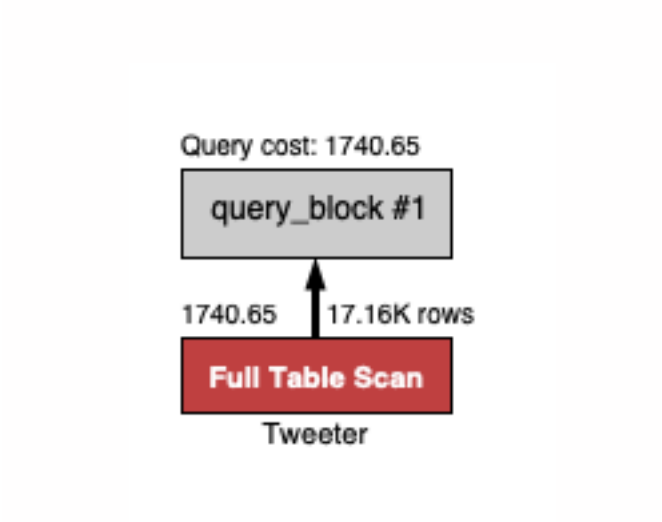
1. To start down the path of exploring physical database designs (indexing) and query plans, start by checking the query plans for each of the following queries against the database without any indexes using the EXPLAIN function (see instructions). For each query, take snapshots of the EXPLAIN results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

   a)

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

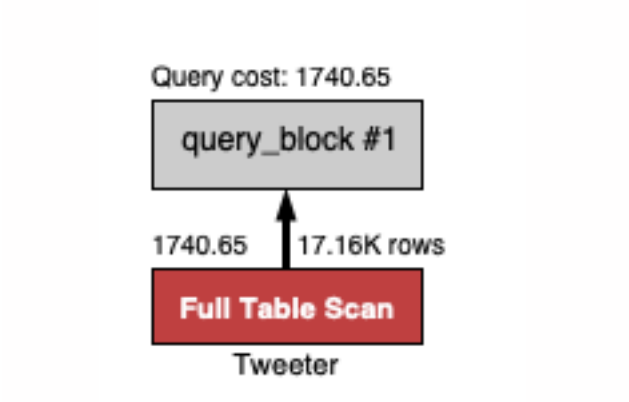
   Query Plan:



   b)

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

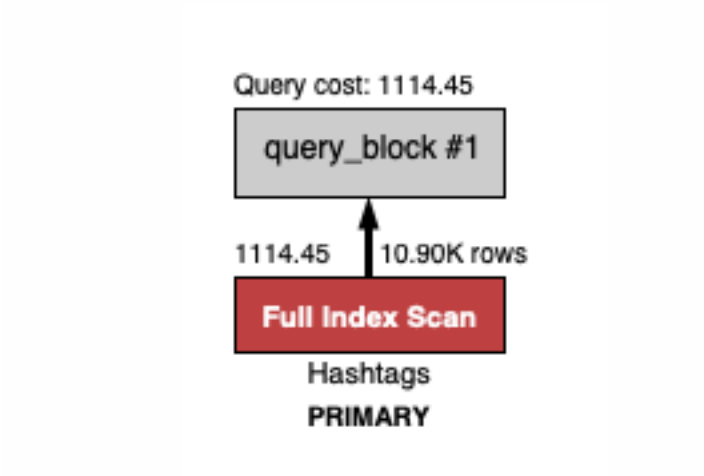
   Query Plan:



c)

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

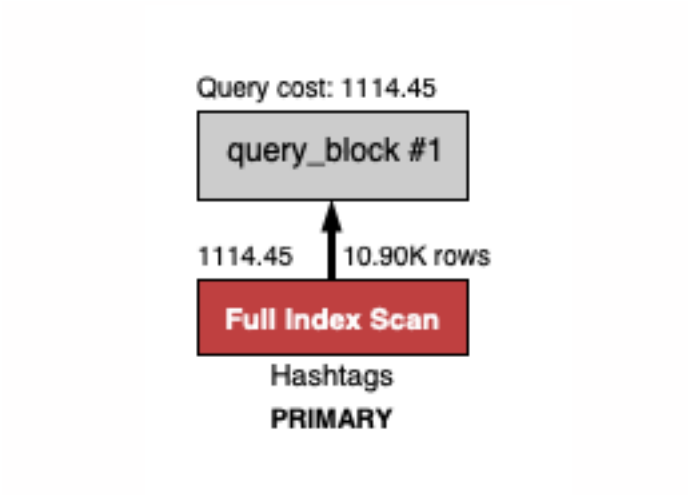
Query Plan:



d)

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

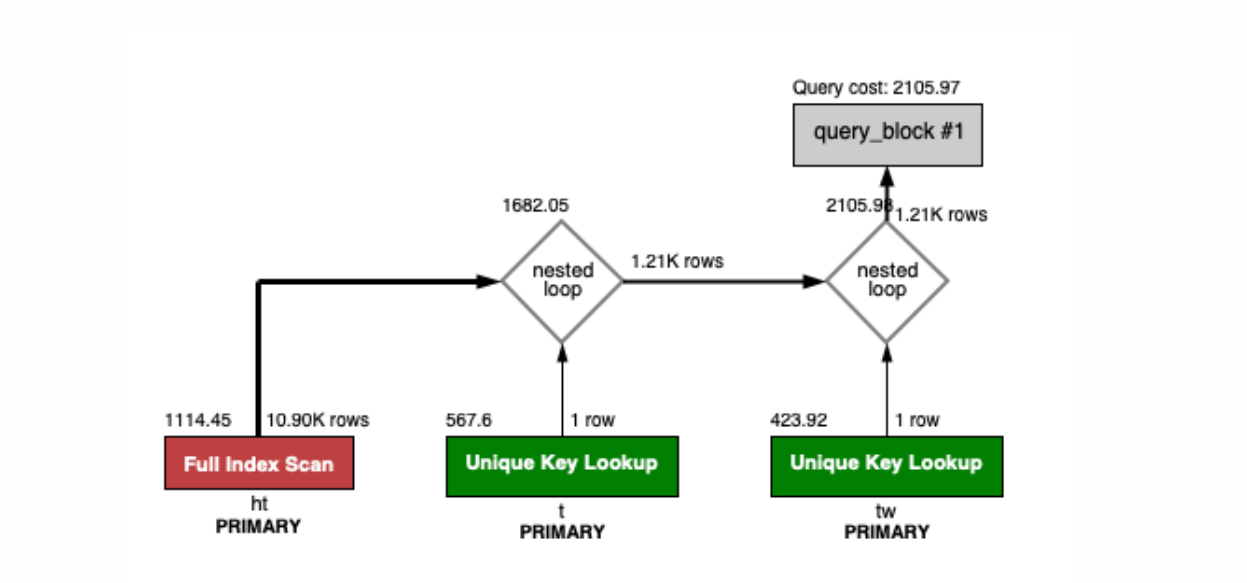
Query Plan:



e)

```
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 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.

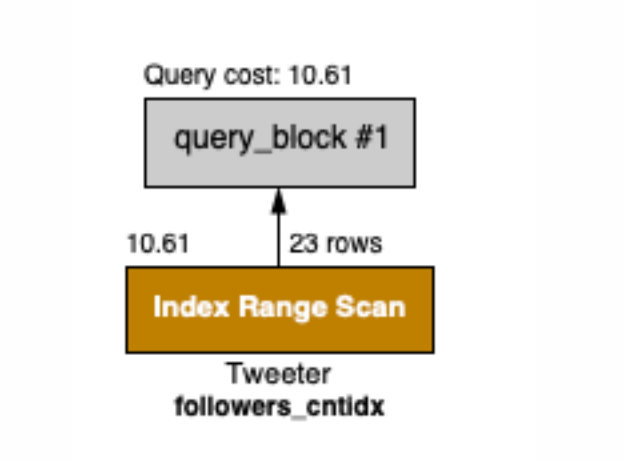
```
1 • CREATE INDEX followers_cntidx on Tweeter (followers_count) USING BTREE;
2 • CREATE INDEX hashtagidx on Hashtags (hashtag) USING BTREE;
```

3. 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)

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

Query Plan:



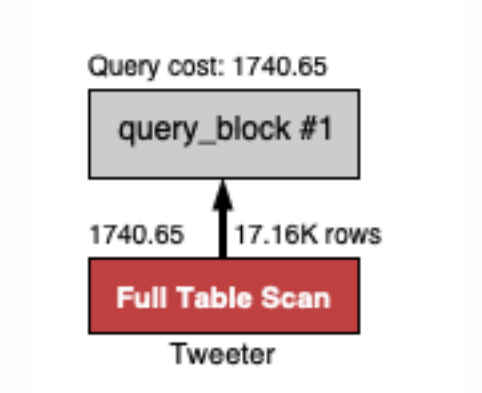
Is index used? (y/n) : Yes

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

b) [1.5 pts]

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

Query Plan:



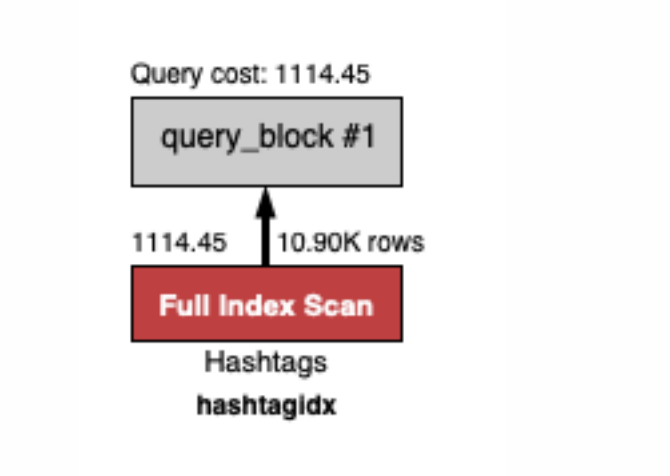
Is index used? (y/n) : No

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

c) [3 pts]

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

Query Plan:



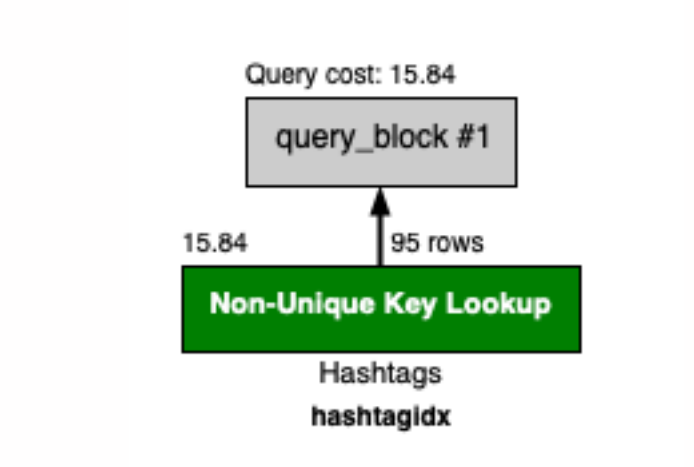
Is an index used? (y/n) : Yes

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

d) [3 pts]

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

Query Plan:



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

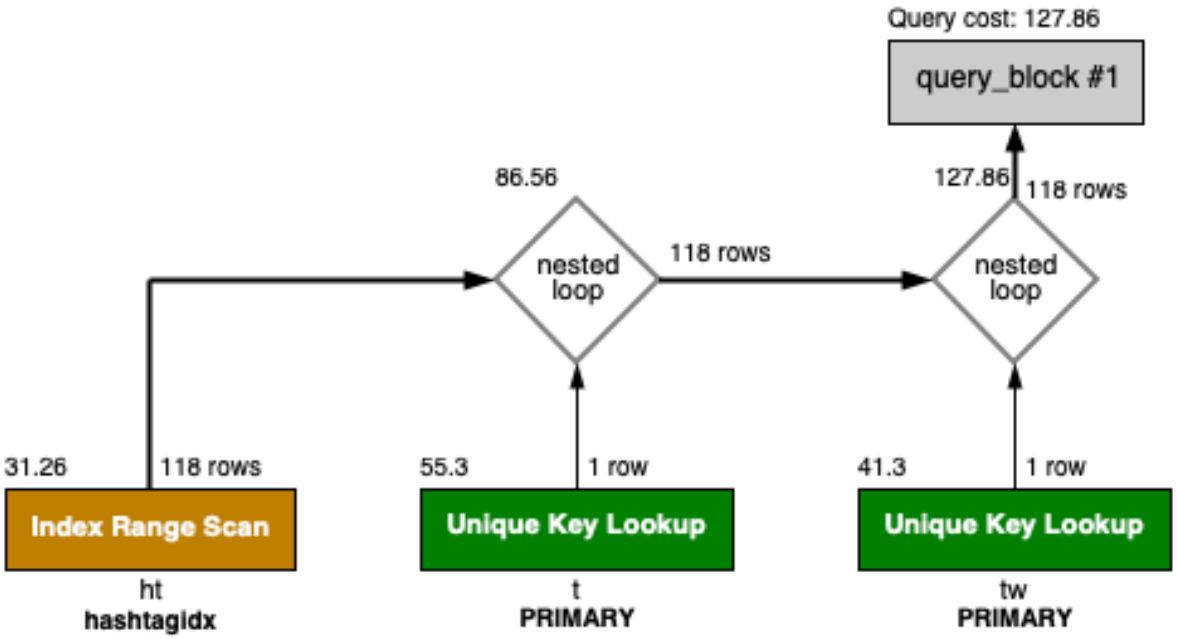
Is the plan index only? (y/n) : Yes

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. 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) For both queries 3a and 3b:

i) Explain the differences between their produced plans:

3a performs an index range scan using index hashtag_cntindx, while 3b does a full table scan.

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

Because the results in part 3b encompasses almost the whole table (16305 out 16929 rows), but the returned results in part 3a includes just a small portion of contingent rows (23 rows) using hashtag_cntindx.

b) 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:

No, because the index sorts the hashtag based on lexicographical order. As the first letter in the LIKE expression is '%', it could match with any letter and it doesn't change the range of search.

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:

It's because that in this LIKE expression the first letter is specified as 'C' and using the already built index hashtagidx, it will narrow down the search to all of the tags that they start with 'C'. As the number of result records selected by the index is extremely small compared to the total number of records in the file, this will significantly reduce the query cost.

c) 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, the index is very selective and significantly narrows down the search space. More specifically, instead of joining all of the hashtags with the tweet table, we just join the hashtags that they have 'COV' as their prefix.

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)?

Hashtag, Tweet, Tweeter

What kinds of joins are being performed? Why?

Index Nested loop Join, because it applies the joins in the nested fashion by first looking at the outer table Hashtag, and then using the common column between Hashtag and Tweet to bind on Tweet_id which is Tweet's primary key. Finally, biding the result on Tweeter_id from Tweeter which is its primary key. Since, the number of hashtags like 'COV%' is extremely small, this will significantly narrows down the search space.

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?

Index Nested loop joins in both joins

Is this query cheaper to perform (w.r.t the join order) compared to the join order picked by MySQL? Why?

No, it isn't cheaper. This will start with a full scan on Tweet table which is very slow, and then it joins it with table Tweeter, which returns around 20k rows and the final join would be between the resulting massive 20k rows and all the hashtags meeting LIKE 'COV%' condition.

5. 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) 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.

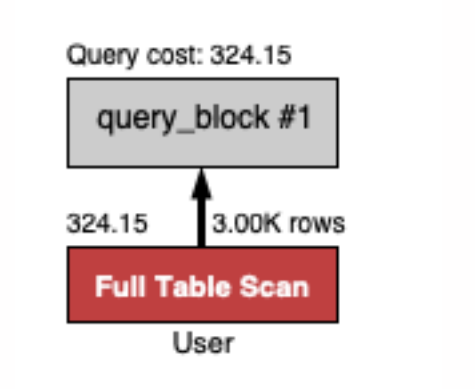
```
1  CREATE INDEX nameidx on User (name_first, name_last) USING BTREE;
```

b) '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';
```

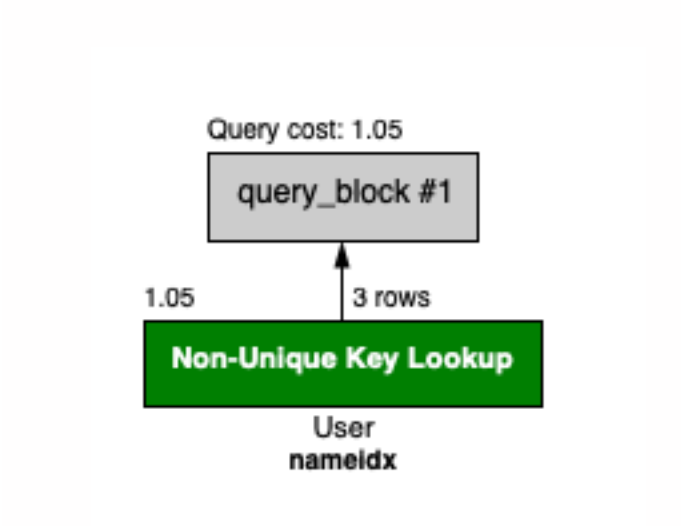
Query Plan:



Query 2)

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

Query Plan:



c) 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 composite index is first built on name_first and it's not specified here.

Query 2)

Is index used? (y/n) : Yes

Explanation:

Because it uses the composite index using the first_name and last_name and narrows down the search space to the Users that their first_name = Michael and last_name = Burton.

6. 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)

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

Answer:

```
CREATE CLUSTERED INDEX user_ididx ON Phone (user_id);
```

b) (**Note**: CheckedTweets.org started their service in 2020…)

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

Answer:
No Index.

c)

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

Answer:

```
CREATE UNCLUSTERED INDEX user_idEvidenceFromidx ON EvideceFrom (user_id);
```