Final

CSE 410— Spring 2024

Name:
UBIT:

Academic Integrity

My signature on this cover sheet indicates that I agree to abide by the academic integrity policies of this course, the department, and university, and that this exam is my own work.

Signature: ________________________ Date: ________________

Instructions

Write your name and UBIT above, sign the Academic Integrity notice, and wait for course staff to begin the exam.

Answer each question on this exam to the best of your ability. You may make notes or perform calculations in the margins or any blank area on the bottom or margins of exam pages, on the designated scratch pages, or on the back of this cover sheet. If you mis-mark an answer and need to correct it, *draw a line through the mis-marked answer and circle the corrected answer.*

Questions vary in difficulty. *Do not get stuck on one question.* When you are finished, check to ensure that you have answered all questions, then turn in the entire exam (including all scrap pages used) to course staff.

Part a: NYC Taxi Trip Data

The New York City Taxi & Limousine Commission releases a yearly dataset, recording every taxi trip taken in New York City over the course of the year. For example, the 2018 dataset contains 112 million rows, each with 18 columns. Column types include **number**, **plain text**, and timestamps (treat as numbers). An incomplete list of example columns includes: (a) Vendor ID, (b) Pickup Timestamp, (c) Drop-Off Time, (d) Passenger Count, (e) Payment Type, (f) Pickup location, (g) Dropoff location, (h) Store and Forward Flag, (i) Fare, Tip, Tolls, Fees, Total Amount. The dataset is provided in no particular order.

Question A1 [**10 points**]

Propose a strategy, at the level of individual files, pages, and bytes, and sort order, for storing the 2018 NYC T&LC dataset on disk. Use diagrams wherever helpful. A good measure of whether you have a complete answer is whether a reader can unambiguously infer where the individual bytes of each field of each record are located within a file. Your answer does not need to enumerate every individual attribute above; you may instead provide generic guidelines for how numbers and plain text are to be handled.

Answer

The question is open-ended, so there is no one correct answer. However, as an example of the class of answer this question was looking for:
The dataset is stored row-wise in a paged layout in a single file. Each page uses an indexed layout, with a header containing pointers to each record. Individual records are stored using an index header to identify the location of each cell.

Point Breakdown

- **(5 pt)** The answer clearly describes a correct strategy for laying out data on disk.
- **(5 pt)** The strategy is reasonable for the data proposed.

Question A2 [**10 points**]

Propose a second strategy, distinct from your answer to Question 1. Clearly identify a situation (e.g., workload, disk style, etc...) where your new strategy would be preferable, and clearly identify a situation where your original strategy would be preferable.

Answer

The question is open-ended, so there is no one correct answer. However, as an example of the class of answer this question was looking for:
The dataset is stored column-wise with one file per column. Columns with fixed-size datatypes are stored directly as arrays. Columns with variable-size datatypes are encoded with a dictionary encoding and stored directly as arrays.

Point Breakdown

- **(5 pt)** The answer clearly describes a correct strategy for laying out data on disk.
- **(5 pt)** The answer clearly describes a situation in which the strategy would be preferable to that outlined in A1.

Part b: SQL

Each of the following parts will provide a SQL query and identify a table used by the query. For the identified table, answer the attached Yes/No questions, and provide a justification *in no more than* ***one sentence***. Unless otherwise specified, assume that all tables are stored as an unsorted collection of records (e.g., an unsorted array).

Question B1 [**5 points**]

```
SELECT COUNT(*) FROM students WHERE credits > 12;
```

Answer the following questions with respect to the `students` table.

1. Would the query run faster if the table were instead stored in a B+Tree?

Circle One		Justification
Yes	**No**	

2. If a bloom filter were available for this table, could it be used to make the query run faster.

Circle One		Justification
Yes	**No**	

3. If a fence pointer table were available for this table, could it be used to make the query run faster.

Circle One		Justification
Yes	**No**	

Answer

1. **Yes**; The query is looking for a range of values. A B+ tree indexed on credits could be used to efficiently (log time + linear in the result size) enumerate the subset of the result table that matches the query
2. **No**; Bloom filters support testing for the presence of individual elements, not ranges.
3. **Yes**; A sorted array with a fence pointer table over it works like a B+ Tree. Answers that noted that a fence pointer table didn't necessarily imply sortedness of the underlying data got full credit for this part.

Point Breakdown

- **(1 pt)** (Yes) for parts 1, 3
- **(2 pt)** A justification related to the support for range-based filtering for parts 1, 3
- **(2 pt)** (No) for part 2 with a justification related to the lack of support for range-based filtering.

Question B2 [**5 points**]

```
SELECT * FROM students WHERE id = 23;
```

Answer the following questions with respect to the `students` table.

1. Would the query run faster if the table were instead stored in a B+Tree?

Circle One	Justification
Yes **No**	

2. If a bloom filter were available for this table, could it be used to make the query run faster.

Circle One	Justification
Yes **No**	

3. If a fence pointer table were available for this table, could it be used to make the query run faster.

Circle One	Justification
Yes **No**	

Answer

1. **Yes**; The query is looking for a specific value. A B+ tree indexed on credits could be used to efficiently (log time) locate the record in question, if it exists.
2. **Yes**; A bloom filter over the `id` attribute could determine if there was no record `id = 23`, saving a trip to disk. Answers of **No** who's justification noted that record lookups by `id` were likely to be present (thus negating the value of the bloom filter) got full credit.
3. **Yes**; A sorted array with a fence pointer table over it works like a B+ Tree.

Point Breakdown

- **(1 pt)** (Yes) for parts 1, 3
- **(2 pt)** A justification related to the support for range-based filtering for parts 1, 3
- **(2 pt)** (Yes) for part 2 with a justification related to the query being a single-record lookup

Question B3 [**5 points**]

```
SELECT COUNT(*) FROM students JOIN enrollment
ON student.id = enrollment.student_id
```

Answer the following questions with respect to the `enrollment` table.

1. Would the query run faster if the table were instead stored in a B+Tree?

Circle One		Justification
Yes	**No**	

2. If a bloom filter were available for this table, could it be used to make the query run faster.

Circle One		Justification
Yes	**No**	

3. If a fence pointer table were available for this table, could it be used to make the query run faster.

Circle One		Justification
Yes	**No**	

1. Would the query run faster if the table were instead stored in a B+Tree?

Circle One		Justification
Yes	**No**	

2. If a bloom filter were available for this table, could it be used to make the query run faster.

Circle One		Justification
Yes	**No**	

3. If a fence pointer table were available for this table, could it be used to make the query run faster.

Circle One		Justification
Yes	**No**	

Answer

1. **Yes**; The query is iterating over all records; an index on `student_id` would allow lookups without having to construct hash-tables; or would necessitate that data be sorted, allowing the use of sort-merge join. Answers of **No** that explicitly related the runtime complexity of 1p or 2p hash join to the potential value of the B+ Tree in reducing lookup cost received full credit.
2. **Yes**; A bloom filter over `student_id` could be used to pre-filter rows of the `student` table, reducing memory complexity, and potentially opening 2 pass hash join. Answers of `No` that explicitly called out the low likelihood that there would be no `enrollment` records for a given `student_id` received full credit.
3. **Yes**; A sorted array with a fence pointer table over it works like a B+ Tree.

Point Breakdown

- **(2 pt)** (Yes) for all 3
- **(2 pt)** A justification related to the use of existing indexes instead of rebuilding a new one for the hash join

Question B4 [**5 points**]

```
SELECT COUNT(*) FROM students JOIN enrollment
ON student.id = enrollment.student_id
```

Identify two different join algorithms that could be used to implement the query above. For each algorithm you identify state a property of students, enrollment, and/or the query result, where the algorithm you identified would be preferable.

Answer

The question is open-ended, so there is no one correct answer. However, as a few examples of algorithms and ideal use cases:

- **Sort Merge Join**: Linear time if `student` and `enrollment` are already sorted on the `id` field.
- **1 pass Hash Join**: Lowest overall IO if sufficient memory exists to hold `students` or `enrollment` entirely in memory.
- **2 pass Hash Join**: Lowest IO complexity if neither `students` nor `enrollment` will fit entirely in memory.

Point Breakdown

- **(2+1 pt)** 2 different join algorithms indicated
- **(1+1 pt)** Correct justification for each algorithm

Part c: The RAM/EM Models

Consider each of the following algorithms, with the explicitly listed algorithm parameters. For each:

1. Identify every line of pseudocode that allocates memory, and identify where in the program that memory may be released.
2. Identify every line of pseudocode that performs IO (i.e., reads from/writes to disk) and state the IO complexity of the operation.
3. Identify the point in the algorithm where the maximum amount of memory has been allocated.
4. Set up a summation for the total IO performed during the algorithm.
5. State the worst-case (Big-O) Memory and IO complexity of the algorithm.

Question C1 [**10 points**]

The following algorithm performs the first part of sorting a dataset R initially provided as an on-disk file. The algorithm is provided in two parts. Your answer for this question should provide an analysis exclusively with respect to this first part of the algorithm. Complexity measures should be given in terms of $|R|$ (the number of records in the input file), B (buffer-size), and K (fan-in).

```
buffer ← a new B-element buffer
sorted_runs ← a new, empty queue
while R has more data do
    Read up to B records from R into buffer (or less if fewer records exist in R)
    Sort buffer in-place, in memory
    run ← a newly created file
    Write buffer to run
    Enqueue run to sorted_runs
end while
```

Answer

1. Allocations include `buffer` ($O(B)$; freed at end), `sorted_runs` ($O(1)$; produced as output), and data enqueued into `sorted_runs` ($O(1) \cdot O(\frac{|R|}{B})$ times; produced as output).
2. IOs include reading records from R into `buffer`, and writing `buffer` to `run`.
3. Max memory at end, with $O(\frac{|R|}{B})$ entries in `sorted_runs`.
4. Reading $\sum^{\frac{|R|}{B}} O(B) = O(|R|)$ and writing a like amount.
5. $O(|R|)$ IOS, $O(\frac{|R|}{B})$ memory

Point Breakdown

- **(1 pt)** Every allocation identified
- **(1 pt)** Every deallocation identified
- **(1 pt)** Every IO identified
- **(1 pt)** Complexity of every IO correct
- **(2 pt)** Point of max memory allocation correctly identified
- **(2 pt)** Correct summation for IO
- **(1 pt)** Correct Mem complexity
- **(1 pt)** Correct IO complexity

Question C2 [**10 points**]

The following algorithm performs the second part of sorting a dataset R initially provided as an on-disk file. The algorithm is provided in two parts. Your answer for this question should provide an analysis exclusively with respect to this second part of the algorithm (ignore memory allocated during the first part). Complexity measures should be given in terms of $|R|$ (the number of records in the input file), B (buffer-size), and K (fan-in).

```
while |sorted_runs| > 1 do
    current_level ← a vector containing up to K elements dequeued from sorted_runs
    For each file in current_level seek to the start of the file.
    output ← a newly created file
    while At least one file in current_level has more data do
        r ← the result of reading the least value that would be read next from any file in current_level.
        Write r to output
    end while
    Enqueue output to sorted_runs
end while
Dequeue from sorted_runs and return the result
```

Answer

1. Allocations include `current_level` ($O(K)$; freed at end), `r` ($O(1)$; released after while loop body), and data enqueued into `sorted_runs` (based on the observation that every enqueue follows K dequeues, memory usage shrinks)
2. IOs include reading one record at a time from `current_level` ($O(1)$) and writing one record at a time to `output` ($O(1)$). Observing that each iteration through the outer while loop dequeues K elements and enqueues 1 element, you can conclude that the outer while loop runs $O(\frac{|R|}{K})$ times. The inner while loop is bounded by $|R|$, since each record is read/written at most once. A slightly more intricate approach would be to note that, since this is merge sort, you can model the first $(\frac{|R|}{K})$ iterations as performing $|R|$ IOs, the next $\frac{|R|}{K^2}$ iterations as performing $|R|$ IOs, and so forth, leading to $\log_K |R|$ layers, each performing $|R|$ IOs.
3. Max memory at start, with $O(\frac{|R|}{B})$ entries in `sorted_runs`.
4. Depending on the answer to 2, either, $\sum^{\frac{|R|}{K}} O(|R|) = \frac{|R|^2}{K}$ or $\sum_{i=1}^{\log_K |R|} \sum^{\frac{|R|}{K^i}} O(|R|) = O(|R| \log_K(|R|))$.
5. $\frac{|R|^2}{K}$ or $O(|R| \log_K(|R|))$ IOS, $O(\frac{|R|}{B})$ memory

Point Breakdown

- **(1 pt)** Every allocation identified
- **(1 pt)** Every deallocation identified
- **(1 pt)** Every IO identified
- **(1 pt)** Complexity of every IO correct
- **(2 pt)** Point of max memory allocation correctly identified
- **(2 pt)** Correct summation for IO
- **(1 pt)** Correct Mem complexity
- **(1 pt)** Correct IO complexity

Question C3 [**10 points**]

The following algorithm performs a depth-first traversal of a graph G to build a spanning tree stored in the output file. The graph's adjacency list (i.e., out-edges) is stored in an on-disk B+Tree, using the vertex ID as a key. Complexity measures should be given in terms of $|G|$ (the number of vertices) and D (the maximum out-degree of any vertex in G). You may assume that the graph is fully connected.

```
queue ← a new, empty queue containing the vertex ID of an arbitrary vertex.
output ← a new, empty on-disk B+Tree
while queue is non-empty do
    currentID ← dequeue from queue
    out_edges ← read out edges for vertex currentID
    for edge in out_edges do
        if output does not contain edge.destinationID then
            Write (edge.destinationID → currentID) to output
            Enqueue edge.destinationID
        end if
    end for
end while
```

Answer

1. Allocations include `queue` ($O(1)$; freed at end), `out_edges` ($O(D)$; released after while loop body), and data enqueued into `queue` (recall, this is capped at $|G|$, since each node is enqueued at most once)
2. IOs include reading `out_edges` ($O(\log|\texttt{out\_edges}| + D) < O(\log|G| + D)$, at most $O(|G|)$ times) and writing (`edge.destinationID` → `currentID`) ($O(\log|\texttt{out\_edges}|) < O(\log|G|)$ at most $O(|G|)$ times).
3. Max memory at enqueue to `queue`; at worst, $O(|G|)$.
4. $\sum^{|G|} O(D + \log|G|) + O(\log|G|)$
5. $O(|G|\log|G| + |G|D)$ IOS, $O(|G|)$ memory

Point Breakdown

- **(1 pt)** Every allocation identified
- **(1 pt)** Every deallocation identified
- **(1 pt)** Every IO identified
- **(1 pt)** Complexity of every IO correct
- **(2 pt)** Point of max memory allocation correctly identified
- **(2 pt)** Correct summation for IO
- **(1 pt)** Correct Mem complexity
- **(1 pt)** Correct IO complexity

Part D: Concurrency

For each of the following schedules identify whether:

- The schedule is a serial schedule. Give the serial order of the processes
- The schedule is a conflict-serializable schedule. Show the happens-before graph.
- The schedule could have been created by 2-phase locking (with standard, mutex-style, locks). Show where the locks would be placed.
- The schedule could have been created by 2-phase locking (with reader/writer locks). Show where the locks would be placed.

Question D1 [10 points]

→ **time** →

P1	W(B)			R(A)		
P2		R(B)			R(A)	
P3			R(B)			W(A)

→ **time** →

Serial	Conflict Ser.	2PL	Reader-Writer 2PL
Yes No	Yes No	Yes No	Yes No

Answer

see below

Point Breakdown

- **(1 pt)** (No) Serial Schedule
- **(2 pt)** (Yes) Conflict Serializable
- **(1 pt)** Happens-before graph
- **(3 pt)** (No) 2-phase locking
- **(2 pt)** (Yes) 2-phases R/W locking
- **(1 pt)** R/W Locks identified

Question D2 [10 points]

→ **time** →

P1	W(B)	R(A)				
P2					R(B)	R(A)
P3			R(B)	W(A)		

→ **time** →

Serial	Conflict Ser.	2PL	Reader-Writer 2PL
Yes No	Yes No	Yes No	Yes No

Answer

see below

Point Breakdown

- **(1 pt)** (Yes) Serial Schedule with order
- **(2 pt)** (Yes) Conflict Serializable
- **(1 pt)** Happens-before graph
- **(2 pt)** (Yes) 2-phase locking
- **(1 pt)** Locks identified
- **(2 pt)** (Yes) 2-phases R/W locking
- **(1 pt)** R/W Locks identified

Question D3 [10 points]

→ **time** →

P1	W(B)			R(A)		
P2		R(B)			W(A)	
P3			R(B)			W(A)

→ **time** →

Serial	Conflict Ser.	2PL	Reader-Writer 2PL
Yes No	Yes No	Yes No	Yes No

Answer

see below

Point Breakdown

- **(1 pt)** (No) Serial Schedule
- **(2 pt)** (Yes) Conflict Serializable
- **(1 pt)** Happens-before graph
- **(3 pt)** (No) 2-phase locking
- **(3 pt)** (No) 2-phases R/W locking (was technically a way, credit was given if justified)