COMP 421: Files & Databases

Fall 2025, Instructor Benjamin Berg, COMP421.001.FA25

Final Exam

Name:

PID:

ONYEN:

Question	Points Recvd / Points Possible
Q1: Mult. Choice	/ 10
Q2: Query Execution	/ 20
Q3: Concurrency Control	/ 25
Q4: MVCC	/ 20
Q5: Logging + Recovery	/ 25
TOTAL	/100

This is your final exam. There are many like it, but this one is yours. There are 5 questions, each with several parts/sub-questions. Detailed point values are given for each part in the exam. We do not intend this exam to take the full three-hour period, but you are welcome to use the full three hours if you like. Note the point values and don't get bogged down on any one question.

Before you begin, please read and sign:

In accordance with the UNC honor code, I certify that I will not give or receive help on this closed-book exam. No calculators, computers, or mobile devices of any kind are allowed. You may have unlimited blank paper for calculations or extended answers. Mark any extended answers clearly. Name: ____________________________

Part I. Multiple Choice

Each question is worth 2 points.

Q1. Consider a database table

    Order(o_id, customer_id, order_date, total_amount),

You need to optimize for the following workload

- **Lookup** a specific order by o_id
- **Range query:** Find orders placed in a specific date range
- **Ordered Lookup:** Find the 10 most recent orders for a specific customer

You can choose exactly **one** index for this table. Which choice offers the best overall performance?

A. Hash table index on customer_id
B. B+ Tree index on total_amount
C. B+ Tree index on (customer_id, order_date)
D. Hash table index on order_date

Q2. Consider a table `Price(item, value)`, where two transactions T1 and T2 execute the following schedule:

Time	Transaction T1	Transaction T2	Value of A
1	READ(A)		1.00
2		READ(A)	1.00
3		WRITE(A, 2.00)	2.00
4		COMMIT	2.00
5	READ(A)		2.00

T1 sees inconsistent values between steps 1 and 5. Which isolation guarantee is violated in this example?
A. Conflict serializability
B. View serializability
C. Snapshot isolation
D. All of the above

Q3. A DBMS uses a buffer pool and write-ahead logging (WAL) with:

- STEAL: Dirty pages from uncommitted transactions may be written to disk.
- NO-FORCE: At commit, not all updated pages are guaranteed to be on disk.

When the database crashes, which of the following operations needs to take place?
A. Rollback uncommitted transactions that were active at the time of the crash
B. Rollback committed transactions whose commit records were in the WAL but that still had dirty pages in the buffer pool at the time of the crash
C. Rollback any Compensation Log Records (CLRs) for transactions that were being undone at the time of the crash
D. Rollback all transactions that were committed before the most recent checkpoint

Q4. You are doing an external merge sort on a dataset that is 10 times larger than memory. What will be the effect of doubling the number of buffer pages used, *B*?

A. It increases the I/O cost by writing bigger chunks to disk
B. It completely removes the need for a merge phase
C. It creates larger initial sorted runs, reducing the number of runs to merge
D. It only helps CPU cache behavior and does not affect I/O

Q5. Consider the SQL query:

```
SELECT *
FROM Orders O
JOIN Customers C ON O.cust_id = C.id
WHERE C.region = ‘US’;
```

Which query plan is usually the most efficient?
A. Join O and C first, then filter C.region = ‘US’
B. Filter Orders on O.num_items first, then join with all Customers
C. Replace the join with a cross product and apply the join condition at the end
D. Filter Customers on region = ‘US’ first, then join with O

Part II. Query Execution and Optimization

Q1. Compare the **iterator** execution model (e.g., SQLite/PostgreSQL) and the **vectorized** execution model (e.g., DuckDB). Which model tends to be faster on large analytical scans, and why? (4 pts)

Q2. Your database executes the following query:

```
SELECT A.id, B.name
FROM A JOIN B ON A.id = B.id
WHERE B.score > 90;
```

You notice that the optimizer plans on using a block nested-loop join, but you think that a simple hash join would work better.

(i) If a B+ Tree index exists on B.id, why might you *not* prefer an index nested loop join to a hash join? (2 pts)

(ii) Describe what happens during the **Build Phase** of the hash join versus what happens during the **Probe Phase**? (3 pts)

(iii) In the iterator execution model, why is the build phase of a hash join considered a “pipeline breaker”? (3 pts)

Q3. Consider a database table
`Students(sid, name, age, dept),`
where we run a SQL query:

```
SELECT sid, name
FROM Students
WHERE dept = 'CS' AND age < 25;
```

Assume there is a B+-tree index on **`Students.dept,`** but no index on **`Students.age`**.
SQLite’s operators follow these notations:

- SeqScan_R for a full table scan over relation R
- IndexScan_R(X = v) for an index-based access using attribute X
- σ_{predicate} for selection
- π_{attributes} for projection

SQLite might choose the following logical query plan:

```
     π_{sid, name}
           |
σ_{dept = 'CS' ∧ age < 25}
           |
   SeqScan_Students
```

(i) Describe the steps in the above logical plan, and why this is inefficient in this case. (2 pts)

(ii) Using the operator notation defined above, draw a more efficient operator tree, and briefly justify why your plan is more efficient than the logical plan above. (3 pts)

(iii) If we instead had separate B+Tree indexes on both **`Students.dept,`** and **`Students.age`**, explain how a bitmap index scan in PostgreSQL can efficiently answer queries such as this that have multiple predicates. (3 pts)

Part III. Concurrency Control

Q1. Consider a database with three data items: A, B, and C. Two transactions, T1 and T2, execute the operations listed below using the **Strong Strict 2PL** concurrency control protocol.

- T1: Transfer data from A to B.
- T2: Read C and then update A.

Complete the remaining "Locks Held" columns. Indicate which locks are acquired (**S** for Shared, **X** for Exclusive) and exactly when they are released (**RLS** for release). If a transaction waits, mark the time when it starts and stops waiting and acquires its lock. The first few rows are completed as an example. (5 pts)

Time	Transaction 1	Transaction 2	Locks Held by T1	Locks Held by T2
1	READ(A)		**S(A)**	
2		READ(C)	**S(A)**	**S(C)**
3	WRITE(A)		**X(A)**	**S(C)**
4		WRITE(C)	**X(A)**	**X(C)**
5	READ(B)			
6		READ(A)		
7	WRITE(B)			
8	COMMIT			
9				
10		WRITE(A)		
11		COMMIT		

Q2. Consider a database with two objects, A and B, and initial values A = 0, B = 1. We run two transactions according to the following schedule:

Time	Transaction T1	Transaction T2
1	BEGIN	
2	READ(A)	
3	READ(B)	
4		BEGIN
5		READ(A)
6	WRITE(A:=B)	
7		READ(B)
8		WRITE(B:=A)
9	COMMIT	
10		COMMIT

(i) Assume we use MVCC to provide snapshot isolation. For both T1 and T2, state whether each transaction commits or aborts under the above schedule. (3 pts)

(ii) Does the above schedule obey conflict serializability? Sketch a dependency/precedence graph to prove your answer. Label the edges to denote the operations that cause each dependency. (4 pts)

(iii) What does this example show about conflict serializability versus snapshot isolation? (4 pts)

Q3. Let's consider how the design of a concurrency control mechanism depends on the workload and use case.

(i) Consider an OLAP system that generates business intelligence reports every night by running a read-heavy workload. Would you prefer OCC or 2PL in this case? Give a one-sentence justification for why. (3 pts)

(ii) Consider an OLTP system that is used for ticketing at Taylor Swift concerts. Would you prefer OCC or 2PL in this case? Give a one-sentence justification for why. (3 pts)

(iii) Zhongrui is writing the deadlock resolution function for his 2PL implementation. He decides to resolve deadlocks by giving priority to the transaction with the most other locks held. That is, the transaction holding fewer locks will be killed (if they are tied, we flip a coin). Describe a scenario where this would be a good deadlock resolution policy. (3 pts)

Part IV. MVCC

A banking database uses append-only MVCC (the style of MVCC used in PostgreSQL). We track the balances of accounts A and B, where initially **A=100** and **B=100**.

A developer from Zelle inadvertently splits one transfer M(A, B) into two transactions:

- T1a (debit A): read A; write A := A − 10; commit
- T1b (credit B): read B; write B := B + 10; commit

Other transactions are running at the same time as the above transfer:

- T2 (report): read A; read B; output A + B; commit
- T3 (snapshot): read-only query that reads A and B

Q1. Consider the following schedule of the above transactions:

Time	Transaction T1a	Transaction T1b	Transaction T2	Transaction T3
1	BEGIN			
2	READ(A)			
3	WRITE(A)			
4	COMMIT			
5				BEGIN
6				READ (A, B)
7		BEGIN		
8		READ(B)		
9		WRITE(B)		
10		COMMIT		
11			BEGIN	
12			READ(A)	
13			READ(B)	
14			OUTPUT A+B	
15			COMMIT	

(i) Under MVCC, what does T3 read for A and B? (3 pts)

(ii) What number does T2 output? (3 pts)

(iii) Assume the developer fixes their error and writes a single transaction T1 composed of the operations of both T1a and T1b. Assuming snapshot isolation, can T3 observe the same values of A and B once this error has been fixed? Briefly explain your answer. (3 pts)

(iv) Besides providing snapshot isolation, what is one other potential benefit of using MVCC in this situation? (3 pts)

Q2. Eventual consistency
A handful of modern large-scale databases purposefully allow some temporary inconsistency in the database. However, they guarantee that if that database runs for long enough without crashing, it will *eventually* return to a consistent state. This model is known as eventual consistency.

(i) What is a potential advantage of allowing the database to temporarily enter an inconsistent state? (2 pts)

(ii) What are some potential drawbacks of using an eventual consistency model? Where might these drawbacks be acceptable versus unacceptable? For example, would eventual consistency be ok for the database in Q1? Why or why not? (3 pts)

(iii) The idea of eventual consistency was initially very popular about 20 years ago, but it has lost favor over time. Why might developers have moved away from this model? (3 pts)

Part V. Logging and Crash Recovery

You are helping maintain a simple banking database system that stores account balances in a table:

`Account(id, balance)`

The system uses **Write-Ahead Logging** (WAL) on disk for crash recovery. Each log record is appended to a sequential log file before the corresponding change is written to the data pages on disk.

For this question, assume the WAL contains four basic types of log records:

- **`BEGIN(T)`** - marks the start of transaction T
- **`UPDATE(T, id, old_balance, new_balance)`** - T changes Account(id).balance from old_balance to new_balance
- **`COMMIT(T)`** - marks that transaction T has committed
- **`ABORT(T)`** - marks that transaction T was rolled back

Assume a steal / no-force buffer policy: dirty pages may be written to disk before commit, and committed pages may still be only in memory at commit time.

Q1. Let's start with the core concepts of logging.
(i) When using a WAL, when is it safe to tell the outside world that a transaction has committed? Be specific about what is happening with the log buffer, the log on disk, and any dirty pages. (5 pts)

(ii) System designers generally prefer a STEAL / NO-FORCE buffer pool design compared to a NO-STEAL / FORCE design. What are two advantages of this design? (4 pts)

Q2. Daniel is concerned that the log seems to grow without bound over time. To keep the log small, he suggests introducing some form of checkpoints. Consider the following simple checkpointing scheme: At a checkpoint, the system writes a `CHECKPOINT<L>` record to the log where L is the set of all transactions that are currently active (started, but not committed or rolled back) at the time of the checkpoint.

To keep things simple, before the system writes `CHECKPOINT<L>` to the log, all other transactions in the system are paused, and the buffer pool is flushed completely.

(i) When recovering from a crash under this scheme, let's say the end of the log (from the most recent checkpoint onward) contains log records for three transactions:

- T1: appears in the checkpoint, and later has a COMMIT(T1) record
- T2: appears in the checkpoint, but has no COMMIT(T2) record
- T3: does not appear in the checkpoint, but starts and commits after the checkpoint (you see BEGIN(T3) and COMMIT(T3) after the checkpoint)

During recovery, which of T1, T2, and T3 might need their actions to be redone, and which might need to be undone? Give a brief justification for each transaction. (4 pts)

(ii) Given that you will write out checkpoints periodically during normal operation, you also want to trim the log periodically to prevent it from taking up too much space on disk. When you trim the log, how do you decide where to trim? (2 pts)

Q3. (i) The simple checkpointing scheme in Q2 has several negative impacts on system performance. The ARIES recovery algorithm became popular for addressing many of these issues. Identify one specific performance issue with the simple scheme in Q2 that ARIES improves. (3 pts)

(ii) Explain the mechanism that ARIES uses to address the problem you have identified. A high-level description is fine, you do not need to walk through the recovery algorithm. However, you should give a real description of the mechanism, what it does, and how it differs from the simple recovery case above. Do not just state the name of the mechanism. If it helps, you can sketch an example or a picture. (5 pts)

(iii) Describe what performance metric you would expect to improve between the simple scheme and ARIES when using the mechanism you described. (2 pts)

</END OF EXAM>

</END OF COURSE>