COMP 421: Files & Databases

Fall 2026, Instructor Benjamin Berg, COMP421.001.FA26

Midterm Exam SOLUTIONS

Name:

PID:

ONYEN:

Question	Points Recvd / Points Possible	Time Estimate
Q1: Mult. Choice	/ 20	10 minutes
Q2: Joins + Hashing	/ 20	15 minutes
Q3: Storage Models	/ 30	20 minutes
Q4: B+ Trees	/ 30	25 minutes

This is your midterm exam. There are many like it, but this one is yours. There are 4 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.

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: _____

Q1: Hard Disk Drive Like Jehu

Let's begin with some multiple-choice questions on a variety of topics (4 questions, each question worth *5 pts*).

1. Suppose a query asks to sequentially scan an entire table whose pages are stored *sequentially* on disk. The HDD spins at 12,000 RPM, has an average seek time of 8 ms, and a sequential read speed of 200 MB/s.

Assuming an average rotational delay of half a rotation and negligible controller and software overhead, what is the total time required to read the 5 GB table? **(Select one)**

A. 10.5 ms
B. 16.02 ms
C. 25.0105 seconds
D. 10.5 seconds

Answer: C. 25.0105 seconds

2. Consider the table created by the following SQL statement:

```
-- Create the managers table
CREATE TABLE managers ( mid INT NOT NULL, name VARCHAR(255) NOT NULL,
PRIMARY KEY (mid) );

-- Create the employees table referencing the managers table
CREATE TABLE employees ( eid INT NOT NULL, name VARCHAR(255) NOT NULL, mid
INT, PRIMARY KEY (eid), FOREIGN KEY (mid) REFERENCES managers(mid) );
```

Let E = ρ_E(employees) and M = ρ_M(managers).
Which relational algebra expression is equivalent to the following SQL query? **(Select one)**

```
SELECT e.name FROM employees e
JOIN managers m ON e.mid = m.mid
WHERE e.name = 'Alice';
```

A. $\pi_{E.name}(\sigma_{E.name='Alice'}(E \bowtie_{E.mid=M.mid} M))$

B. $\pi_{E.name}(E \bowtie_{E.mid=M.mid} M)$

C. $\pi_{M.name}(\sigma_{M.name='Alice'}(E \times M))$

D. $\pi_{E.name}(\sigma_{E.name='Alice'}(E \bowtie_{E.mid=M.eid} M))$

Answer: A. $\pi_{E.name}(\sigma_{E.name='Alice'}(E \bowtie_{E.mid=M.mid} M))$

3. A database administrator is evaluating whether to use tuple-oriented storage (NSM) or log-structured storage for an application with heavy write traffic, including frequent inserts and updates.

Which of the following statements best describes the trade-off between these storage architectures? **(Select one)**

A. Tuple-oriented storage is generally better for write-heavy workloads because updating existing pages avoids the overhead of maintaining multiple copies of the data.

B. Log-structured storage can improve write performance by batching writes into sequential disk operations, but may introduce additional read and compaction overhead.

C. Log-structured storage improves I/O performance by ensuring previously stored data is never re-written (i.e., zero write amplification).

D. Tuple-oriented storage generally requires more disk space because log-structured storage automatically removes outdated records whenever new updates arrive.

Answer: B. Log-structured storage can improve write performance by batching writes into sequential disk operations, but may introduce additional read and compaction overhead.

4. A Buffer Pool Manager (BPM) has three frame slots, all currently occupied:

- Page 1: Pinned (pin_count = 1), Clean
- Page 2: Unpinned (pin_count = 0), Dirty
- Page 3: Unpinned (pin_count = 0), Clean

The BPM receives a request to load Page 4, which is not currently in memory.

Which of the following statements correctly describes how the BPM handles page replacement? **(Select one)**

A. Page 1 can be evicted if it is the least recently used page in the buffer pool.

B. Page 2 can be evicted once its modified contents are written back (flushed) to disk.

C. Page 2 cannot be evicted until the application that wrote the page explicitly calls `flush()`.

D. Page 3 is the only evictable page in this scenario.

Answer: B. Page 2 can be evicted, but its modified contents must first be written back (flushed) to disk before the frame is reused.

Q2: Mudhoney

You have decided to create ChopGPT, a model trained to chat with users to create new recipes. To train this model, you scrape the internet to build the ultimate recipe database to use for model training and information retrieval. You are staggered to learn that Google estimates there are *hundreds of millions* of recipes on the internet. To track them, you create the following tables in a Postgres database:

```
CREATE TABLE Ingredients ( ingredient_id BIGINT PRIMARY KEY, name
VARCHAR(100) NOT NULL );

CREATE TABLE Recipes ( recipe_id BIGINT PRIMARY KEY, name VARCHAR(100) NOT
NULL, portion_size SMALLINT );

CREATE TABLE Recipe_Ingredients ( recipe_id BIGINT, ingredient_id BIGINT,
ingredient_ounces SMALLINT, PRIMARY KEY (recipe_id, ingredient_id),
FOREIGN KEY (recipe_id) REFERENCES Recipes (recipe_id), FOREIGN KEY
(ingredient_id) REFERENCES Ingredients (ingredient_id) );
```

There are B+Tree indices on all three tables. Note the composite key used on the Recipe_Ingredients table:

```
CREATE UNIQUE INDEX idx_recipes_recipe_id ON Recipes USING BTREE
(recipe_id);

CREATE UNIQUE INDEX idx_ingredients_ingredient_id ON Ingredients USING
BTREE (ingredient_id);

CREATE UNIQUE INDEX idx_recipe_ingredients_recipe_id ON Recipe_Ingredients
USING BTREE (recipe_id, ingredient_id);
```

The buffer pool manager manages 1 GB of memory.

Because you will only make updates to the database offline, you cluster the Recipes and Recipe_Ingredients tables using their respective indexes:

```
CLUSTER Recipes USING idx_recipes_recipe_id;

CLUSTER Recipe_Ingredients USING idx_recipe_ingredients_recipe_id;
```

Joining Ingredients

For each scenario below, describe the join algorithm you would use to execute the given join. Explain why you chose that algorithm in 1-2 sentences. If arguing that one algorithm is faster, it is sufficient to give an asymptotic estimate of I/O operations. It is *not necessary* to explicitly compute a number of I/O operations.

1a. Each recipe's **recipe_id** is assigned chronologically, so that more recently added recipes have a higher id number. For the first 20,000,000 recipes added to the website, list the names of each recipe together with the IDs of each ingredient used in the recipe. Return the results in the order the recipes were added. (5 pts)

Answer:
Sort-Merge: Recipes and Recipe_Ingredients are already sorted by recipe_id due to storage order. Recipes table is too large for simple hash. GRACE doesn't preserve order, requiring additional work to sort the output. Index nested Loop join will work with tweaks for range scanning.

1b. The site wants a report of every ingredient **name** together with the **recipe_id** of all recipes that use it. The Ingredients table is about 20 GB, Recipe_Ingredients is about 100 GB, and Recipes is also about 20 GB. The order of the output does not matter. (5 pts)

Answer: GRACE/partitioned hash join. None of the tables fit in memory, so no simple hash join will work regardless of table order. Sort-merge will be slow because Recipe_Ingredients is sorted by recipe_id, not ingredient_id. Wrong index on Recipe_Ingredients for index nested loop join. Index nested loop join will suffer lots of random I/O.

Hash Browns

Daniel wants to develop a specialized “hash brown” index for fast lookups in the Recipe_Ingredients table. This index will consist of a hash table (which will be larger than the buffer pool) using **recipe_id** as the *key* and **ingredient_id** as the *value*.

We want to make it faster to, given a single **recipe_id** (denoted ‘XX’), pull the list of ingredients for that recipe:

```
SELECT Ingredients.name FROM Recipe_Ingredients JOIN Ingredients ON
Recipe_Ingredients.ingredient_id = Ingredients.ingredient_id WHERE
Recipe_Ingredients.recipe_id = 'XX';
```

To do this, we’ll perform an index nested loop join using the hash brown index.

2a. You are debating between a linear probe hash table and a chaining hash table. In both cases you plan to handle duplicates by storing redundant keys (that is, just inserting multiple records with the same key). Tests show that the given query runs significantly faster given the linear probe hash table. Why do you think that is? (4 pts)

Answer/s:
Using recipe_id hashes the ingredients to the same starting slot, making accesses to duplicate keys sequential. Chaining here will have worse spatial locality, leading to more random reads.

The linear probe hash table suffers from a poor 95th percentile query runtime, indicating that a small set of recipes have slow lookup times. This is because, under linear probing, long chains of filled slots can build up due to a combination of hash collisions and random chance.

2b. Ben proposes a variant of the linear probe hash table. Each lookup/insert/delete uses two hash functions: h_1 and h_2. To lookup/insert/delete **x**, we first go to the location pointed to by $h_1(\mathbf{x})$. If it is full, instead of scanning linearly to find a free spot, we take $h_2(\mathbf{x})$ and next check the location pointed to by $h_1(\mathbf{x}) + h_2(\mathbf{x})$. If that is full, we check $h_1(\mathbf{x}) + 2\ h_2(\mathbf{x})$, and so on, so that the nth location checked is $h_1(\mathbf{x}) + n\ h_2(\mathbf{x})$. As soon as a free spot is found, **x** is inserted. Why might this improve the slow lookup times we saw under linear probing? What is one drawback of this approach? (6 pts)

Answer/s:

Long runs caused by collisions are the culprit under linear probing. Runs must be explored exhaustively when storing duplicates in a linear probe hash table until a free slot is found.

The proposal is known as double-hashing. It breaks long chains that would accumulate under linear probing. This comes at the cost of slightly worse locality, since duplicates cannot be stored sequentially. So, it probably improves tail performance at the cost of some average-case performance. Better for in-memory hash tables.

Q3: Parquet Courts

A DBMS stores a large Customer table containing millions of records, each with 10 attributes, including customer_id, name, address, balance, and region.

The application mainly submits two types of queries to the DBMS:

Transactional Queries — Customer Lookup, for example

```
SELECT * FROM Customer WHERE customer_id = 'XX';
```

Analytical Queries — Account Analysis, for example

```
SELECT MEDIAN(balance) FROM Customer WHERE region = 'YY';
```

The developer observes that the transactional query performs better with a **row-oriented storage model (NSM)**, while the analytical query performs significantly better with a **column-oriented storage model (DSM)**.

1. Explain why these performance differences occur. In your answer, discuss at least one specific characteristic or optimization used in each case to improve performance for the corresponding workload (10 pts).

Answer:

NSM (Row-oriented): Stores all attributes of a tuple together. The OLTP query benefits because loading the page containing the customer provides access to all 10 attributes. This reduces the number of pages needed to reconstruct the complete record, making NSM suitable for point lookups.

DSM (Column-oriented): Stores values from the same attribute together. The OLAP query only needs region and balance, so the DBMS can read pages containing these two columns without loading the other eight attributes. This reduces unnecessary disk I/O and improves data locality, making DSM efficient for analytical scans.

We need to support both the transactional queries and the analytical queries. Zhenyu proposes the following solution: keep customer records in Postgres for the OLTP queries. Nightly, dump contents of the Postgres database into Apache Parquet files that use a **PAX** storage format. These files divide rows into 1 GB row groups and store data within a row group in a columnar format. The nightly Parquet file can be read by DuckDB (a column store) to run our analytical queries.

2a. Customer **region** is tracked as one of the 3,244 counties in the US (stored as a VARCHAR(100)). If we want to compress the values in this column, what compression algorithm might be a good choice and why? (4 pts)

Answer:

Dictionary compression, maybe RLE but probably not great, German strings could work

2b. Zhenyu notices that the data in the Parquet files are not sorted by default. He suggests doing some sorting of the data while it is being dumped from Postgres to the Parquet file. How could sorting improve your answer to (2a)? (4 pts)

Answer:

Sort by region, RLE the column.

Ben argues that sorting the data every night takes too long, but he is worried that we will scan lots of useless data when running the example analytical query above. Then he sees an interesting feature for Parquet files. We can choose, on a per-column basis, to store Bloom filters in the file footer that describe which **region** values are in each row group.

3a. How would you design DuckDB to use these Bloom filters to accelerate the example analytical query above? (6 pts)

Answer:

Load one row group at a time, before loading, use the Bloom filter to "push down" the predicate by asking if the row group contains data from the target region. If yes, read the row group. If no, skip it.

3b. Bloom filters can have false positives. Will your design return the exact median, or just an approximation? Explain why. (4 pts)

Answer:

Exact median since no false negatives occur. False positives cause unnecessary row group reads, no bias in the computed aggregate.

3c. Parquet allows you to set a desired false positive rate for your Bloom filters. What changes

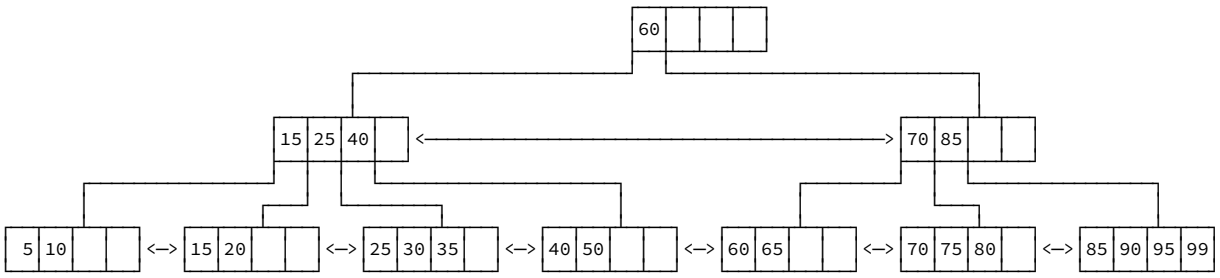
with the Bloom filter structure as you adjust this target false positive rate? (2 pts)

Answer:

More bits and probes to keep collision/false positive probability low. More bits generally leads to fewer false positives.

Q4: The Screaming B+Trees

This B+Tree has at most 4 keys per node. Every node except the root must contain a minimum of 2 keys.



1a. Above is an example of a B+Tree with sibling pointers in every level. Explain how a DELETE(50) operation would run, including what latches it takes. Are any splits/merges/sibling steals required? (6 pts)

Answer:

Write latch the root, write latch child, release parent if child is "safe" (contains more than 2 keys). Recurse down to the leaf. Write latch the leaf, delete the key, now the node is underfull. Write latch the left sibling, steal key 35, change the parent split pointer, release all latches.

1b. A user wants to retrieve all record ids for keys < 75. You do this using the B+ Tree iterator you implemented on Project 2. You also benchmark this operation on a table that uses Postgres's HASH INDEX. The table with the hash table index is much slower for this scan. Why do you think that is? (4 pts)

Answer:

Hash index is unsorted, so range scan requires a full table scan. B+ Tree just traverses all the way left and rips across the leaf nodes using sibling pointers.

Now, you want to run leaf node scans concurrently with INSERT/FIND/DELETE operations. The iterator you are writing in Project 2 is not thread safe, so we propose the following design: As the iterator scans the leaf nodes, it first takes a read latch on the first leaf node in the scan. Then, while scanning, it takes a read latch on the **next** leaf node *before* releasing its latch on the **current** node.

2a. We run the delete operation and scan operation from (1), using this thread-safe scan algorithm. The code hangs (gets stuck and doesn't return). What may have happened? Be specific, indicating the states of different latches on the tree. (6 pts)

Answer:

Deadlocked. T1 (delete) acquires the write latch on 50's leaf node. T2 (scan) acquired read latch on the left sibling. T1 tries to write latch the sibling and blocks. T2 tries to read latch 50's node and blocks.

2b. Propose one idea to fix this issue. Describe how it fixes the issue, and also one potential drawback to your solution. (4 pts)

Answer:

Many possible options. Timeouts could work. Make the scanning algorithm non-blocking (restart on blocking). Using a lock manager to detect deadlock is probably too heavyweight for this depth of the system, but we'll accept any version of this.

Timeouts/retries do not guarantee forward progress. Also, a delete that needed to acquire many latches can get sent back to the beginning if we are not careful.

This B+ Tree indexes a table by storing keys associated with record ids in the leaf nodes. You run your scan iterator and, as you find each record id that matches **key < 75**, you read that tuple by sending a read request to the buffer pool manager.

3. To compare your index-based scan to some baseline, you benchmark a simple scan that iterates over every page in the table in order, returning tuples with **key < 75**. Even though only 10% of the keys match **key < 75**, the full scan ends up being more than *10 times faster* than using the index. What went wrong? (10 pts)

Answer:

Let's say the table is X bytes of data stored sequentially on disk. Sequential read speed is Y bytes/second, so the full scan takes X/Y seconds. Index scan only reads about .1X bytes in the best case scenario (in reality, it will be more). Reads are random, so read speed is roughly Y/100 bytes/second. The index scan then takes about .1X/(Y/100) = 10X/Y seconds.

The main point is: the naive index scan did random I/O which is two orders of magnitude slower than the sequential scan per byte.

</END OF EXAM>