CSCI143 Final, Spring 2022

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:

---

1 True/False Questions

For each question below, circle either True or False. Each correct answer will result in +2 points, each incorrect answer will result in -2 points, and each blank answer in 0 points.

1. True **FALSE** A table that takes up 832KB on disk has 109 pages.
2. **TRUE** False An index only scan may have to access heap table pages to determine whether a tuple is visible. This is likely to happen on a table that has had many recent UPDATE/DELETE operations and no recent VACUUM operation.
3. True **FALSE** Every heap page must have at least one live tuple.
4. True **FALSE** If you run the TRUNCATE command to delete the contents of a table, you must run a subsequent VACUUM FULL command to free up the disk space for other processes to use.
5. **TRUE** False Postgres automatically compresses large TEXT values.
6. True **FALSE** Postgres documentation recommends disabling autovacuum if you encounter the transaction id wraparound problem.
7. **TRUE** False Decreasing the `fillfactor` for a table from the default value of 100 will make HOT tuple updates more likely.
8. True **FALSE** The autovacuum process runs the VACUUM FULL command at regular intervals in order to automatically free up disk space from dead tuples.
9. **TRUE** False A btree index created on an SMALLINT column will have higher fanout than the same index created on a BIGINT column.
10. True **FALSE** In the postgres documentation, TID is an abbreviation for transaction identifier.
11. True **FALSE** Dirty reads are possible in Postgres's read committed isolation level.
12. **TRUE** False For very small tables, the postgres query planner is likely to choose sequential scan instead of an index scan.
13. **TRUE** False The hash index supports the bitmap scan access method.
14. **TRUE** False A database stored using HDDs should have a higher value for the `random_page_cost` system parameter than a database stored using SSDs.
15. True **FALSE** A denormalized representation of data tends to take up less disk space than a normalized representation.

16. **TRUE** False The nested loop join strategy can be used to join tables on an equality constraint.

17. **TRUE** False The hash join strategy can be used for self joins.

18. **TRUE** False A hash index can be used to speed up a nested loop join.

19. True **FALSE** A btree index can be used to speed up a CHECK constraint.

20. True **FALSE** It is possible to INSERT a NULL value into a column labeled as the PRIMARY KEY.

21. True **FALSE** It is not possible to create a GIN index to enforce a UNIQUE constraint, but it is possible to create a RUM index to enforce a UNIQUE constraint.

22. **TRUE** False One advantage of the RUM index over the GIN index is that the former supports index scans and the latter does not. This implies that the RUM index can be used to speed up queries using the `LIMIT` clause, but the GIN index cannot.

23. True **FALSE** If postgres crashes while a DELETE/INSERT/UPDATE statement is modifying a RUM index, the index becomes corrupted and must be regenerated.

24. **TRUE** False The ANALYZE command collects statistics on the values in the table which the query planner uses when selecting which scan algorithm to use for a query.

25. True **FALSE** A hash index can return tuples in sorted order.

26. **TRUE** False The GroupAggregate algorithm can be used if one of the SELECT columns contains COUNT(DISTINCT *).

27. True **FALSE** When a table has been CLUSTERed on an index, inserting new tuples causes them to be inserted in the order specified by the index.

28. True **FALSE** In the current version of postgres (versrion 14), index only scans can be parallelized, but index scans cannot be parallelized.

29. True **FALSE** If you run EXPLAIN ANALYZE on an UPDATE statement, then the database is guaranteed not to modify your data.

30. True **FALSE** The order of columns matters in a multi-column GIN index.

2 Integrated Questions

The questions below relate to the following simplified normalized twitter schema. There are 12 subproblems in total, each worth 5 points.

```
CREATE TABLE users (
    id_users BIGINT PRIMARY KEY,
    created_at TIMESTAMPTZ CHECK (created_at > '2000-01-01'),
    name TEXT UNIQUE NOT NULL,
    description TEXT
);

CREATE TABLE tweets (
    id_tweets BIGINT PRIMARY KEY,
    id_users BIGINT REFERENCES users(id_users),
    retweet_count SMALLINT NOT NULL,
    country_code VARCHAR(2),
    lang VARCHAR(2) NOT NULL,
    text TEXT NOT NULL
);

CREATE TABLE tweet_mentions (
    id_tweets BIGINT REFERENCES tweets(id_tweets),
    id_users BIGINT REFERENCES users(id_users),
    PRIMARY KEY(id_tweets, id_users)
);
```

1. 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 tweets_mentions (id_tweets, id_users);
```

2. Create index(es) so that the following query will run as efficiently as possible. Do not create any unneeded indexes; if no new indexes are needed, say so and explain why.

```
SELECT count(*)
FROM users
WHERE name=lower(:name);
```

Solution: No new index is needed. The index on `users(name)` can be used. The fact that there is a function call to the right of the equals doesn't matter.

Common mistakes:

(-5pts) Many students suggested the following index:

```
CREATE INDEX ON users(lower(name));
```

This index cannot be used to speed up the query, however, because the expression `lower(name)` that appears in the index does not appear in the WHERE clause. A similar looking expression `lower(:name)` does appear, but `:name` is different than `name`.

3. Create index(es) so that the following query will run as efficiently as possible. Do not create any unneeded indexes; if no new indexes are needed, say so and explain why.

```
SELECT max(retweet_count)
FROM tweets
WHERE
    id_users = :id_users;
```

Solution:

```
CREATE INDEX ON tweets(id_users, retweet_count);
```

Common mistakes:

(-3) Not creating an index with the `retweet_count` column. Not including this column means we cannot do an index only scan.

(-4) Listing the columns in the wrong order. This prevents the index from being used.

(-5) Creating an index that includes `max(retweet_count)`. Indexes cannot be created on aggregate functions.

4. Create index(es) so that the following query will run as efficiently as possible. Do not create any unneeded indexes; if no new indexes are needed, say so and explain why.

```
SELECT DISTINCT id_users
FROM tweets
WHERE country_code = :country_code
   OR lang = :lang
LIMIT 10;
```

Solution:

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

Note: Due to the OR condition above, there is no way to create a single index on both `country_code` and `lang` that will result in an index/index only scan. Instead, we must create the two indexes above and do a bitmap scan. The bitmap scan cannot take advantage of additional columns to output results in a sorted order, and so an explicit sort must be called to implement the DISTINCT clause. Furthermore, the bitmap scan cannot return tuples one at a time, and so we cannot take advantage of the LIMIT clause.

Common Mistakes:

(-5pts) Creating only a single index.

(-2pts) Adding additional columns into the created indexes. Because we are using a bitmap scan, the additional columns cannot be used. Instead, they will take up additional space in the index and cause a variety of small performance problems. These include reducing the fanout of the btree and increasing the disk usage.

5. Create index(es) so that the following query will run as efficiently as possible. Do not create any unneeded indexes; if no new indexes are needed, say so and explain why.

```
SELECT id_users, id_tweets
FROM tweets
WHERE retweet_count > :minimum_retweet_count
  AND country_code = :country_code
ORDER BY retweet_count DESC, id_users ASC
LIMIT 10;
```

Solution:

```
CREATE INDEX ON tweets(
    country_code,
    retweet_count DESC,
    id_users ASC,
    id_tweets
    );
```

Common mistakes:

(-3) If you created multiple indexes that can use a bitmap scan to speed up the query. This won't be as fast as an index only scan, and in particular won't be to exploit the ORDER BY or LIMIT clauses.

(between -2 to -5) If you ordered your columns incorrectly. If the mistake is so grievous that the proposed index cannot be used at all, you got -5; smaller penalties were for orderings that could still be used somewhat but were not optimal.

6. Create index(es) so that the following query will run as efficiently as possible. Do not create any unneeded indexes; if no new indexes are needed, say so and explain why.

```
SELECT name
FROM users
JOIN tweet_mentions USING (id_users)
WHERE
    id_tweets = :id_tweets
ORDER BY name;
```

Solution: We need the following index

```
CREATE INDEX ON users (id_users, name);
```

Additionally, we would need to create the following index as well, but it is already created by the PRIMARY KEY constraint

```
CREATE INDEX ON tweet_mentions (id_tweets, id_users);
```

With these two indexes, we can do index only scans and a merge join, and the results will already be sorted according to the ORDER BY clause so no explicit sort is needed.

Common Mistakes:

(-1) Relying on the index on `users(id_users)` instead of creating `users(id_users, name)`. The former will not have results in sorted order by `name`, and so cannot take advantage of the ORDER BY clause.

(-3) Getting the column order wrong on the index `(tweet_mentions (id_tweets, id_users))`.

7. Create index(es) so that the following query will run as efficiently as possible. Do not create any unneeded indexes; if no new indexes are needed, say so and explain why.

```
SELECT country_code, count(*)
FROM tweets
JOIN users USING (id_users)
WHERE name = :name
GROUP BY country_code
ORDER BY count(*) DESC;
```

Solution:

```
CREATE INDEX ON users (name, id_users);
CREATE INDEX ON tweets (id_users, country_code);
```

Note: The indexes above allow a merge join, returning results in sorted order based on the `country_code`. Because the results are pre-sorted, the GroupAggregate algorithm can be used to implement the grouping and `count(*)`. There is no way to speed up the ORDER BY clause, however.

Common Mistakes:

(-2) Not creating multicolumn indexes. This can prevent the use of the merge join and GroupAggregate algorithms. If you created one index multicolumn and one index not multicolumn, you got -1 point.

8. Create index(es) so that the following query will run as efficiently as possible. Do not create any unneeded indexes; if no new indexes are needed, say so and explain why.

```
SELECT count(*)
FROM tweets
WHERE retweet_count > 0
  AND country_code = :country_code
  AND to_tsvector('english', text) @@ to_tsquery('english', :query)
LIMIT 10;
```

Solution: There are many possible solutions to this problem with near-identical performance. The following are two full credit solutions.

Example 1: The following RUM index allows an index scan and so can take advantage of the LIMIT clause:

```
CREATE INDEX ON tweets using RUM(
    to_tsvector('english', text),
    country_code
    )
    WHERE retweet_count > 0;
```

Example 2: The following pair of GIN indexes can work together in a bitmap scan. This example is likely to be slightly slower than the example above, however, as it cannot take advantage of the LIMIT clause.

```
CREATE INDEX ON tweets(
    country_code
    )
    WHERE retweet_count > 0;

CREATE INDEX ON tweets USING GIN(
    to_tsvector('english', text)
    )
    WHERE retweet_count > 0;
```

Common mistakes:

(-1) If you didn't take advantage of the fact that the `retweet_count > 0` clause is hardcoded and create a partial index. If you didn't take advantage of this column at all, you received an additional -2 penalty.

(-1) If you used the `ATTACH='...', TO='...'` syntax for the RUM index incorrectly. In order to get speed up from filtering on the `country_code` column, you cannot attach this value.

(-5) If you did not use a GIN/RUM index at all.

9. Consider the following SQL query.

```
SELECT count(*)
FROM users
WHERE name ILIKE '%Smith'
```

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

Solution: Only a btree index can be used to speed up ILIKE/LIKE clauses. The leading wildcard in the ILIKE expression means that when scanning the btree, we cannot prune any branches from the search and must check every value for a match.

The above answer is full credit, but it's also important for the second part of the problem that the ILIKE operator must do more comparisons than the LIKE operator because ILIKE is case-insensitive. By replacing ILIKE with an equivalent LIKE expression, we can reduce the number of comparisons when the wildcard comes at the end of the string.

Common Mistakes:

(-3) Only mentioning case sensitivity.

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.

Solution: The key idea is to use the `reverse` function to move the wildcard to the end of the string and the `lower` function to replace the ILIKE operator with a LIKE operator.

```
SELECT count(*)
FROM users
WHERE reverse(lower(name)) LIKE reverse('%smith');

CREATE INDEX ON users (reverse(lower(name)));
```

Common Mistakes:

(-1) Not changing ILIKE to LIKE and adding the call to `lower`, which is needed because `ILIKE` is case-insensitive.

(-3) Only fixing the case sensitivity, and not the position of the wildcard.

(between -5 to -2) If you created a query/index (often using FTS) that is not equivalent to the one above. The exact penalty is absed on how similar your query is to the original.

10. Consider the following SQL query, which returns tweets where either the text or the description of the user match a FTS 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 condition in the WHERE clause mentions columns from multiple tables in the same expression. To speed up a query, the expression in an index must exactly match the expression on one side of the operator in the condition, and you cannot build an index on an expression that mentions multiple tables.

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.

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)
      )
;
```

Then 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.