Midterm

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: Complexity

Consider the following two algorithms, which take, as input, two files (**input1** and **input2**) containing unsorted arrays of records in delimited format. The algorithms both emit outputs to a delimited file (**output**). The second algorithm takes an additional parameter (**B**) as an input. Both algorithms produce the same set of outputs, albeit in different orders.

Algorithm 1 Unbuffered "Nested Loop" Join

```
for a ∈ input1 do
    for b ∈ input2 do
        if a.key = b.key then
            Append ⟨a, b⟩ to output
        end if
    end for
end for
```

Algorithm 2 Buffered "Block-Nested Loop" Join

```
buffer ← Initialize an array of size B
while More records exist to be read from input1 do
    for $i ∈ 0..$B do
        buffer[i] ← Read record from input1
    end for
    for b ∈ input2 do
        for a ∈ buffer do
            if a.key = b.key then
                Append ⟨a, b⟩ to output
            end if
        end for
    end for
end while
```

Answer the following questions with respect to the size of the three files: $N = |$**input1**$|$, $M = |$**input2**$|$, and $O = |$**output**$|$; and as appropriate the parameter B.

Question A1 [**10 points**]

What is the asymptotic memory complexity of each of the algorithms?

Answer

- Algorithm 1: $O(1)$
- Algorithm 1: $O(\mathbf{B})$

Point Breakdown

- **(1 pt)** Answers recognize that iterating over a file requires constant memory.
- **(1 pt)** Answers recognize that appending to a file requires constant memory.
- **(3 pt)** Answer for Algorithm 1 recognizes that the method allocates no additional memory.
- **(3 pt)** Answer for Algorithm 2 recognizes that the buffer consumes a non-constant amount of memory.
- **(2 pt)** Answer for Algorithm 2 recognizes that, apart from the buffer, no additional memory is allocated.

Question A2 [**10 points**]

What is the asymptotic IO complexity of each of the algorithms?

Answer

- Algorithm 1: $O(N \cdot M)$
- Algorithm 1: $O(\frac{N \cdot M}{\mathbf{B}})$

Point Breakdown

- **(1 pt)** Both answers recognize that iterating over a file requires linear IO.
- **(1 pt)** Both answers recognize that appending to a file requires linear IO.
- **(2 pt)** Both answers recognize that the nested for loop requires multiple iterations over one of the files.
- **(2 pt)** Answer for Algorithm 1 is correct ($O(N \cdot M)$ or equivalent)
- **(2 pt)** Answers recognize that buffering reduces the IO complexity.
- **(2 pt)** Answer for Algorithm 2 is correct ($O(\frac{N \cdot M}{B})$ or equivalent).

Question A3 [**5 points**]

What is the asymptotic runtime complexity of each of the algorithms?

Answer

$O(N \cdot M)$ in both cases.

Point Breakdown

- **(3 pt)** Answer for Algorithm 1 is correct.
- **(2 pt)** Answer for Algorithm 2 is correct.

Part b: Scenarios

Review each of the following use cases. For each use case, for each data data structure listed: (i) state whether the data structure would be appropriate for the use case, (ii) in at most 1-2 sentences, identify features of the use case that justify your choice.

- **Array+FP**: Sorted Array with In-Memory Fence Pointer Table
- **Array+FP+BF**: Sorted Array with In-Memory Fence Pointer Table and Bloom Filter
- **B+Tree**: B+ Tree
- **LSM Tree**: An LSM Tree who's sorted runs are sorted arrays with In-Memory Fence Pointer Tables and Bloom Filters.
- $\beta\epsilon$ **Tree**: Beta-Epsilon Tree

Question B1 [**15 points**]

Sales Records: The data being stored contains a large number of large (∼500B) records, each identified by a small (u32) key. The list of records is updated weekly in a large process, late at night when few people need to access the list. Accesses consist primarily of requests for specific keys that are likely to be present in the data, or records falling in a range of keys.

Answer

Key features of this use case include:

- **Offline-only updates** mean that the cost of updating an Array representation is low (pro: Array+FP, Array+FP+BF), and that there is little value in accepting the overheads of the write-optimized data structures (con: LSM Tree, $\beta\epsilon$ Tree).
- **Very large number of records** mean that a data structure with an in-memory component that's linear in the size of the data may exceed the available memory (con: Array+FP, Array+FP+BF; pro: B+Tree)
- **Large record sizes** mean that an indexing scheme (e.g, like a B+Tree or FP table) will help a lot, even if entirely in memory.
- **Support for range access** means that data structures that don't randomly scatter data around the file will be faster (con: B+Tree, $\beta\epsilon$ Tree; pro: Array+FP, Array+FP+BF, LSM Tree)
- **Records likely to be present** means that the primary value proposition of bloom filters is irrelevant (con: Array+FP+BF)

An ideal answer would argue for Array+FP and/or B+Tree.

Point Breakdown

- **(6 (up to 2x) pt)** Valid justification for using one of more of the good data layouts.
- **(3 pt)** Valid justifications for not using the other data layouts.

Question B2 [**15 points**]

Idealized TikToks: The data being stored is updated frequently. Records are of moderate size (∼100B), and inserted with a monotonically increasing `u64` key (each inserted key is greater than any previously inserted key). Accesses consist primarily of requests for: (i) The 1000 most recently inserted keys, (ii) A record with a specific key where the key is recent 80% of the time, and old 20% of the time.

Answer

Key features of this use case include:

- **Frequent appends** The data is modified frequently, but always by appending (not really a con: Array+FP, Array+FP+BF). Because data is always appended, the 'buffer' of a $\beta\epsilon$ tree is only used on the rightmost branch (con: $\beta\epsilon$ tree).
- **Records likely to be present** means that the primary value proposition of bloom filters is irrelevant (con: Array+FP+BF)
- **Most accesses are to recent records**, so LSM trees and Arrays behave comparably.

An ideal answer would argue for Array+FP and/or LSM Tree.

Point Breakdown

- **(6 (up to 2x) pt)** Valid justification for using one of more of the good data layouts.
- **(3 pt)** Valid justifications for not using the other data layouts.

Question B3 [**15 points**]

Realistic TikToks: As question 2, but records are inserted in a *mostly monotonic order*. That is, most insertions will be in order, but some insertions may be arbitrarily out-of-order.

Answer

Key features of this use case include:

- **Frequent updates** The data is modified frequently (con: Array+FP, Array+FP+BF).
- **Records likely to be present** means that the primary value proposition of bloom filters is irrelevant (con: Array+FP+BF)
- **Most accesses are to recent records**, so LSM trees and Arrays behave comparably.

An ideal answer would argue for LSM Tree or B+Tree.

Point Breakdown

- **(6 (up to 2x) pt)** Valid justification for using one of more of the good data layouts.
- **(3 pt)** Valid justifications for not using the other data layouts.

Question B4 [15 points]

Inventory: The data being stored consists of `u64` keys, paired with a `u64` integer count. Most accesses consist of three steps: (i) Check if a key exists (it may not), (ii) Decrement the count paired with the key, and (iii) if the count reaches 0, remove the record. Infrequently (about once per day), the inventory is updated with a two step process: (i) Check if the key exists, (ii) Add a number to the count if present, or insert the key if not.

Answer

Key features of this use case include:

- **Frequent updates** The data is modified frequently with records being inserted and removed (con: Array+FP, Array+FP+BF; pro: $\beta\epsilon$ Tree, LSM Tree, B+Tree)
- **Records not always present** Bloom filters can be used to reduce lookup times for missing records (pro: Array+FP+BF)

An ideal answer would argue for a $\beta\epsilon$ Tree, LSM Tree, or B+Tree.

Point Breakdown

- **(6 (up to 2x) pt)** Valid justification for using one of more of the good data layouts.
- **(3 pt)** Valid justifications for not using the other data layouts.

Part c: Data Layout and Access

Consider the algorithm below, which acts on the following two inputs:

- **keys**: An in-memory list of K keys of type `u32`. The keys are provided in arbitrary order.
- **file**: An on-disk file storing N records, each 1KB in size. Each record has a unique `u32` key.

```
1: total ← 0
2: for key ∈keys do
3:     record ← find and load record with key key from file
4:     total ← total + record.value
5: end for
6: return total
```

Question C1 [10 points]

In no more than 2-4 short sentences, how would you organize **file** (and how would you implement line 3) to minimize the IO complexity of the algorithm above *without changing the algorithm further.*

Answer

We need efficient *random* read-only access to specific records. Of the structures discussed in class, a B+Tree, ISAM index, or Sorted Array would allow us to access records in logarithmic time (the best of any structures discussed). Of these, the ISAM index or B+Tree have the best log base.

Point Breakdown

- **(3 pt)** The answer recognizes that the workload is read-only
- **(2 pt)** The answer recognizes that the workload is focused on random-access to data
- **(3 pt)** The answer selects a data structure with at most a logarithmic cost per record.
- **(2 pt)** The answer selects a data structure that has better than logarithmic point access, with a logarithmic base better than 2.

Question C2 [10 points]

Assume that $K \approx N$. That is, each $4kb$ page will likely be read about 4 times. In at most 1-2 sentences, how would you modify both the algorithm and the layout of **file** to ensure that each page is read at most once.

Answer

Sort the data, sort the keys, use a merge-sort-style scan.

Point Breakdown

- **(5 pt)** The answer recognizes that a single-pass over the data is possible.
- **(5 pt)** The answer adjusts the algorithm to allow a single-pass implementation.

Part D: Tradeoffs

Later in the semester, we'll talk about a type of database called a column store. For now, what you need to know is that column stores store (on disk) arrays of values. The column store needs to be able to:

- Retrieve the value at index i.
- Retrieve all indices where the value is equal to v.
- Retrieve all indices where the value is between ℓ and h.
- Append a new value v to the end of the array.

Although the column store needs to be able to support all of these, for some use workloads, one or two of these operations may be invoked much more frequently than the others. You can also assume that the values being stored are fixed size (e.g., integers).

Consider the following two strategies for storing the array on-disk. The first is a simple fixed-size array layout. If each value is V bytes wide, the ith value is stored at byte $i \cdot V$.

The second layout is one that we have not yet discussed in class; it's something called run-length encoding. Instead of storing the array directly, we only store one copy of each distinct value, along with a list of the indices at which the value appears. Specifically, in this layout, we store the array as collection of records, each containing (i) The value; (ii) The number of times the value appears; and (iii) A list of indices at which the value appears.

Note that the records are variable length (since each distinct value may appear a different number of times in the array). These records are stored in what we called 'indexed page' layout (i.e., each page has a set of pointers to the start of each record on the page). We maintain a fence pointer table in-memory, splitting on the lowest value stored on each page.

Question D1 [**5 points**]

In at most 2-3 sentences, identify a set of properties (of the dataset, workload) for which the first (fixed-size) layout would be preferable.

Answer

Legitimate reasons include:

- Access to the array by index is constant-time/IO, while it would be linear for run-length encoding.
- Appends to the array are constant-time/IO, but up to linear for run-length encoding.
- The amount of space used is V per individual record, as opposed to V plus 2 integers, plus an additional integer for every duplicate; For arrays where the value is integer-sized, or where there are few duplicates, the run-length encoded version is larger.

Point Breakdown

- **(2 pt)** The answer identifies a property that would argue for the first encoding.
- **(3 pt)** The answer includes a reasonable justification.
- **(3; partial credit pt)** The answer discusses properties of fixed encodings in general, and not specific to this answer.

Question D2 [**5 points**]

In at most 2-3 sentences, identify a set of properties (of the dataset, workload) for which the second (run-length encoded) layout would be preferable.

Answer

If the value is large, and there are many duplicates, the run-length encoded version is smaller; Fewer IOs are required to access the data or any sub-range of it.

Point Breakdown

- **(2 pt)** The answer identifies a property that would argue for the first encoding.
- **(3 pt)** The answer includes a reasonable justification.