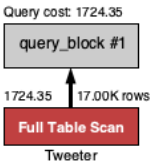
Last Name: SUH First Name: Joowon Student ID:44414081

1. [12 pts] To start down the path of exploring physical database designs (indexing) and query plans, start by checking the query plans for each of the following queries against the database without any indexes using the EXPLAIN function (see instructions). For each query, take snapshots of the EXPLAIN results and paste them into your copy of the HW7 template file. (Just take a snapshot of the query plan, not the whole screen.)

a) [1.5 pts]

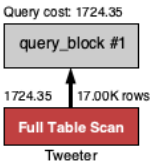
```
SELECT * FROM Tweeter WHERE followers_count BETWEEN 20000 AND 21000;
```



Query Plan:

b) [1.5 pts]

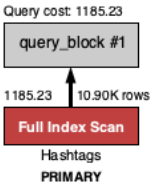
```
SELECT * FROM Tweeter WHERE followers_count BETWEEN 0 AND 21000;
```



Query Plan:

c) [3 pts]

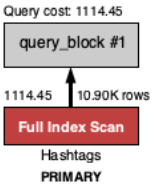
```
SELECT * FROM Hashtags WHERE hashtag LIKE '%vid%';
```



Query Plan:

d) [3 pts]

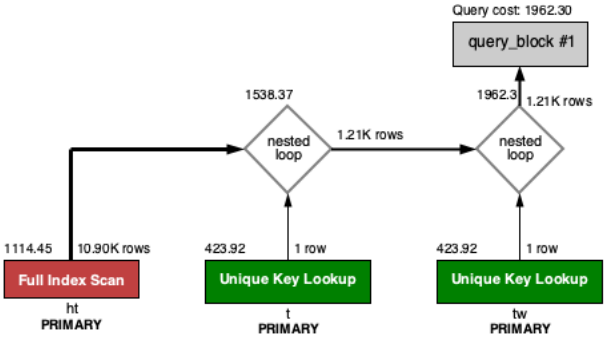
```
SELECT COUNT(*) FROM Hashtags WHERE hashtag = 'COVID19';
```



Query Plan:

e) [3 pts]

```
SELECT tw.handle
FROM Hashtags ht, Tweet t, Tweeter tw
WHERE ht.hashtag LIKE 'COV%'
AND ht.tweet_id = t.tweet_id
AND t.tweeter_id = tw.tweeter_id;
```


Query cost: 1962.30
query_block #1
1962.3
1.21K rows
1538.37
nested loop
1.21K rows
nested loop
1114.45
10.90K rows
Full Index Scan
ht
PRIMARY
423.92
1 row
Unique Key Lookup
t
PRIMARY
423.92
1 row
Unique Key Lookup
tw
PRIMARY

Query Plan:

2. [10 pts] Now create secondary indexes (which are B+ trees, under the hood of MySQL) on the Tweeter.followers_count attribute and Hashtags.hashtag *separately*. (I.e., create two indexes, one per table.) Paste your CREATE INDEX statements below.

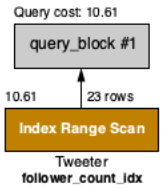
   CREATE INDEX follower_count_idx ON Tweeter(followers_count) USING BTREE;

   CREATE INDEX hashtagidx ON Hashtags(hashtag) USING BTREE;

3. [12 pts] Re-"explain" the queries in Q1 and **indicate whether the indexes you created in Q2 are used**, and if so, **whether the uses are index-only plans or not**. Copy and paste the query plan after each query, as before.

a) [1.5 pts]

```
SELECT * FROM Tweeter WHERE followers_count BETWEEN 20000 AND 21000;
```



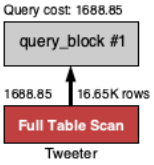
Query Plan:

Is index used? (y/n) : Yes

Is the plan index only? (y/n) : no

b) [1.5 pts]

```
SELECT * FROM Tweeter WHERE followers_count BETWEEN 0 AND 21000;
```



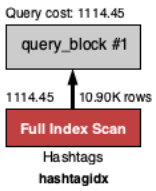
Query Plan:

Is index used? (y/n) : No

Is the plan index only? (y/n) : No

c) [3 pts]

```
SELECT * FROM Hashtags WHERE hashtag LIKE '%vid%';
```



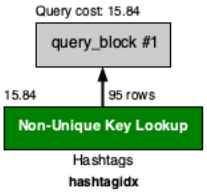
Query Plan:

Is an index used? (y/n) : Yes

Is the plan index only? (y/n) : no

d) [3 pts]

```
SELECT COUNT(*) FROM Hashtags WHERE hashtag = 'COVID19';
```



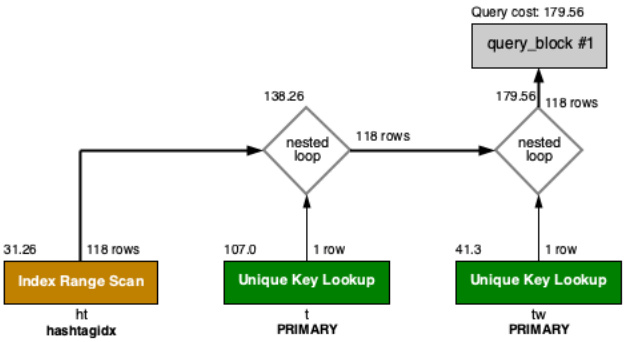
Query Plan:

Is an index used? (y/n) : yes

Is the plan index only? (y/n) : yes

e) [3 pts]

```
SELECT tw.handle
FROM Tweet t, Tweeter tw, Hashtags ht
WHERE ht.hashtag LIKE 'COV%'
AND ht.tweet_id = t.tweet_id
AND t.tweeter_id = tw.tweeter_id;
```



Query Plan:

Is index used? (y/n) : Yes

For which table? Hashtags

4. [21 pts] Examine **each** of the above queries **with** and **without** the use of an index. Please **briefly** answer the following questions after pondering what you have seen.

a) [5 pts] For both queries 3a and 3b:

i) Explain the differences between their produced plans:

3a has less cost than the 3b, 3a is using index while 3b is not using index.

ii) Why are the two queries (3a and 3b) producing different plans?

Because 3a starts from the page which has followers_count=20000 as 3b starts from the beginning.

b) [6 pts] For the LIKE query (**c**), explain **whether** an index is useful and **why** or **why not.**

i)

Is an index useful? (y/n) : No

Explanation: The “like” in this query compares if the record has the “~vid~” part inside, which means it will compare with all the records because the first letter is not specified for searching. even though we make index.

ii)

What if the query is changed to something like:

```
SELECT * FROM Hashtags WHERE hashtag LIKE 'C%19';
```

In your explanation, if an index is used and is beneficial to perform the query, include the search key that would be used. Assume the number of result records selected by the index (if an index is used) is extremely small compared to the total number of records in the file.

Is an index useful? (y/n) : Yes

Explanation: For this example, System will try to find hashtag that starts with letter ”C” and finish with “19”, which means we can reduce cost by using index, because system does not have to check hashtag that is not starting with letter “C” and ending with “19”.

c) [10 pts] For join queries (e.g., query 3**e**), answer the following questions briefly (assuming the number of tweets that contain hashtags like 'COV%' is extremely small compared to the total number of tweets in the file).

Is an index useful? (y/n) : yes

Why is the index useful or not useful? (explain briefly => 2 sentences): Because with index, system does not have to access the records which does not start with “COV”. In other words, system can reduce cost.

In what order the join was performed (your answer’s format should be TblName1, TblName2, TblName3, when TblName1 and TblName2 are joined first and TblName3 joined after)? Hashtags, Tweet, Tweeter

What kinds of joins are being performed? Why? Index Nested Loop Join has been used. What we want to get from this query is handle of the tweeter who tweeted with hashtag starts with “COV”. First, find hashtag that has “COV~”. Next check equality on the tweet_ids, this can be done nested loop, and so on. This logic is well fit with INLJ.

Use the query below to inspect the plan that produces the other join order. Having **STRAIGHT_JOIN** after **SELECT** forces the join order to be performed as the tables’ names appear in the **FROM** clause.

```
SELECT STRAIGHT_JOIN tw.handle
FROM Tweet t, Tweeter tw, Hashtags ht
WHERE ht.hashtag LIKE 'COV%'
AND ht.tweet_id = t.tweet_id
AND t.tweeter_id = tw.tweeter_id;
```

What kinds of joins are being performed in this case? Same INLJ

Is this query cheaper to perform (w.r.t the join order) compared to the join order picked by MySQL? Why? No, since if loop starts with tweet we cannot narrow down the entries to look up, which means this query will start with reading all the entries of tweet and joining them with other table’s entry. So it needs more time than the join order of MySQL

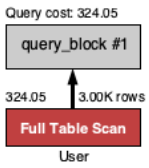
5. [21 pts] It's time to go one step further and explore the notion of a "*composite Index*", which is an index that covers several fields together.

a) [5 pts] Create a composite index on the attributes name_first and name_last (*in that order!*) of the User table. Paste your CREATE INDEX statement below.
CREATE INDEX user_name_idx ON User(name_first, name_last)

b) [6 pts] 'Explain' the queries 1) and 2) below. Copy and paste the query plan of each query into your response, as before. Be sure to look carefully at each plan.

Query 1)

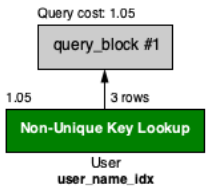
```
SELECT * FROM User WHERE name_last = 'Hernandez';
```



Query Plan:

Query 2)

```
SELECT * FROM User WHERE name_first = 'Michael' AND name_last = 'Burton';
```



Query Plan:

c) [10 pts] For each query, explain whether the composite index is **used or not** and **why**.

Query 1)

Is index used? (y/n) : No

Explanation: Because the columns in the query is not forming the leftmost prefix of the index.

Query 2)

Is index used? (y/n) : Yes

Explanation: Because coloumns in the query is forming leftmost prefix of the index(first, last)

6. [24 pts] Now assume you are working with a different system (that isn't MySQL) that allows you to specify if an index is clustered or not. To create a *clustered* index in this hypothetical system on User (first_name), one would execute the statement:

```
CREATE CLUSTERED INDEX userFirstNameIdx ON User(first_name);
```

To create an *unclustered* index on User (last_name), one would execute the statement:

```
CREATE UNCLUSTERED INDEX userLastNameIdx ON User(last_name);
```

For each of the queries below, specify the most appropriate CREATE INDEX statement that would build a helpful index to accelerate that query. If an index won't be useful for a given query, then write 'No index'.

a) [8 pts]

```
SELECT * FROM Phone WHERE user_id = 2939;
```

Answer: `CREATE CLUSTERED INDEX user_id_idx ON Phone (user_id);`

b) [8 pts] (**Note**: CheckedTweets.org started their service in 2020…)

```
SELECT * FROM Checker WHERE checker_since > '2019-01-01 06:12:35';
```

Answer: No index

c) [8 pts]

```
SELECT user_id, COUNT(*) as cnt FROM EvidenceFrom GROUP BY user_id;
```

Answer `CREATE UNCLUSTERED INDEX user_id_idx ON EvidenceFrom (user_id);`