miniDB

Team Participants:

ΤΙΓΚΙΛΗΣ ΝΙΚΟΛΑΟΣ **Π17142**
ΜΑΝΑΚΟΣ ΑΛΕΞΑΝΔΡΟΣ **Π19093**
ΧΑΤΖΟΠΟΥΛΟΣ ΚΙΜΩΝΑΣ **Π19184**

Documentation:

Issue 1:

#1 - Enrich WHERE statement by supporting (a) NOT and BETWEEN operators (5/50) and (b) AND and OR operators (10/50):

Currently, miniDB does not support the use of (a) the simple NOT and BETWEEN operators as well as (b) complex where conditions combined with AND or OR operators. The target of this issue is to add this functionality, both at mSQL level and internally, i.e., in table.py and database.py

#1 Implementation:

Extra: a != b SUPPORT:

In order to support **Logical** NOT we also had to first implement **!= operation**. Now miniDB supports queries like:

- **select * from department where budget != 50000 order by budget;** (numeric support)

and

- **select * from department where building != "watson" order by building;** (string support)

```
dept_name (str) #PK#    building (str)      budget (int)
--------------------    ---------------   --------------
physics                 watson                   70000
music                   packard                  80000
elec. eng.              taylor                   85000
biology                 watson                   90000
comp. sci.              taylor                  100000
finance                 painter                 120000

(smdb)> select * from department where budget != 50000 order by budget;
```

Figure 1(Record with budget=5000 is missing)

```
dept_name (str) #PK#    building (str)      budget (int)
----------------------  ----------------  --------------
music                   packard                    80000
finance                 painter                   120000
history                 painter                    50000
comp. sci.              taylor                    100000
elec. eng.              taylor                     85000

(smdb)> select * from department where building != "watson" order by building;
```

Figure 2(Records with building="watson" is missing)

Implementation of "!=" is achieved by adding != (*operator.ne* from python's operator library) to the corresponding lists at get_op, reverse_operator, reverse_op.

LOGICAL NOT IMPLEMENTATION:

Now that we have implemented the "Not Equals" operation, we can use the **reverse_op** function to reverse the final missing "Equals'' operation (<, >, <=, >= reversion was already implemented in miniDB). Detection of NOT and the sequential handling is implemented again inside misc.py, with **split_condition'**s loop **(for op_key in ops.keys()**). If **Not is found** then the next operation – **and only the next** - will be **reversed** using **reverse_operator(op_key).** Notice that we have to **remove each NOT from the condition** after each detection and finally reset the Boolean ***isNOT*** in order for it to properly function using multiple conditions. This means we have implemented NOT functionality even with queries which contain AND/OR Operators.

To demonstrate that, we will use the implemented logical AND (that we will explain further on):

- **select * from department where NOT budget = 70000 and budget = 80000 order by budget;**

will transform to:

- **select * from department where budget != 70000 and budget = 80000 order by budget;**

★ Only the record with budget = 80000 passes both conditions:

```
dept_name (str) #PK#    building (str)      budget (int)
----------------------  ----------------  --------------
music                   packard                    80000

(smdb)> select * from department where NOT budget = 70000 and budget = 80000 order by budget;
```

Figure 3 Query result

Logical NOT works with **all operators** (=, !=, >, <, >=, <=, **BETWEEN**) ,both numbers (ints,floats, etc.) and strings.

EXAMPLE SYNTAX: (case **insensitive**, works with both lower-upper case)

- **select * from department where NOT dept_name = "music";**

```
dept_name (str) #PK#    building (str)      budget (int)
--------------------    ---------------    -------------
biology                 watson                    90000
comp. sci.              taylor                   100000
elec. eng.              taylor                    85000
finance                 painter                  120000
history                 painter                   50000
physics                 watson                    70000

(smdb)> select * from department where NOT dept_name = "music";
```

Figure 4 Condition: NOT dept_name = "music" transformed to dept_name != "music"

- **select * from department where NOT budget < 60000 order by budget;**

```
dept_name (str) #PK#    building (str)      budget (int)
--------------------    ---------------    -------------
physics                 watson                    70000
music                   packard                   80000
elec. eng.              taylor                    85000
biology                 watson                    90000
comp. sci.              taylor                   100000
finance                 painter                  120000

(smdb)> select * from department where NOT budget < 60000 order by budget;
```

COMPARING REAL SQL with our Logical NOT Implementation:

When running logical Not operations with regular SQL:
NOT of **">="** **is "<"**, and **NOT** of **"<" is ">="**.

E.g.: **SELECT * FROM Products WHERE NOT Price < 18 ORDER BY Price;**

ProductID	ProductName	SupplierID	CategoryID	Unit	Price
35	Steeleye Stout	16	1	24 - 12 oz bottles	18
1	Chais	1	1	10 boxes x 20 bags	18
76	Lakkalikööri	23	1	500 ml	18
39	Chartreuse verte	18	1	750 cc per bottle	18
40	Boston Crab Meat	19	8	24 - 4 oz tins	18.4

(NOT <18 **includes** 18)

Our miniDB implementation works the same:

- **select * from department where NOT budget < 70000 order by budget;**

```
dept_name (str) #PK#    building (str)      budget (int)
----------------------  ----------------  --------------
physics                 watson                   70000
music                   packard                  80000
elec. eng.              taylor                   85000
biology                 watson                   90000
comp. sci.              taylor                  100000
finance                 painter                 120000

(smdb)> select * from department where NOT budget < 70000 order by budget;
```

Figure 5 Not of < 70000 becomes >= 70000

– NA VGALW TA PRINTS

– NA DIPLOTSEKARW TA COMMENTS, + ‘’’’’

– NA ALAKSUME TO TEMPLATE GIA TO CHANGELOG STO EPISIMO TOY PAPEI.

BETWEEN IMPLEMENTATION:

Implemented the Between operator at misc.py. Firstly it was added in the ops list inside **def get_op** function. Unlike the already implemented ‘=’ operation equals to: operator.eq, between is not a “ready to use” operation from python’s library “operator”. To resolve this issue, we have to manually implement “between” functionality. For this reason, we add **'between': between** to ops list and then, define the function inside the same file (misc.py).

The function:

def between(value,range) takes two arguments, **the range of between** and the **value**, which is given internally, def between runs in a loop for every table record. Returns a corresponding True or False for each value check. To pass the range argument, we have to modify parse condition in table.py, when between operation is detected, input after between will be firstly labeled as a string (“20000 & 10000”) and like that, it will be passed as an argument to between function. There it will be split and handled internally, identifying the true type of limits.

Between works with:

1) Integers
2) Floats
3) Mixed one Integer - one Float
4) Works even when the two limits are given in reverse: example: **between “20000 & 10000”** will work like **between “20000 & 10000”** (as it should).
5) Works with Strings although it uses python’s logic of string comparison.

- It checks and handles wrong user input.

Syntax:

[ACTION] FROM [TABLE_NAME] WHERE [COLUMN_NAME] **BETWEEN "VALUE1 & VALUE2";**

(case **insensitive**, works with both lower-upper case)

Example of correct syntax:

- **select * from department where budget between "49000.6 & 80000"**

COMPARING REAL SQL with our Between Implementation:

<u>Case 1: Limit == Value:</u>

Regular SQL:

SELECT * FROM Products WHERE Price BETWEEN **18** AND 20 ORDER BY PRICE;

Result:

Number of Records: 10

ProductID	ProductName	SupplierID	CategoryID	Unit	Price
76	Lakkalikööri	23	1	500 ml	18
39	Chartreuse verte	18	1	750 cc per bottle	18
35	Steeleye Stout	16	1	24 - 12 oz bottles	18
1	Chais	1	1	10 boxes x 20 bags	18
40	Boston Crab Meat	19	8	24 - 4 oz tins	18.4
36	Inlagd Sill	17	8	24 - 250 g jars	19

Figure 6 Under Limit: 18 must be included.

SQL includes 18 which is equal to the under-limit .

Our miniDB implementation has the same functionality:

select * from department where budget between "50000 & 80000" order by budget;

```
dept_name (str) #PK#    building (str)      budget (int)
---------------------   ---------------   --------------
history                 painter                    50000
physics                 watson                     70000
music                   packard                    80000
```

Figure 7 miniDB: Under and Upper Limits are included

<u>Between also works with NOT logic:</u>

SYNTAX FOR NOT + BETWEEN:

[ACTION] FROM [TABLE_NAME] WHERE [COLUMN_NAME] **NOT BETWEEN "VALUE1 & VALUE2";** (case **insensitive**, works with both lower-upper case)

```
dept_name (str) #PK#    building (str)      budget (int)
----------------------  ----------------  --------------
history                 painter                    50000
elec. eng.              taylor                     85000
biology                 watson                     90000
comp. sci.              taylor                    100000
finance                 painter                   120000

(smdb)> select * from department where budget not  between "60000 & 80000" order by budget;
```

Figure 8 NOT BETWEEN example

There is a second defined function: not_between(value,range) which is called when NOT is detected in front of the between operation. Not Between is the reverse of Between logic.

e.g.: select * from department where budget NOT between "60000 & 80000" order by budget;

To achieve this **'between' : 'not_between'** was added to the reverse_operator list and **'not_between' : not_between** was added to the get_op list.

Case 2: NOT BETWEEN and Limits:

Regular SQL:

SELECT * FROM Products WHERE Price NOT BETWEEN 19 AND 20;

Number of Records: 72

ProductID	ProductName	SupplierID	CategoryID	Unit	Price
1	Chais	1	1	10 boxes x 20 bags	18
3	Aniseed Syrup	1	2	12 - 550 ml bottles	10
4	Chef Anton's Cajun Seasoning	2	2	48 - 6 oz jars	22
5	Chef Anton's Gumbo Mix	2	2	36 boxes	21.35

While using NOT logic, **SQL excludes limits** from range. (while there is record with price = 19, it had been excluded)

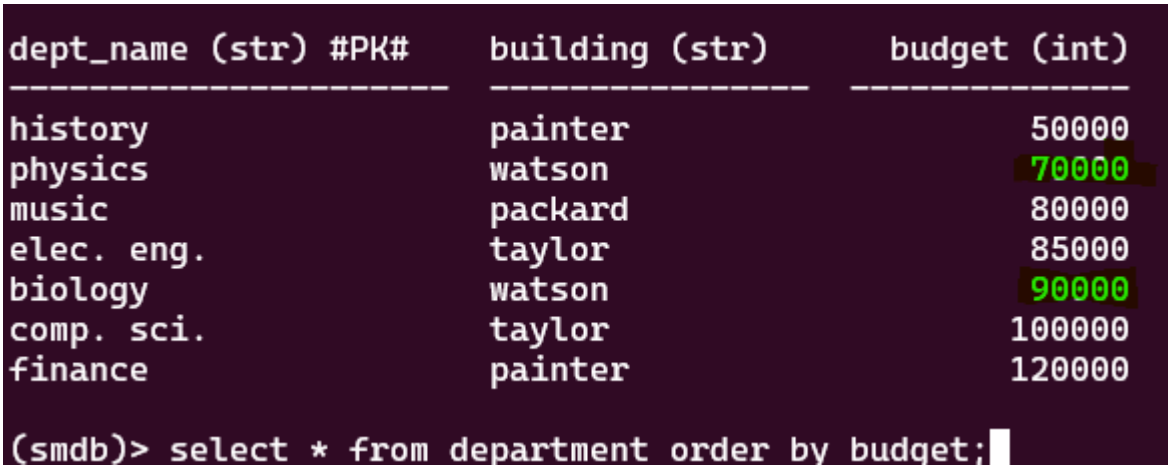
Our miniDB implementation has the same functionality:

select * from department where budget not between "70000 & 90000" order by budget;

```
dept_name (str) #PK#    building (str)      budget (int)
----------------------  ----------------  --------------
history                 painter                    50000
comp. sci.              taylor                    100000
finance                 painter                   120000

(smdb)> select * from department where budget not  between "70000 & 90000" order by budget;
```

By selecting everything from department we can see that rows with budget=70000 and 90000 do exist.



& is called the **split_key** and is defined at the start of between function, this way it can be easily modified. Although modifying the split_key to upper-lower-case “AND” without further changes to the code will result in a functionality failure as it will intervene with logical AND’s/OR’s.

LOGICAL AND/OR IMPLEMENTATION:

To achieve this , the interpreter function now checks if logical operators (AND’s/OR’s) exist inside the query. If yes, the user entered query has to split into multiple new ones:

select * from department where budget < 80000 or budget = 100000;

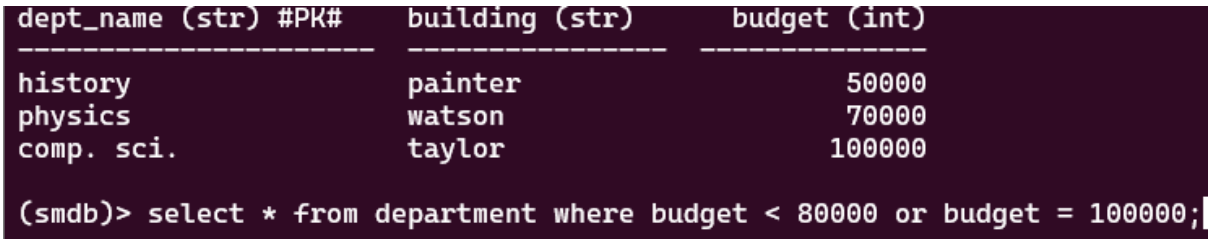
“or” will be detected and they query will produce:

select * from department where budget < 80000;

and

select * from department where budget = 100000;

These queries will be passed to the query planner and will continue operation as normal. Both queries will be run and all results will be saved in the list called allQueryResults. These results will be processed according to the logical operator given (mdb.py main function), and then printed as normal in the terminal.



select * from department where budget = 80000 or budget = 100000 and budget < 110000;

```
dept_name (str) #PK#    building (str)      budget (int)
----------------------  ----------------  --------------
comp. sci.              taylor                    100000
music                   packard                    80000

(smdb)> select * from department where budget = 80000 or budget = 100000 and budget < 110000;
```

More information on the result comparison process:

OR:

If we run:

select * from department where budget = 80000 or budget < 100000; with one record of budget=80000 stored in the database, we have to make sure that the output won't show the same record two times as it passes both conditions (since the user entered query split into two separate queries, both queries have the same record in their result)

```
['music', 'packard', 80000] (OR) OK
['biology', 'watson', 90000] (OR) OK
['elec. eng.', 'taylor', 85000] (OR) OK
['history', 'painter', 50000] (OR) OK
['music', 'packard', 80000] (OR) DUPLICATE (Denied)
['physics', 'watson', 70000] (OR) OK
dept_name (str) #PK#    building (str)      budget (int)
----------------------  ----------------  --------------
music                   packard                    80000
biology                 watson                     90000
elec. eng.              taylor                     85000
history                 painter                    50000
physics                 watson                     70000

(smdb)> select * from department where budget = 80000 or budget < 100000;
```

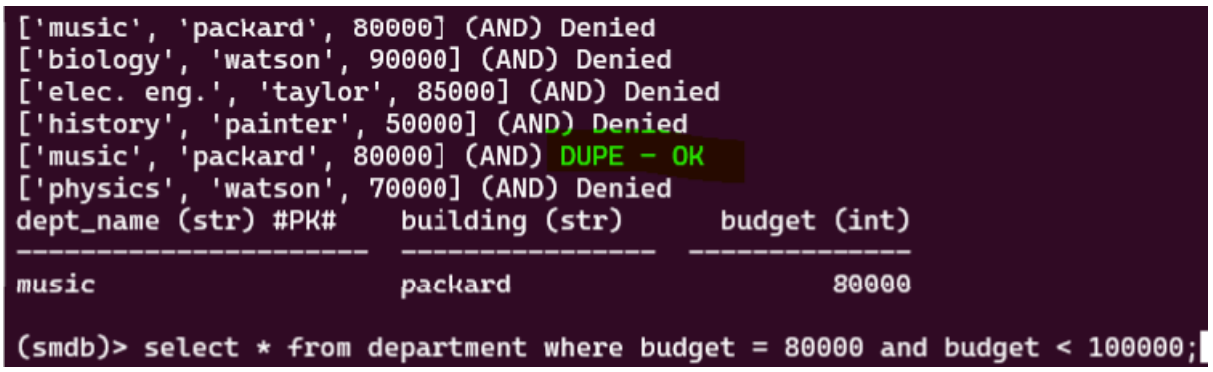
Using the debug screen we can see, the duplicate is detected, denied and therefore not showing in the final output table.

OR Operations use a <u>single</u> temporary list of records:

1) result_table: All results enter one by one to the result_table, during each addition, the whole table is checked to see if there is an identical record already saved. If yes, the dupe is detected and the record is left outside.

Opposite process for AND operations, we need to approve only duped records:

select * from department where budget = 80000 and budget < 100000;



AND Operations use <u>two</u> temporary lists of records.

1) result_table which contains the final table that will be shown to the user
2) temp_table: Every record is added to this buffer list, during each record addition to this buffer, the whole table is checked to see if there is an exact same record in the table already. Only if a duplicate is found in this table, the record passes to the result_table.

The first time, record: music, packard, 80000 was denied (passing to result_table) as there was no other identical record stored in temp_table.

Issue 2:

#2 - Enrich indexing functionality by supporting (a) BTree index over unique (non-PK) columns (10/50) and (b) Hash index over PK or unique columns (10/50) Currently, miniDB supports BTree indexing over the primary key of a table. The target of this issue is to (a) add support for BTree index over other columns as well, if they are declared as 'unique', and (b) add support for Hash index over the PK or a unique column of a table (implement extendible hashing, either MSB or LSB variant, by using a hash function based on the modulo operator "%" – dict is a hash struct in python that can be used in your implementation). The required changes affect the SQL parser mdb.py as well as the maintenance of the meta_table "meta_indexes". In either case, as a preparatory action, the CREATE TABLE statement should be enriched to support the declaration of a column as 'unique'.

When working on issue 2 we need to split it into 3 parts that took 3 different approaches.

<u>#2 Implementation:</u>

<u>Part1:</u>

Our implementation currently supports creation of 1 unique column per table , but more could be added by expanding the current unique tables columns in order to accept more and also for the show table function in table.py to append the #unique tags to the columns .

<u>Creation of UNIQUE columns on tables and handling similar rows being added to a unique table:</u>

This can be done by inserting a query like :

create table table_name **(**column_name1 **str unique(there must be a space after the word UNIQUE) ,** column_name2 **str primary key ,** column_name3 **str)**

Afterwards when showing the table we check in the pkl files to see if there is a unique one with the table name and we attach the #unique tag

Inside mdb.py we check for the words unique inside the create table action query and afterwards we create a pkl table that saves the table name the unique column and its pk if it has one.We achieve this by checking the arguments inside ArgNopk variable when checking for action==create table.If 'unique' is found we then begin treating that column using different handling methods that will always cover the constraint..

Afterwards when showing the table we check in the pkl files to see if there is a unique one with the table name and we attach the #unique tag , we also create a unique index that is saved and we use it to complete the aforementioned process.

Example of usage:

- **create table test_table(one str unique , two str)**

Checking for similarities:

If there is another value with the same one as the inserted already inside a tables unique column we check that by : before the insertion inside the insert_into function of the database.py we use the **check_uniques** functions to extract and compare data inside and data to be added to table and check for unique column.

The **check_uniques function :** A function to check if there are any unique constraints in a table and ensure that the new data being inserted does not violate any of these constraints.

this happens by checking the temporary data that has been inserted in the table , we extract the needed data , table name etc. from the unique table.pkl compare and then give an exception or continue:

- **insert into test_table values(value1,value2);**
- **insert into test_table values(value3,value2);**
- **insert into test_table values(value4,value2);**
- **insert into test_table values(value5,value2);**
- **select * from test_table**

```
one (str) #UNIQUE#    two (str)
--------------------  -----------
value1                value2
value3                value2
value4                value2
value5                value3

(smdb)> select * from test_table
```

- **insert into test_table values(value5,value3);**

EXCEPTION!

```
Traceback (most recent call last):
  File "/home/ngc224/gitclones/githubs_miniDB/mdb.py", line 480, in <module>
    result = execute_dic(dic)
  File "/home/ngc224/gitclones/githubs_miniDB/mdb.py", line 369, in execute_dic
    return getattr(db, action)(*dic.values())
  File "/home/ngc224/gitclones/githubs_miniDB/miniDB/database.py", line 345, in insert_into
    self._check_unique(row,table_name)
  File "/home/ngc224/gitclones/githubs_miniDB/miniDB/database.py", line 870, in _check_unique
    raise Exception("error value exsists in unique column no insertion will be done ")
Exception: error value exsists in unique column no insertion will be done

(smdb)> insert into test_table values(value5,value3)
```

Part2:

Here is the way we made the b+tree index allow unique columns , the way of writing the query is

Create index index_name **on** table_name(unique_column) **using btree**

In the mdb.py file in the interpret function we check if a query is similar to the above) , we extract the table name , the index name like normally and also we save the unique column name in a unique_index.pkl file(temporarily because with every new index on a unique column we overwrite the data) afterwards the create index function gets called and before we get to the part that checks for pk , we check if the attributes of the called function coincide inside the temp pkl table that has all the tables with a unique constraint .Then we call the **construct_index_for_uniques_using_btree** that does the same as the construct index but on the specified column .

Example of usage:

Let's say we have table test_table that has these values and a unique constraint , For the b+tree index example we are using these values , and we were asked for the interpreter to support index on unique column:

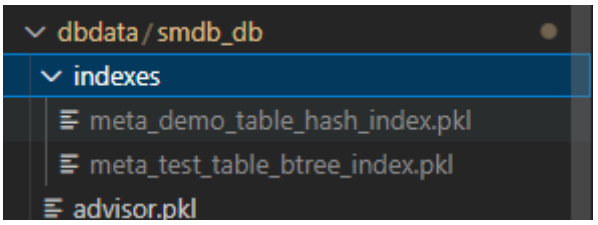
- **select * from test_table**

```
one (str) #UNIQUE#    two (str)
--------------------  -----------
value1                value2
value3                value2
value4                value2
value5                value3

(smdb)> select * from test_table
```

- **create index test_table_btree on test_table(one) using btree**

```
index made on unique column
(smdb)> create index test_table_btree on test_table(one) using btree
```



- **select * from test_table**

```
Select using b+tree with a unique column
one
one (str) #UNIQUE#    two (str)
--------------------  -----------
value1                value2
value3                value2
value4                value2
value5                value3

(smdb)> select * from test_table
```

Part3: Extendible Hash index

Here we created another class named hashindex inside the database.py that we call its attributes from the create index and construct index function of database.py.

This is an implementation of extendible hash index. This class has a BUCKET_SIZE constant that determines the maximum size of the bucket and a HASH_SIZE constant for the hash key. The class has an init method that takes optional depth and lsb parameters, sets up an empty bucket dictionary to store key-value pairs, and initializes the depth and lsb properties.

The split_bucket function takes a hash as input and splits the bucket with the given hash. It increments the depth variable, then re-evaluates all items in the old bucket with the updated depth value.

The Find method looks up the value associated with a particular key in a hash index. If the key is a 2-tuple, a range query is performed and all values between the start and end of the range are returned. If start or end is None, returns all values less than, greater than, or equal to the specified value. If the key is a single value, returns the value associated with that key in the hash index, or None if the key is not found.

The delete method takes a key as input and deletes the pair with the given key from the hash index.

The Insert method takes a key-value pair as input and hashes the key using the _hash_lsb or _hash_msb function, depending on the value of lsb. If the resulting hash is already in the bucket dictionary, it appends the key-value pair to the appropriate bucket. If not, it creates a new bucket with the given hash and saves the key-value pair there. If the length of the bucket exceeds BUCKET_SIZE, call the split_bucket function to split the bucket.

The get_all method returns a list of all key-value pairs in the hash index. It does this by iterating over all the buckets in the hash table and appending each key-value pair to a list of pairs. Eventually it returns this list of pairs.

Example of usage:

create table demo_table

```
(smdb)> create table demo_table(one str primary key , two str)
Created table "demo_table".
```

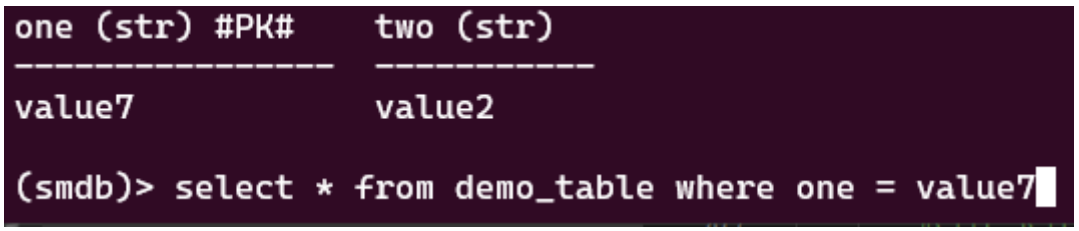
demo_table values :

```
(smdb)> select * from demo_table

one (str) #PK#    two (str)
----------------  -----------
value5            value3
value6            value3
value7            value2
value8            value77
```

Hash index creation :

```
Index made
(smdb)> create index demo_table_hash on demo_table using hash;
```



Part4:

There was an error when getting data from the meta indexes pkl file and we reformed it , the error was that we were only getting the first row of it when searching what index to use when searching , as it was always picking the first one and it would show errors . And inside the drop table and index functions we implemented the necessary pkl file handling mechanisms so as when a table that has an index on a unique column gets deleted or its index gets dropped , we delete all the necessary things from the pkl tables

Final notes: Inside our branch you will find a file named PARADOXES.txt inside of it is the way of writing the new and usable SQL commands concerning the implementations