Questions on Indexes and Access Algorithms

1 True/False Questions

For each question below, circle either True or False. On your final exam, each correct answer will result in +1 point, each incorrect answer will result in -1 point, and each blank answer in 0 points. For this homework assignment, you can uncomment the following line in the tex file to view the answers:

```
\printanswers
```

and so these questions do not need to be submitted. You should still try to complete them, however, to check your understanding. Approximately 4/5 of these questions are answered in class, and the remaining 1/5 you'll have to refer to the postgres documentation / supplementary material for the answers.

BTree

1. True **FALSE** Increasing the fanout of a BTree will increase the depth of the tree.
2. **TRUE** False For all integers $B, n \geq 2$, it is true that $B \log_B n \geq \log_2 n$.
3. **TRUE** False Postgres uses B+ Trees in the BTree index.
4. True **FALSE** When analyzing the performance of an algorithm on BTrees, the most import metric to consider is the number of comparision operations.
5. **TRUE** False On a typical HDD, seeks are expensive operations.
6. True **FALSE** Balanced binary search trees like the AVL Tree or Red-Black Tree tend to perform better than BTrees for large datasets that cannot fit in memory, and must be stored on disk.
7. **TRUE** False You have created a BTree index on an INTEGER column. The fanout of the tree will typically be in the hundreds.
8. True **FALSE** All nodes in a BTree, as implemented in postgres, are guaranteed to have the same fanout.
9. True **FALSE** Binary trees typically have a lower height than BTrees.
10. **TRUE** False B+ Trees support faster range queries than BTrees.
11. **TRUE** False For HDDs with a very slow seek time but fast sequential read time, it makes sense to have higher fanout when using a BTree.
12. **TRUE** False NULL values are stored in postgresql indexes by default.

Scan Algorithms

13. True **FALSE** An index can be used to speed up every slow query.
14. True **FALSE** Creating an indexes on a table will make INSERT statements faster on that table.
15. True **FALSE** When it is possible to perform both an index scan and a bitmap index scan, the bitmap index scan is guaranteed to be faster.
16. **TRUE** False When it is possible to perform both an index only scan and a bitmap index scan, the index only scan is guaranteed to be faster.
17. **TRUE** False When it is possible to perform both an index only scan and an index scan, the index only scan is guaranteed to be faster.

18. True **FALSE** When it is possible to perform both an index scan and sequential scan, the index scan is guaranteed to be faster.

19. True **FALSE** When it is possible to perform both a bitmap index scan and sequential scan, the bitmap index scan is guaranteed to be faster.

20. **TRUE** False Bitmap scans are the only scan method that can take advantage of multiple indexes.

21. True **FALSE** When it is possible to perform both an index only scan and sequential scan, the index only scan is guaranteed to be faster.

22. True **FALSE** There exist situations where it is possible to perform an index only scan, but it is not possible to perform an index scan.

23. **TRUE** False There exist situations where it is possible to perform an index scan, but it is not possible to perform an index only scan.

24. True **FALSE** When performing an index only scan, postgres only needs to consult the index file, and never needs to consult the table file.

25. **TRUE** False Postgres uses a table's VM when performing an index only scan.

26. True **FALSE** Postgres indexes contain enough metainformation for each tuple in order to determine the tuple's visibility.

27. True **FALSE** In some situations, an index scan can access fewer table pages than a bitmap scan.

28. **TRUE** False For databases stored on HDDs, the query planner will choose to perform sequential scans instead of index scans relatively more often than when the database is stored on SSDs.

29. True **FALSE** When inserting large amounts of data into an empty table, it is faster to first create your indexes, then insert the data.

30. **TRUE** False When performing an index scan on a BTree Index, the number of comparison operations performed is always less than or equal to the number of pages accessed.

31. **TRUE** False Reducing the value of `random_page_cost` system parameter will increase the number of situations where the query planner will use an index only/index/bitmap scan over a seq scan.

Sorting / Grouping Algorithms

32. **TRUE** False A BTree index can be used to speed up SELECT statements with the ORDER BY clause.

33. **TRUE** False The query planner will typically prefer an index scan over a bitmap index scan on SELECT queries that use a LIMIT clause with a small value.

34. True **FALSE** If a SELECT statement requires an explicit SORT operation in the query plan, then adding a LIMIT clause to the SELECT statement is likely to significantly improve performance.

35. **TRUE** False A GROUP BY clause can always be implemented with either a GroupAggregate or a HashAggregate.

36. **TRUE** False A BTree index can be used to speed up a GroupAggregate operation.

37. True **FALSE** A BTree index can be used to speed up a HashAggregate operation.

38. True **FALSE** The HashAggregate requires less memory than the GroupAggregate.

39. **TRUE** False Increasing the `work_mem` parameter will cause the query planner to more likely prefer a HashAggregate operation.

40. **TRUE** False Increasing the `work_mem` parameter too high can cause the operating system to unexpectedly kill worker processes.

41. True **FALSE** If the `work_mem` parameter is lower than the amount of memory needed to complete a computation, the process will be killed by the OS.

42. True **FALSE** The HashAggregate algorithm can be used if one of the SELECT columns contains COUNT(DISTINCT *).

Join Strategies

43. True **FALSE** An index scan can be used to compute the join between two tables.

44. **TRUE** False It's always possible to use a nested loop join.

45. **TRUE** False It's always possible to use a hash join.

46. True **FALSE** It's always possible to use a merge join.

47. **TRUE** False The nested loop join can implement joins using a $<$ condition.

48. True **FALSE** The hash join can implement joins using a $<$ condition.

49. True **FALSE** The merge join can implement joins using a $<$ condition.

50. **TRUE** False The nested loop join can implement self joins.

51. **TRUE** False The hash join can implement self joins.

52. **TRUE** False The merge join can implement self joins.

53. **TRUE** False All three join algorithms (nested loop, hash, and merge) can implement joins using on an equality condition.

54. True **FALSE** When it's posible to do both a hash join and a merge join, the hash join will always be faster.

55. True **FALSE** When it's posible to do both a hash join and a merge join, the merge join will always be faster.

56. **TRUE** False A SQL full outer join can be computed using either the nested loop join, hash join, or merge join algorithms.

57. **TRUE** False The order that the query planner chooses to join tables together can have a significant impact on the runtime of the join.

58. **TRUE** False Increasing the size of `work_mem` can improve the performance of a hash join.

59. True **FALSE** Appropriately created indexes can speed up hash joins.

The `CLUSTER` command

60. **TRUE** False The CLUSTER command can greatly speed up bitmap scans by reducing the number of table pages accessed.

61. 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.

62. True **FALSE** You can insert into a table while the CLUSTER command is being run.

63. **TRUE** False You can insert into a table while the CREATE INDEX CONCURRENTLY command is being run.

64. True **FALSE** You can insert into a table while the CREATE INDEX command (without the CONCURRENTLY option) is being run.

65. True **FALSE** The `maintenance_work_mem` system parameter should be set to a low value in order to make CLUSTER run faster.

66. **TRUE** False It is always recommended to run the ANALYZE command after running the CLUSTER command.

The `ANALYZE` command

67. **TRUE** False Running the ANALYZE command on a table helps the query planner choose which scan algorithm to implement.

68. True **FALSE** The ANALYZE command should be run after every INSERT command for optimal performance.

69. **TRUE** False The ANALYZE command should be run after large bulk inserts for optimal performance.

70. True **FALSE** The ANALYZE command is never run automatically.

71. True **FALSE** If a database table hasn't changed, but we've created several new indexes on the table, we should run the ANALYZE command so that the query planner knows how to best use those indexes.

72. True **FALSE** If a database table has changed because a significant fraction of rows have been inserted, but we have not created any new indexes, then the ANALYZE command will not do anything.

73. **TRUE** False Running the ANALYZE command on the tables that are used in a sequence of JOINs can help the query planner choose which order to perform the joins in.

Constraints

74. True **FALSE** It is possible to have a UNIQUE constraint without an index.

75. **TRUE** False It is possible to have a CHECK constraint without an index.

76. **TRUE** False It is possible to have a NOT NULL constraint without an index.

77. True **FALSE** It is possible to have a FOREIGN KEY constraint without an index on the target table/column(s).

78. True **FALSE** It is possible to have a FOREIGN KEY constraint without an index on the source table/column(s).

Parallelism

The questions below all refer specifically to Postgres version 13.

79. True **FALSE** It is always more efficient to run a parallelized operation than an unparallelized one, when both methods are available.

80. **TRUE** False Sequential scans can be parallelized.

81. **TRUE** False Index only scans can be parallelized.

82. **TRUE** False Index scans can be parallelized.

83. **TRUE** False Bitmap scans can be parallelized.

84. **TRUE** False Both the inner and outer sides of a nested loop join can be parallelized.

85. True **FALSE** Both the inner and outer sides of a merge join can be parallelized.

86. True **FALSE** Both the inner and outer sides of a hash join can be parallelized.

87. True **FALSE** For all join strategies, the inner side can be parallelized.

88. **TRUE** False For all join strategies, the outer side can be parallelized.

2 Integrated Questions

Consider the following simplified normalized twitter schema.

```
CREATE TABLE users (
    id_users BIGINT PRIMARY KEY,
    created_at TIMESTAMPTZ,
    username TEXT
);

CREATE TABLE tweets (
        id_tweets BIGINT PRIMARY KEY,
        id_users BIGINT REFERENCES users(id_users),
        in_reply_to_user_id BIGINT REFERENCES users(id_users),
        created_at TIMESTAMPTZ,
        text TEXT
);

CREATE TABLE tweets_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 these constraints.

2. List all of the scan methods that could possibly be used for the following SQL query.

```
SELECT count(*) FROM tweets WHERE id_user=:id_user;
```

3. List all of the scan methods that could possibly be used for the following SQL query.

```
SELECT id_users,username FROM users WHERE id_user=:id_user;
```

4. Explain why the following SQL query is likely to be inefficient, and create an index that will speed up the query.

```
SELECT id_tweets
FROM tweets_mentions
WHERE id_users=:id_users;
```

5. Create index(es) so that the following query could use an index only scan.

   Do not create any unneeded indexes; if no new indexes are needed, say so.

```
SELECT count(*)
FROM tweets
WHERE id_user=:id_user
  AND created_at < :hi
  AND created_at >= :lo;
```

6. Create index(es) so that the following query can use an index only scan, avoid an explicit sort, and take advantage of the LIMIT clause for faster processing.

   Do not create any unneeded indexes; if no new indexes are needed, say so.

```
SELECT id_tweets, created_at
FROM tweets
WHERE id_users=:is_users
ORDER BY created_at DESC
LIMIT 10;
```

7. Construct index(es) so that the following query can use an index only scan, and the `users(username)` column will have a UNIQUE constraint.

   Do not create any unneeded indexes; if no new indexes are needed, say so.

```
SELECT created_at FROM users WHERE username=:username;
```

8. Construct a single index so that the following query can be answered as quickly as possible.

```
SELECT id_tweets
FROM tweets
WHERE id_user=:id_user
  AND created_at >= '2020-01-01 00:00:00'
  AND created_at <  '2021-01-01 00:00:00'
ORDER BY
  created_at ASC,
  id_reply_to_user_id DESC
```

9. Construct index(es) to speed up the following JOIN, assuming a merge join is used.

   Do not create any unneeded indexes; if no new indexes are needed, say so.

```
SELECT *
FROM tweets_mentions
JOIN tweets USING (id_tweets);
```

10. Construct index(es) to speed up the following JOIN, assuming a merge join is used. Your index(es) should take advantage of the WHERE clause.

    Do not create any unneeded indexes; if no new indexes are needed, say so.

```
SELECT id_tweets
FROM tweets_mentions
JOIN users USING (id_users)
WHERE username=:username;
```

11. Your goal is to answer the following query quickly.

```
SELECT count(*)
FROM tweets
WHERE id_user=:id_user
  AND created_at >= '2020-01-01 00:00:00';
```

You have the option of creating either of the following two indexes:

```
CREATE INDEX tweets_idx1 ON tweets(id_user)
        WHERE created_at >= '2020-01-01 00:00:00';
CREATE INDEX tweets_idx2 ON tweets(id_user,created_at);
```

If the tweets table is large (several TBs), which index will result in the fewest page reads when answering the SELECT query? Why?

Which index will use the least amount of disk space?

12. You are considering adding more information to the tweets table by redefining the schema as:

```
CREATE TABLE tweets (
    id_tweets BIGINT PRIMARY KEY,
    id_users BIGINT REFERENCES users(id_users),
    created_at TIMESTAMPTZ,
    in_reply_to_status_id BIGINT,
    in_reply_to_user_id BIGINT REFERENCES users(id_users),
    quoted_status_id BIGINT,
    retweet_count SMALLINT,
    favorite_count SMALLINT,
    quote_count SMALLINT,
    withheld_copyright BOOLEAN,
    withheld_in_countries VARCHAR(2)[],
    source TEXT,
    text TEXT,
    country_code VARCHAR(2),
    state_code VARCHAR(2),
    lang TEXT,
    place_name TEXT,
    geo geometry
);
```

How will this affect the number of pages accessed during (and therefore the runtime of) a

1. sequential scan?

2. index only scan?

3. index scan?

4. bitmap scan?

5. GroupAggregate?

6. HashAggregate?

7. Nested Loop join?

8. Hash join?

9. Merge join?