COMP 421: Files & Databases

Midterm Exam

Name:

PID:

ONYEN:

Question	Points Recvd / Points Possible	Time Estimate
Q1: Mult. Choice	/ 10	10 minutes
Q2: Storage Manager	/ 25	13 minutes
Q3: LSM Trees	/ 25	12 minutes
Q4: B+ Trees	/ 27	23 minutes
Q5: Joins	/ 13	12 minutes

This is your midterm 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 if you are curious. We are providing time estimates for each question based on how long the TAs took when demoing the exam. Of course, you can allocate your time however you want, and different people work at different speeds. The questions are ordered by when the material was discussed in class.

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.

Name: _____

The Choice Is Yours (Revisited)

Let's begin with some multiple-choice questions about high-level DBMS design, SQL, and the relational algebra (each question is worth 2 pts).

1. In a disk-oriented DBMS, which of the following statements is FALSE?
Choices:
A. Random access on disk is much slower than sequential access.
B. Database page sizes, OS page sizes, and disk page sizes are not guaranteed to have the same size
C. The buffer pool manages pages fetched from disk into memory.
D. The OS knows not to double-cache pages held in memory by the buffer pool manager

Answer: D.

2. Consider a database with the following relations:
Student:

pid (BIGINT)	onyen (VARCHAR(10))	dept (VARCHAR(10))

Enroll

pid (BIGINT) (FOREIGN KEY REFERENCES (Student(pid)))	course_number (VARCHAR(10))

Consider the relational algebra expression:

$$\pi_{\text{name}}(\sigma_{\text{dept}='CS'}(Student) \bowtie Enroll)$$

This statement returns the result set requested by which of the following SQL queries (check all that apply):

A. SELECT dept FROM Student WHERE name='CS';
B. SELECT name FROM Student AS s INNER JOIN Enroll AS e ON s.pid=e.course_number WHERE s.dept='CS' ;
C. SELECT name FROM Student AS s NATURAL JOIN Enroll WHERE s.dept='CS';
D. SELECT name FROM Student AS s, Enroll AS e WHERE s.dept='CS' AND s.pid=e.pid;

Answer:
C, D

3. A hard disk spins at 15,000 RPM, with an average seek time of 2 ms, and a sequential read speed of 300 MB/s. We assume the read head waits for half a rotation on average before the desired sector arrives. Estimate the total time to read a 4 KB block from disk.
Choices:
A. 0.5 ms
B. 4 ms
C. 9 ms
D. 12 ms
Answer:
B. 4 ms

4. Which of the following statements about buffer pool management is true?
Choices:
A. A dirty page must be written back immediately when modified.
B. The DBMS can prefetch pages based on the query plan to minimize I/O stalls.
C. Most production systems rely on the mmap syscall to implement their buffer pool manager
D. The buffer pool manager handles requests for tuple data, but index data is always managed separately
Answer:
B. The DBMS can prefetch pages based on the query plan to minimize I/O stalls.

5. Which of the following statements accurately compares the n-ary storage model (NSM, row store) and the decomposition storage model (DSM, column store). Check all that apply.

Choices:
A. The NSM is better suited for write-heavy (insert, update, delete) workloads, while the DSM is better suited for read-only analytics workloads.
B. The NSM commonly uses an unsorted heap file, while the DSM stores data in sorted order
C. The NSM easily supports relational algebra / SQL queries, while the DSM is used in “no SQL” databases
D. The NSM tends to allow for better data compression than the DSM

Answer:
A. The NSM is better suited for write-heavy (insert, update, delete) workloads, while the DSM is better suited for read-only analytics workloads.

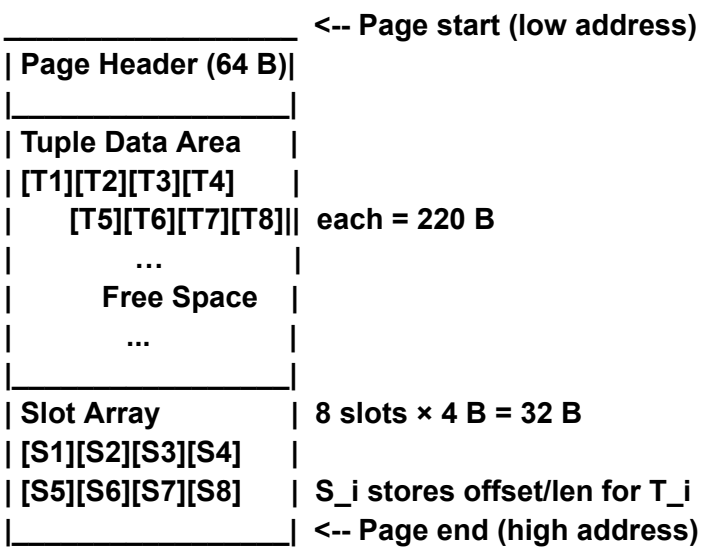
Cache Rules Everything Around Me

For this question, we will consider the heap file storage model discussed in the lecture. Assume we are in a disk-oriented database composed of DRAM and a hard disk drive (HDD). Assume the database uses 4 KB pages, which are accessed via a buffer pool manager with *B* buffers/frames.

(a) Slotted Page Arithmetic

A 4 KB page uses a slotted-page layout. Tuples are a fixed length of 220 B, including tuple headers, the NULL bitmap, and all tuple metadata.
Given a page with the following specifications:
(i) Page header: 64 B (ii) Slot in the slot array: 4 B (iii) 8 tuples

1. Draw a diagram of the page that clearly labels the three above components. You can label additional components if you need to provide context for items (i), (ii), (iii) (3 pts)
Answer:



2. How much free space remains in this page? How many additional tuples will fit in this page? (3 pts)

Answer:

Used = 64 + (8×220) + (8×4) = 1,856 B Free space = 4,096 − 1,856 = 2,240 B.

Additional tuples fit = 2,240 / (220+4) = 10 tuples

3. Assume that a user deletes the second tuple in the page (as indexed by position in the slot array). Explain one way to perform this delete operation. You can refer back to your drawing in (1) if it helps. There are a few choices of how to implement a delete operation from a slotted page, so describe one pro and one con for the method you chose. (3 pts)

Answer:

Multiple possible answers

Option 1- Invalidate Slot (simple delete)

Mark the slot as invalid (offset = -1) and mark tuple data as deleted.

Pro: O(1) delete, no data movement.

Con: leaves holes -> fragmentation until compaction

Option 2- Tombstone Record

Leave slot valid but replace tuple with a tombstone marker.

Pro: keeps stable RIDs for indexes/concurrency.

Con: adds space overhead; needs vacuum to reclaim.

Option 3- Compaction

Physically remove tuple and shift others to close gap; update slot offsets.

Pro: no wasted space; page stays compact.

Con: causes multiple writes and memory copies; O(n) cost; invalidates cached offsets.

(b) Buffer Pool Manager

The following blocks of code operate on page 0 and page 1, which you can assume have been allocated by the storage manager (e.g., in the disk_manager in bustub). Assume the buffer pool has a *single* frame/buffer (*B=1*) that begins empty. Consider the following code, which executes sequentially in a single thread:

```
// Transaction T1
{
   auto writeGuard = bpm->WritePage(0);
   std::string p0 = “page0”;
   //copy p0 intro page p
   CopyString(p.GetDataMut(), p0)
}
// END T1; writeGuard went out of scope and called unpin on page 0.
//
//============================
//============================
//============================
//
// Transaction T2
{
   auto readGuard = bpm->ReadPage(1);
   // doing something with this page’s data…
   // …
   //
}
// END T2; writeGuard went out of scope and called unpin on page 0.
```

Consider the state of the buffer pool once execution reaches the comment “`END T1;`”.

1. After T1 ends, what are the pin count and dirty flag values for page 0? (2 pts)

Answer: After T1 ends: page 0 pin count = 0; dirty flag = true.

2. If the system crashes between the comments “`END T1;`” and “`START T2;`”, is T1’s data (“page0” stored on page_id 0) preserved? (3 pts)

Answer: No

3. If yes, describe exactly when “page0” was written out to disk. If no, describe a solution (e.g., what code or mechanisms you would add to the storage manager) to ensure that this data is written to disk by the end of T1. (3 pts)

Answer: force a flush at end of T1 (ex: bpm->FlushPage(0) and ensure fsync), or implement a commit path that does WAL flush then forces dirty page writeback.

(c) Buffer Pool Optimization

1. Recent analytical DBMSs (e.g., DuckDB, or PostgreSQL with “direct I/O”) allow large sequential table scans to bypass the buffer pool and read directly from disk into user memory. Why can bypassing the buffer pool improve performance for workloads that include large scans, and what is one metric you might use to measure this improvement? (4 pts)

Answer: Skipping the buffer pool keeps one-time scan pages out of the cache, so they don’t evict hot pages we actually reuse. It also avoids extra work (like hash looksups, latches into the bp), and skips double buffering with the OS page cache. This is correct for read-only scans because no pages are dirtied. Since, nothing to WAL/flush, recovery rules aren’t affected.

2. Most buffer pools use LRU or some LRU approximation to evict pages. Another strategy for mitigating the impact of large scans is to change the eviction policy in some way. How would you change the eviction policy to better handle large scans, and why would you expect this to work? If it helps, you can sketch an example. (4 pts)

Answer:

- **MRU for scans: when you detect a forward sequential scan, evict the most recently used page next so scan pages don’t linger.**

- **Promote-on-second-hit / LRU-K: new pages start “cold” and only move to “hot” after a second access, so single-touch scan pages get dropped quickly.**
- **Per-query ring buffer: give the scan a small circular pool so it recycles its own pages instead of evicting hot data.**

Why it works: large scans have low reuse (each page is touched once). These policies keep hot pages in cache and prevent thrashing caused by LRU during scans.

A$AP RocksDB

(a) LSM Design

RocksDB is a popular open source storage system based on the Log-structured Merge Tree (LSM) we discussed in lecture. Let's talk about how this thing works and some optimizations that have been added over the years.

1. The default storage engine for MySQL is InnoDB, which famously uses B+Trees to track the location of each tuple. What are some use-cases/workloads for which you would expect RocksDB to outperform InnoDB, and why? Include which metric you would expect to be better for RocksDB. (4 pts)

Answer:
When there are write-heavy ingest scenarios, LSMs buffer writes in memory and flush sequentially to SSTs, so they avoid random in-place page updates of B+Trees. Metric: sustained write throughput (ingest MB/s or ops/s) and tail write latency under load.

2. A potential pitfall of the LSM design is high *Read Amplification*. Explain what read amplification is, how it can occur in an LSM, and one technique used to mitigate it. (4 pts)

Answer:
Extra reads beyond the minimum to answer a lookup (having to probe many SST files/levels for one key). Data is spread across memtable + immutable memtables + multiple SSTs across levels. With overlapping files (e.g., Level 0 or tiered compaction), a point lookup may need to check many files before confirming presence/absence.
How to mitigate: Bloom filters per SST, level compaction

(b) Compaction

Because compaction is an important aspect of the LSM design, there have been many compaction algorithms/variants designed to optimize the process.

In lecture, we discussed a simple compaction algorithm. In the literature, the simple algorithm we discussed is known as *tiered compaction*. Tiered compaction works by waiting for the size of a level to exceed some threshold, and then merging all SSTs on that level to form one SST on the next level.

1. When using tiered compaction, how many SSTs need to be searched in the worst case when performing a lookup operation? Select an option and explain your answer. (4 pts)

a) None
b) All
c) One per level
d) Two per level

Answer: b)
In tiered compaction, multiple SSTs within the same level can have overlapping key ranges. A lookup may need to check every SST at L0 and at other levels until it either finds the key or proves it's absent.

RocksDB famously uses *leveled compaction* as its default compaction strategy. Here, when data is flushed from memory, it is inserted at level 0. In level 0, the SSTs can contain overlapping ranges of keys. When the size of any level exceeds its threshold, we pick one file from that level and merge it with one or more files on the *next* level. Merging is performed such that in levels 1 or higher, SSTs *within* a level contain disjoint (non-overlapping) ranges of keys. No guarantee is made about overlapping ranges *between SSTs in different levels*.

2. RocksDB keeps metadata to track the smallest and largest keys for every SST in the system. In a system with leveled compaction, how would you use this metadata to reduce the number of reads needed during a lookup operation? (5 pts)

Answer:
For each SST, we maintain [min_key, max_key].
At L0, we use min/max to narrow to only those L0 files whose ranges contain k (and consult each file's Bloom filter to avoid data reads when possible).
At L1+, do a binary search (or a small index) over the file-range metadata to pick exactly one SST in that level whose range could contain k, then check that file.

4. A downside to leveled compaction is that it is generally shown to have higher write amplification than tiered compaction. Why do you think that is? To show why, it may help to draw a picture of what leveled compaction has to do during compaction to maintain non-overlapping SSTs within each level. (5 pts)

Answer:
To keep non-overlapping SSTs within a level, compacting one file from level i into i+1 often overlaps several files at i+1; the compaction must rewrite all overlapping data to produce a new, disjoint set of SSTs in i+1. Over a key's lifetime, it can be rewritten multiple times as it descends through levels (L0->L1->…->Lk), increasing write amplification.

Diagram: in leveled compaction, imagine sliding a "window" file from Li across multiple adjacent files in L_{i+1}; to preserve non-overlap, you must rewrite everything under that window each time.

(c) Alternative Storage Models

1. A drawback of RockDB is that, for the most part, it only compresses data at the block level for SST files. Name a storage model that you would expect to be far better at leveraging compression, and explain why/how this model enables compression. (3 pts)

Answer:
Columnar storage (Parquet/ORC). By laying out values column-wise, the data in each block is homogeneous (same type, similar distributions), enabling dictionary encoding, run-length encoding, with higher compression ratios.

B+Trees Are the Perfect Place for Shade

Assume a disk-oriented DBMS with a Buffer Pool Manager (BPM) and a thread-safe B+Tree that utilizes latch crabbing/coupling, as discussed in lecture. Assume that each node has sibling pointers, meaning this is technically a “B-Link Tree” variant of the original B+Tree.

If it helps, you can use the latching notation:
R(X) acquire read latch; **W(X)** acquire/upgrade write latch; **−X** release

(a) Latch Crabbing

Let’s describe latch crabbing at a high level. For each operation, describe the rules for latching as you traverse the tree from root to leaf. State the rule for when a node is considered “safe” for each operation, and which latches get released when a “safe” node is found. For this question, ignore any splits or merges that may happen once you reach the leaf nodes. (6 pts)

FIND(X)

Answer:
R(Root) -> pick child -> R(Child) -> −Root -> … repeat until leaf. Always acquire R on the next node before releasing the current one (“crabbing”). All nodes are “safe” (no structural change for a read). As soon as you R-latch the child, you may release the parent R latch. At each step: acquire R(next) then −current. Hold only the current node in R; never hold ancestors.

INSERT(X)

Answer:
Start with R(Root). At each step before moving down: acquire W on the child you will modify (since you may insert/split), then check if the child is safe. If safe, immediately release all ancestor latches (including any R/W on the parent) and continue down holding only W(child). If unsafe, you must keep a W latch on the parent while you descend (so you can install the split separator upward). Continue crabbing: R/W(parent) -> W(child), test safety, release or hold accordingly, until leaf.
A node is safe if it has at least one free entry (i.e., not full). In a safe child, an insert won’t split, so ancestors aren’t needed.
If the child is safe: −all ancestors, keep only W(child). If the child is unsafe: keep W(parent) while moving down; after handling split and posting the separator to the parent, release.

DELETE(X)

Answer:
Start R(Root). Before moving down, acquire W(child) and check if the child is safe. If safe, −ancestors and continue with only W(child). If unsafe, keep a W on the parent while descending so you can merge/redistribute and fix the separator.
A node is safe if it is strictly above the minimum occupancy (i.e., it can lose one entry without underflow). In a safe child, a deletion won't need merge/redistribute or parent updates.
If the child is safe: −all ancestors, keep only W(child).
If the child is unsafe: keep W(parent) until the leaf deletion and any needed merge/redistribute and separator fix are done; then release.

(b) B+Tree Design

Consider a B+Tree where each inner node can hold m node pointers. This B+Tree stores n key-value pairs that fit perfectly into N leaf nodes. Assume the inner nodes of the tree are completely full, meaning every inner node has exactly m pointers and m-1 keys.

1. Out of the total number of nodes in the B+Tree, what fraction are leaf nodes? You should write your answer in terms of m and N. (6 pts)
Hint: In your answer, you may want to use the geometric partial sum formula:

$$S_L = \sum_{i=0}^{L} m^i = \frac{m^{L+1}-1}{m-1}$$

Answer:

Level k has m^k nodes (root is level 0).

Let the leaves be at level L. Since there are N leaves, $m^L = N,\ L = \log_m N$.

Based on the geometric partial sum formula, Total nodes are

$$\frac{m^{L+1}-1}{m-1}$$

Then fraction in leaves are:

$$\frac{N}{total\ nodes} = \frac{N(m-1)}{Nm-1} \geq \frac{m-1}{m}.$$

2. Assume N is large. Does this fraction increase or decrease with m? Given that a real system might have 8 KB pages holding hundreds of pointers, what does that tell you about the fraction of leaf nodes in a real-world B+Tree? (3 pts)

Answer:
For large N, it increases with m and approaches 1.With 8 KB pages and hundreds of pointers per internal node (large m), almost all nodes are leaves; internal levels are very shallow and few.

Zhongrui and Ben are working on a database paper. They want to optimize the *average* index lookup time. Ben proposes using a B+Tree with m=32 pointers per inner node to index N leaf nodes. Zhongrui disagrees and makes the following argument: "B+Trees are fine in *worst-case*, but maybe we can improve the *average* lookup time by moving some of the data from leaf nodes to the inner nodes. That way, some FIND operations don't have to traverse the whole tree". Zhongrui proposes an alternative index, called a Z+Tree, that uses half of each *inner node* to store key-value data. To make space for the added data in each inner node, the Z+Tree stores only m=16 pointers in each inner node instead of 32. The Z+Tree still stores copies of all key-value pairs in the leaf nodes to allow for fast sequential scans, so there are still N leaf nodes. Assume that N is large for the following questions.

3. Using your formula above, how does the fraction of nodes that are leaf nodes compare between the B+Tree and the Z+Tree? (3 pts)

Answer:
With m = 16, at least 93.75% of nodes are leaves using the formula from (1).

4. How does the height of the B+Tree compare to the height of the Z+Tree? (3 pts)

Answer:
Z+Tree: log_16 N
B+Tree: log_32 N

5. Based on your answers to (3) and (4), do you think that the Z+Tree will improve the average time for a lookup operation? Will Zhongrui's paper get published? (6 pts)

Answer:
The root-to-leaf time ratio between Z+Tree and B+Tree is:

$$\frac{\log_{16} N}{\log_{32} N} = 1.25.$$

To lower-bound the average Z+Tree lookup time, we have that >90% of accesses take 25% longer. Hence, the average is at least 1.25 * 93.75% = 1.17 > 1 (assumed partial traversal equals to 0), which means the Z+Tree is slower on average.

A Spike Lee Join

For this question, we'll consider doing some joins on the IMDB dataset. We are interested in the following two (simplified) tables.

Directors

director_id (BIGINT)	name (VARCHAR (100))	born (DATE)

Films

film_id (BIGINT)	director_id (BIGINT, references Directors (director_id))	title (VARCHAR (200))	released (DATE)

There is a B+Tree index on the Directors table, on the director_id key.
There is a B+Tree index on the Films table, on the film_id key.

(a) Join algorithm selection

For each scenario below, state which of the following algorithms you would use, and why, in one or two sentences. It is not necessary to compute explicit run times for this question.

Possible Join Algorithms

- **Simple Hash Join**
- **Partition/GRACE Hash Join**
- **Sort-Merge Join**
- **Indexed Nested Loop Join**

Scenarios

1. You want to find the names of the 24 feature films directed by Spike Lee. Your SQL query execution plan is first to filter the rows of **Directors** to find Spike's **director_id**, and then join this result set against the **Films** table on the **director_id** column. (2 pts)

Answer:
Simple Hash Join: Build a tiny in-memory hash table on the single Spike row, then scan Films once and probe. No index on Films.director_id, so an indexed nested loop wouldn't help.

2. You want to find all films directed by people born since 1957. You again begin by filtering the **Directors** table and find that there are tens of thousands of these directors. You then join this result set against the **Films** table on the **director_id** column. Assume that each director has directed multiple films during their career. (2 pts)

Answer:
We will be lenient on this one and accept partitioned hash join or simple hash join. However, the math here is ridiculous! 100000 directors sounds like a big number, but each row is guaranteed to be less than 1KB (816 B), so 100k of these is still only 100 MB! This is easily fitting in DRAM unless you are running on the old Gateway Windows machine I grew up with.

Partition/GRACE Hash Join: The build side (thousands of directors) will not fit in memory comfortably, and Films has no index on director_id. GRACE partitions both sides to ensure each build partition fits in memory, keeping the access largely sequential and avoiding buffer thrash.

3. Assume that both **director_id** and **film_id** are assigned chronologically, so that more recently debuting films and directors have higher ID numbers. You want to return the films from the 1000 most recently-debuting directors in chronological order. (2 pts)

Answer:
Sort-Merge Join: Since director_id and film_id increase with time, sorting both inputs on director_id lets Sort-Merge Join produce a naturally ordered stream with minimal post-join sorting. This plays well with sequential I/O (reads in order) and avoids random access.

(b) I/O cost computations

You write a SQL query to generate the set of all films released since 1989, along with the names of their directors. Your query plan suggests a block nested loop join. After filtering the films table by date, you see the following statistics

Films (filtered by date):
1400 pages, 25000 tuples

Directors (unfiltered):
600 pages, 20000 tuples

1. Assuming you have 102 buffer pages, estimate the number of I/Os for this join. Explain which table you would use as the inner/outer table and why. (3 pts)

Answer:
600 + ceil(600 / (102 - 2)) * 1400 = 9000

Directors as outer and Films as inner.
Choose the smaller table as the outer to minimize the number of outer chunks.

2. You believe a hash join of some kind would work better here. Describe the hash join algorithm you would use here and why. Specify which table is used as the inner/outer table in your solution. Estimate the number of I/Os your solution requires. You can assume perfectly uniform (unskewed) hashing. (4 pts)

Answer:
Use grace hash join.
We build on Directors (the smaller side) and probe with Films.
Pass 1 (partition): read each relation once and write its partitions -> 2(B_F + B_D) I/Os.
Pass 2 (probe per partition): read each partition pair once -> (B_F + B_D) I/Os.
Total IO is 3 * (1400 + 600) = 6000