COMP 115
Databases
Fall 2018

Exam 2

December 14, 2018
7:00–9:00 PM

1. B-Trees

If you load a set of records into a B-tree in random order, then the expected storage utilization is 70%.

1.1. What is the storage utilization if the records are inserted in increasing key order? (Hint: figure out the storage utilization at the leaf level and ignore the other levels. This is justified since the number of internal nodes is relatively tiny.)

1.2. Suppose you knew in advance that the next N inserts to a B-tree were all going to be at the end of the B-tree, and in increasing order. (E.g. the largest key present so far is 1199, and you insert keys 1271, 1309, 1414, 1506, 1672, 1788, …) How would you modify the insertion algorithm to obtain 80% storage utilization for the leaf pages containing the newly inserted records? (Don’t write out the entire algorithm, just describe the one detail that changes.)

2. Index Usage

We have a table:

```
create table T(
    id int primary key not null,
    x int,
    y int,
    z int,
    u int,
    v int)
```

2.1. Consider this query:

```
select *
from T
where x = 10
and   y between 20 and 22
```

Assume that each restriction is highly selective, e.g. `x = 10` identifies < 1% of the rows, and the same is true for `b between 20 and 22`.

There are a variety of indexing strategies that could be considered:

a) An index on `(x)` only.
b) An index on `(y)` only.
c) An index on `(x)` and another index on `(y)`.
d) An index on `(x, y)`.
e) An index on `(y, x)`.
f) No indexes involving `x` or `y`.

Which of these indexes are candidates for use this query? Describe the query plans and rank them from fastest to slowest.

2.2. Same as above, but for this slightly different query (replacing the `and` with an `or`):

```
select *
from T
where x = 10
or    y between 20 and 22
```

2.3. For each of the following sets of queries, identify a set of indexes that can be used to produce the fastest possible execution. (Assume each restriction is highly selective.) The goal is to use as few indexes as possible in each case, ideally one.

a)

- `select * from T where x = 1 and y = 2`
- `select * from T where y < 3`

b)

- `select * from T where y < 4`
- `select * from T where y = 5 and z < 6`
- `select * from T where x < 9 and y = 7 and z = 8`

c)

- `select * from T where x = 1 and y = 2`
- `select * from T where y = 3 and z = 4`
- `select * from T where x = 5 and z < 6`
- `select * from T where y = 7 or z = 8`

3. Query processing

The SQL `union` operator computes a set union of two sub-queries, eliminating duplicates, e.g.

```
select a, b, c from R
union
select x, y, z from S
```

The `union all` operator computes the same thing except that duplicates are preserved.

The most obvious implementation of `union all` copies the rows from `X` to output, then copies the rows of `Y` to output. The most obvious implemenation of `union` simply relies on the `union all` and `unique` operators, i.e. `union(X, Y) = unique(union_all(X, Y))`.

But there are faster ways to implement `union`.

3.1. Write a pseudo-code algorithm for `union(X, Y)`, where `X` and `Y` are input streams. Just give an algorithm that writes all output rows into a set – you don't have to write the algorithm as an iterator. Your goal is to come up with an algorithm faster than the naive one described above. Explain why your algorithm is faster.

3.2. For the algorithm you described in **3.1**, do you get better performance if the smaller input is on the left? On the right? Or does it not matter? Explain your answer.

4. Query Optimization

Consider these tables:

```
create table A(aid int not null primary key,
               ...);

create table B(bid int not null primary key,
               ...,
               aid int references A);

create table C(cid int not null primary key,
               ...
               bid int references B);

create table D(did int not null primary key,
               ...
               bid int references B);

create table E(eid int not null primary key,
               ...
               did int references D);
```

We have indexes on the primary keys:

```
create index on A(aid)
create index on B(bid)
create index on C(cid)
create index on D(did)
create index on E(eid)
```

And on foreign keys:

```
create index on B(aid)
create index on C(bid)
create index on D(bid)
create index on E(did)
```

4.1. For this query:

```
select E.*
from A
join B using (aid)
join C using (bid)
join D using (bid)
join E using (did)
where ...
```

Draw the diagram showing the enumeration of left-deep join plans, generated by dynamic programming.

4.2. Which of the following join plans would *not* be considered at any point during this enumeration of joins? For each join ruled out, explain why it would not be considered.

j1	`join(A, B)`
j2	`join(B, A)`
j3	`join(C, D)`
j4	`join(A, join(B, D))`
j5	`join(join(B, D), A)`
j6	`join(join(A, B), E)`
j7	`join(join(join(B, A), D), E)`
j8	`join(join(A, join(B, D)), E)`
j9	`join(join(A, B), join(D, E))`

4.3. These tables are similar to those from Assignment 8 (the assignment in which you used EXPLAIN find the indexing strategy minimizing estimated query cost):

```
create table T(tid int not null primary key,
               z int,
               ...)

create table S(sid int not null primary key,
              y int,
              ...
              tid int references T,
              uid int references U)

create table R(rid int not null primary key,
              x int,
              ...
              sid int references S)
```

`R` has 2,000,000 rows, `S` and `T` have 200,000 rows each.

```
explain select S.y, T.x
from R join S using (sid)
       join T using (tid)
where R.x = 1
```

If we have only indexes on the primary keys, we get this execution plan (row width information has been removed for readability):

```
Nested Loop  (cost=0.84..41756.31 rows=10)
   ->  Nested Loop  (cost=0.42..41751.47 rows=10)
         ->  Seq Scan on r  (cost=0.00..41667.00 rows=10)
               Filter: (r_10 = 1)
         ->  Index Scan using s_pkey on s  (cost=0.42..8.44 rows=1)
               Index Cond: (s_id = r.s_id)
   ->  Index Scan using t_pkey on t  (cost=0.42..0.47 rows=1)
         Index Cond: (t_id = s.t_id)
```

a) What single index should result in the biggest improvement in query performance?

b) Describe the change to the query plan from adding this index.

c) Describe indexing changes that could allow the optimizer to make use of at least one *covering index* optimization, (i.e., using an `Index Only Scan` instead of an `Index Scan`). The more opportunities you can find, the better. (*Hint:* An indexing change could involve either adding an index, or modifying an existing index.)

5. Strict Two-Phase Locking

We have three transactions that read (R), or write (W), three rows, a, b, and c.

```
T1: Ra, Rc
T2: Wa, Wb
T3: Wc, Rb
```

The scheduling of these steps, mediated by shared (S) and exclusive (X) locks, determines the serialization order of the transactions. What is the serialization order for each of the following schedules?

5.1.

T1	T2	T3
Sa		
Ra		
	Xa	
Sc		
		Xc
Rc		
COMMIT		
	Wa	
		Wc
	Xb	
		Sb
	Wb	
	COMMIT	
		Rb
		COMMIT

5.2.

T1	T2	T3
		Xc
		Wc
	Xa	
Sa		
		Sb
	Wa	
	Xb	
		Rb
		COMMIT
	Wb	
	COMMIT	
Ra		
Sc		
Rc		
COMMIT		

5.3.

T1	T2	T3
		Xc
		Wc
	Xa	
Sa		
	Wa	
	Xb	
		Sb
	Wb	
	COMMIT	
		Rb
Ra		
		COMMIT
Sc		
Rc		
COMMIT		

5.4. The following schedule has a bug in it. What went wrong? Is the final database state correct in spite of the bug? A yes/no answer is not sufficient, explain your reasoning.

T1	T2	T3
		Xc
		Wc
	Xa	
Sa		
Ra		
	Wa	
	Xb	
		Sb
	Wb	
	COMMIT	
		Rb
		COMMIT
Sc		
Rc		
COMMIT		

6. Multi-Version Concurrency Control

We have a Postgres database, which implements multi-version concurrency control. The database starts with one table containing one column, with 5 rows: `a, b, c, d, e`. The initial value of each row is `(1)`.

We have the following schedule, in which all transactions are executed with serializable isolation:

T1	T2	T3	T4	T5
	Ra			
	Rb			
	Wa=2			
Ra				
	Wb=2			
	COMMIT			
Rb				
				Ra
			Re	
			We=4	
			Rd	
		Ra		
		Wa=3		
			Wd=4	
			COMMIT	
		Rc		
Rc				
				Wa=5
		Wc=3		
		Rd		
Rd				
		Wd=3		
		COMMIT		
Re				
COMMIT				

Rx: Read row x

Wx=3: Update the value row of x in the database to 3.

6.1. After this schedule executes, and before the database is vacuumed, how many versions of `a` are there? List the versions in chronological order.

6.2. After the transactions complete, there is a vacuum, and then no more transactions begin. How many of versions of `a` exist in the database?

6.3. What values of `a, b, c, d, e` are read by `T1`?

6.4. What are the final values of `a, b, c, d, e`?

6.5. `T5` gets an error message when it tries `Wa=5`. Why?