Στοιχεία Φοιτητή:

Ονοματεπώνυμο: Γιώργος Βλάχος

Αριθμός Μητρώου: Π20034

Σύντομη Περιγραφή του παραδοτέου:

Έχω απαντήσει ολοκληρωμένα στα πρώτα 2 issues της εκφώνησης και έχω κάνει μια μικρή αρχή στο τρίτο issue όσον αφορά το κομμάτι του Optimizer.

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

Για τον σκοπό αυτόν έχω δημιουργήσει ένα αρχείο με όνομα 'condition_handler.py' το οποίο περιέχει διάφορες μεθόδους ώστε να πραγματοποιηθεί το παραπάνω ζητούμενο. Η κάθε μέθοδος διαθέτει αναλυτική περιγραφή για τον τρόπο λειτουργίας της εντός του κώδικα. Ακολουθούν συνοπτικές πληροφορίες για τις κυριότερες από αυτές.

def get_condition_plan(condition): Αυτή είναι από τις σημαντικότερες συναρτήσεις καθώς χρησιμοποιείτε σε συνδιασμό με μερικές άλλες για να τις λειτουργίες insert, update, delete, select. Δέχεται ώς είσοδο ένα string που είναι το condition μέρος του query και ανδρομικά κατασκευάζει και επιστρέφει ένα dictionary που αποτελείτε από όλην την απαραίτητη πληροφορία για να μπορούμε να έχομε πιο σύνθετα quiries.

Στην γενική μορφή το επιστρεφόμενο λεξικό έχει ώς εξής:

{ first_condtion: [left_operand, operator, right_operand, keyword_operator , empty string or another dictionary of the same form] }

π.χ.0

Για το condition: 'tot_cred > 80'

Η συνάρτηση επιστρέφει:

{'tot_cred > 80': ['tot_cred', '>', '80', '']}

Που σημάινει ότι το πρώτο μέρος (και μοναδικό) είναι το ‘tot_cred > 80’, τα tokens από τα οποία αποτελείται αυτό είναι [‘tot_cred’, ‘>’, ‘80’, ‘’] όπου το κενό string στο τέλος σηματοδοτεί ότι δεν ακολουθεί άλλο μέρος condition.

π.χ.1

Για το condition:

4 > 3 and 12 not between 40 and 60 or 12 between 20 and 100

Η συνάρτηση επιστέφει την εξής δομή:

{'4 > 3': ['4', '>', '3', 'and', {'12': ['12', 'not', {'12 between 40 and 60': ['12', 'between', '40', 'and', '60', 'or', {'12 between 20 and 100': ['12', 'between', '20', 'and', '100', '']}]}]}]}

Σημείωση: Στην περίπτωση του not ώς key του dictionary δεν είναι ολόκληρο το 1ο condition αλλά μόνο το όνομα του αριστερού operand. Αυτό έχει επειδή πρέπει σε οποιοδήποτε condition να μπορούμε να απομονώσουμε τον αριστερότερο operand για να τσεκάρουμε αργότερα ποιο είναι το condition column.

π.χ.2

Για το condition:

id > 3 and tot_cred not between 40 and 60 or tot_cred between 80 and 100

Η συνάρτηση επιστρέφει την εξής δομή:

{'id > 3': ['id', '>', '3', 'and', {'tot_cred': ['tot_cred', 'not', {'tot_cred between 40 and 60': ['tot_cred', 'between', '40', 'and', '60', ‘or’, {'tot_cred between 80 and 100': ['tot_cred', 'between', '80', 'and', '100', '']}]}]}]}

Επιπλέον υπάρχει η συνάρτηση: **def evaluate_condition(condition),** που χρησιμοποιείτε σε συνδιασμό με την παραπάνω και χρησιμοποιεί και μια εμφωλευμένη συνάρτηση την **def evaluate_basic_condition**. Δέχετε ένα condition_plan όπως αυτό που επιστρέφει η get_condition_plan και επιστρέφει True ή False. Η συνάρτηση επίσης λειτουργεί αναδρομικά μπαίνοντας στο πιο μέσα dictionary του plan και επιστρέφοντας την λογική πράξη μεταξύ αυτού και το προηγούμενο dicitonary. Χρησιμοποιεί επίσης και κάποιες άλλες συναρτήσεις που έχω ορίσει στο αρχείο όπως η between.

Με αυτόν τον τρόπο μπορούμε να έχουμε queries που περιέχουν τα keywords and, or, between, not αλλά και συνδυασμούς αυτών, ενώ παράλληλα μέσω της αναδρομής και κάποιων ακόμα ελέγχων διατηρείται και η σωστή προτεραιότητα μεταξύ τους.

Με βάση τα παραπάνω έχει αλλάξει ο τρόπος που γίνονται parsed και evaluated τα conditions στις μεθόδους select, insert, updata, delete, κ.λπ. Για παράδειγμα για τα select το κομμάτι του parsing και evaluation έχει γίνει ώς εξής:

```
range_query = False
if condition is not None:
# we need to store the first subcondition of the full condition
# if condition: id > 3 and name = john, the generated condition_plan is: { 'id > 3': [ 'name', '=', 'john', '' ] }
# we want to get the first part of the condition, meaning -> 'id > 3' then split it into a list
tokens = list(condtion_handler.get_condition_plan(condition).keys())[0].split()
print(f'tokens: {tokens}, {condtion_handler.get_condition_plan(condition).keys()}')
condition_column = tokens[0]
range_query = condtion_handler.is_range_query(tokens) # we need to know where or not the query is a range query in case we're using EHash Index
#condition_column = split_condition(condition)[0]
else:
condition_column = ''
```

(Η condition_handler.is_range_query είναι επίσης μια συνάρτηση που έχω φτιάξει και θα αναφερθεί περισσότερο στο 2b)

Ώς αποτέλεσμα αυτών των αλλαγών και με μερικά prints έχουμε το εξής:

<u>π.χ.0 απλό between</u>

Για το query: **select * from student where tot_cred between 30 and 90;**

```
(smdb)> select * from student where tot_cred between 30 and 90;
Dict in create query: {'select': '*', 'from': 'student', 'where': 'tot_cred between 30 and 90', 'distinct': None, 'order by': None, 'limit': None, 'desc': None}
tokens: ['tot_cred', 'between', '30', 'and', '90'], dict_keys(['tot_cred between 30 and 90'])
USING LINEAR
{'102 between 30 and 90': ['102', 'between', '30', 'and', '90', '']}
{'102 between 30 and 90': ['102', 'between', '30', 'and', '90', '']} False
{'32 between 30 and 90': ['32', 'between', '30', 'and', '90', '']}
{'32 between 30 and 90': ['32', 'between', '30', 'and', '90', '']} True
{'80 between 30 and 90': ['80', 'between', '30', 'and', '90', '']}
{'80 between 30 and 90': ['80', 'between', '30', 'and', '90', '']} True
{'110 between 30 and 90': ['110', 'between', '30', 'and', '90', '']}
{'110 between 30 and 90': ['110', 'between', '30', 'and', '90', '']} False
{'56 between 30 and 90': ['56', 'between', '30', 'and', '90', '']}
{'56 between 30 and 90': ['56', 'between', '30', 'and', '90', '']} True
{'46 between 30 and 90': ['46', 'between', '30', 'and', '90', '']}
{'46 between 30 and 90': ['46', 'between', '30', 'and', '90', '']} True
{'54 between 30 and 90': ['54', 'between', '30', 'and', '90', '']}
{'54 between 30 and 90': ['54', 'between', '30', 'and', '90', '']} True
{'38 between 30 and 90': ['38', 'between', '30', 'and', '90', '']}
{'38 between 30 and 90': ['38', 'between', '30', 'and', '90', '']} True
{'0 between 30 and 90': ['0', 'between', '30', 'and', '90', '']}
{'0 between 30 and 90': ['0', 'between', '30', 'and', '90', '']} False
{'58 between 30 and 90': ['58', 'between', '30', 'and', '90', '']}
{'58 between 30 and 90': ['58', 'between', '30', 'and', '90', '']} True
{'60 between 30 and 90': ['60', 'between', '30', 'and', '90', '']}
{'60 between 30 and 90': ['60', 'between', '30', 'and', '90', '']} True
{'98 between 30 and 90': ['98', 'between', '30', 'and', '90', '']}
{'98 between 30 and 90': ['98', 'between', '30', 'and', '90', '']} False
{'120 between 30 and 90': ['120', 'between', '30', 'and', '90', '']}
{'120 between 30 and 90': ['120', 'between', '30', 'and', '90', '']} False
  id (str) #PK#  name (str)    dept_name (str)      tot_cred (int)
---------------  ------------  -----------------  ----------------
         12345  shankar       comp. sci.                       32
         19991  brandt        history                          80
         44553  peltier       physics                          56
         45678  levy          physics                          46
         54321  williams      comp. sci.                       54
         55739  sanchez       music                            38
         76543  brown         comp. sci.                       58
         76653  aoi           elec. eng.                       60

(smdb)>
```

Εδώ βλέπουμε το condition plan που δημιουργήται για κάθε γραμμή κατά την διάρκεια του select και δίπλα το αποτέλεμά του (το αποτέλεσμα προκύπτει απο την evaluate_condition).

π.χ.1 between και not

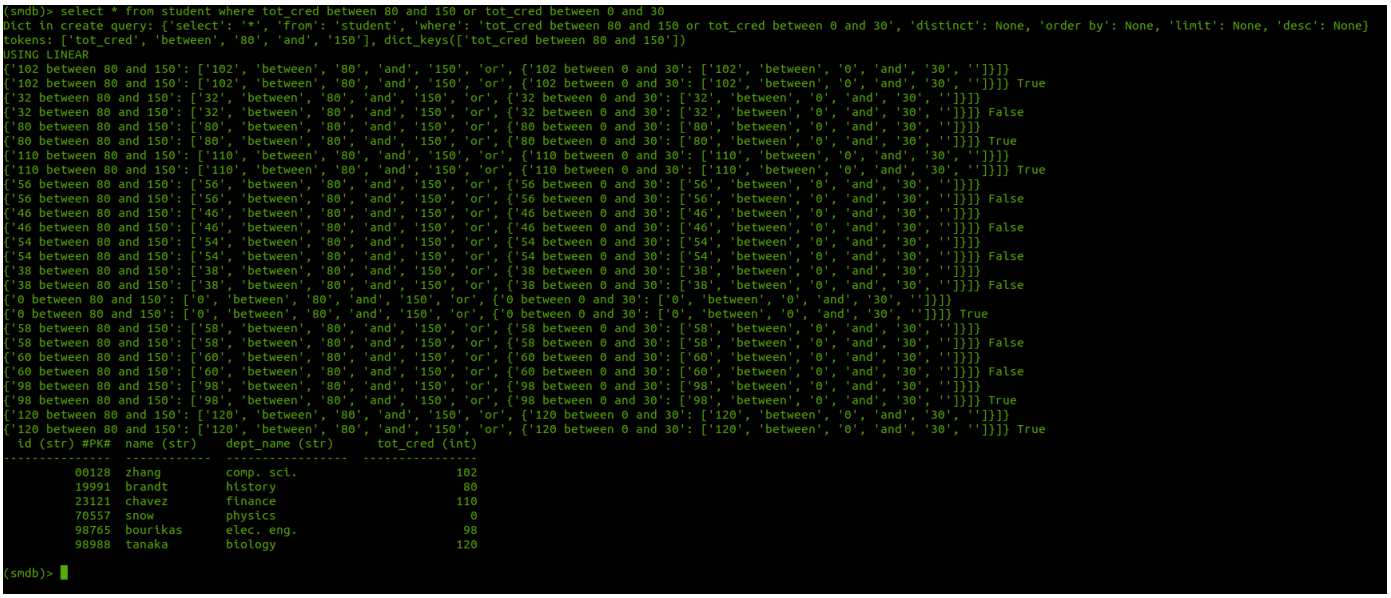
Για το query: **select * from student where tot_cred not between 30 and 60;**

```
(smdb)> select * from student where tot_cred not between 30 and 60;
Dict in create query: {'select': '*', 'from': 'student', 'where': 'tot_cred not between 30 and 60', 'distinct': None, 'order by': None, 'limit': None, 'desc': None}
tokens: ['tot_cred'], dict_keys(['tot_cred'])
USING LINEAR
{'102': ['102', 'not', {'102 between 30 and 60': ['102', 'between', '30', 'and', '60', '']}]}
{'102': ['102', 'not', {'102 between 30 and 60': ['102', 'between', '30', 'and', '60', '']}]} True
{'32': ['32', 'not', {'32 between 30 and 60': ['32', 'between', '30', 'and', '60', '']}]}
{'32': ['32', 'not', {'32 between 30 and 60': ['32', 'between', '30', 'and', '60', '']}]} False
{'80': ['80', 'not', {'80 between 30 and 60': ['80', 'between', '30', 'and', '60', '']}]}
{'80': ['80', 'not', {'80 between 30 and 60': ['80', 'between', '30', 'and', '60', '']}]} True
{'110': ['110', 'not', {'110 between 30 and 60': ['110', 'between', '30', 'and', '60', '']}]}
{'110': ['110', 'not', {'110 between 30 and 60': ['110', 'between', '30', 'and', '60', '']}]} True
{'56': ['56', 'not', {'56 between 30 and 60': ['56', 'between', '30', 'and', '60', '']}]}
{'56': ['56', 'not', {'56 between 30 and 60': ['56', 'between', '30', 'and', '60', '']}]} False
{'46': ['46', 'not', {'46 between 30 and 60': ['46', 'between', '30', 'and', '60', '']}]}
{'46': ['46', 'not', {'46 between 30 and 60': ['46', 'between', '30', 'and', '60', '']}]} False
{'54': ['54', 'not', {'54 between 30 and 60': ['54', 'between', '30', 'and', '60', '']}]}
{'54': ['54', 'not', {'54 between 30 and 60': ['54', 'between', '30', 'and', '60', '']}]} False
{'38': ['38', 'not', {'38 between 30 and 60': ['38', 'between', '30', 'and', '60', '']}]}
{'38': ['38', 'not', {'38 between 30 and 60': ['38', 'between', '30', 'and', '60', '']}]} False
{'0': ['0', 'not', {'0 between 30 and 60': ['0', 'between', '30', 'and', '60', '']}]}
{'0': ['0', 'not', {'0 between 30 and 60': ['0', 'between', '30', 'and', '60', '']}]} True
{'58': ['58', 'not', {'58 between 30 and 60': ['58', 'between', '30', 'and', '60', '']}]}
{'58': ['58', 'not', {'58 between 30 and 60': ['58', 'between', '30', 'and', '60', '']}]} False
{'60': ['60', 'not', {'60 between 30 and 60': ['60', 'between', '30', 'and', '60', '']}]}
{'60': ['60', 'not', {'60 between 30 and 60': ['60', 'between', '30', 'and', '60', '']}]} False
{'98': ['98', 'not', {'98 between 30 and 60': ['98', 'between', '30', 'and', '60', '']}]}
{'98': ['98', 'not', {'98 between 30 and 60': ['98', 'between', '30', 'and', '60', '']}]} True
{'120': ['120', 'not', {'120 between 30 and 60': ['120', 'between', '30', 'and', '60', '']}]}
{'120': ['120', 'not', {'120 between 30 and 60': ['120', 'between', '30', 'and', '60', '']}]} True
  id (str) #PK#  name (str)    dept_name (str)      tot_cred (int)
---------------  ------------  -----------------  ----------------
         00128  zhang         comp. sci.                      102
         19991  brandt        history                          80
         23121  chavez        finance                         110
         70557  snow          physics                           0
         98765  bourikas      elec. eng.                       98
         98988  tanaka        biology                         120

(smdb)>
```

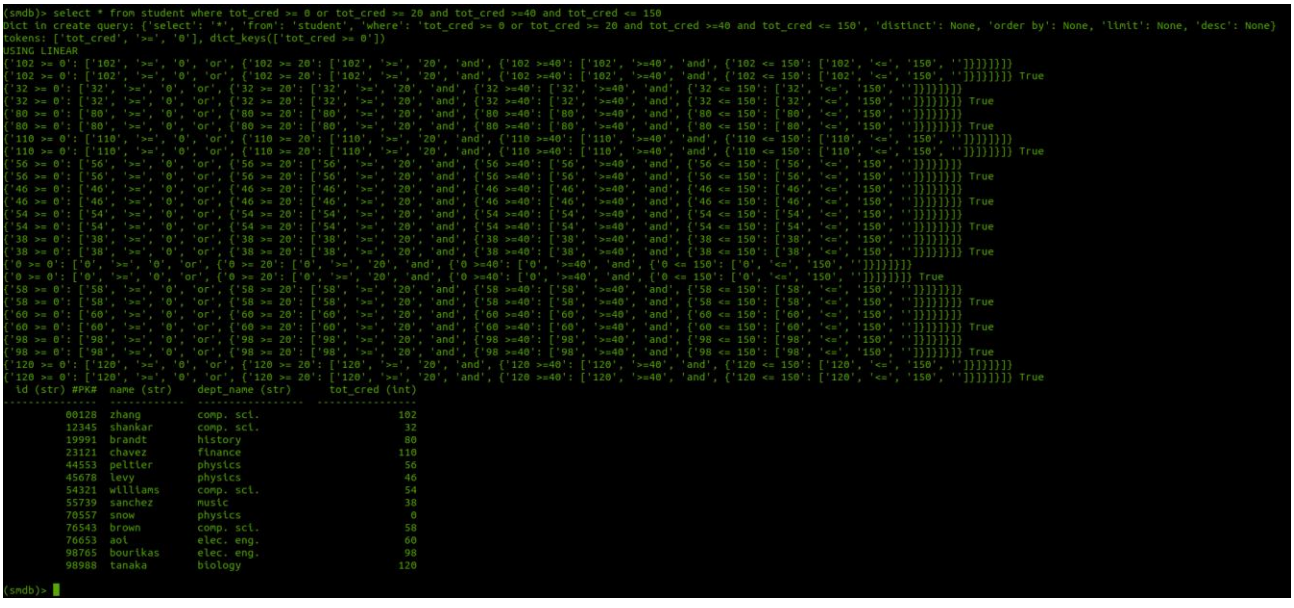
π.χ.2 Δύο between και or

Για το query: **select * from student where tot_cred between 80 and 150 or tot_cred between 0 and 30**



π.χ.3 Πολλαπλά or και and

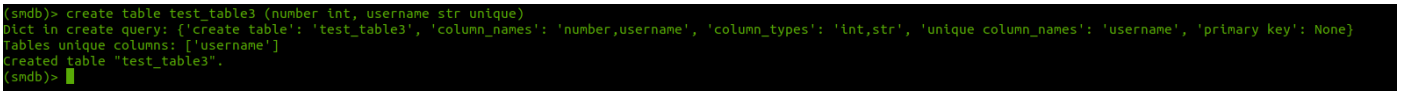
Για το query: **select * from student where tot_cred >= 0 or tot_cred >= 20 and tot_cred >=40 and tot_cred <= 150**



- **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)**

Για αυτό το ερώτημα αρχικά έχουν γίνει αλλαγές για να επιτρέπεται η δημιουργεία unique columns πέρα από το primary key και η δημιουργία index σε unique column. Έχω προσθέσει ώς optional argument στην δημιουργία ενός table μια λίστα με τα ονόματα των unique columns του table. Έτσι ο χρήστης μπορεί δίπλα από το όνομα μιας στήλης κατά την δημιουργία του table να γράψει το keyword 'unique' ώστε η συγκεκριμένη στήλη να θεωρηθεί unique, να αποτρέπετε ο χρήστης από το να εισάγει records που κάνουν overlap με βάση αυτή και να μπορεί να δημιουργεί indexes πάνω σε αυτή.

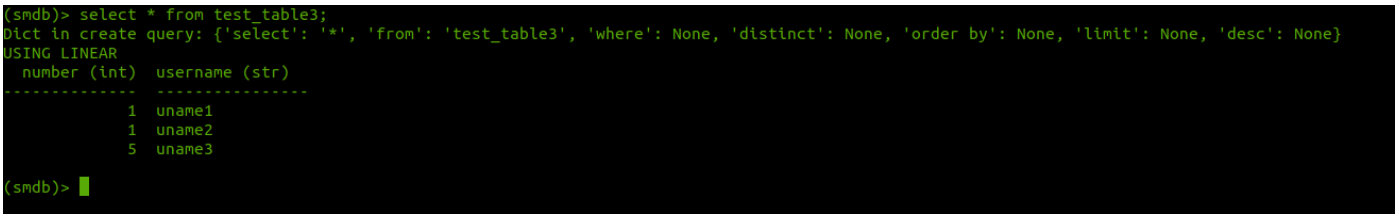
Παράδειγμα δημιουργίας table με όνομα test_table3 στύλες number (int), και username (string και unique).



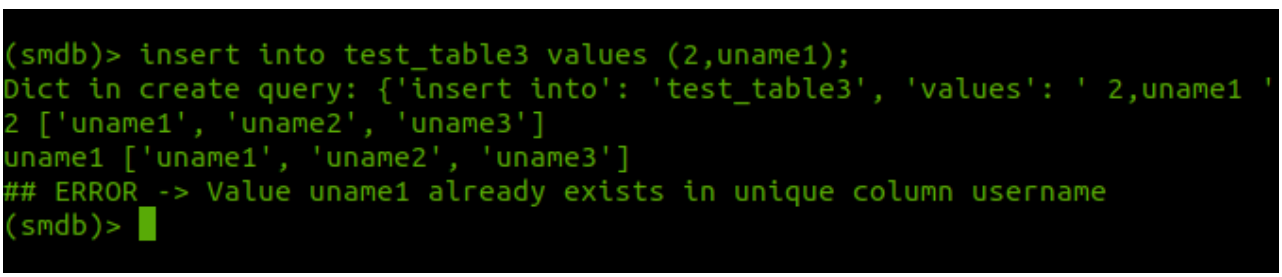
Μέσω μερικών print statements βλέπουμε ότι το dictionary που αποθηκεύει το query αποθηκεύει την επιπλέον πληροφορία 'unique_column_names' που στην συγκεκριμένη περίπτωση είναι η 'username', και ενημερώμαστε και παρακάτω για αυτό.

Τώρα θα βάλουμε μερικά records στο test_table3.

Μετά από μερικά inserts το test_table3 έχει ώς εξής:



Αν τώρα προσπαθήσουμε να βάλουμε το record (2, uname1) θα πάρουμε error καθώς υπάρχει ήδη record με username = uname1.



Μετά από μερικά ακόμα inserts το test_table3 έχει ώς:

number (int)	username (str)
1	uname1
1	uname2
5	uname3
2	uname4
10	uname5
12	uname6

Για το Hash Index:

Για την υποστήριξη Index με Extentible Hash έχω φτιάξει το αρχείο extendible_hash.py που περιέχει την κλάση EntentibleHash. Η κλάση αυτή λειτουργεί με τον τρόπο που ορίζεται στις διαφάνειες του μαθήματος και υπάρχουν πολλά comments στο αντίστοιχο αρχείο για την λειτουργία του. Παρόλο που με τον τρόπο που έχει φτιαχτεί θα μπορούσε να υποστιρίζει δυπλότηπα records για τις απαιτήσεις της εκφώνησης έχω προσθέσει τους απαραίτητους περιορισμούς για να μην συμβαίνει αυτό.

Ο χρήστης μπορεί να φτιάξει ένα ExtentibleHash index σε κάποια unique στήλη ώς εξής:

create index index_name on table_name**.**column_name using index_type

Όπου η '.' σηματοδοτεί το table και την στήλη που θέλουμε να φτιαχτεί ο index πάνω και index_type είναι είτε btree είτε ehash.

Δημιουργία ExtentibleHash index:

```
(smdb)> create index test_ind3 on test_table3.username using ehash;
Dict in create query: {'create index': 'test_ind3', 'on': 'test_table3', 'using': 'ehash', 'column': 'username'}
Creating index: test_ind3, on: test_table3.username
Saving index: test_ind3, indexed column: username
(smdb)>
```

Εδώ καταφέραμε να φτιάξουμε το ehash index , και με ένα print στην save index ενημερωνόμαστε ότι επίσης αποθηκεύτηκε.

Σημείωση: Πλέον η save_index αποθηκεύει μια λίστα που αποτελείτε από το index object και την στήλη πάνω στην οποία είναι.

Διάφορες μεθόδοι όπως create_index, construct_index save_index, drop_index έχουν αλλάξει ελάχιστα (και έχουν προστεθεί μερικά ακόμα arguments) με βάση αυτό, συγκεκριμένα η construct_index έχει αλλάξει ώς εξής:

```
def _construct_index(self, table_name, index_name, index_type, column_name):
'''
Construct a btree on a table and save.
Args:
table_name: string. Table name (must be part of database).
index_name: string. Name of the created index.
index_type: string. The type of the created index, either btree or ehash
column_name: string. The name of the indexed column
'''
index = None
if index_type == 'btree':
index = Btree(3) # 3 is arbitrary
elif index_type == 'ehash':
index = ExtendibleHash(method='lsb')
#if on_pk[0]:
# # for each record in the primary key of the table, insert its value and index to the btree
# for idx, key in enumerate(self.tables[table_name].column_by_name(self.tables[table_name].pk)):
# if key is None:
# continue
# index.insert(key, idx)
#else:
```

```
# for idx, key in enumerate(self.tables[table_name].column_by_name(on_pk[1])):
# if key is None:
# continue
# index.insert(key, idx)
for idx, key in enumerate(self.tables[table_name].column_by_name(column_name)):
if key is None:
continue
index.insert(key, idx)
self._save_index(index_name, column_name, index) # saving : name of the index, name of index column, index object
```

Και ο έλεγχος για δημιουργία index πάνω σε unique ή primary key γίνεται στην create_index έτσι:

```
if column_name != self.tables[table_name].pk and column_name not in self.tables[table_name].unique_column_names:
raise Exception('Cannot create index. Given column is not unique or primary key.')
```

Επιπλέον έχει δημιουργηθεί στο table.py η μέθοδος select_where_with_ehash η οποία λειτουργεί με πολύ όμοιο τρόπο με την select_where_with_btree

Παράδειγμα χρήσης ExtendibleHash index:

```
(smdb)> select * from test_table3 where username = uname3;
Dict in create query: {'select': '*', 'from': 'test_table3', 'where': 'username = uname3', 'distinct': None, 'order by': None, 'limit': None, 'desc': None}
tokens: ['username', '=', 'uname3'], dict_keys(['username = uname3'])
tokens: ['table_name=test_table3'], dict_keys(['table_name=test_table3'])
USING LINEAR
Loading index: test_ind3, indexed_column: username
USING EHASH
condition_plan: {'uname1 = uname3': ['uname1', '=', 'uname3', '']}, result: False
condition_plan: {'uname2 = uname3': ['uname2', '=', 'uname3', '']}, result: False
condition_plan: {'uname3 = uname3': ['uname3', '=', 'uname3', '']}, result: True
condition_plan: {'uname4 = uname3': ['uname4', '=', 'uname3', '']}, result: False
condition_plan: {'uname5 = uname3': ['uname5', '=', 'uname3', '']}, result: False
condition_plan: {'uname6 = uname3': ['uname6', '=', 'uname3', '']}, result: False
Rows from ehash find: [2], final_rows: [2]
  number (int)  username (str)
--------------  ---------------
             5  uname3

(smdb)>
```

Όπως βλέπουμε εδώ από το μήνυμα USING EHASH ενημερωνόμαστε από την database select ότι υπάρχει ehash index πάνω στην στήλη που ρωτάμε. Στην select_where_with_ehash έχω προσθέσει ένα print statement που τυπώνει το condition plan που δημιουργείτε και το αποτέλεσμα αυτού. Από το 'Rows from ehashfind [7]' ενημερωνόμαστε ότι υπάρχει 1 μόνο εγγραφή με αριθμό 7 (όπως αναμέναμε). Σαφώς με την ίδια λογική με παραπάνω μπορούμε και εδώ να κάνουμε σύνθετα ερωτήματα με πολλαπλά and, or κ.λπ . (Το 'USING LINEAR' αναφέρεται στο select που κάνει το meta_indexes table που δεν περιέχει κάποιο index).

Για να γίνει το select με χρήση ehash πρέπει το query να μην είναι ερώτηση διαστήματος αλλά ισότητας. Αυτό το ελέγχουμε μέσω της condition_hanlder.is_range_query.

Επιπλέον πρέπει στο πρώτο subcondition να βρίσκεται η στήλη στην οποία υπάρχει ο index. Δηλαδή αν είχαμε το query: id > 3 and name = john και το index υπήρχε στην στήλη name η αναζήτηση θα γινόταν σειριακά. Για να λυθεί αυτό το πρόβλημα στο αρχείο optimizer.py υπάρχει η μέθοδος swap_condition_tokens_of_where που θα αντιστρέψει το query και θα γίνει έτσι: name = john and id > 3 ώστε να αξοιοποιηθεί το index.

Σημείωση: Με τον ίδιο τρόπο μπορούμε σαφώς να φτιάξουμε και Btree index σε unique columns πέρα από το primary key

- **Implement miniDB's query optimiser by (a) building equivalent query plans based on respective RA expressions (10/50) and (b) choosing the optimal query execution plan using a cost-based evaluation (20/50)**

Για τον ερώτημα αυτό έχω φτιάξει την κλάση Optimizer στο αρχείο optimizer.py.

Δύστυχος δεν μπόρεσα να ολοκληρώσω τα ερωτήματα και το μόνο πράγμα που μπορεί να κάνει ο optimizer είναι να αντιστρέψει την σειρά των tokens όπως ανέφερα παραπάνω ώστε να αξιοποιηθεί τυχόν index.

Σημείωση: Για αναλυτικότερες πληροφορίες σχετικά με τις αλλαγές στα αρχεία και τον τρόπο λειτουργίας διάφορων κλάσεων/συναρτήσεων παρακαλώ κοιτάξτε τα αντίστοιχα κομμάτια κώδικα, ελπίζω να αρκούν.