Question 1. (42 points) SQL queries and indexes. We have a database that stores information about a simplified twitter application. When a user creates an account, he or she is assigned a user id (uid) and is added to the User table. Users may create tweets. Each of these is assigned a unique tweet id (tid) and is stored in the Tweet table. A Tweet may be posted as many times as desired, and each posting is recorded in the Post table (and a tweet can be posted by anyone, not just the user who created it). Finally, a user can follow other users and that information is recorded in the Follow table. For example, if user A is following user B (i.e., A is a follower of B), then a row with id of user A as uid and the id of user B as followsuid is added to the Follow table.

User(uid, name)
Tweet(tid, content)
Post(uid, tid, time)
Follow(uid, followsuid)

- The underlined attribute(s) represent the primary key for each relation.
- Post.uid is a foreign key that references User.uid.
- Post.tid is a foreign key that references Tweet.tid.
- Follow.uid is a foreign key that references User.uid.
- Follow.followsuid is a foreign key that references User.uid.

The queries you write must be proper SQL that would be accepted by SQL Server and any other SQL implementation. You should not use incorrect SQL, even if sqlite might produce an apparently correct answer from the buggy SQL.

(a) (16 points) Write a SQL query that retrieves the names of users who have posted more than 5 tweets that are at least 100 characters long. (Hint: If s is a string attribute, length(s) will return its length.)

```
SELECT u.name
FROM User u, Tweet t, Post p
WHERE u.uid = p.uid AND t.tid = p.tid AND length(t.content) >= 100
GROUP BY u.uid, u.name
HAVING count(*) > 5;
```

(continued next page)

Question 1. (cont)

User(<u>uid</u>, name)
Tweet(<u>tid</u>, content)
Post(<u>uid</u>, <u>tid</u>, <u>time</u>)
Follow(<u>uid</u>, <u>followsuid</u>)

(b) (16 points) An active user is one who is following more than 50 users. Write a SQL query that retrieves the names of users who are being followed by one or more active users.

Here are two possible solutions:

```
SELECT u1.name
FROM (SELECT u.uid
        FROM User u, Follow f
        WHERE u.uid = f.uid
        GROUP BY u.uid
        HAVING count(*) > 50) as au, User u1, Follow f1,
WHERE u1.uid = f1.followsuid AND f1.uid = au.uid
GROUP BY u1.uid, u1.name;

SELECT u1.name
FROM User u1, Follow f1,
WHERE u1.uid = f1.followsuid
        AND f1.uid   =  (SELECT u.uid
                              FROM User u, Follow f
                              WHERE u.uid = f.uid
                              GROUP BY u.uid
                              HAVING count(*) > 50)
GROUP BY u1.uid, u1.name;
```

(c) (10 points) Suggest **two indexes** that would be most likely to speed up execution of the queries in your answers to the previous parts of this question. Give a brief justification for your answer.

This question doesn't have a unique "best" answer. Answers were evaluated on whether the proposals were reasonable and on the quality of the justification, in particular if it showed an understanding of how a particular index could be used to facilitate queries.

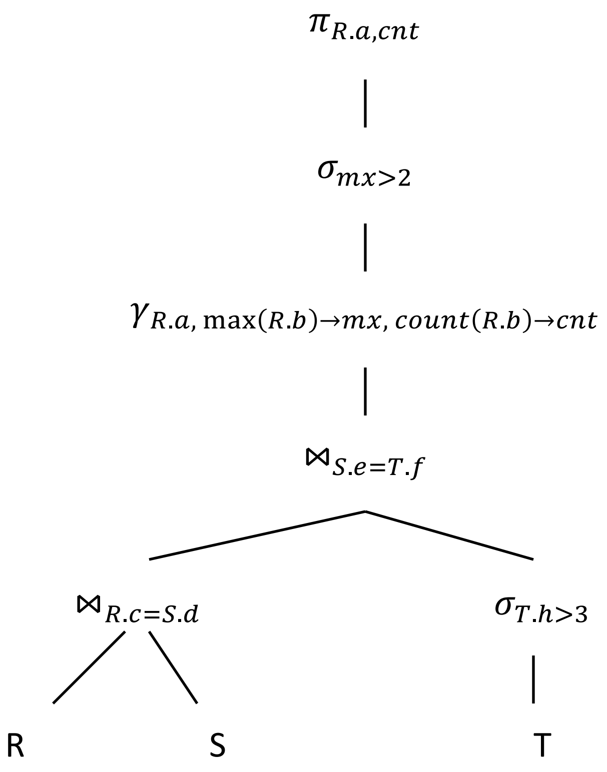
- **User.uid should almost certainly be indexed. Matches on User.uid are probably the most common ones in the queries.**
- **Answers to part (a) should benefit from an index on Post.tid since we need to find all tweets posted by each user.**
- **Answers to part (b) should benefit from an index on Follows.followsuid since we want to quickly find all users followed by particular users.**

Question 2. (40 points) Relational algebra and query plans. Consider the following schema for the relations R, S, and T to answer part (a), (b) and (c).

R(a, b, c)
S(d, e)
T(f, g, h)

(a) (16 points) Draw a tree giving a relational algebra query plan corresponding to the SQL query below. (Recall that the main relational algebra operators are ⋈, join; σ, select; Π, project; γ, grouping and aggregation; δ, duplicate elimination; and –, difference or subtract.)

```
SELECT R.a, count(R.b)
FROM R, S, T
WHERE R.c = S.d AND S.e = T.f AND T.h > 3
GROUP BY R.a
HAVING max(R.b) > 2;
```

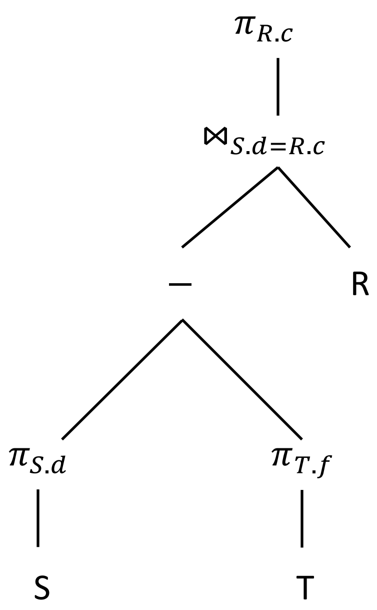


[There are many logical plans that will produce the correct result. This is just one of them and any other correct answer received full credit.]

Question 2 (cont.) R(a, b, c)
S(d, e)
T(f, g, h)

(b) (16 points) Draw a tree giving a relational algebra query plan corresponding to the SQL query below.

```
SELECT R.c
FROM R, S
WHERE R.c = S.d AND NOT EXISTS (SELECT *
                                FROM T
                                WHERE S.d = T.f);
```



[As with the other part of the question, there are other plans that also work and received credit. We awarded generous partial credit on this problem since it is tricky to get right.]

(c) (8 points) Suppose relations R, S, and T contain the following data:

a	b	c
A	3	1
B	4	2
C	1	3

d	e
1	1
1	4
2	3
3	1
4	2

f	g	h
4	5	4
3	1	5
1	2	1

What output is produced by the query in part (b) when it is executed using this data?

2

Question 3. (18 points) Query implementation. For each of the following, either circle the right answer (true/false, etc.) or give a short answer to the question, as appropriate. You do not need to provide an explanation for your answer unless the question asks for one.

[Brief explanations are given in the solutions for (a)-(d), but were not required.]

(a) (2 points) A relation can have clustered indexes on two or more attributes if this will speed up queries.

True / [✓ False]

[A relation can only be physically ordered on a single attribute or set of attributes. If there is more than one index, only one of them will be clustered.]

(b) (2 points) Which type of index is more suitable for speeding up queries that involve both exact matches (a=v) and range queries (a>v1 and a<v2)? (circle)

hash index / [✓ B+-tree index]

[A hash index supports fast access to individual values, but does not speed up access to nearby ones.]

(c) (2 points) Two measures of a relation R are the number of blocks B(R) and number of tuples T(R). For typical tables holding textual data like credit card transactions or the Twitter database from problem 1, what is the expected relationship between the sizes of B(R) and T(R)? (circle the correct choice)

- [✓ B(R) is normally significantly smaller than T(R)]
- B(R) and T(R) are normally about the same
- B(R) is normally significantly larger than T(R)

[Disk blocks are normally 4K bytes each or more, while tuples for these applications typically are a few dozen to a few hundred bytes each.]

(d) (2 points) Once a database engine translates a SQL query to a relational algebra logical query plan, that choice determines how the physical query will be executed.

True / [✓ False]

[There are usually several possible physical query plans for a given logical plan.]

(continued next page)

Question 3. (cont) (e) (5 points) When scanning a table to perform a join, the database can either do a full scan of the table or use an index to directly access tuples as needed. Will indexed access always be cheaper than a full scan? Answer yes or no and give a brief explanation supporting your answer.

No. Accesses to disk blocks are slow (milliseconds) and cost the same whether a single tuple or all of the tuples in the block are processed. If we use an index to access a table, the cost will be proportional to the number of tuples read. That can be significantly more expensive than a sequential scan of the full table if a large percentage of the tuples in the table are retrieved.

(f) (5 points) Suppose we execute a join operation $\mathbf{R} \bowtie_{R.c = S.d} \mathbf{S}$ using a hash join. Assuming that at least one of relations **R** or **S** will fit entirely in main memory, and that no suitable indexes are available to use, what is the estimated cost of the hash join? Give your answer in terms of quantities like B(R), T(R), etc., and give a brief explanation supporting your answer (i.e., how does a hash join access the data?).

Assuming that one relation will fit in memory, a hash join will read the smaller relation into a hash table in memory, then read the blocks of the other relation one at a time and use the hash table to find tuples in the hash table that join tuples in the block. The total cost will be B(R)+B(S), i.e., every block of each relation is read once.