Relational Databases

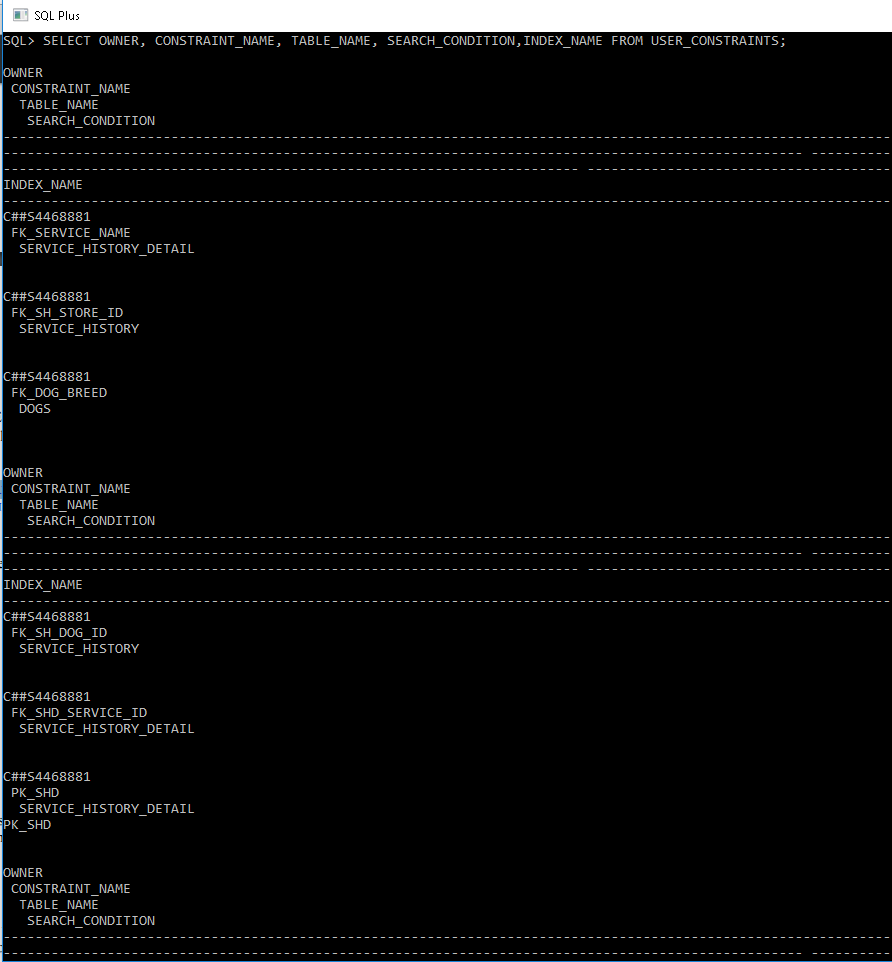
# Oracle SQL Assignment

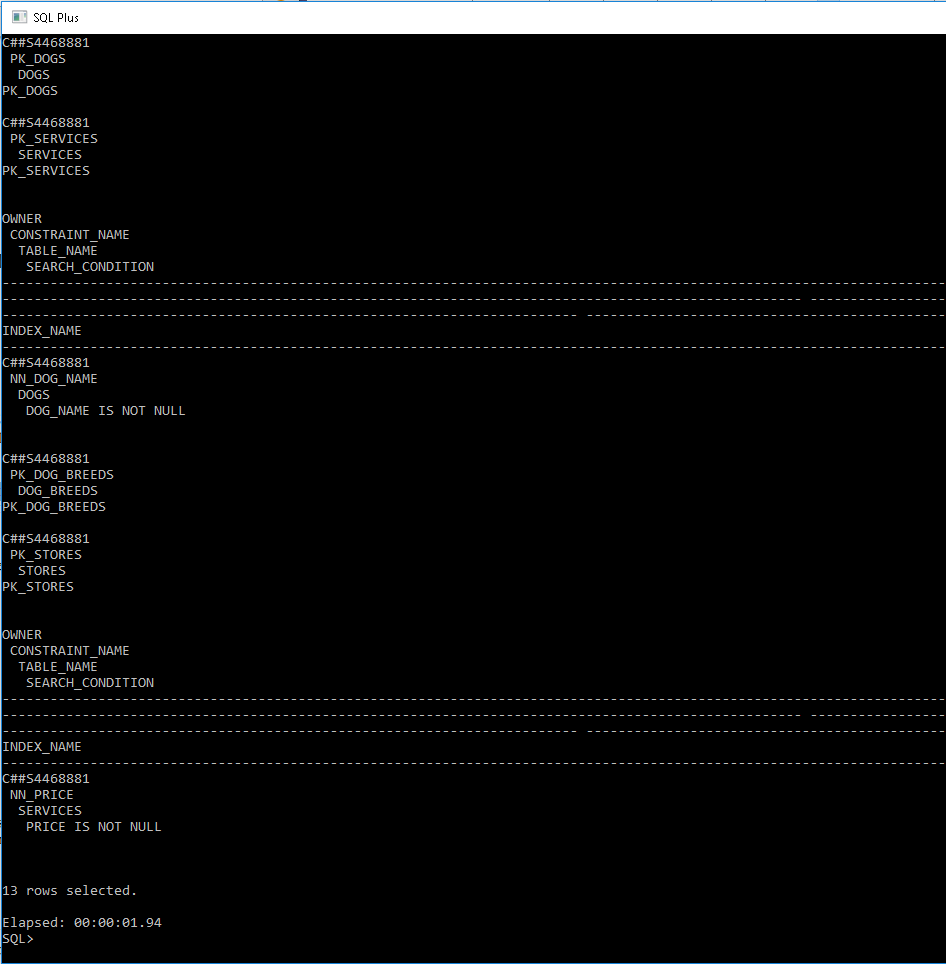
# Doggo Paradise

# Task 1: Database Constraints

## A: All Constraints on database

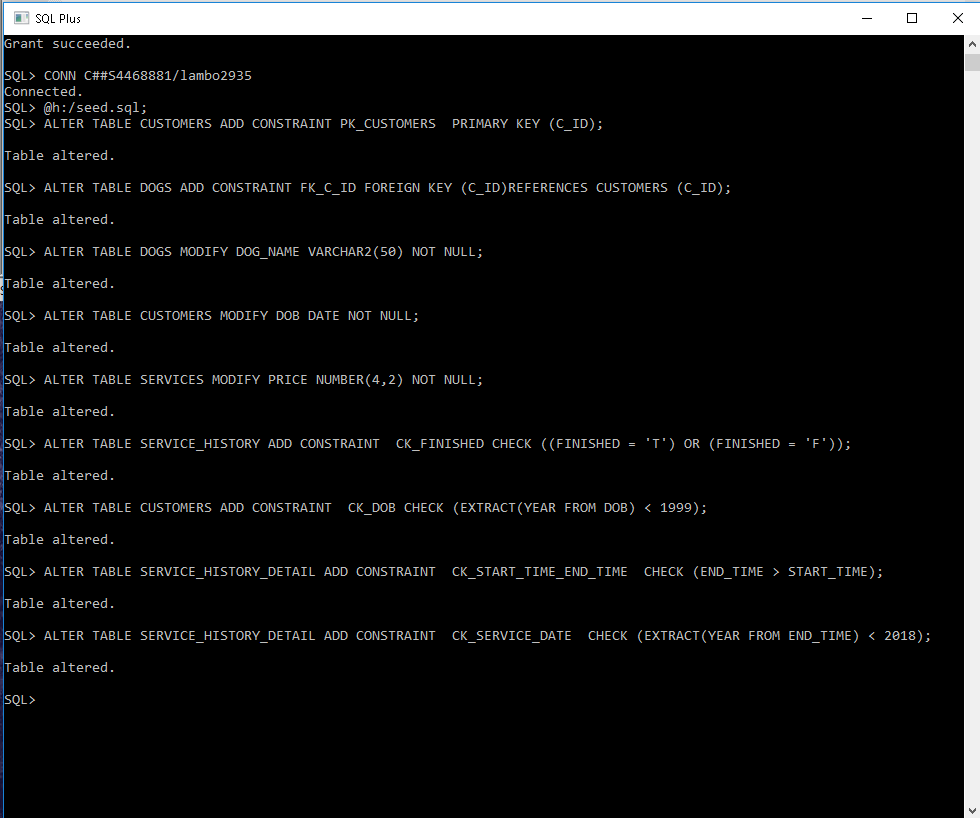
**SELECT OWNER**, **CONSTRAINT\_NAME**, **TABLE\_NAME**, SEARCH\_CONDITION,  
INDEX\_NAME **FROM** USER\_CONSTRAINTS;





## B: Missing Constraints

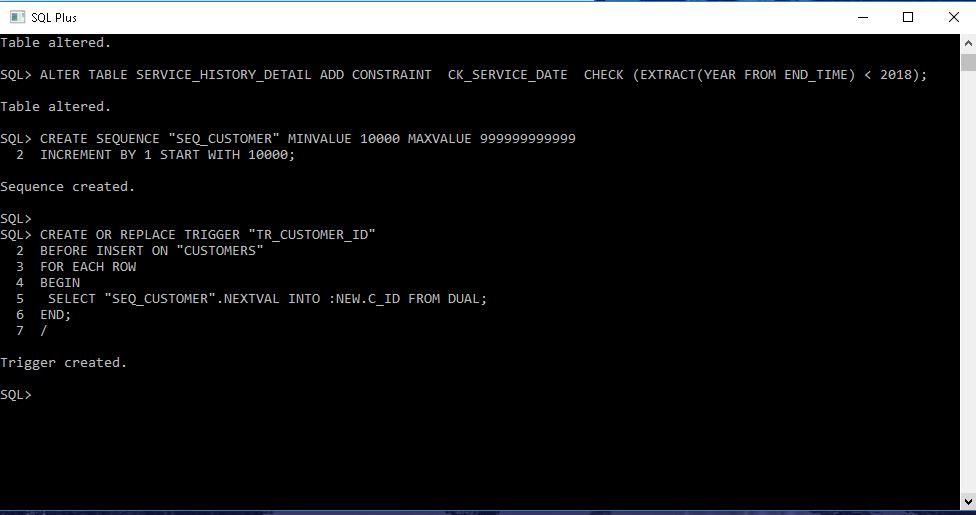
**ALTER TABLE** CUSTOMERS   
**ADD CONSTRAINT** PK\_CUSTOMERS **PRIMARY KEY** (C\_ID);   
  
**ALTER TABLE** DOGS **ADD CONSTRAINT** FK\_C\_ID **FOREIGN KEY** (C\_ID)  
**REFERENCES** CUSTOMERS (C\_ID);   
  
**ALTER TABLE** DOGS **MODIFY** DOG\_NAME VARCHAR2(50) **NOT NULL**;  
  
**ALTER TABLE** CUSTOMERS **MODIFY** DOB **DATE NOT NULL**;  
  
**ALTER TABLE** SERVICES **MODIFY** PRICE **NUMBER**(4,2) **NOT NULL**;  
  
**ALTER TABLE** SERVICE\_HISTORY **ADD CONSTRAINT** CK\_FINISHED   
**CHECK** ((FINISHED = 'T') **OR** (FINISHED = 'F'));   
  
**ALTER TABLE** CUSTOMERS **ADD CONSTRAINT** CK\_DOB   
**CHECK** (*EXTRACT*(**YEAR FROM** DOB) < 1999);   
  
**ALTER TABLE** SERVICE\_HISTORY\_DETAIL **ADD CONSTRAINT** CK\_START\_TIME\_END\_TIME   
**CHECK** (END\_TIME > START\_TIME);   
  
**ALTER TABLE** SERVICE\_HISTORY\_DETAIL **ADD CONSTRAINT** CK\_SERVICE\_DATE   
**CHECK** (*EXTRACT*(**YEAR FROM** END\_TIME) < 2018);



# Task 2: Triggers

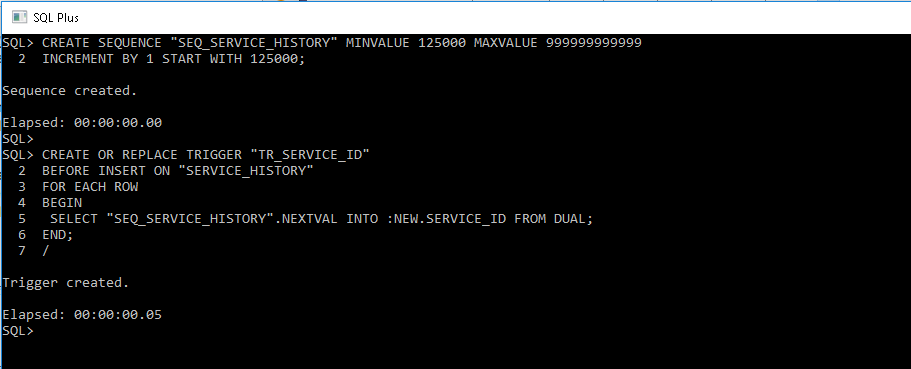
## A: SEQ\_CUSTOMER AND TR\_CUSTOMER\_ID

**CREATE** SEQUENCE "SEQ\_CUSTOMER" MINVALUE 10000 **MAXVALUE** 999999999999  
INCREMENT **BY** 1 **START WITH** 10000;  
  
**CREATE OR REPLACE TRIGGER** "TR\_CUSTOMER\_ID"  
**BEFORE INSERT ON** "CUSTOMERS"  
**FOR EACH ROW  
BEGIN  
 SELECT** "SEQ\_CUSTOMER".NEXTVAL **INTO** :NEW.C\_ID **FROM DUAL**;  
**END**;  
/



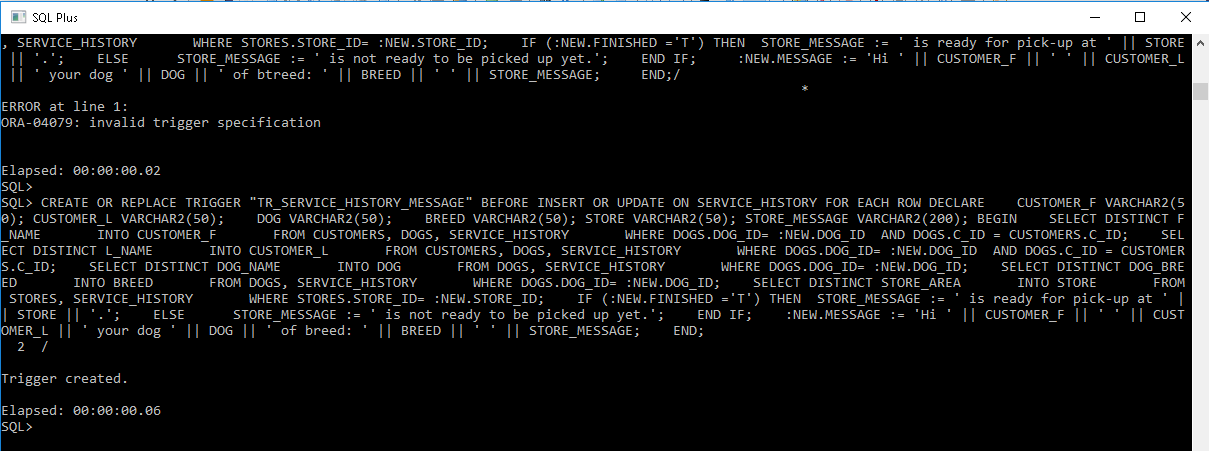
## B: SEQ\_SERVICE\_HISTORY AND TR\_SERVICE\_ID

**CREATE** SEQUENCE "SEQ\_SERVICE\_HISTORY" MINVALUE 125000 **MAXVALUE** 999999999999  
INCREMENT **BY** 1 **START WITH** 125000;  
  
**CREATE OR REPLACE TRIGGER** "TR\_SERVICE\_ID"  
**BEFORE INSERT ON** "SERVICE\_HISTORY"  
**FOR EACH ROW  
BEGIN  
 SELECT** "SEQ\_SERVICE\_HISTORY".NEXTVAL **INTO** :NEW.SERVICE\_ID **FROM DUAL**;  
**END**;  
/



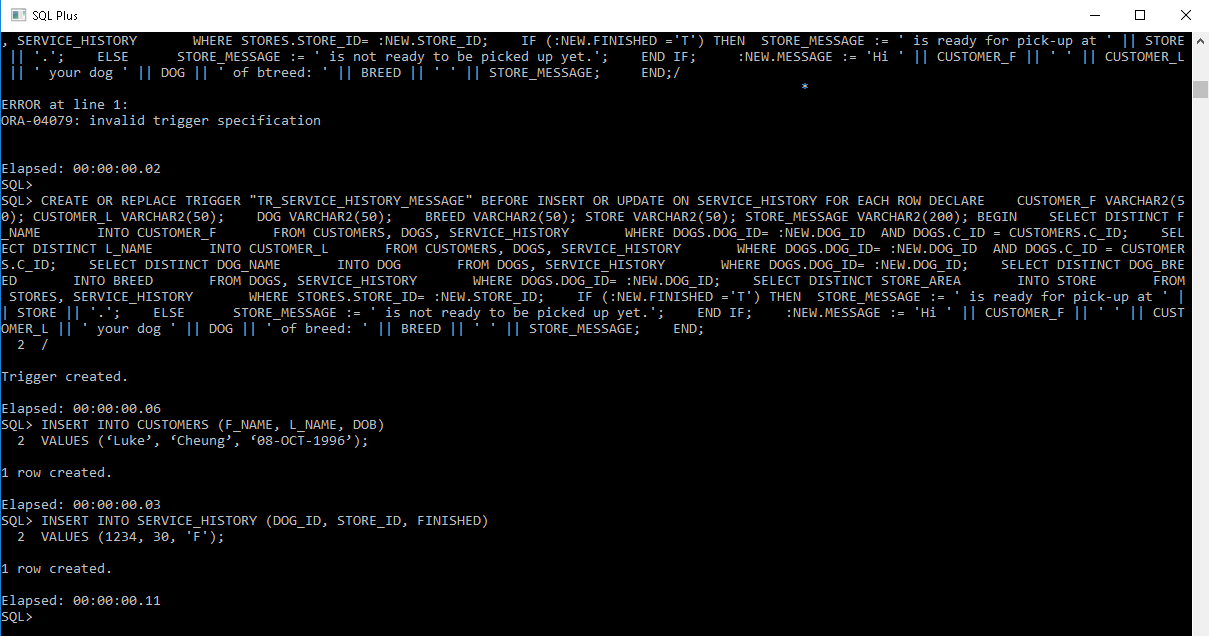
## C: TR\_SERVICE\_HISTORY\_MESSAGE

**CREATE OR REPLACE TRIGGER** "TR\_SERVICE\_HISTORY\_MESSAGE"  
**BEFORE INSERT OR UPDATE ON** SERVICE\_HISTORY  
**FOR EACH ROW  
DECLARE** CUSTOMER\_F VARCHAR2(50);  
 CUSTOMER\_L VARCHAR2(50);  
 DOG VARCHAR2(50);  
 BREED VARCHAR2(50);  
 STORE VARCHAR2(50);  
 STORE\_MESSAGE VARCHAR2(200);  
**BEGIN  
  
 SELECT DISTINCT** F\_NAME   
 **INTO** CUSTOMER\_F   
 **FROM** CUSTOMERS, DOGS, SERVICE\_HISTORY   
 **WHERE** DOGS.DOG\_ID= :NEW.DOG\_ID  
 **AND** DOGS.C\_ID = CUSTOMERS.C\_ID;  
   
 **SELECT DISTINCT** L\_NAME   
 **INTO** CUSTOMER\_L   
 **FROM** CUSTOMERS, DOGS, SERVICE\_HISTORY   
 **WHERE** DOGS.DOG\_ID= :NEW.DOG\_ID  
 **AND** DOGS.C\_ID = CUSTOMERS.C\_ID;  
   
 **SELECT DISTINCT** DOG\_NAME   
 **INTO** DOG   
 **FROM** DOGS, SERVICE\_HISTORY   
 **WHERE** DOGS.DOG\_ID= :NEW.DOG\_ID;  
   
 **SELECT DISTINCT** DOG\_BREED   
 **INTO** BREED   
 **FROM** DOGS, SERVICE\_HISTORY   
 **WHERE** DOGS.DOG\_ID= :NEW.DOG\_ID;  
   
 **SELECT DISTINCT** STORE\_AREA   
 **INTO** STORE   
 **FROM** STORES, SERVICE\_HISTORY   
 **WHERE** STORES.STORE\_ID= :NEW.STORE\_ID;  
  
 **IF** (:NEW.FINISHED ='T') **THEN** STORE\_MESSAGE := ' is ready for pick-up at ' || STORE || '.';  
 **ELSE** STORE\_MESSAGE := ' is not ready to be picked up yet.';  
 **END IF**;  
  
 :NEW.MESSAGE := 'Hi ' || CUSTOMER\_F || ' ' || CUSTOMER\_L || ' your dog ' || DOG || ' of breed: ' || BREED || ' ' || STORE\_MESSAGE;  
  
 **END**;  
/



## D: TRIGGER TESTING

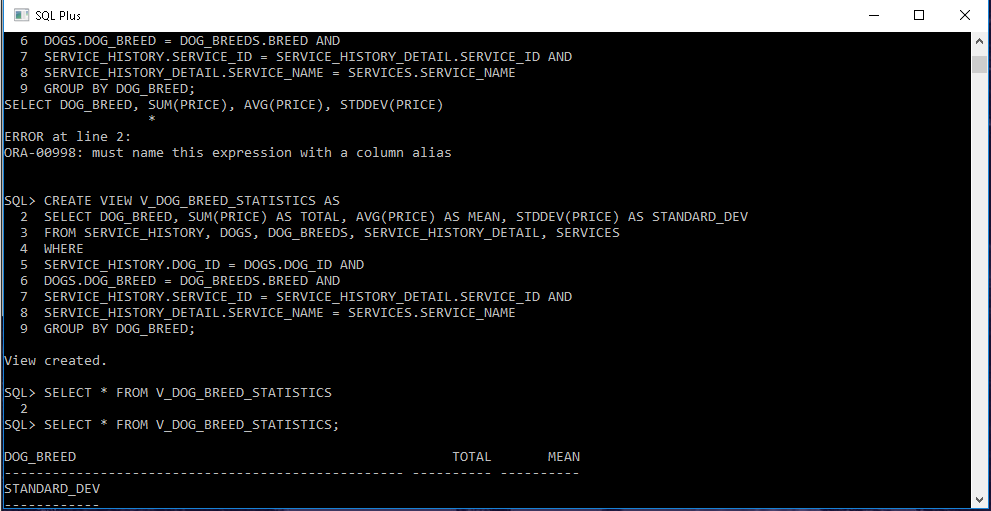
**INSERT INTO** CUSTOMERS (F\_NAME, L\_NAME, DOB)  
**VALUES** (‘Luke’, ‘Cheung’, ‘08-OCT-1996’);  
  
**INSERT INTO** SERVICE\_HISTORY (DOG\_ID, STORE\_ID, FINISHED)  
**VALUES** (1234, 30, 'F');



# Task 3: VIEWS

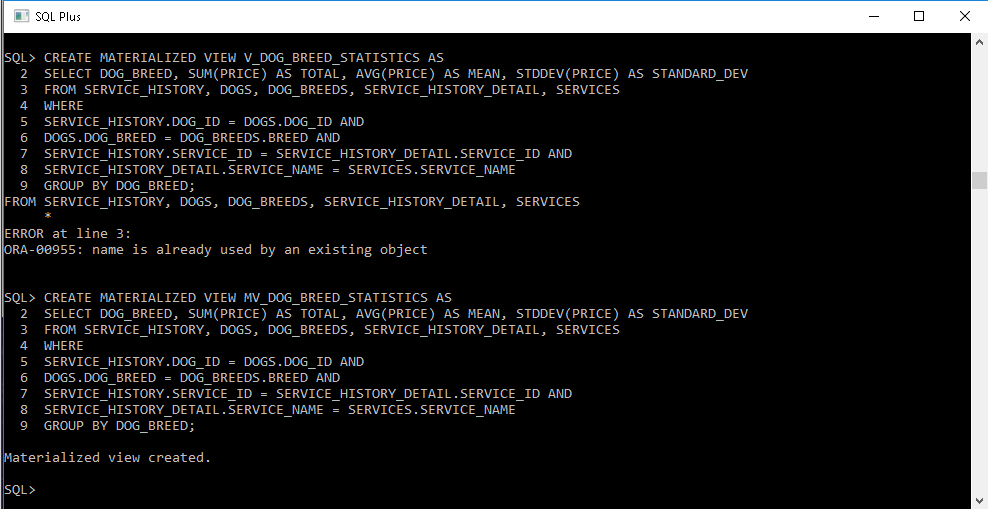
## A: V\_DOG\_BREED\_STATISTICS

**CREATE VIEW** V\_DOG\_BREED\_STATISTICS **AS  
SELECT** DOG\_BREED, *SUM*(PRICE) **AS** TOTAL, *AVG*(PRICE) **AS** MEAN, *STDDEV*(PRICE) **AS** STANDARD\_DEV  
**FROM** SERVICE\_HISTORY, DOGS, DOG\_BREEDS, SERVICE\_HISTORY\_DETAIL, SERVICES  
**WHERE**SERVICE\_HISTORY.DOG\_ID = DOGS.DOG\_ID **AND**DOGS.DOG\_BREED = DOG\_BREEDS.BREED **AND**SERVICE\_HISTORY.SERVICE\_ID = SERVICE\_HISTORY\_DETAIL.SERVICE\_ID **AND**SERVICE\_HISTORY\_DETAIL.SERVICE\_NAME = SERVICES.SERVICE\_NAME  
**GROUP BY** DOG\_BREED;

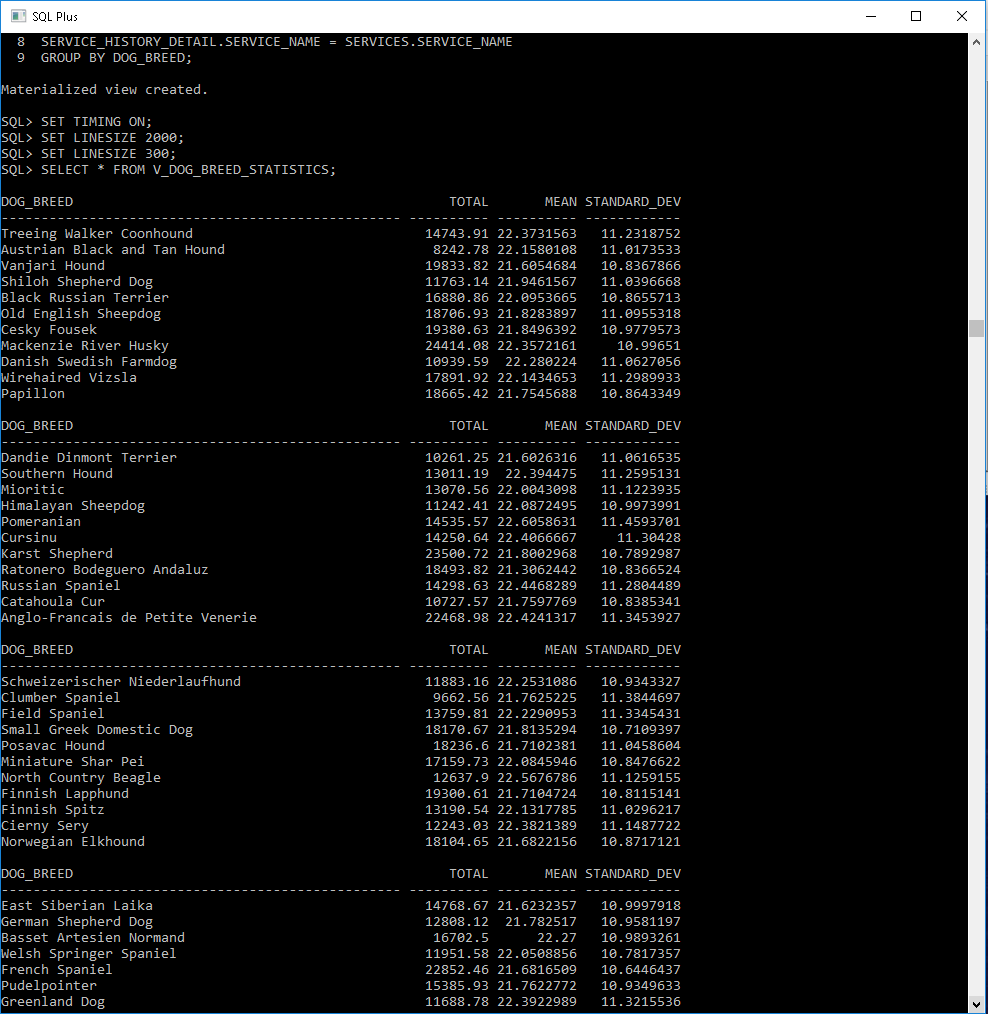


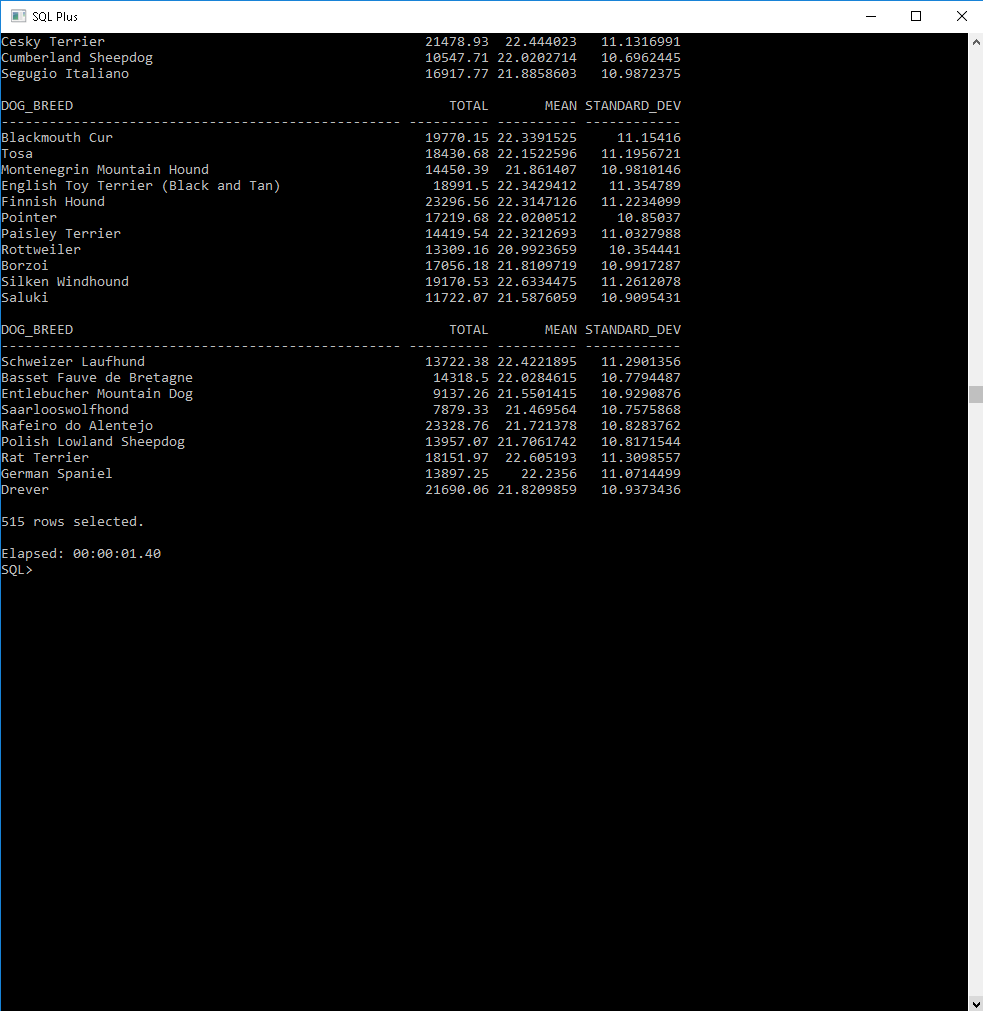
## B: MV\_DOG\_BREED\_STATISTICS

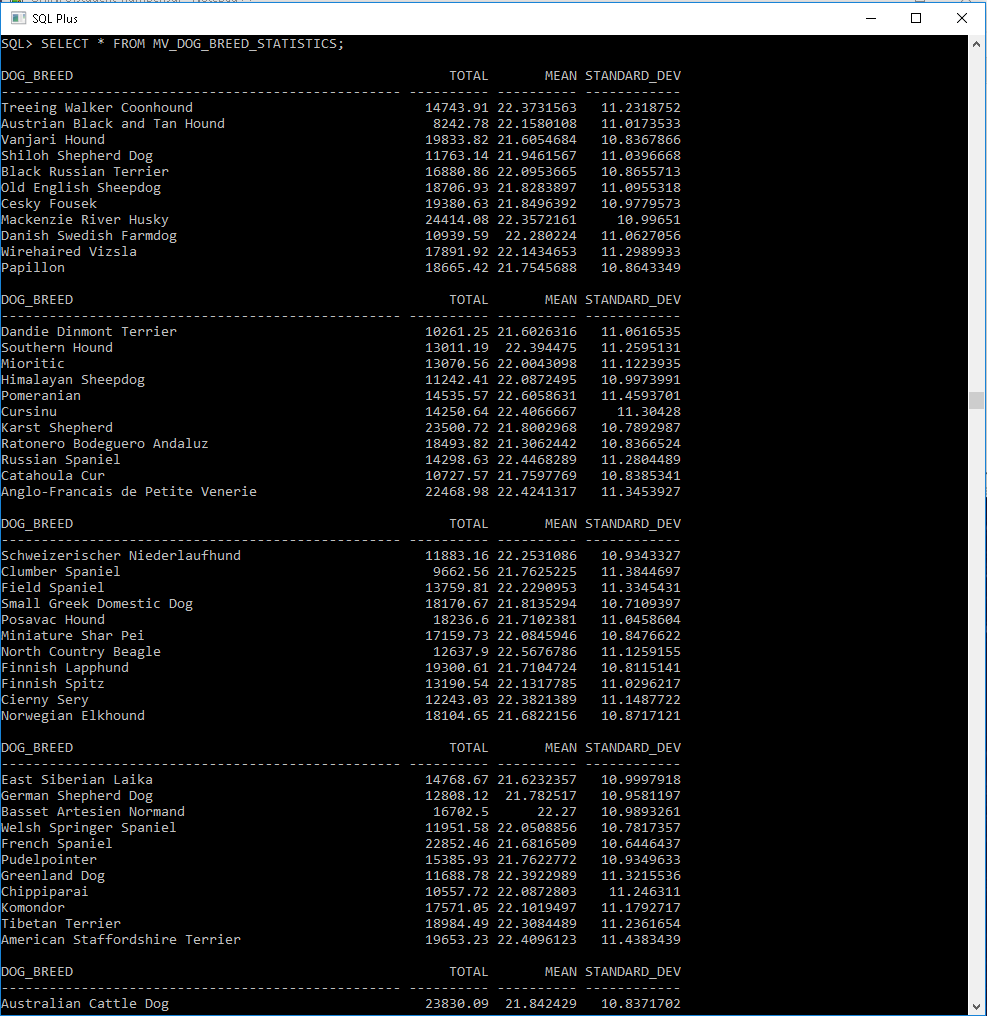
**CREATE** MATERIALIZED **VIEW** MV\_DOG\_BREED\_STATISTICS **AS  
SELECT** DOG\_BREED, *SUM*(PRICE) **AS** TOTAL, *AVG*(PRICE) **AS** MEAN, *STDDEV*(PRICE) **AS** STANDARD\_DEV  
**FROM** SERVICE\_HISTORY, DOGS, DOG\_BREEDS, SERVICE\_HISTORY\_DETAIL, SERVICES  
**WHERE**SERVICE\_HISTORY.DOG\_ID = DOGS.DOG\_ID **AND**DOGS.DOG\_BREED = DOG\_BREEDS.BREED **AND**SERVICE\_HISTORY.SERVICE\_ID = SERVICE\_HISTORY\_DETAIL.SERVICE\_ID **AND**SERVICE\_HISTORY\_DETAIL.SERVICE\_NAME = SERVICES.SERVICE\_NAME  
**GROUP BY** DOG\_BREED;

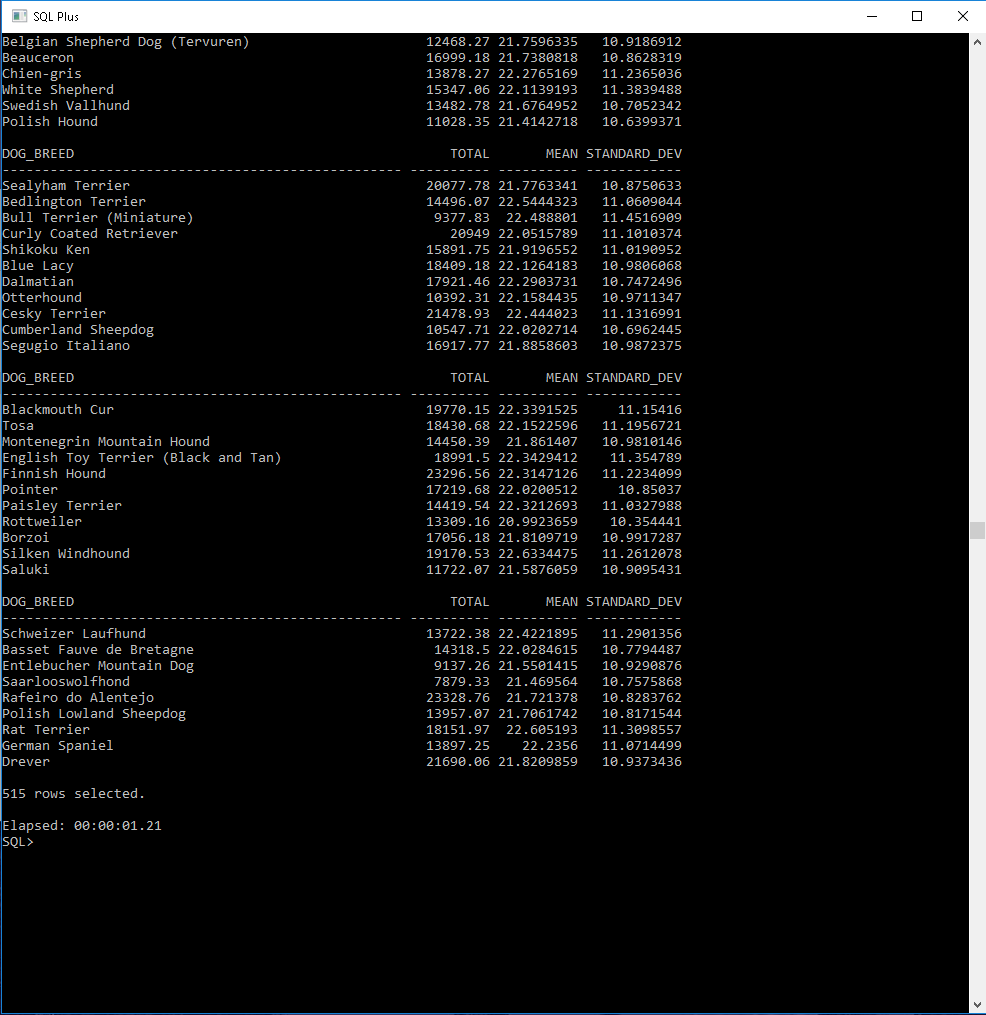


## C: Performance of Materialised views vs Virtual views









The query on Materialised view is running faster than that of Virtual view because Materialised views are separate tables where results of the view queries are saved on physical memory. Whereas, Virtual views are nothing but a save query where the results are displayed by querying on an actual table.

As the size of Materialised view table is much smaller in size than the actual table, the query on Materialized view runs faster as compared to Virtual view where the query is on actual table.

## D: View Updatability

The first view is a virtual view or a “Named Query”. This means that each time a query is made on view, the DBMS gets the result by query modification method and querying on base tables. This ultimately means that this view is always updated as the base table is updated.

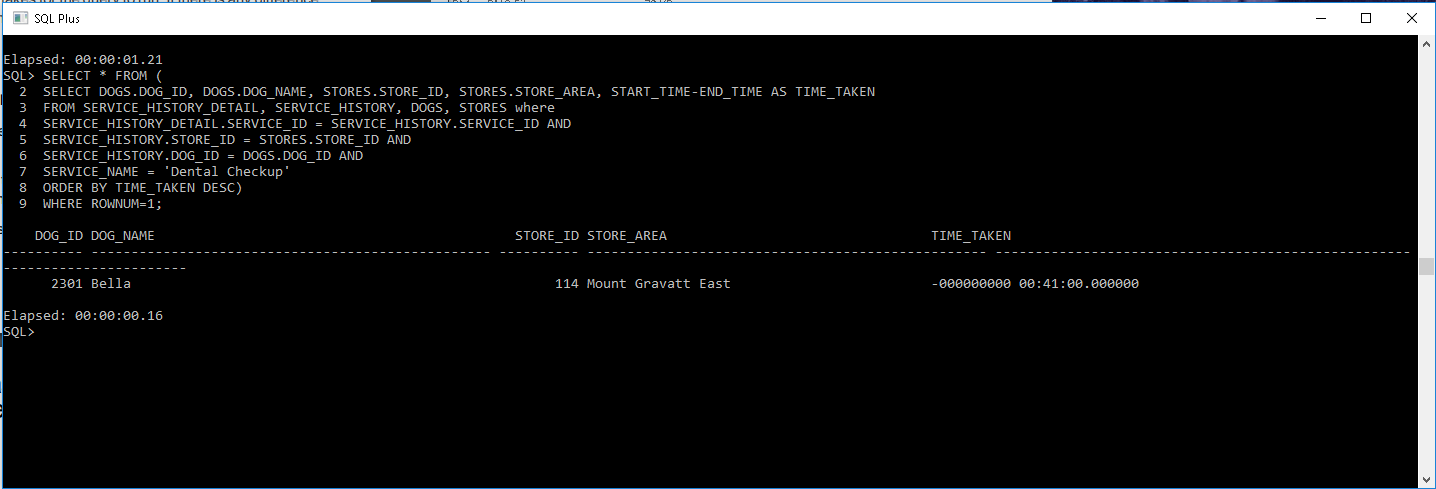
On the other hand, the second view is Materialised view which is an actual table with view query results stored on physical memory. This view is not automatically updated whenever there is a change in base tables. In order to update this materialised view, we have to define a refresh policy to trigger when we want the view to get refreshed.

A simple example can be to use “REFRESH FAST ON COMMIT;” command when creating the materialised view to get the view refreshed.

# TASK 4: FUNCTION BASED INDEXES

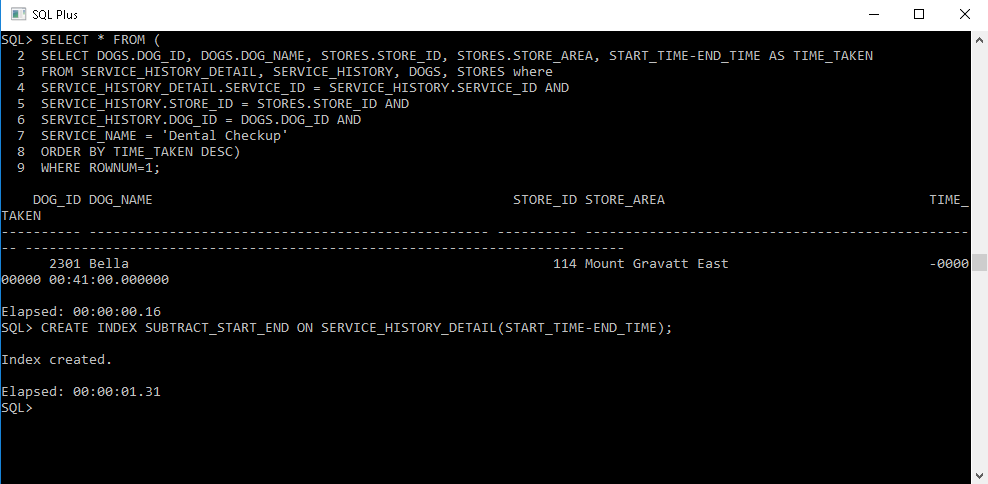
## A: DENTAL CHECKUP QUERY

**SELECT** \* **FROM** (  
  
**SELECT** DOGS.DOG\_ID, DOGS.DOG\_NAME, STORES.STORE\_ID, STORES.STORE\_AREA, START\_TIME-END\_TIME **AS** TIME\_TAKEN  
**FROM** SERVICE\_HISTORY\_DETAIL, SERVICE\_HISTORY, DOGS, STORES  
**WHERE**SERVICE\_HISTORY\_DETAIL.SERVICE\_ID = SERVICE\_HISTORY.SERVICE\_ID **AND**SERVICE\_HISTORY.STORE\_ID = STORES.STORE\_ID **AND**SERVICE\_HISTORY.DOG\_ID = DOGS.DOG\_ID **AND**SERVICE\_NAME = 'Dental Checkup'  
**ORDER BY** TIME\_TAKEN **DESC**)  
  
**WHERE** ROWNUM=1;

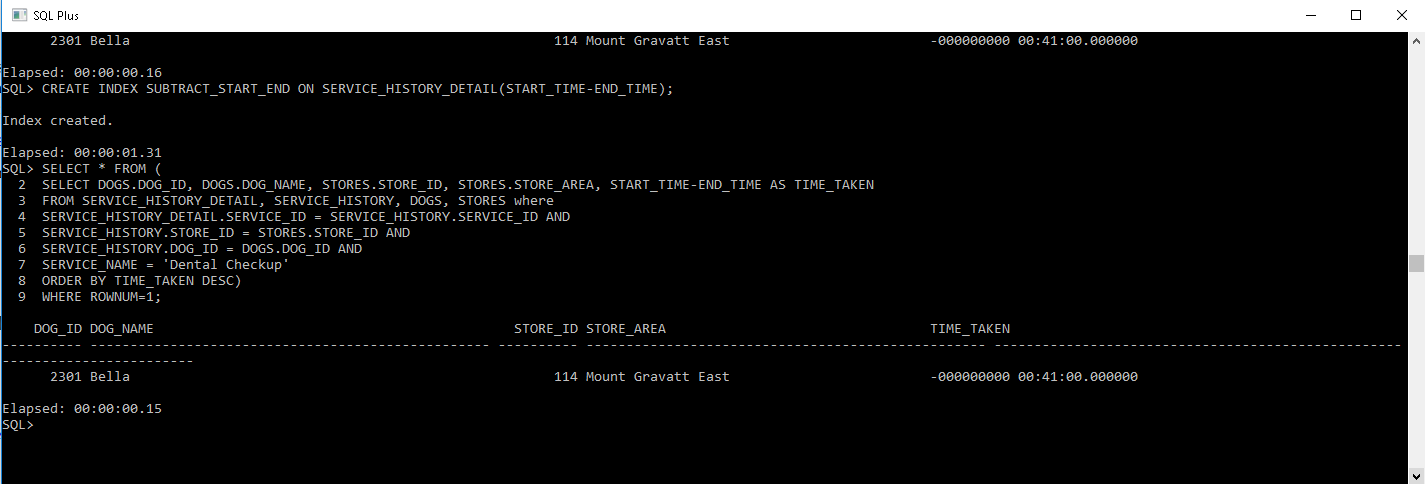


## b: function based index

**CREATE INDEX** SUBTRACT\_START\_END **ON** SERVICE\_HISTORY\_DETAIL(START\_TIME-END\_TIME);



## c: query execution times after index



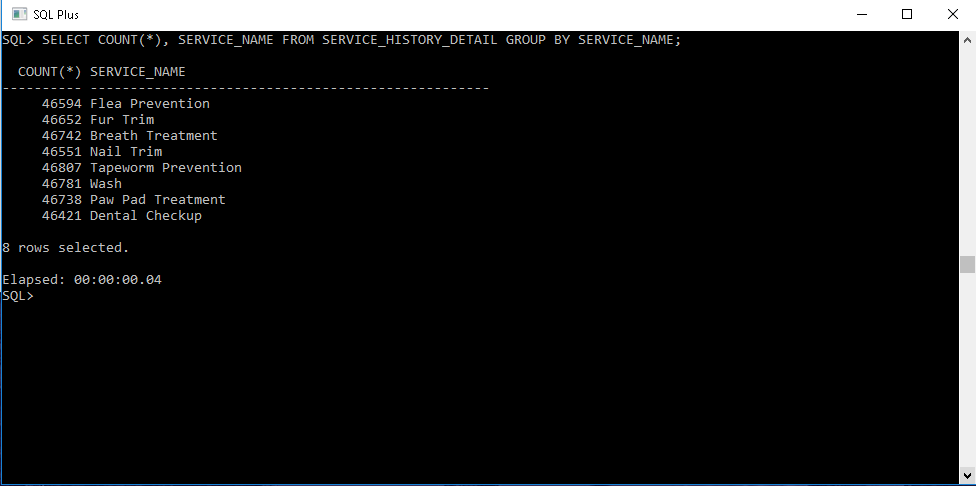
There is not much but a slight difference in query execution time before creating the function based index which computes the difference of two columns. The Function based indices actually is useful where it involves complex and expensive computations when querying the data and performing those operations simultaneously. Here in out query, the difference operation is not much expensive in terms of computing power. This is the reason we are not seeing any significance difference in query execution times.

This is also true because of the approach used to get the result. If we could have used MAX operation on the column differences, creating function based index on that operation might speed things up.

# TASK 5: BITMAP INDEXING

## A: TOTAL NUMBER OF EACH SERVICE PERFORMED

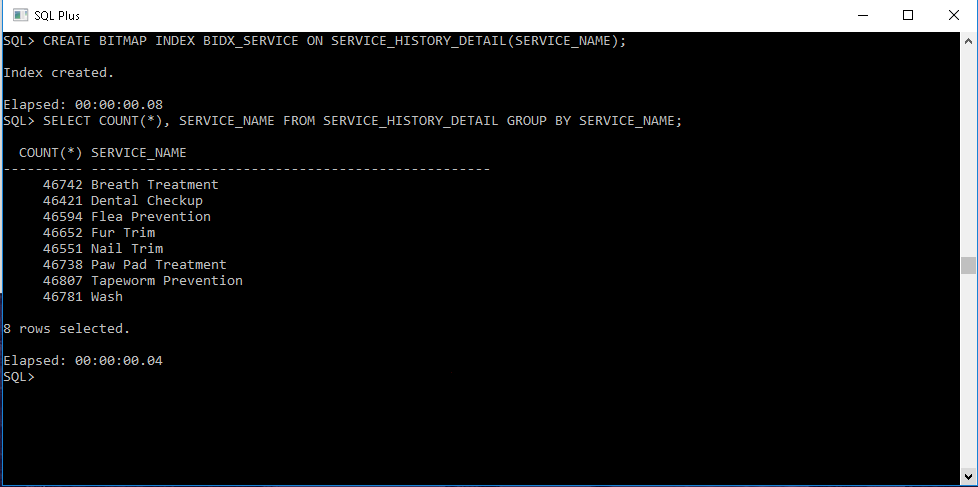
**SELECT** *COUNT*(\*), SERVICE\_NAME   
**FROM** SERVICE\_HISTORY\_DETAIL   
**GROUP BY** SERVICE\_NAME;



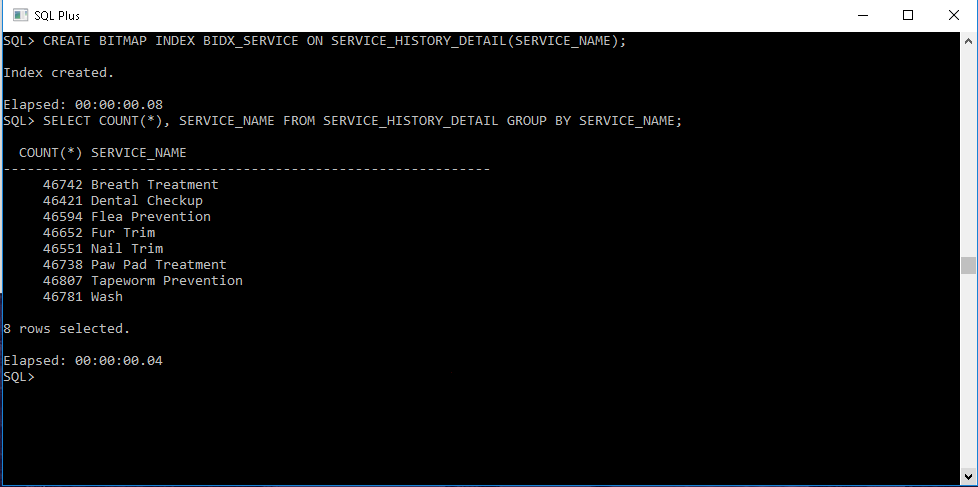
The query execution time to find the total number of each service performed is 4 milliseconds.

## B: BITMAP INDEX BDX\_SERVICE

**CREATE** BITMAP **INDEX** BIDX\_SERVICE **ON** SERVICE\_HISTORY\_DETAIL(SERVICE\_NAME);



## C: DIFFERENCE IN QUERY EXECUTION TIMES



For finding the total number of each service performed, we chose service name column for the bitmap index. This is because this is the only column present in service\_history\_detail that can be categorised. But the problem here is that the number of categories of services is too small as compared to the data. For this reason, the created bitmap index does not affect query execution time significantly.

This tells us that in order to have an efficient and effective bitmap index, the column should not have too sparse or too dense categorisation.

## D: ADVANTAGES AND DISADVANTAGES OF CONSTRUCTING THIS INDEX

### Advantages

* The biggest advantage for this index is that it is created on service\_name column which contains the service name categorization. This means that whenever a query involves operations to fetch the data related to category of serive\_name, this index is a great help to speed up query execution time.
* Moreover, this bitmap index is update efficient in terms of service name updatability. This is true because the business Doggo Paradise may not change their service names too frequently. This will ultimately require less update operations on index.
* The index can be compressed which means it offers space efficiency as well. Furthermore, it breaks down complex data into simple and easy to read data to aid processor work efficiently.

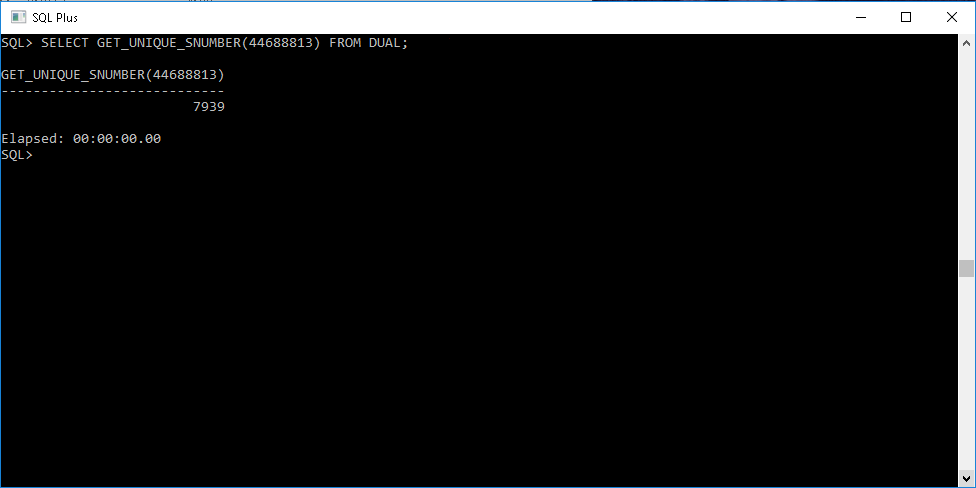
### Disadvantages

* This bitmap index affects query execution times less significantly. This is because of the number of actual data records and the number of categories service\_name should be in some optimal range in order to feel the difference between execution times. There might be a significant difference if the actual records were in gigabytes of size.
* This index is update inefficient in terms of adding new service\_history\_detail records. This is true because each time the business performs a service on some dog, the index needs to be updated and cater for new dog service details. This accounts for expensive operations of rearranging the data and hence accounts for update inefficiency.

# TASK 6: EXECUTION PLAN AND ANALYSIS

## A: GETTING UNIQUE DOG\_ID

**SELECT** GET\_UNIQUE\_SNUMBER(44688813) **FROM DUAL**;



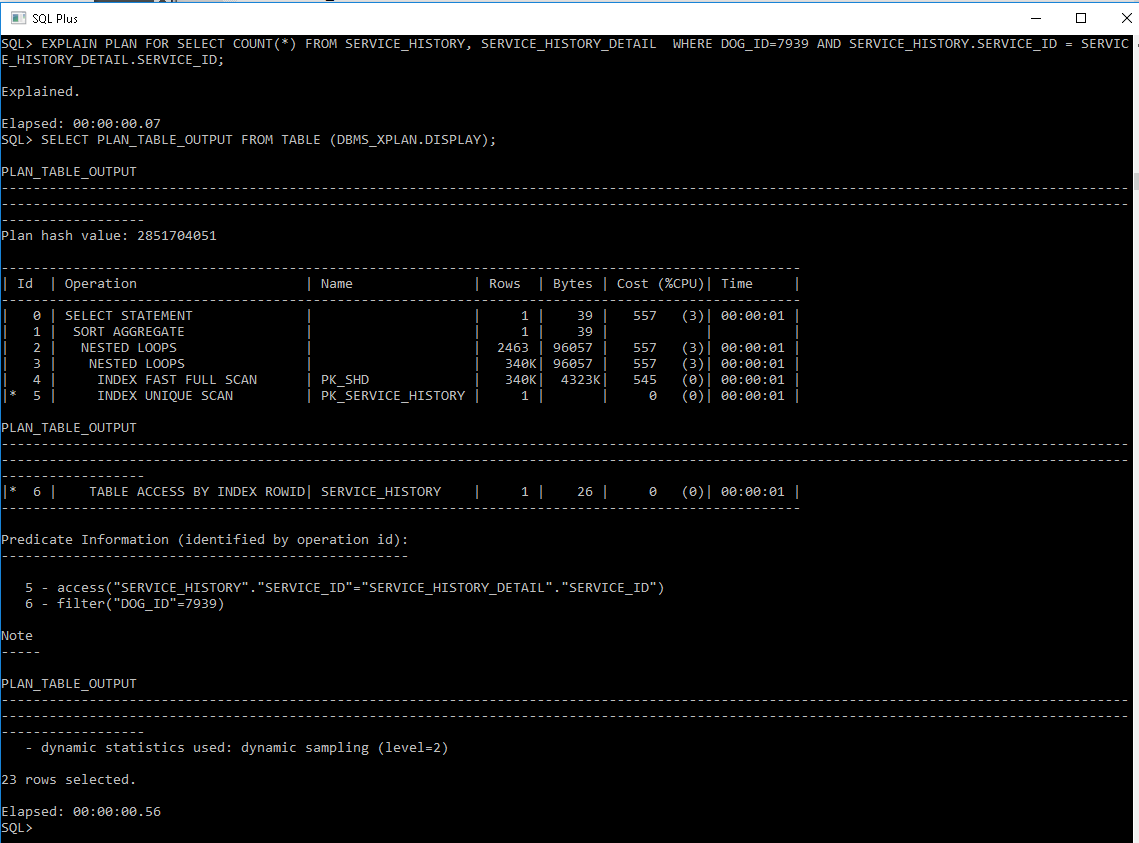
The unique DOG\_ID is 7939.

## B: FIND B+ TREE INDICES

**SELECT** \*  
**FROM** (**SELECT** A.INDEX\_NAME, A.TABLE\_NAME, A.COLUMN\_NAME, B.INDEX\_TYPE  
 **FROM** USER\_IND\_COLUMNS A, USER\_INDEXES B  
 **WHERE** A.TABLE\_NAME='STORES' **AND** A.INDEX\_NAME=B.INDEX\_NAME **AND** B.INDEX\_TYPE='NORMAL'),   
   
 (**SELECT** A.INDEX\_NAME, A.TABLE\_NAME, A.COLUMN\_NAME, B.INDEX\_TYPE  
 **FROM** USER\_IND\_COLUMNS A, USER\_INDEXES B  
 **WHERE** A.TABLE\_NAME='SERVICE\_HISTORY' **AND** A.INDEX\_NAME=B.INDEX\_NAME **AND** B.INDEX\_TYPE='NORMAL'),  
   
 (**SELECT** A.INDEX\_NAME, A.TABLE\_NAME, A.COLUMN\_NAME, B.INDEX\_TYPE  
 **FROM** USER\_IND\_COLUMNS A, USER\_INDEXES B  
 **WHERE** A.TABLE\_NAME='SERVICE\_HISTORY\_DETAIL' **AND** A.INDEX\_NAME=B.INDEX\_NAME **AND** B.INDEX\_TYPE='NORMAL');

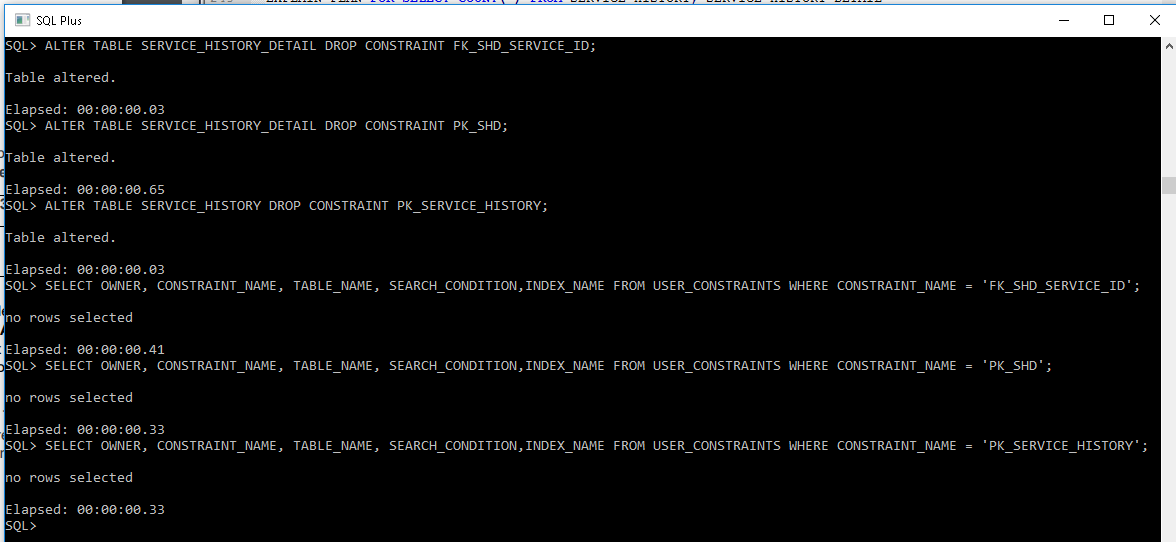
## C: ALL VISITS TO DOGGO PARADISE

**EXPLAIN** PLAN **FOR SELECT COUNT**(\*) **FROM** SERVICE\_HISTORY, SERVICE\_HISTORY\_DETAIL   
**WHERE** DOG\_ID=7939 **AND** SERVICE\_HISTORY.SERVICE\_ID = SERVICE\_HISTORY\_DETAIL.SERVICE\_ID;  
  
  
**SELECT** PLAN\_TABLE\_OUTPUT **FROM TABLE** (DBMS\_XPLAN.DISPLAY);



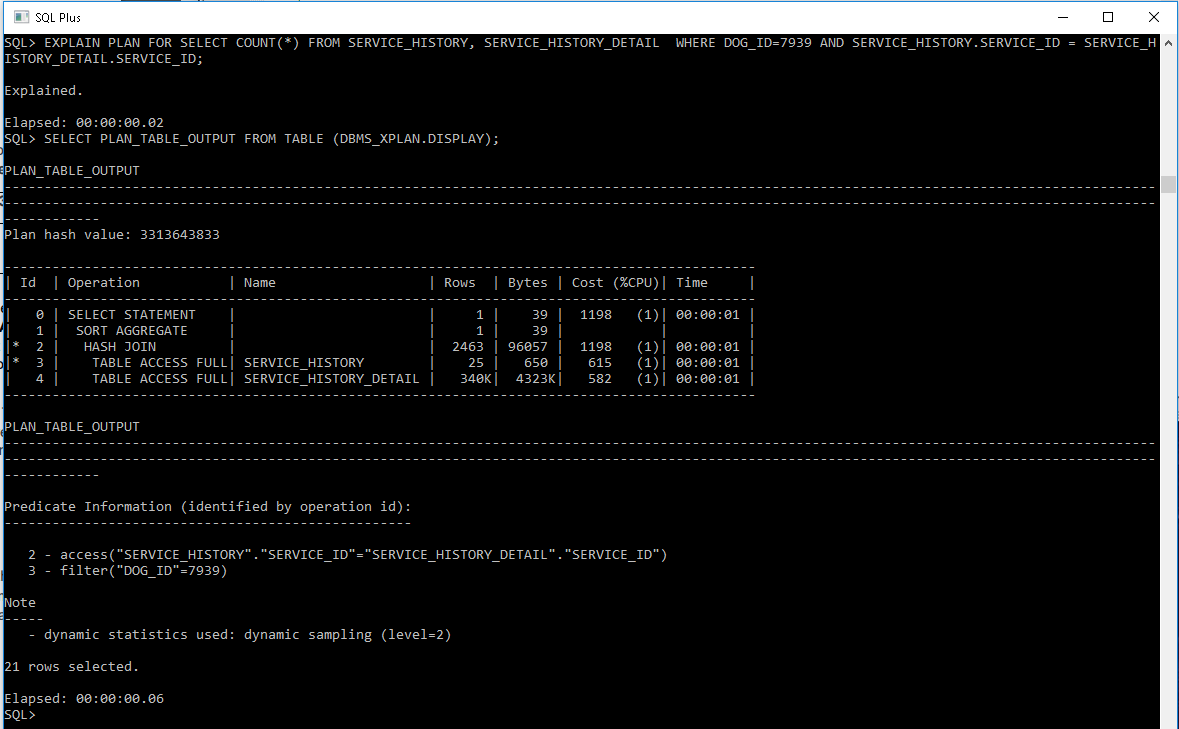
## d: difference in query execution

**ALTER TABLE** SERVICE\_HISTORY\_DETAIL **DROP CONSTRAINT** FK\_SHD\_SERVICE\_ID;  
  
**ALTER TABLE** SERVICE\_HISTORY\_DETAIL **DROP CONSTRAINT** PK\_SHD;  
  
**ALTER TABLE** SERVICE\_HISTORY **DROP CONSTRAINT** PK\_SERVICE\_HISTORY;  
  
**SELECT OWNER**, **CONSTRAINT\_NAME**, **TABLE\_NAME**, SEARCH\_CONDITION,  
INDEX\_NAME **FROM** USER\_CONSTRAINTS **WHERE CONSTRAINT\_NAME** = 'FK\_SHD\_SERVICE\_ID';  
  
**SELECT OWNER**, **CONSTRAINT\_NAME**, **TABLE\_NAME**, SEARCH\_CONDITION,  
INDEX\_NAME **FROM** USER\_CONSTRAINTS **WHERE CONSTRAINT\_NAME** = 'PK\_SHD';  
  
**SELECT OWNER**, **CONSTRAINT\_NAME**, **TABLE\_NAME**, SEARCH\_CONDITION,  
INDEX\_NAME **FROM** USER\_CONSTRAINTS **WHERE CONSTRAINT\_NAME** = 'PK\_SERVICE\_HISTORY';



### Execution Plan Difference

**EXPLAIN** PLAN **FOR SELECT COUNT**(\*) **FROM** SERVICE\_HISTORY, SERVICE\_HISTORY\_DETAIL   
**WHERE** DOG\_ID=7939 **AND** SERVICE\_HISTORY.SERVICE\_ID = SERVICE\_HISTORY\_DETAIL.SERVICE\_ID;  
  
  
**SELECT** PLAN\_TABLE\_OUTPUT **FROM TABLE** (DBMS\_XPLAN.DISPLAY);

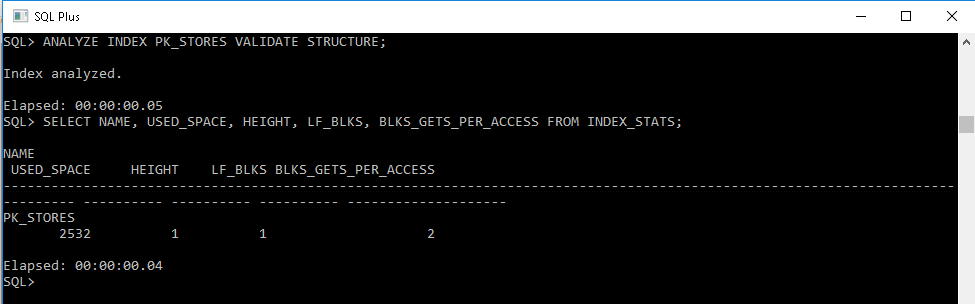


The query execution plans for the query with B-tree indexes on primary keys and without that have a significant difference. If we look at the operations column for both execution plans, we can clearly see that the plan with B-tree indexes on primary keys is scanning the indexes on Service\_History and Serive\_History\_Detail tables. As searching through index is faster, we can clearly see the magnitude of bytes scanned and cost (% CPU time) that it is pretty low. That’s a good thing about indexing - fast access to data without scanning the entire table for matching records.

On the other hand, if we look at the execution plan after dropping the B-tree indexes of primary keys from both tables, it is significantly different. Again, if we look at the operations column, the system is canning both tables entirely for matching records. This is clearly a disadvantage in terms of bytes scanned and cpu cost. The bytes scanned and their relative cpu cost is much greater in this plan than the execution plan with B-tree indexes.

## e: index analysis

**ANALYZE INDEX** PK\_STORES VALIDATE STRUCTURE;  
  
**SELECT NAME**, USED\_SPACE, HEIGHT, LF\_BLKS, BLKS\_GETS\_PER\_ACCESS **FROM** INDEX\_STATS;



Height of Index = 1

Used Space = 2532

Leaf Blocks = 1

Blocks per access = 2