

Questions on Indexes and Table Storage

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.

Table Storage

1. **TRUE** False A table that takes up 160KB on disk has 20 pages.
2. True **FALSE** TID stands for *Transaction ID*, and every transaction is assigned a unique TID.
3. **TRUE** False OID stands for *Object ID*, and every table is assigned a unique OID.
4. True **FALSE** There is a hard limit of 100 tuples per page.
5. True **FALSE** Very large tuples can span multiple pages.
6. True **FALSE** If a tuple exists on disk, then there is guaranteed to be some transaction that can see the tuple.
7. True **FALSE** In order to remove dead tuples from a table, you must manually run the VACUUM command on that table.
8. True **FALSE** You should disable autovacuum to improve the performance of your database.
9. True **FALSE** When you use the INSERT command to insert multiple rows into a table at once, these rows are guaranteed to be inserted into the same page.
10. True **FALSE** A postgres database cluster can span dozens of computers.
11. True **FALSE** Postgres is using too much disk space, and you need to free up some space. You identify that there is a large 10TB table that contains about 90% dead tuples. Running the VACUUM command on this table will free up several terabytes of disk space.
12. **TRUE** False Postgres uses a table's FSM to quickly determine which page it should insert a tuple into.
13. **TRUE** False Postgres uses a table's VM to speed up VACUUMing.
14. True **FALSE** If you are inserting large text strings into postgres, it makes sense to compress them first in order to save space.

15. True **FALSE** The DELETE command deletes tuples directly from the page files.
16. True **FALSE** NoSQL databases like MongoDB and CassandraDB use ACID compliant transactions.
17. **TRUE** False Postgres transactions are ACID compliant, and therefore slower than non-ACID databases.
18. **TRUE** False The `synchronous_commit` system setting can be used to disable postgresql's ACID guarantees. This speeds up transactions, but may result in data loss if the server crashes.
19. True **FALSE** If the write ahead log (WAL) grows very large, it is safe to delete it in order to free up disk space.
20. True **FALSE** If the transaction log (clog/xact) grows very large, it is safe to delete it in order to free up disk space.
21. **TRUE** False Given the choice between (A) having tables on an SSD and indexes on an HDD, and (B) having tables on an HDD and indexes on an SDD, option (A) will generally be faster.
22. **TRUE** False If the database cluster is being stored on an SSD, then the `random_page_cost` system parameter should be reduced from its default value of 4.
23. **TRUE** False The default `fillfactor` for tables is 100, and for btree indexes is 90.
24. **TRUE** False Tables that have INSERTs but no UPDATEs should use a `fillfactor` of 100, but for tables with many updates, it may make sense to decrease the `fillfactor`.
25. True **FALSE** The order of columns in a multicolumn index has no effect on which queries the index can speed up.
26. **TRUE** False Multicolumn indexes will have a smaller fanout than a single column index created on any of the indexes columns.
27. True **FALSE** The advantage of using a partial index is that it saves disk space, but the disadvantage is that more pages are accessed during an index only, index, or bitmap scan.

BTree

28. True **FALSE** Increasing the fanout of a BTree will increase the depth of the tree.
29. **TRUE** False For all integers $B, n \geq 2$, it is true that $B \log_B n \geq \log_2 n$.
30. **TRUE** False Postgres uses B+ Trees in the BTree index.
31. True **FALSE** When analyzing the performance of an algorithm on BTrees, the most import metric to consider is the number of comparision operations.
32. **TRUE** False On a typical HDD, seeks are expensive operations.
33. 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.
34. **TRUE** False You have created a BTree index on an INTEGER column. The fanout of the tree will typically be in the hundreds.

- 35. True **FALSE** All nodes in a BTree, as implemented in postgres, are guaranteed to have the same fanout.
- 36. True **FALSE** Binary trees typically have a lower height than BTrees.
- 37. **TRUE** False B+ Trees support faster range queries than BTrees.
- 38. **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.
- 39. **TRUE** False NULL values are stored in postgresql indexes by default.

Scan Algorithms

- 40. True **FALSE** An index can be used to speed up every slow query.
- 41. True **FALSE** Creating an indexes on a table will make INSERT statements faster on that table.
- 42. 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.
- 43. **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.
- 44. **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.
- 45. True **FALSE** When it is possible to perform both an index scan and sequential scan, the index scan is guaranteed to be faster.
- 46. 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.
- 47. **TRUE** False Bitmap scans are the only scan method that can take advantage of multiple indexes.
- 48. 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.
- 49. 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.
- 50. **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.
- 51. True **FALSE** When performing an index only scan, postgres only needs to consult the index file, and never needs to consult the table file.
- 52. **TRUE** False Postgres uses a table's VM when performing an index only scan.
- 53. True **FALSE** Postgres indexes contain enough metainformation for each tuple in order to determine the tuple's visibility.
- 54. True **FALSE** In some situations, an index scan can access fewer table pages than a bitmap scan.
- 55. **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.
- 56. True **FALSE** When inserting large amounts of data into an empty table, it is faster to first create your indexes, then insert the data.

57. **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.
58. **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

59. **TRUE** False A BTree index can be used to speed up SELECT statements with the ORDER BY clause.
60. **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.
61. 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.
62. **TRUE** False A GROUP BY clause can always be implemented with either a GroupAggregate or a HashAggregate.
63. **TRUE** False A BTree index can be used to speed up a GroupAggregate operation.
64. True **FALSE** A BTree index can be used to speed up a HashAggregate operation.
65. True **FALSE** The HashAggregate requires less memory than the GroupAggregate.
66. **TRUE** False Increasing the `work_mem` parameter will cause the query planner to more likely prefer a HashAggregate operation.
67. **TRUE** False Increasing the `work_mem` parameter too high can cause the operating system to unexpectedly kill worker processes.
68. 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.
69. True **FALSE** The HashAggregate algorithm can be used if one of the SELECT columns contains COUNT(DISTINCT *).

Join Strategies

70. True **FALSE** An index scan can be used to compute the join between two tables.
71. **TRUE** False It's always possible to use a nested loop join.
72. **TRUE** False It's always possible to use a hash join.
73. True **FALSE** It's always possible to use a merge join.
74. **TRUE** False The nested loop join can implement joins using a < condition.
75. True **FALSE** The hash join can implement joins using a < condition.
76. True **FALSE** The merge join can implement joins using a < condition.
77. **TRUE** False The nested loop join can implement self joins.
78. **TRUE** False The hash join can implement self joins.

- 79. **TRUE** False The merge join can implement self joins.
- 80. **TRUE** False All three join algorithms (nested loop, hash, and merge) can implement joins using on an equality condition.
- 81. True **FALSE** When it's possible to do both a hash join and a merge join, the hash join will always be faster.
- 82. True **FALSE** When it's possible to do both a hash join and a merge join, the merge join will always be faster.
- 83. **TRUE** False A SQL full outer join can be computed using either the nested loop join, hash join, or merge join algorithms.
- 84. **TRUE** False The order that the query planner chooses to join tables together can have a significant impact on the runtime of the join.
- 85. **TRUE** False Increasing the size of `work_mem` can improve the performance of a hash join.
- 86. True **FALSE** Appropriately created indexes can speed up hash joins.

The CLUSTER command

- 87. **TRUE** False The CLUSTER command can greatly speed up bitmap scans by reducing the number of table pages accessed.
- 88. 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.
- 89. True **FALSE** You can insert into a table while the CLUSTER command is being run.
- 90. **TRUE** False You can insert into a table while the CREATE INDEX CONCURRENTLY command is being run.
- 91. True **FALSE** You can insert into a table while the CREATE INDEX command (without the CONCURRENTLY option) is being run.
- 92. True **FALSE** The `maintenance_work_mem` system parameter should be set to a low value in order to make CLUSTER run faster.
- 93. **TRUE** False It is always recommended to run the ANALYZE command after running the CLUSTER command.

The ANALYZE command

- 94. **TRUE** False Running the ANALYZE command on a table helps the query planner choose which scan algorithm to implement.
- 95. True **FALSE** The ANALYZE command should be run after every INSERT command for optimal performance.
- 96. **TRUE** False The ANALYZE command should be run after large bulk inserts for optimal performance.
- 97. True **FALSE** The ANALYZE command is never run automatically.
- 98. 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.

99. 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.
100. **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

101. True **FALSE** It is possible to have a UNIQUE constraint without an index.
102. **TRUE** False It is possible to have a CHECK constraint without an index.
103. **TRUE** False It is possible to have a NOT NULL constraint without an index.
104. True **FALSE** It is possible to have a FOREIGN KEY constraint without an index on the target table/column(s).
105. 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.

106. True **FALSE** It is always more efficient to run a parallelized operation than an unparallelized one, when both methods are available.
107. **TRUE** False Sequential scans can be parallelized.
108. **TRUE** False Index only scans can be parallelized.
109. **TRUE** False Index scans can be parallelized.
110. **TRUE** False Bitmap scans can be parallelized.
111. **TRUE** False Both the inner and outer sides of a nested loop join can be parallelized.
112. True **FALSE** Both the inner and outer sides of a merge join can be parallelized.
113. True **FALSE** Both the inner and outer sides of a hash join can be parallelized.
114. True **FALSE** For all join strategies, the inner side can be parallelized.
115. **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. List all the tables/columns that have indexes created on them.

Solution:

```
users (id_users)  
tweets (id_tweets)  
tweets_mentions (id_tweets, id_users)
```

2. List the scan methods applicable for the following SQL query.

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

Solution: seq scan

3. List the scan methods applicable for the following SQL query.

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

Solution: seq scan, index scan, bitmap scan

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;
```

Solution: Explanation: We have an index on the column `id_users`, but `id_users` is the second column in the index, it is therefore only used in ordering to break ties on the first column `id_tweets`. This implies that the btree will not be ordered by `id_users`, and so there is no efficient algorithm for finding entries in the index satisfying `id_users=:id_users`.

An index that fixes this problem and allows index only scans is

```
CREATE INDEX idx ON tweets_mentions(id_users,id_tweets);
```

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;
```

Solution:

```
CREATE INDEX idx ON tweets(id_user, created_at);
```

Note that the order of the columns above is important.

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;
```


Solution:

```
CREATE INDEX idx ON tweets(id_user, created_at, id_tweets);
```

Note that the order of the columns above is important.

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;
```

Solution:

```
CREATE UNIQUE INDEX idx ON users(username) INCLUDE (created_at);
```

Note that the `INCLUDE` statement is necessary to ensure the `UNIQUE` constraint applies to only the `username` column, but the query can still use an index only scan.

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
```

Solution:

```
CREATE INDEX idx ON tweets
( id_user
  , created_at ASC
  , id_reply_to_ser_id DESC
  , id_tweets
);
```

The `ASC/DESC` ordering conditions are not strictly necessary, and may or may not improve performance.

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);
```

Solution: No new indexes are required, as we already have indexes on the `id_tweets` column for both tables.

An index on `tweets(id_tweets)` will not significantly improve join performance, but you would get credit either for including or not including this index. In a real application, however, you would not want to create this index due to the increased overhead of INSERT/UPDATE/DELETE operations.

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;
```

Solution:

```
CREATE INDEX idx1 ON users (username,id_users);
CREATE INDEX idx2 ON tweets_mentions (id_users,id_tweets);
```

The ordering of the columns in both indexes is important. If you did not create `idx2` but mention it's created in a previous problem, that's fine.

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?

Solution: `tweets_idx1` results in the fewest page reads for two reasons. First, because it is a partial index, the number of tuples will be smaller, so the height of the tree will be smaller, so fewer pages need to be traversed to reach a leaf node where the TID information is stored. Second, it has only a single column, so more tuples can be stored per page, so the fanout is higher, so the height of the tree is smaller, so fewer pages are traversed to reach a leaf.

Which index will use the least amount of disk space?

Solution: `tweets_idx1` for the same reasons as above.

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?

Solution: Increase: More data per tuple implies fewer tuples per page implies more pages for the same number of tuples, and a sequential scan must visit all tuples.

2. index only scan?

Solution: Stay the same: Assuming the table has been recently vacuumed, then the index only scan will not touch the heap pages. Adding more columns to a tuple doesn't affect its representation in an index, so the number of index pages accessed will remain the same.

3. index scan?

Solution: Stay the same: In an index scan, the tuples are not returned in a sorted order, so we must access 1 heap page per tuple returned. The fact that the tuples are now more spread out on the heap doesn't affect this.

4. bitmap scan?

Solution: Increase: In a bitmap scan, we first scan the index to identify which heap pages need to be accessed, and access those heap pages only once. Since tuples are now spread out across more pages in the heap, it is less likely that tuples will share pages, and so we will have to access more pages.

You should also understand (but do not need to explain) why the number of pages accessed during a GroupAggregate/HashAggregate or Nested Loop/Hash/Merge join are not affected.