COMP 115
Databases
Spring 2019

Exam 2

May 6, 2019
7:00–9:00 PM

1. Query Optimization – Theory (25 points)

Query optimization relies on algebraic equivalences. For example, if `R` and `S` are tables, and `p` is a predicate on `R`, then

```
select(join(R, S), p) = join(select(R, p), S)
```

is an equivalence that can be used to generate execution plans to be considered by the optimizer. The following questions are about equivalences involving SQL UNION. Consider this query:

```
select c1, ..., cn from R where p
union
select c1, ..., cn from R where q
```

1.1. (5 points) Rewrite the above query *without* the union.

1.2. (5 points) What is the algebraic expression for the original query?

1.3. (5 points) What is an equivalence that eliminates the union operator?

1.4. (5 points) What is the expression resulting from applying the equivalence of 1.3 to the expression of 1.2?

1.5. (5 points) If `R` has *no* indexes that can be used for predicate `p` or for predicate `q`, then would it be better to evaluate the query using the execution plan containing the `union` operator (your answer to 1.2)? Or should the rewritten plan (your answer to 1.4) be used instead? Explain your reasoning.

2. Query Optimization – Practice (15 points)

These tables are similar to those from Assignment 7 (in which you tried to get the best possible performance for a set of queries):

```
create table U(u_id int not null primary key,
               f int,
               u_filler varchar);

create table T(t_id int not null primary key,
               e int,
               t_filler varchar);

create table S(s_id int not null primary key,
               c int,
               d int,
               s_filler varchar,
               t_id int references T,
               u_id int references U);
```

Suppose we have the primary key indexes, an index on `U.f`, and no other indexes.

The execution plan for this query:

```
select S.c, S.d, T.e
from S
     join T using(t_id)
     join U using(u_id)
where U.f = 1
```

is this:

```
Gather  (cost=1024.39..3980.94 rows=2)
  Workers Planned: 1
  ->  Nested Loop  (cost=24.39..2980.74 rows=1)
    ->  Hash Join  (cost=23.97..2980.27 rows=1)
          Hash Cond: (s.u_id = u.u_id)
          ->  Parallel Seq Scan on s  (cost=0.00..2647.47 rows=117647)
          ->  Hash  (cost=23.91..23.91 rows=5)
            ->  Index Scan using u_f_idx on u  (cost=0.42..23.91 rows=5)
                  Index Cond: (f = 1)
    ->  Index Scan using t_pkey on t  (cost=0.42..0.47 rows=1)
          Index Cond: (t_id = s.t_id)
```

2.1. (3 points) Draw the left-deep join tree corresponding to this plan.

2.2. (3 points) Which of the joins is most expensive? Why is it expensive?

2.3. (3 points) What indexing changes can be made to make the slower join faster?

2.4. (3 points) How would the execution plan change once you add the index? (Explain the change in query processing, don't try and predict the costs.)

2.5. (3 points) How can you modify the indexes to make use of at least one covering index optimization, (i.e., so that the plan contains an Index Only Scan)?

3. Transactions (12 points)

In the initial state of a database, we have A = 10, B = 20, C = 30. Consider the following transactions:

T1	T2
begin	
read A	
read B	
read C	
	begin
	write A = 11
	delete C
	abort
	begin
	write A = 12
	delete C
	insert X = 40
	commit
read A	
read B	
read C	
read X	
commit	

The following questions ask about isolation levels. Assume that the database system implements the *minimum* correct behavior for each isolation level. So READ COMMITTED transactions do *not* guarantee repeatable reads, (unlike Postgres, which implements more than the minimum).

3.1. (4 points) If the transactions are run with SERIALIZABLE isolation, what values of A, B, C, and X are read at the end of T1?

3.2. (4 points) If the transactions are run with REPEATABLE READ isolation, what values of A, B, C, and X are read at the end of T1?

3.1. (4 points) If the transactions are run with READ COMMITTED isolation, what values of A, B, C, and X are read at the end of T1?

4. Transactions (10 points)

We have a database with four items, A, B, C, D, all with value 10. There are three transactions whose actions need to be scheduled:

```
T1:
    read(A)
    A += 1
    write(A)
    read(B)
    B *= 2
    write(B)
    read(C)
    C += 2
    write(C)
    read(D)
    D *= 3
    write(D)
T2:
    read(A)
    A *= 2
    write(A)
    read(B)
    B += 3
    write(B)
T3:
    read(C)
    C *= 3
    write(C)
    read(D)
    D += 4
    write(D)
```

4.1. (5 points) What are the final values for A, B, C, D if the transactions are executed with a serial execution order of T1 < T2 < T3?

4.2. (5 points) What are the final values for A, B, C, D if the transactions are executed with a serial execution order of T2 < T3 < T1?

5. Concurrency Control (18 points)

The table below shows a schedule of the actions of 6 transactions, being run in Postgres, which uses multi-version concurrency control. The transactions are run using the SERIALIZABLE isolation level. A, B, and C all have the initial value of 10. Use the schedule to answer the following questions:

5.1. (2 points) What value does T2 write for D?

5.2. (2 points) T3 blocks when it tries to write A. Why?

5.3. (2 points) When does T3 unblock?

5.4. (2 points) What happens to T3 once it is unblocked?

5.5. (2 points) When does the original version of A, (the version which has the value 10), become obsolete? I.e., when is it no longer possible for any transaction to see that version of A?

5.6. (2 points) When does the original version of B become obsolete?

5.7. (2 points) When does the original version of C become obsolete?

5.8. (2 points) What is the value of A written by T5?

5.9. (2 points) What is the value of A read by T6?

time	T1	T2	T3	T4	T5	T6
0	begin					
1	read A					
2	A = 11					
3				begin		
4		begin				
5				read A		
6	write A					
7			begin			
8		read A				
9			read A			
10			A = 14			
11			write A			
12	read B					
13				read B		
14						
15		read B				
16		read C				
17	B = 12					
18		D = A+B+C				
19	write B					
20		write D				
21		commit				
22	read C					
23	C = 13					
24	write C					
25	commit					
26					begin	
27					read A	
28					A = A + 5	
29					write A	
30						begin
31						read A
32						commit
33					commit	
34				commit		

6. Database Resilience (20 points)

We have a database with three items, A = 10, B = 20, C = 30. The database system uses undo logging, and we have the following sequence of events:

time	action	log record
0		`begin(T1)`
1	`T1: read(A, v1)`	
2	`T1: v1 += 5`	
3	`T1: write(A, v1)`	`update(T1, A, 10)`
4		`begin(T2)`
5	`T2: read(B, v2)`	
6	`T2: v2 += 5`	
7	`T2: write(B, v2)`	`update(T2, B, 20)`
8	`T2: flush()`	
9	`T2: output(B)`	
10		`commit(T2)`
11	`T1: read(B, v1)`	
12	`T1: v1 += 8`	
13	`T1: write(B, v1)`	`update(T1, B, 25)`
14		`begin(T3)`
15	`T3: read(C, v3)`	
16	`T3: v3 += 5`	
17	`T3: write(C, v3)`	`update(T3, c, 30)`
18		`abort(T3)`
19	`T1: read(C, v1)`	
20	`T1: v1 += 7`	
21	`T1: write(C, v1)`	`update(T1, C, 30)`
22	`T1: flush()`	
23	`T1: output(A)`	
24	`T1: output(B)`	
25	`T1: output(C)`	
26		`commit(T1)`

Recall that:

- **`read(X, v)`** calls `input(X)` if necessary, to ensure that X is present in a disk buffer, and then assigns X to variable v.
- **`write(X, v)`** writes v back to its disk buffer.
- **`output(X)`** writes the modified page containing X back to disk.
- **`flush()`** forces to disk any log records that haven't been written to disk so far.

The following questions ask about undo logging and recovery without checkpointing.

6.1. (4 points) If the database crashes immediately after time 26, what is the sequence of undo actions applied to restore the database?

6.2. (4 points) If the database crashes at time 25 (instead of `T1: output(C)`), what is the sequence of undo actions applied to restore the database?

6.3. (4 points) If the database crashes at time 18, (instead of `abort T3`), what is the sequence of undo actions applied to restore the database?

6.4. (4 points) What could go wrong if steps 22 and 23 were swapped, (e.g. due to a bug in the database system)?

6.5. (4 points) Suppose the database crashes *during recovery*, i.e., as we are applying the undo log to recover the state of the database. How would the logging and recovery process need to be modified to allow for this possibility?