не в определенном порядке. При извлечении данных из таблицы порядок, в котором они туда поступали, может отличаться от порядка, в котором они

извлекаются. Данные в такой таблице представляют собой неупорядоченные коллекции данных

Task 2 – Understanding low level of data abstraction: Heap Table Segments

Step 1: Clear the table *USER_SEGMENTS* (figure 2.1). Create table *t* (figure 2.2).

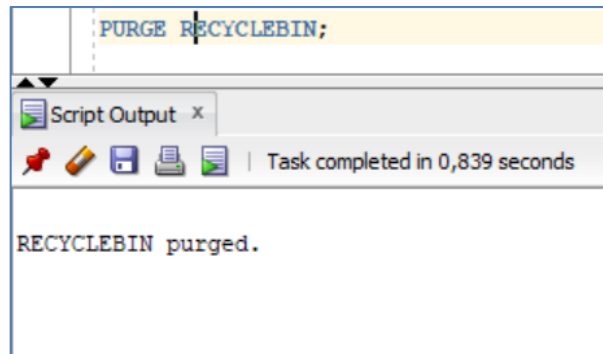


Figure 2.1

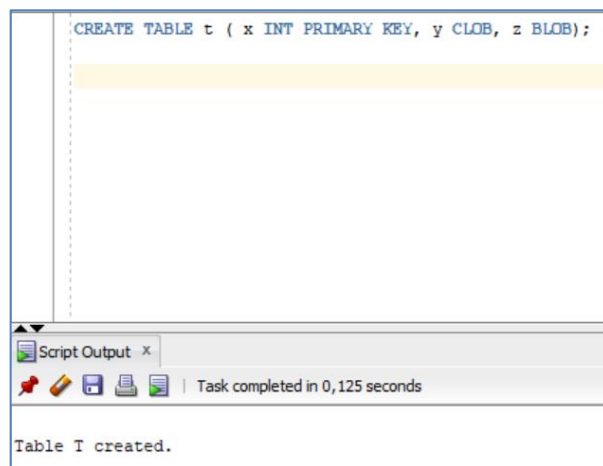


Figure 2.2

Step 2: result of selection from table *user_segments* (figure 2.3).

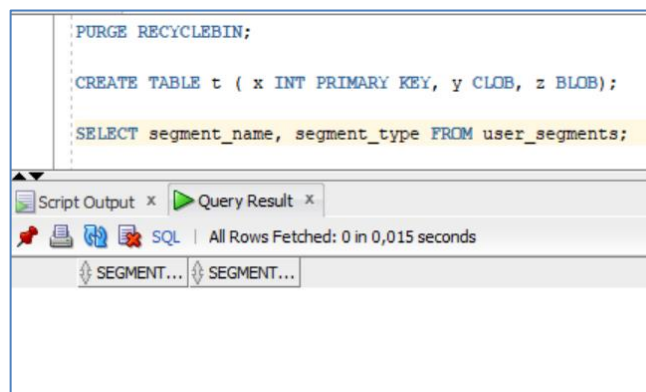


Figure 2.3

Step 3: to create the same table, but already add a *SEGMENT CREATION IMMEDIATE*, we need to drop the previously created table (figure 2.4).

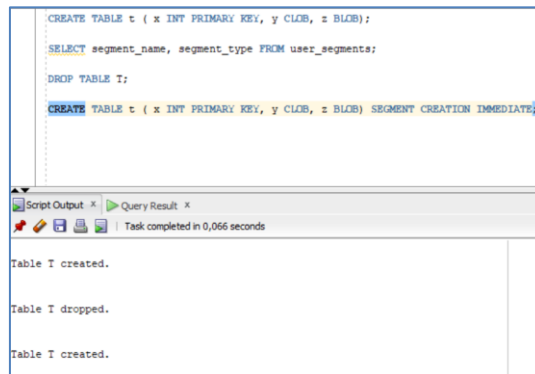


Figure 2.4

Step 4: as a result of selection, we see that 6 rows have been added to the table (figure 2.5-2.6). This is because the segment was forcibly allocated during table creation.

```

CREATE TABLE t ( x INT PRIMARY KEY, y CLOB, z BLOB) SEGMENT CREATION IMMEDIATE;
SELECT segment_name, segment_type FROM user_segments;
SELECT DBMS_METADATA.GET_DDL('TABLE','T') FROM dual;
  
```

Script Output x Query Result x

All Rows Fetched: 6 in 0,02 seconds

| | SEGMENT_NAME | SEGMENT_TYPE |
|---|-----------------------------|--------------|
| 1 | SYS_C007080 | INDEX |
| 2 | SYS_IL0000066685C00002\$\$ | LOBINDEX |
| 3 | SYS_IL0000066685C00003\$\$ | LOBINDEX |
| 4 | SYS_LOB0000066685C00002\$\$ | LOBSEGMENT |
| 5 | SYS_LOB0000066685C00003\$\$ | LOBSEGMENT |
| 6 | T | TABLE |

Figure 2.5

Step 5: query result (figure 2.6). Using the standard supplied package `DBMS_METADATA`, we query the definition of table and see the verbose syntax. The nice thing about this trick is that it shows many of the options for our `CREATE TABLE` statement. We just have to pick data types and such; Oracle will produce the verbose version for us.

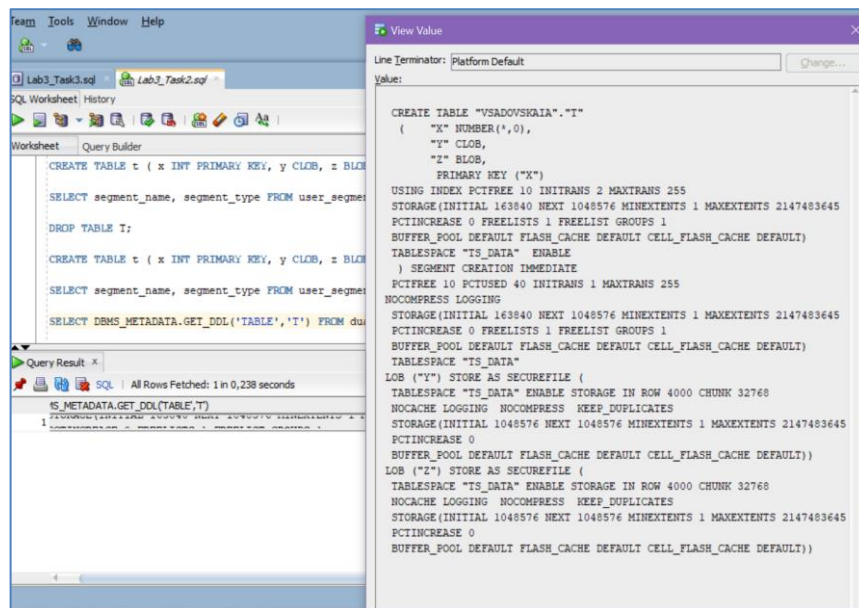


Figure 2.6

Task 3 – Compare performance of using IOT tables

Step 1: create table EMP (figure 3.1). Create index (figure 3.2). Calculate statistic (figure 3.3).

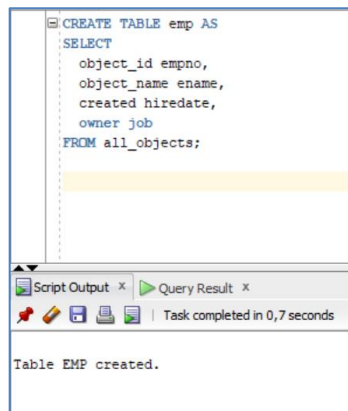


Figure 3.1

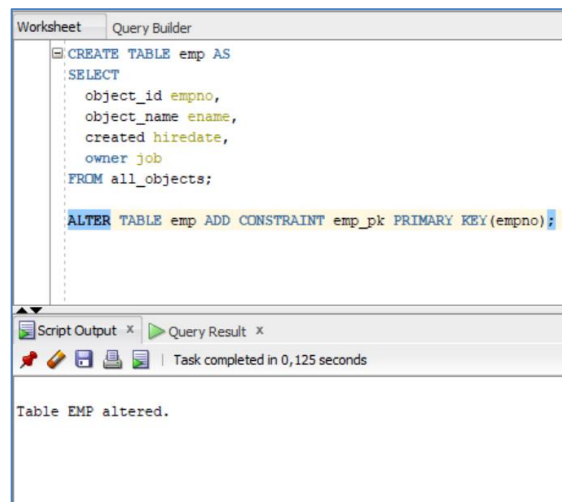


Figure 3.2

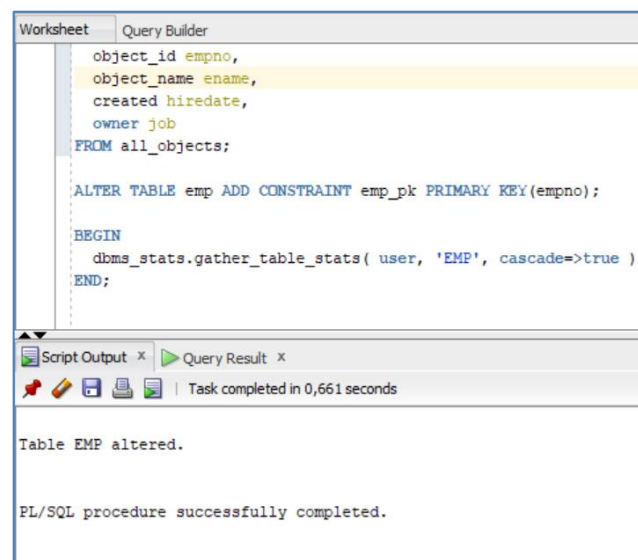


Figure 3.3

Step 2: implement the child table as a conventional heap table, HEAP_ADDRESSES (figure 3.4).

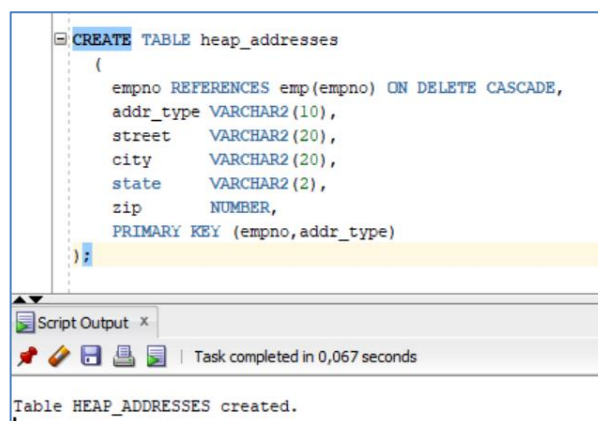


Figure 3.4

Step 3: implement the child table as an IOT, IOT_ADDRESSES (figure 3.5).

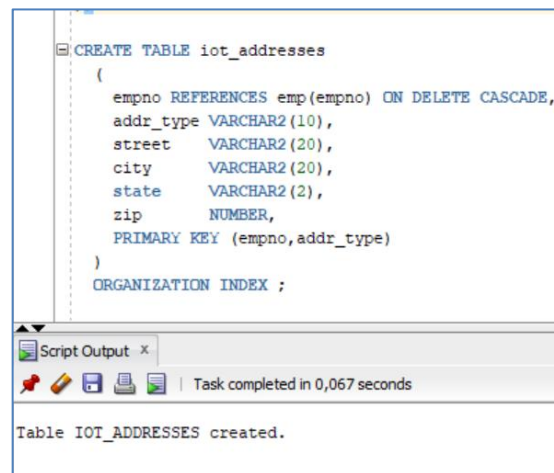


Figure 3.5

Step 4: populate HEAP_ADDRESSES and IOT_ADDRESSES tables (figure 3.6) by inserting into them a work address for each employee, then a home address, then a previous address, and finally a school address. 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.

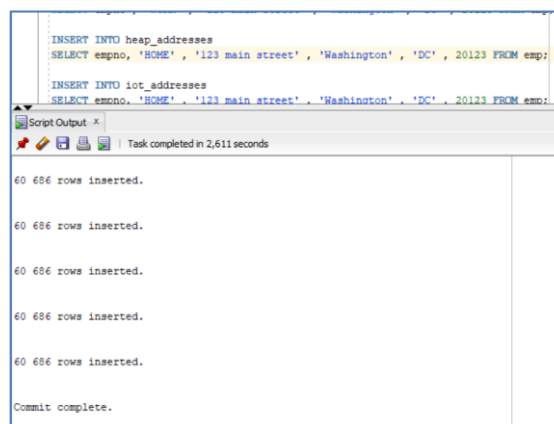


Figure 3.6

Step 5: Calculate statistic (figure 3.7)


```
EXEC dbms_stats.gather_table_stats( user, 'HEAP_ADDRESSES' );

EXEC dbms_stats.gather_table_stats( user, 'IOT_ADDRESSES' );
```

Script Output x

Task completed in 0,264 seconds

PL/SQL procedure successfully completed.

PL/SQL procedure successfully completed.

Figure 3.7

Step 6: Compare trace and performance. Plan table output for table HEAP_ADDRESSES is on figure 3.8. Plan table output for table IOT_ADDRESSES is on figure 3.9.

```
EXPLAIN PLAN FOR
SELECT * FROM emp, heap_addresses
WHERE emp.empno = heap_addresses.empno
AND emp.empno = 42;
SELECT * FROM TABLE(dbms_xplan.display);
```

Script Output x Query Result x

SQL All Rows Fetched: 18 in 0,093 seconds

PLAN_TABLE_OUTPUT

1 Plan hash value: 2136243436

2

3

| Id | Operation | Name | Rows | Bytes | Cost (%CPU) | Time |
|----|-------------------------------------|----------------|------|-------|-------------|----------|
| 0 | SELECT STATEMENT | | 4 | 400 | 7 (0) | 00:00:01 |
| 1 | NESTED LOOPS | | 4 | 400 | 7 (0) | 00:00:01 |
| 2 | TABLE ACCESS BY INDEX ROWID | EMP | 1 | 54 | 2 (0) | 00:00:01 |
| 3 | INDEX UNIQUE SCAN | EMP_PK | 1 | | 1 (0) | 00:00:01 |
| 4 | TABLE ACCESS BY INDEX ROWID BATCHED | HEAP_ADDRESSES | 4 | 184 | 5 (0) | 00:00:01 |
| 5 | INDEX RANGE SCAN | SYS_C007117 | 4 | | 1 (0) | 00:00:01 |

14 Predicate Information (identified by operation id):

15

Figure 3.8

```
EXPLAIN PLAN FOR
SELECT * FROM emp, iot_addresses
WHERE emp.empno = iot_addresses.empno
AND emp.empno = 42;
SELECT * FROM TABLE(dbms_xplan.display);
```

Script Output x Query Result x

SQL All Rows Fetched: 17 in 0,039 seconds

PLAN_TABLE_OUTPUT

1 Plan hash value: 1603601150

2

3

| Id | Operation | Name | Rows | Bytes | Cost (%CPU) | Time |
|----|-----------------------------|-------------------|------|-------|-------------|----------|
| 0 | SELECT STATEMENT | | 4 | 400 | 3 (0) | 00:00:01 |
| 1 | NESTED LOOPS | | 4 | 400 | 3 (0) | 00:00:01 |
| 2 | TABLE ACCESS BY INDEX ROWID | EMP | 1 | 54 | 2 (0) | 00:00:01 |
| 3 | INDEX UNIQUE SCAN | EMP_PK | 1 | | 1 (0) | 00:00:01 |
| 4 | INDEX RANGE SCAN | SYS_IOT_TOP_66721 | 4 | 184 | 1 (0) | 00:00:01 |

13 Predicate Information (identified by operation id):

14

15

Figure 3.8

Both queries fetched exactly the same number of rows, but the HEAP table performed considerably more logical I/O. As the degree of concurrency on the system goes up, we would likewise expect the CPU used by the HEAP table to go up more rapidly.

After comparing the prices of queries from IOT and Heap table($COST_IOT < COST_HEAP$), we came to the obvious result that queries from IOT tables are processed faster, because we know this advantages of an IOT:

- As an IOT has the structure of an index and stores all the columns of the row, accesses via primary key conditions are faster as they don't need to access the table to get additional column values.
- As an IOT has the structure of an index and is thus sorted in the order of the primary key, accesses of a range of primary key values are also faster.
- As the index and the table are in the same segment, less storage space is needed.

Index organized table: таблицы, которые хранятся в индексной структуре. Тогда, как в heap table данные никак не упорядочены, в IOT данные сохранены и отсортированы по первичному ключу. Концепция IOT: индекс – данные, данные – индекс. Индекс содержит физический адрес строки, на которую он указывает, т.е. идентификатор строки (rowid). Структура индекса представляет собой дерево, а листовые блоки (хранящие данные) в действительности являются двусвязным списком, упрощающим перемещение по узлам по порядку после того, как мы найдем место с которого хотим начать.

Task 4 – Analyses cluster storage by blocks

Step 1: create cluster (figure 4.1). A CREATE CLUSTER statement looks a lot like a CREATE TABLE statement with a small number of columns (just the cluster key columns).

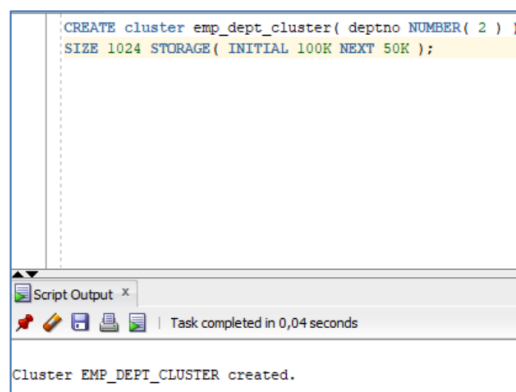


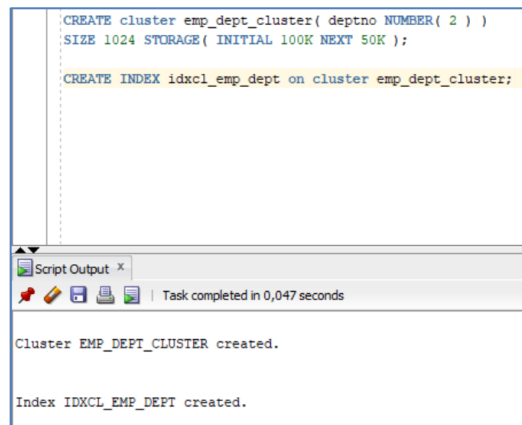
Рисунок 4.1

we have created an index cluster (figure 4.1). The clustering column for this cluster will be the DEPTNO column. SIZE 1024 option used to tell Oracle that we expect about 1,024 bytes of data to be associated with each cluster key value.

Oracle will use that to compute the maximum number of cluster keys that could fit per block. The SIZE parameter therefore controls the maximum number of cluster keys per block. It is the single largest influence on the space utilization of our cluster. Set the size too high, and we'll get very few keys per block and we'll use more space than we need. Set the size too low, and we'll get excessive chaining of data, which offsets the purpose of the cluster to store all of the data together on a single block. It is the most important parameter for a cluster.

Step 2: create cluster key (figure 4.2).

The cluster index's job is to take a cluster key value and return the block address of the block that contains that key. It is a primary key, in effect, where each cluster key value points to a single block in the cluster itself.



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

CREATE INDEX idxcl_emp_dept on cluster emp_dept_cluster;
```

Script Output x

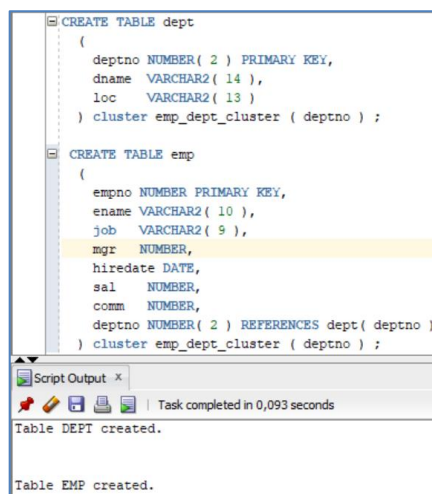
Task completed in 0,047 seconds

Cluster EMP_DEPT_CLUSTER created.

Index IDXCL_EMP_DEPT created.

Рисунок 4.2

Step 3: create tables dept and emp (figure 4.3). The only difference from a normal table is that we used the CLUSTER keyword and told Oracle which column of the base table will map to the cluster key in the cluster itself.



```
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 );
```

Script Output x

Task completed in 0,093 seconds

Table DEPT created.

Table EMP created.

Рисунок 4.3

Step 4: initialize inserting rows (figure 4.4).

```
INSERT INTO DEPT VALUES (10,'ACCOUNTING','NEW YORK');
INSERT INTO DEPT VALUES (20,'RESEARCH','DALLAS');
INSERT INTO DEPT VALUES (30,'SALES','CHICAGO');
INSERT INTO DEPT VALUES (40,'OPERATIONS','BOSTON');

COMMIT;

INSERT INTO EMP VALUES
(7369,'SMITH','CLERK',7902,to_date('17-12-1980','dd-mm-yyyy'),800,NULL,20);
INSERT INTO EMP VALUES
(7499,'ALLEN','SALESMAN',7698,to_date('20-2-1981','dd-mm-yyyy'),1600,300,30);
INSERT INTO EMP VALUES
(7521,'WARD','SALESMAN',7698,to_date('22-2-1981','dd-mm-yyyy'),1250,500,30);
INSERT INTO EMP VALUES
(7566,'JONES','MANAGER',7839,to_date('2-4-1981','dd-mm-yyyy'),2975,NULL,20);
```

Script Output x Query Result x

Task completed in 0,137 seconds

1 row inserted.

1 row inserted.

Рисунок 4.4

Step 5: selection (figure 4.5). That was exactly our goal—to get every row in the EMP table stored on the same block as the corresponding DEPT row.

```
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;
```

Script Output x Query Result x

All Rows Fetched: 12 in 0,04 seconds

| | DEPT_BLK | EMP_BLK | FLAG | DEPTNO |
|----|----------|-------------|------|--------|
| 1 | 3100 | 3100 (null) | | 10 |
| 2 | 3100 | 3100 (null) | | 10 |
| 3 | 3100 | 3100 (null) | | 10 |
| 4 | 3100 | 3100 (null) | | 20 |
| 5 | 3100 | 3100 (null) | | 20 |
| 6 | 3100 | 3100 (null) | | 20 |
| 7 | 3100 | 3100 (null) | | 30 |
| 8 | 3100 | 3100 (null) | | 30 |
| 9 | 3100 | 3100 (null) | | 30 |
| 10 | 3100 | 3100 (null) | | 30 |
| 11 | 3100 | 3100 (null) | | 30 |
| 12 | 3100 | 3100 (null) | | 30 |

Рисунок 4.5

Step 6: drop tables emp and dept (figure 4.6).

```
DROP TABLE emp;

DROP TABLE dept;
```

Script Output x Query Result x

Task completed in 0,052 seconds

Table EMP dropped.

Table DEPT dropped.

Рисунок 4.6

Clusters are groups of one or more tables, physically stored on the same database blocks, with all rows that share a common cluster key value being stored physically near each other. Two goals are achieved in this structure. First, many tables may be stored physically joined together. Normally, you would expect data from only one table to be found on a database block, but with clustered tables, data from many tables may be stored on the same block. Second, all data that contains the same cluster key value, such as DEPTNO = 10, will be physically stored together. The data is clustered around the cluster key value. A cluster key is built using a B*Tree index.

Clusters are used for storing related data from many tables in the same database block. Clusters can accelerate the implementation of intensive in terms of read operations that always join data together or access related sets of data. Clustered index tables reduce the number of blocks that Oracle has to cache. Instead of allocating ten blocks for ten employees from a single Department, Oracle will put them in a single block and thus increase the efficiency of the buffer cache.

As for the disadvantages: if the value for the SIZE parameter is calculated incorrectly, clusters may be inefficient in terms of space utilization and slow down the execution of DML operations.

Index clustered table: Кластеры - это группы, состоящие из одной или более таблиц, которые физически хранятся в тех же самых блоках базы данных, при этом все строки в них разделяют общее значение кластерного ключа и физически находятся близко друг к другу. Кластерный ключ строится с применением индекса со структурой B-дерева. Обычно в блоке базы данных можно ожидать нахождения данных только из одной таблицы, но в случае кластеризованных таблиц в одном блоке могут храниться данные из нескольких таблиц. Кластер не хранит данные в отсортированном порядке - это задача индекс-таблицы. Он хранит данные, кластеризованные по какому-то ключу, но в куче. Значение, по которому происходит деление на блоки., факторизуется и сохраняется один раз. Затем в этом блоке сохраняются все данные из всех таблиц кластера для этого значения. Если все данные для какого-то значения не укладываются в один блок, то с ним сцепляются дополнительные блоки, которые будут содержать переполнение.

Большинство связанных с объектами данных хранится в единственном кластере, совместно использующих один и тот же блок. В них содержится преимущественно информация, связанная со столбцами, так что вся информация о наборе столбцов любой таблицы или индекса хранится физически в одном и том же блоке. В этом есть смысл, поскольку во время

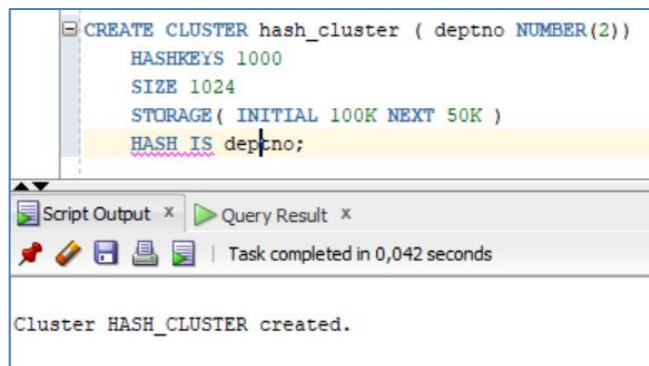
разбора запроса Oracle необходим доступ к данным для всех столбцов в таблице, на которую ссылается запрос. Если бы эти данные были разбросаны по разным местам, требовалось бы определенное время на то, чтобы собрать их вместе.

Кластер является подходящим вариантом, если есть данные, которые по большей части читаются (это вовсе не означает, что они никогда не перезаписываются ; кластерные таблицы вполне допускают модификацию), читаются через индексы - либо индекс на кластерном ключе, либо другие индексы, которые были предусмотрены на таблицах в кластере - и часто соединяются вместе. Ищутся таблицы, которые логически связаны между собой и всегда используются вместе, как поступили разработчики словаря данных Oracle, когда поместили в кластер всю информацию, связанную со столбцами.

Кластеризованные индекс-таблицы предоставляют возможность физически предварительно соединять данные. Кластеры применяются для хранения связанных между собой данных из многих таблиц в одном и том же блоке базы данных. Кластеры могут ускорять выполнение интенсивных в плане чтения операций, которые всегда соединяют данные вместе или получают доступ к связанным наборам данных. Кластеризованные индекс-таблицы сокращают количество блоков, которые Oracle приходится кэшировать. Вместо того чтобы выделять десять блоков для десяти сотрудников из одного отдела, Oracle поместит их в один блок и тем самым увеличит эффективность кеша буферов. Что касается недостатков: если значение для параметра SIZE подсчитано некорректно, то кластеры могут оказаться неэффективными в отношении утилизации пространства.

Task 5 – Analyses cluster storage by blocks (hash)

Step 1: create cluster (figure 5.1). When we create a hash cluster, we'll use the same CREATE CLUSTER statement we used to create the index cluster with different options. we'll just be adding a HASHKEYS option to it to specify the size of the hash table. Oracle will take our HASHKEYS value and round it up to the nearest prime number; the number of hash keys will always be a prime. Oracle will then compute a value based on the SIZE parameter multiplied by the modified HASHKEYS value. It will allocate at least that much space in bytes for the cluster. This is a big difference from the preceding index cluster, which dynamically allocates space as it needs it. A hash cluster pre-allocates enough space to hold (HASHKEYS/trunc(blocksize/SIZE)) bytes of data.



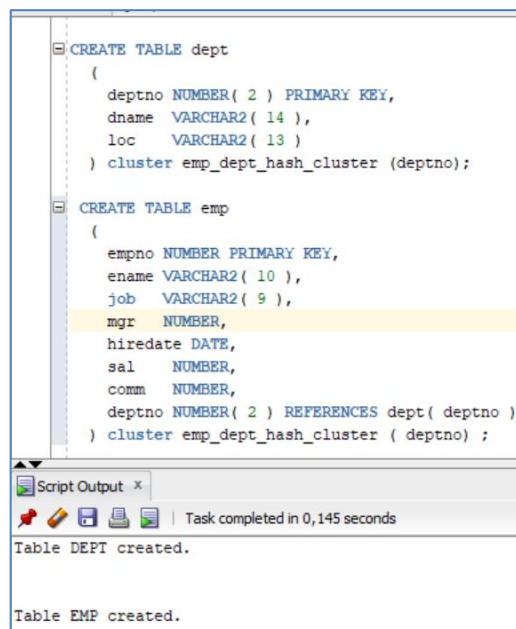
The screenshot shows a SQL script editor with the following code:

```
CREATE CLUSTER hash_cluster ( deptno NUMBER(2))
  HASHKEYS 1000
  SIZE 1024
  STORAGE( INITIAL 100K NEXT 50K )
  HASH IS deptno;
```

Below the script editor, the 'Script Output' tab is active, displaying the message: 'Cluster HASH_CLUSTER created.' The status bar indicates 'Task completed in 0,042 seconds'.

Рисунок 5.1

Step 2: create tables dept and emp (figure 5.2).



The screenshot shows a SQL script editor with the following code:

```
CREATE TABLE dept
(
  deptno NUMBER( 2 ) PRIMARY KEY,
  dname VARCHAR2( 14 ),
  loc VARCHAR2( 13 )
) cluster emp_dept_hash_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_hash_cluster ( deptno );
```

Below the script editor, the 'Script Output' tab is active, displaying the messages: 'Table DEPT created.' and 'Table EMP created.' The status bar indicates 'Task completed in 0,145 seconds'.

Рисунок 5.2

Step 3: initialize inserting rows (figure 5.3).

```

INSERT INTO DEPT VALUES (20,'RESEARCH','DALLAS');
INSERT INTO DEPT VALUES (30,'SALES','CHICAGO');
INSERT INTO DEPT VALUES (40,'OPERATIONS','BOSTON');

COMMIT;

INSERT INTO EMP VALUES
(7369,'SMITH','CLERK',7902,to_date('17-12-1980','dd-mm-yyyy'),800,NULL,20);
INSERT INTO EMP VALUES
(7499,'ALLEN','SALESMAN',7698,to_date('20-2-1981','dd-mm-yyyy'),1600,300,30);
INSERT INTO EMP VALUES
(7521,'WARD','SALESMAN',7698,to_date('22-2-1981','dd-mm-yyyy'),1250,500,30);
INSERT INTO EMP VALUES
(7566,'JONES','MANAGER',7839,to_date('2-4-1981','dd-mm-yyyy'),2975,NULL,20);
INSERT INTO EMP VALUES

```

Script Output x | Task completed in 0,212 seconds

1 row inserted.

1 row inserted.

Commit complete.

Рисунок 5.3

Step 4: selection (figure 5.4).

```

SELECT * FROM
(SELECT dept_blk, emp_blk, CASE WHEN dept_blk <> emp_blk THEN '' END flag, deptno
FROM (SELECT dmsa_rowid.rowid_block_number( dept.rowid ) dept_blk,
dmsa_rowid.rowid_block_number( emp.rowid ) emp_blk, dept.deptno
FROM emp , dept
WHERE emp.deptno = dept.deptno)
) ORDER BY deptno;

```

Script Output x | Query Result x | All Rows Fetched: 12 in 0,341 seconds

| | DEPT_BLK | EMP_BLK | FLAG | DEPTNO |
|----|----------|---------|--------|--------|
| 1 | 7086 | 7086 | (null) | 10 |
| 2 | 7086 | 7086 | (null) | 10 |
| 3 | 7086 | 7086 | (null) | 10 |
| 4 | 7081 | 7081 | (null) | 20 |
| 5 | 7081 | 7081 | (null) | 20 |
| 6 | 7081 | 7081 | (null) | 20 |
| 7 | 7107 | 7107 | (null) | 30 |
| 8 | 7107 | 7107 | (null) | 30 |
| 9 | 7107 | 7107 | (null) | 30 |
| 10 | 7107 | 7107 | (null) | 30 |
| 11 | 7107 | 7107 | (null) | 30 |
| 12 | 7107 | 7107 | (null) | 30 |

Рисунок 5.4

Step 5: drop tables (figure 5.5).

```

DROP TABLE emp;

DROP TABLE dept;

```

Script Output x | Query Result x | Task completed

Table EMP dropped.

Table DEPT dropped.

Рисунок 5.5

Hash clustered table: очень похожи на кластеризованные индекс-таблицы за одним главным исключением: индекс по кластерному ключу заменяется хеш-функцией. Данные в такой таблице являются индексом; какого-либо физического индекса не существует. База данных Oracle будет брать значение ключа для строки, хешировать его с помощью либо внутренней, либо предоставленной нами функции, и затем использовать его для выяснения, где данные должны находиться на диске. Однако одним из недостатков применения алгоритма хеширования для определения местонахождения данных является то, что выполнить просмотр таблицы в хеш-кластере по диапазону, не добавив в нее обычный индекс, не удастся.