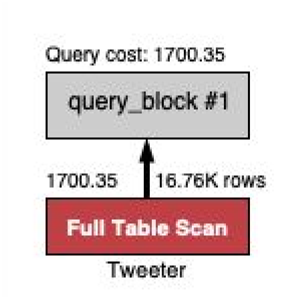
Last Name: First Name: Student ID:

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;
```

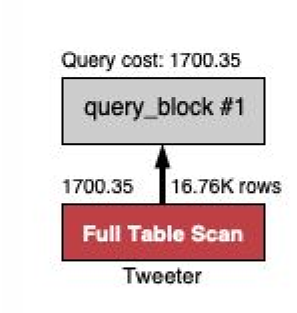
Query Plan:



b) [1.5 pts]

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

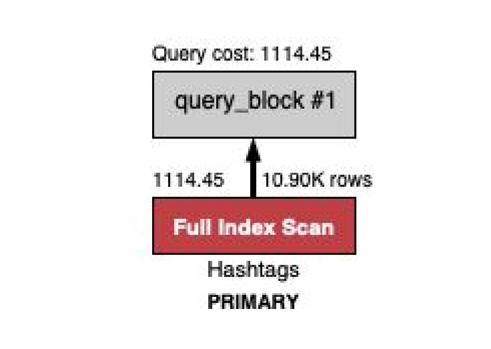
Query Plan:



c) [3 pts]

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

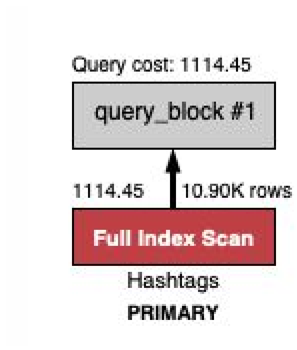
Query Plan:



d) [3 pts]

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

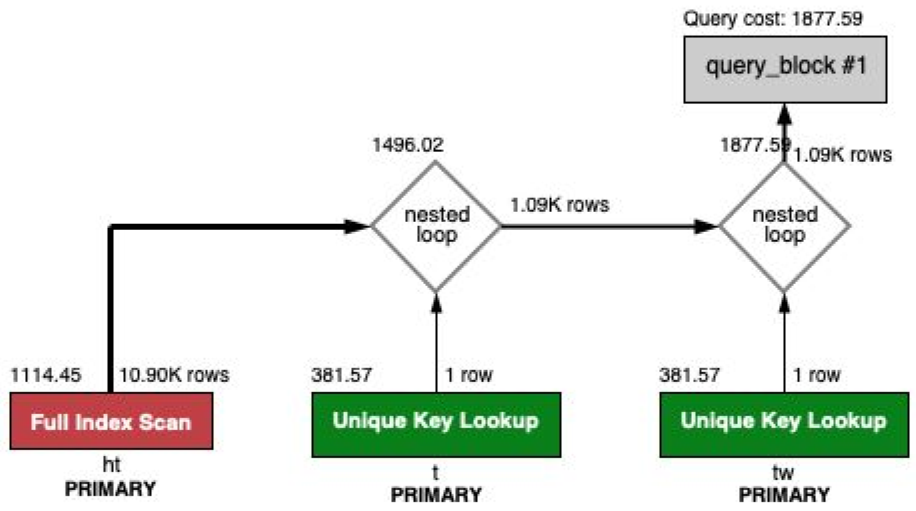
Query Plan:



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 Plan:


Query cost: 1877.59
query_block #1
1877.59
1.09K rows
1496.02
nested loop
1.09K rows
nested loop
1114.45
10.90K rows
Full Index Scan
ht
PRIMARY
381.57
1 row
Unique Key Lookup
t
PRIMARY
381.57
1 row
Unique Key Lookup
tw
PRIMARY

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.

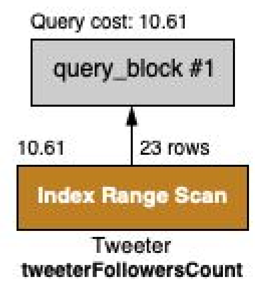
```
CREATE INDEX tweeterFollowersCount ON Tweeter (followers_count);
CREATE INDEX hashtagsIdx ON Hashtags(hashtag);
```

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:



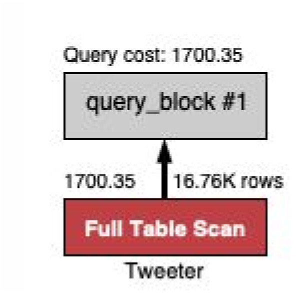
Is index used? (y/n) : y

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

b) [1.5 pts]

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

Query Plan:



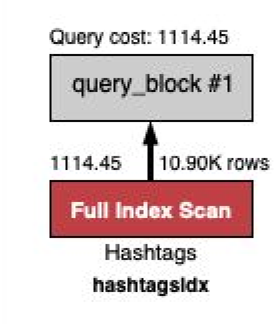
Is index used? (y/n) : n

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

c) [3 pts]

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

Query Plan:



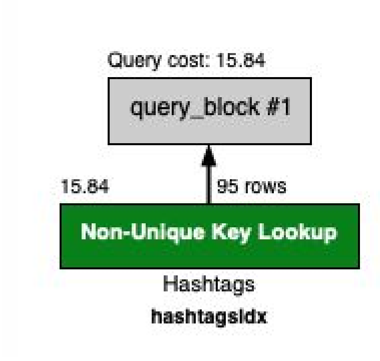
Is index used? (y/n) : y

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

d) [3 pts]

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

Query Plan:



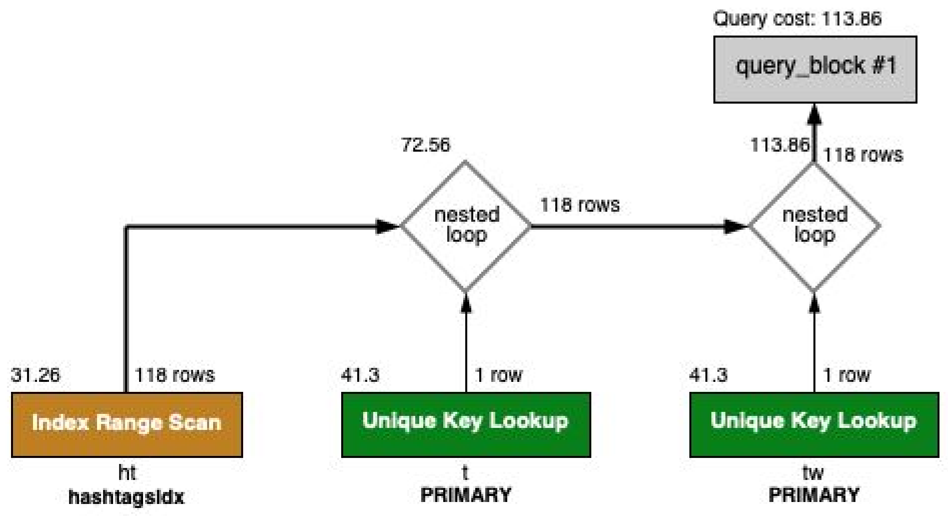
Is index used? (y/n) :  y

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

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) : y

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 plans:

Query 3a uses index tweeterFollowersCount to answer the query whereas query 3b performs a full index scan.

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

The range query 3a (20000 <= followers_count <= 21000) returns only 23 rows out of 16929. Therefore, MySQL will search for all entries that fall within the specific range using the tweeterFollowersCount index, and it will cost less than using a Full Index Scan.

The range query 3b (0 <= followers_count <= 21000) returns 16305 records out of 16929. This is mostly all Tweeters we have in CheckedTweets.org. This would incur a large number of random reads for each entry, which could negatively affect the performance of executing the query.

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) : n

Explanation: The query (LIKE '%vid%') uses the index but does a full index scan, because it isn't a range query that can be handled by a B+ tree index since the condition is not specifically on the prefix but any part of the key.

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) : y

Explanation: The prefix ("C") would be used to perform the index range scan. Since it is extremely selective, it will cost less than using a full table scan.

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) : y

Why is the index useful or not useful? (explain briefly => 2 sentences):

The index is useful because we first use the index hashtagsIdx to filter out all hashtags that do not satisfy (LIKE 'COV%') condition before performing the join with Tweet and Tweeter.

In what order is the join performed (your answer's format should be TblName1, TblName2, TblName3, when TblName1 and TblName2 are joined first and TblName3 joined after)?

Hashtags, Tweets, Tweeter

What kinds of joins are being performed? Why?

Index-Nested-Loop-Joins (INLJ) were performed in both joins.

The first join between **Hashtags** and **Tweet** is on "tweet_id" which is the primary key of Tweet. Since the number of tweets that contain hashtags that satisfy (LIKE 'COV%') condition is small, using INLJ here is efficient to lookup for the tweets that contain hashtags. The second join between the result of joining both Hashtags and Tweet with Tweeter is on “tweeter_id”, which is the primary key of Tweeter. Performing INLJ for the last join is efficient as the result of the first join is small.

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 were performed 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 is not cheaper

In terms of performance, this would be slower as we will do a full table scan on the Tweet table. Then, we join Tweet rows with the table Tweeter. The result cardinality after joining both tables would be 20,000. The final join would be between the resulting 20,000 rows and all hashtags that satisfy (LIKE 'COV%') condition.

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.

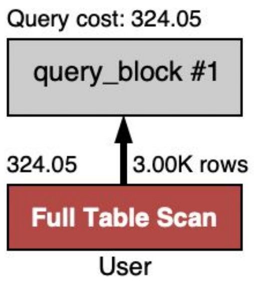
```
CREATE INDEX userNameIdx ON User (name_first, name_last);
```

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';
```

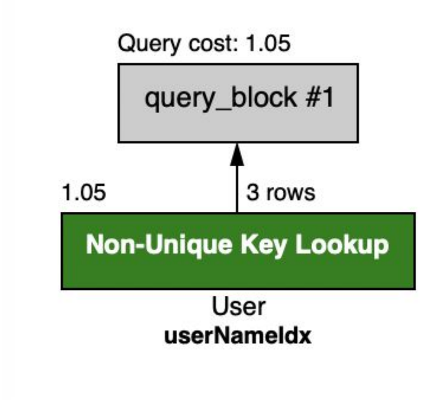
Query Plan:



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) : n

Explanation: The composite index is NOT used, because the index entries are grouped by name_first 'first', and name_last 'second'. In order to find the last name 'Hernandez', we would have to scan the entire index. This is slower than just performing a full table scan (which is what is done here).

Query 2)

Is index used? (y/n) : y

Explanation: The composite index is used, because we minimally specify the first attribute of our index and we are performing an equality search on this attribute. This then becomes a B+ tree lookup for name_first, then name_last (which is faster than doing a full table scan).

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 userIdPhoneIdx 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 userIdEvidenceFromIdx ON EvidenceFrom(user_id);
```