CSCI143 Final, Spring 2023

Collaboration policy:

You may NOT:

1. discuss the exam with any human other than Mike; this includes:
   (a) asking your friend for clarification about what a problem is asking
   (b) asking your friend if they've completed the exam
   (c) posting questions to github

You may:

1. take as much time as needed
2. use any written notes / electronic resources you would like
3. use the lambda server
4. ask Mike to clarify questions via email

Name:

The questions below relate to the following simplified normalized twitter schema.

```
CREATE TABLE users (
    id_users BIGINT PRIMARY KEY,
    name TEXT UNIQUE NOT NULL,
    description TEXT
);

CREATE TABLE tweets (
    id_tweets BIGINT PRIMARY KEY,
    id_users BIGINT REFERENCES users(id_users),
    created_at TIMESTAMPTZ CHECK (created_at > '2000-01-01'),
    country_code VARCHAR(2) NOT NULL,
    lang VARCHAR(2) NOT NULL,
    text TEXT NOT NULL
);

CREATE TABLE tweet_tags (
    id_tweets BIGINT,
    tag TEXT,
    PRIMARY KEY (id_tweets, tag),
    FOREIGN KEY (id_tweets) REFERENCES tweets(id_tweets)
);
```

1. (8pts) Recall that certain constraints create indexes on the appropriate columns. List the equivalent CREATE INDEX commands that are run by the constraints above.

Solution:

```
CREATE UNIQUE INDEX ON users(id_users);

CREATE UNIQUE INDEX ON users(name);

CREATE UNIQUE INDEX ON tweets(id_tweets);

CREATE UNIQUE INDEX ON tweet_tags(id_tweets, tag);
```

Common mistakes:

1. You got full credit whether you included the UNIQUE clause or not in your commands.
2. (-3pts) Getting the column order wrong on the last command. The order of the columns in the last command has no impact on the semantics, but it does impact performance.

2. (8pts) Create index(es) so that the following query will run as efficiently as possible. Do not create any unneeded indexes.

   HINT: Pay careful attention to the column list.

```
SELECT DISTINCT tag
FROM tweet_tags
WHERE
    lower(tag) LIKE 'corona%';
```

Solution:

```
CREATE INDEX ON tweet_tags(lower(tag), tag);
```

OR

```
CREATE INDEX ON tweet_tags(tag)
WHERE lower(tag) LIKE 'corona%';
```

Our goal is to perform an index only scan and not have to do an explicit sort to resolve the DISTINCT clause. Technically, the second index will be quite a bit better than the first: it will use less disk space and have a better fanout. But I promissed in class that you wouldn't need to create a partial index for the final, and so the first index above is also acceptable.

Commomn mistakes:

1. (-6) If you created the index

   ```
   CREATE INDEX ON tweet_tags(lower(tag));
   ```

   then you took advantage of the WHERE clause, but you still needed to perform an explicit sort to resolve the DISTINCT clause. (This is what the HINT was trying to get you to observe.)

2. (-8) If you created the index

   ```
   CREATE INDEX ON tweet_tags(tag);
   ```

   then you don't need to perform an explicit sort, but you still need to scan the entire table or index in order to find which rows match the WHERE clause. Recall that postgres does not take advantage of the semantics of the functions that you use in the WHERE clause in order to speed things up. Instead, the WHERE clause condition must match exactly the columns in the index.

3. (-4) If you wrote

```
CREATE INDEX ON tweet_tags(lower(tag))
WHERE lower(tag) LIKE 'corona%';
```

including the `lower` function both in the column list and in the WHERE clause. This index won't avoid the explicit sort.

3. (16pts) Consider the following two queries, which differ only by the conjunction operation used in the WHERE clause.

   For each query: (1) Create index(es) so that the query will run as efficiently as possible. (2) State which scanning strategy you expect the Postgres query planner will use and explain why. (3) Describe which clauses of the query will be sped up with the table scanning strategy you selected in (2), and which clauses (if any) will not be sped up.

   HINT: One of these queries can be implemented very efficiently with an index only scan, and the other query cannot.

   a.
```
SELECT id_tweets
FROM tweets
WHERE country_code = :country_code
  AND lang = :lang
ORDER BY created_at
LIMIT 10;
```

Solution: Creating either of the following indexes

```
CREATE INDEX ON tweets(
    country_code,
    lang,
    created_at,
    id_tweets
    );
```

or

```
CREATE INDEX ON tweets(
    lang,
    country_code,
    created_at,
    id_tweets
    );
```

will result in an index only scan and take advantage of the WHERE, OORDER BY, and LIMIT clauses. Specifically, the index only scan will only touch the parts of the index that match the WHERE clause, it will return the results in the order specified by the ORDER BY clause, and it can stop early to take advantage of the LIMIT clause.

Common mistakes:

(a) (-4) Adding more columns to the indexes above. When performing a bitmap scan, extra indexes do not help. They actually hurt because they hurt the fanout. (They help for an index scan because they can "upgrade" the algorithm to an index only scan.)

(b) (-4) Adding an additional index on `tweets(created_at)`. The idea is that this index could be used to speed up the ORDER BY clause, but the ORDER BY can not be sped up with an bitmap scan, and everyone who wrote this index specified a bitmap scan as the most likely traversal.

Technically, the index

```
CREATE INDEX ON tweets(
    created_at,
    lang,
    country_code,
    id_tweets
    );
```

can be used with an index only scan. (The index only scan will be used for the ORDER BY clause to avoid a sort, and the WHERE clause will use a separate filter step that doesn't take advantage of the index.) For the particular data in the twitter dataset, if you specify that the `country_code` is the US and the `lang` is EN, then about 90% of all tweets will be returned. Therefore taking advantage of the WHERE clause won't provide much benefit, but taking advantage of the ORDER BY and LIMIT clauses will provide a huge benefit. In this case, the index specified above will be preferred by Postgres with an index only scan.

b.
```
SELECT id_tweets
FROM tweets
WHERE country_code = :country_code
   OR lang = :lang
ORDER BY created_at
LIMIT 10;
```

Solution: We must create both of the following indexes:

```
CREATE INDEX ON tweets(country_code);
CREATE INDEX ON tweets(lang);
```

Because of the OR clause, the best scan method we can achieve is a bitmap scan. The bitmap scan will speed up the WHERE clause, but cannot return results in sorted order, and so we will need an explicit sort for the ORDER BY clause. Also, the bitmap scan must build up a bitmap of size $O(n)$ regardless of the presence of the LIMIT clause, and so the LIMIT will not significantly speed up the scan. The LIMIT will cause the sort used in the ORDER BY clause to go faster because the entire results list will not need to be sorted.

4. (8pts) Create index(es) so that the following query will run as efficiently as possible. Do not create any unneeded indexes.

```
SELECT count(*)
FROM tweets
JOIN tweet_tags USING (id_tweets)
WHERE
    tag = :tag;
```

Solution: We need to create the new index

```
CREATE INDEX ON tweet_tags(tag, id_tweets);
```

and can re-use the existing index on `tweet(id_tweets)`. This will enable an index only can on `tweet_tags` to filter with the WHERE clause and the results will be returned in sorted order to use the merge join strategy.

Common mistakes:

1. (-6) Relying on the existing index on `tweet_tags(id_tweets, tag)` is incorrect because the column order is wrong.
2. (-6) An index on `tweet_tags(tag)` will not speed up the JOIN clause.

5. (8pts) Create index(es) so that the following query will run as efficiently as possible. Do not create any unneeded indexes.

```
SELECT name, count(*)
FROM users
JOIN tweets USING (id_users)
JOIN tweet_tags USING (id_tweets)
WHERE tag = :tag
GROUP BY name;
```

Solution: The correct answer is

```
CREATE INDEX ON users(id_users, name);
CREATE INDEX ON tweets(id_tweets);
CREATE INDEX ON tweets(id_users);
CREATE INDEX ON tag_tweets(tag, id_tweets);
```

Where the second index above can be (optionally) ommitted since it is created automatically by the UNIQUE constraint.

To derive these indexes, first consider the following simpler query that has the column list shortened and the WHERE clause removed:

```
SELECT count(*)
FROM users
JOIN tweets USING (id_users)
JOIN tweet_tags USING (id_tweets)
```

If we first join on `users,tweets`, then we need the following indexes:

```
CREATE INDEX ON users(id_users);
CREATE INDEX ON tweets(id_users);
CREATE INDEX ON tag_tweets(id_tweets);
```

and if we first join on `tweets,tweet_tags`, then we need

```
CREATE INDEX ON users(id_users);
CREATE INDEX ON tweets(id_tweets);
CREATE INDEX ON tag_tweets(id_tweets);
```

Removing duplicates we get that we need the following four indexes:

```
CREATE INDEX ON users(id_users);
CREATE INDEX ON tweets(id_tweets);
CREATE INDEX ON tweets(id_users);
CREATE INDEX ON tag_tweets(id_tweets);
```

To take advantage of the WHERE clause, we need to modify the index on `tag_tweets` to include the `tag` column in the first position. To perform an index only scan on

`users` and prevent us from accessing the heap table, we need to add the `name` column in the second position to the index on `users`. Unfortunately, there is no way to come out of the JOINs with the `name` column in sorted order, and so the GROUP BY clause will have to be implemented with either a hash join or a group aggregate + explicit sort, and there is no way to speed up that clause.

Common mistages:

1. (-6) Missing the `tag` column in the `tag_tweets` index.
2. (-2) Missing the `name` column in the `users` index. This mistake results in fewer points missed because the performance penalty from missing this column will be only a constant-factor; whereas the performance penalty from the mistake above will result in an asymptotic slowdown.

6. (16pts) The following query returns tweets where either the text or the description of the user match a full text search query.

```
SELECT id_tweets
FROM tweets, users
WHERE ( to_tsvector('english', text)
     || to_tsvector('english', description)
      )
      @@ to_tsquery('english', :query);
```

   a. This query cannot be sped up using an index. Why?

> **Solution:** The expression to the left of the `@@` operator mentions two tables. Indexes can only speed up expressions on single tables.
> (Both portions of this problem were also givin in 2022.)

b. Rewrite the query above into an equivalent query that can be sped up with an index. Also provide the index that would speed up the query and explain why the modified query can be sped up.

Solution: The key idea is to "factor out" the `||` operator so that we have multiple conditions, each mentioning only a single table:

```
SELECT id_tweets
FROM tweets, users
WHERE (to_tsvector('english', text)
    @@ to_tsquery('english', :query)
      )
   OR ( to_tsvector('english', description)
     @@ to_tsquery('english', :query)
      )
;
```

We can actually further simplify the query by replacing the cross join with a UNION ALL operator:

```
SELECT id_tweets
FROM tweets
WHERE (to_tsvector('english', text)
    @@ to_tsquery('english', :query)
      )
UNION ALL
SELECT id_tweets
FROM users
WHERE ( to_tsvector('english', description)
     @@ to_tsquery('english', :query)
      )
;
```

For either query, we can build gin indexes like so:

```
CREATE INDEX ON tweets USING GIN(
    to_tsvector('english', text)
    );
CREATE INDEX ON users USING GIN(
    to_tsvector('english', description)
    );
```

Note that it is not wrong to use a RUM index in this case, but there are no particular advantages to using it.