CS222P – Fall 2018, Midterm Exam
Principles of Data Management
Department of Computer Science, UC Irvine
Prof. Chen Li
(Max. Points: 100)

Instructions:
• This exam has five (5) questions.
• This exam is closed book. However, you can use one cheat sheet (A4 size).
• The total time for the exam is 80 minutes, so budget your time accordingly.
• Be sure to answer each part of each question after reading the whole question carefully.
• If you don’t understand something, ask for clarification.
• If you still find ambiguities in a question, write down the interpretation you are taking and then answer the question based on that interpretation.

<table>
<thead>
<tr>
<th>QUESTION</th>
<th>POINTS</th>
<th>SCORE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>TOTAL</td>
<td>100</td>
<td></td>
</tr>
</tbody>
</table>
Question 1: Short questions (20 points)

a) (4 pts) Explain why many database systems use their own buffer manager instead of using the default buffer manager provided by the operating system.

Certain operations in the database have certain unique characteristics. The database buffer manager can utilize these characteristics to develop replacement policies better than the policy by the buffer manager of the OS.

b) (4 pts) Consider a table Student(id, name, age, gpa, email, department). Use this table to explain the concepts of “index-only plans” and “covering indexes”.

Consider a composite unclustered index on (department, gpa)

SELECT department, AVG(gpa)
FROM student
GROUP BY department;
This query can be answered by an index-only plan, which only accesses this index without accessing the original records. The index is called a “covering index” for this query since the index includes all the attributes needed in the query.

c) (4 pts) Explain a main benefit of using relational tables to store catalog information.

We can reuse the logic and functionalities of the record manager to read/write catalog information from/to tables.
d) (4 pts) Explain at least two benefits of using a column store compared to a row store.

- Store values of the same type; better chances to do compression;
- Efficient support of queries that access only a few columns.

e) (4 pts) In our record manager in the course projects, each record ID includes a page number and a slot number. Briefly explain the main advantage of this design.

In case we need to organize the bytes within this page, we can just change the offset stored in the slot of this record without changing the RID of this record.
Question 2: Record Manager (14 points)

Suppose we have a table with the following schema:

\[
\text{Movie (} id \text{ INT, } director \text{ VARCHAR(30), title VARCHAR(150), producer VARCHAR(50), rating FLOAT)}
\]

We store the table as a heap file of variable-length records. Each record is stored as a sequence of bytes with a directory of pointers. Assume that:

1. The directory uses a 2-byte offset to store the ending position for each field.
2. The offset starts from 0, which is the beginning of the directory.
3. The schema is known in all record operations so there is no need to store the number of fields in each record.
4. Integers and floating-point numbers are 4 bytes each.
5. Each NULL value is represented using a special value -1 in the corresponding pointer in the directory.

Consider this record:

\[(1234, \text{“Steven”}, \text{“ET”}, \text{NULL}, 7.9)\]

a) (7 pts) Fill in the following byte array that illustrates how this record is stored using the format stated above. Clearly indicate the number of bytes in each segment and all their values.

The offset value could be the end of the field or the beginning of following field, and we take both as correct as long as your solution is consistent.

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
13 19 21 -1 25 1234 Steven ET 7.9
```

or

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
14 20 22 -1 26 1234 Steven ET 7.9
```

b) (7 pts) Fill in the following byte array using the “\texttt{data}” format defined in “\texttt{insertRecord()}” function of RBFM in our course project, as following:

```
[null-indicator] [value of 1st field] [value of 2nd field] …
```

Each Int/Real uses 4 bytes and each Varchar uses 4 bytes for the length followed by actual values. For the null indicator, show the values of the 0/1 bits. Clearly show the number of bytes in each segment and all their values.
Question 3: B+ Tree (20 points)

Consider the following ISAM tree:

For the following questions, draw the updated ISAM tree after each operation on the original tree. If there is no ambiguity, you can just draw the part that is changed. For each operation, clearly add an “R” (for read) and “W” (for write) on all the affected pages, and “A” (meaning appending a page, which includes its “Write” operation) as well as the original A/B/C/.../I labels.

We assume that each node can hold up to 4 key entries.

The following is an example after inserting 6 on the original tree:
a) (5 pts) Delete 55 from the original tree.

b) (5 pts) Insert 24 to the original tree.
c) (5 pts) Draw the updated B+ tree after inserting 78 to the following tree (For this question you don’t need to write which pages have been read, written, or appended).

![B+ Tree Diagram]


d) (5 pts) In the lectures we discussed string key compression in the intermediate nodes in order to increase the fanout of a B+ tree. Illustrate how that can be done on the following example:

![Example Diagram]

We can compress the key “James Angel” to “Jam” which will hold the B+ tree logic of having (left side < key <= right side)
Question 4: Hash Tables (20 points)

Consider storing a set of course codes in a table, which has a dynamic hash table. Suppose we want to insert the following 10 entries sequentially. We are using a hash function called \( h \), which simply computes a hash value of a course code based on its binary value (their hash values are also provided below). Only 2 records can fit into a bucket/page. Assume that the bits of \( h(k) \) are used in least-to-most order of significance. For example, for a binary value “0010”, we use the two rightmost bits “10” to find the bucket.

<table>
<thead>
<tr>
<th>Course Code</th>
<th>Hash Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>122a</td>
<td>0001</td>
</tr>
<tr>
<td>122b</td>
<td>0010</td>
</tr>
<tr>
<td>122c</td>
<td>0011</td>
</tr>
<tr>
<td>171</td>
<td>0100</td>
</tr>
<tr>
<td>222</td>
<td>0101</td>
</tr>
<tr>
<td>223</td>
<td>0110</td>
</tr>
<tr>
<td>241</td>
<td>0111</td>
</tr>
<tr>
<td>274</td>
<td>1000</td>
</tr>
<tr>
<td>290</td>
<td>1001</td>
</tr>
<tr>
<td>299</td>
<td>1010</td>
</tr>
</tbody>
</table>

a) (10 pts) Suppose the index is based on linear hashing and our policy is to split the page (only once) indicated by the “Next” pointer whenever a new overflow page is created due to an insertion. Below is the state of dynamic hashed index after inserting the first two data entries. Draw a diagram representing the final state of the dynamic hash index after inserting all the 10 given data entries. You are highly recommended to use a pencil and an eraser to work on your solution.

Level = 1, Next = 0
**copied for your convenience.**

<table>
<thead>
<tr>
<th>Course Code</th>
<th>Hash Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>122a</td>
<td>0001</td>
</tr>
<tr>
<td>122b</td>
<td>0010</td>
</tr>
<tr>
<td>122c</td>
<td>0011</td>
</tr>
<tr>
<td>171</td>
<td>0100</td>
</tr>
<tr>
<td>222</td>
<td>0101</td>
</tr>
<tr>
<td>223</td>
<td>0110</td>
</tr>
<tr>
<td>241</td>
<td>0111</td>
</tr>
<tr>
<td>274</td>
<td>1000</td>
</tr>
<tr>
<td>290</td>
<td>1001</td>
</tr>
<tr>
<td>299</td>
<td>1010</td>
</tr>
</tbody>
</table>

Level = 3, Next = 2

<table>
<thead>
<tr>
<th>h2</th>
<th>h3</th>
<th>Next</th>
<th>primary pages</th>
<th>overflow pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td></td>
<td>274</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>00</td>
<td></td>
<td>122a, 290</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>01</td>
<td></td>
<td>122b, 223</td>
<td>299</td>
</tr>
<tr>
<td>11</td>
<td>01</td>
<td></td>
<td>122c, 241</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td></td>
<td></td>
<td>171</td>
<td></td>
</tr>
<tr>
<td>101</td>
<td></td>
<td></td>
<td>222</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>111</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
b) (5 pts) Suppose in **extendible hashing**, our split policy is the following: whenever an insertion causes a new overflow page, split the pages with this hash value, *only once*.

Also suppose in **linear hashing**, our split policy is the following: whenever an insertion causes a new overflow page, split the page(s) pointed by the “Next” pointer, *only once*.

Under these assumptions, suppose we have a good hash function that hashes each key uniformly into a hash space and the hash index is used as a unique index, i.e., each key appears only once. Which hash scheme provides a better query performance, extendible hashing or linear hashing? Briefly justify your answer.

Linear hashing will be better since it has no directory and keys are uniformly distributed (meaning overflows will be rare).

We will also take that answer that both methods perform similar, since the directory of extendible will be relatively small and can be cached. However, saying extendible hashing performs better than linear hashing would be wrong.

c) (5 pts) Under the same assumptions about the split policy, suppose the hash function is poorly designed that results in many hash collisions. Which hash scheme provides a better query performance, extendible hashing or linear hashing? Briefly justify your answer.

Extendible hashing will be better since it has bounded I/O cost for lookups, i.e., 1 I/O for directory and 1 I/O for data page. However, long overflow chains could develop for linear hashing.
Question 5: Indexing Performance (26 points)

Consider a table

Tweet (ID, user, msg, lat, long, retweetCount)

that contains 10,000,000 records (10M) organized as a sorted file ordered by ID, and each page contains 100 records. We assume:

- “ID” is an integer value uniformly distributed over the range \([0, 9,999,999]\).
- “user” is a variable-length string of a maximum length of 20 and with a not-null constraint.
- “msg” is a variable-length string of a maximum length of 70.
- “lat” is a double value uniformly distributed over the range \([-90.0, 90.0]\).
- “long” is a double value uniformly distributed over the range \([-180.0, 180.0]\).
- “retweetCount” is an integer value uniformly distributed over the range \([0, 4,999]\).
- The value distributions of multiple attributes are independent.
- Each index page contains 1000 single-key entries.
- Each index page contains 500 composite key entries.
- Non-leaf pages of a B+ tree are always cached by the buffer manager.

a) (5 pts) What is the I/O cost of the full scan over the “Tweet” table?

\[
10M/100 = 100K
\]

For questions b) - d), describe which of the following access methods would work the best for the given workload, and the number of disk IOs. Briefly explain the reason and indicate whether an index-only plan is applicable or not.

1) Sorted-file scan;
2) Unclustered B+ tree index on “lat”;
3) Unclustered B+ tree index on “long”;
4) Unclustered B+ tree index on “retweetCount”;
5) Unclustered B+ tree index on composite key “<lat, long>”;
6) Unclustered B+ tree index on composite key “<retweetCount, ID>”;

b) (7 pts) SELECT ID, user FROM Tweet WHERE retweetCount >= 4990;

4) is the best, total I/O = 20,020
index-only plan not applicable
# of records meet condition = 10M * (5,000 – 4,990) / 5,000 = 20,000
If using 4):
   Index I/O = 20,000 / 1,000 = 20
   Data I/O = 20,000
   * Total I/O = 20,000 + 20 = 20,020

If using 6):
   Index I/O = 20,000 / 500 = 40
   Data I/O = 20,000
   Total I/O = 20,000 + 40 = 20,040

**c) (7 pts)** SELECT * FROM Tweet WHERE lat BETWEEN 33.0 AND 42.0
   AND long BETWEEN -124.0 AND -115.0;

2)+3)+intersection is the best, total I/O = 13250
index-only plan not applicable

# of records meet condition = 10M * 9 * 9 / (360 * 180) = 12500

If using 5):
   # of index entries accessed = 10M * (9 / 180) = 500,000
   Index I/O = 500,000 / 500 = 1000
   Data I/O = 12500
   Total I/O = 12500 + 1000 = 13500

If using 2) + 3) + intersection:
   # of index entries accessed by 2) = 10M * (9 / 180) = 500,000
   # of index entries accessed by 3) = 10M * (9 / 360) = 250,000
   Index I/O = (500,000 + 250,000) / 1000 = 750
   Data I/O = 12500
   * Total I/O = 12500 + 750 = 13250

**d) (7 pts)** SELECT retweetCount, Count(ID)
   FROM Tweet
   WHERE ID BETWEEN 1 AND 1,000,000
   AND retweetCount BETWEEN 1,001 AND 1,005
   GROUP BY retweetCount;

Index-only plan is applicable.

6) is the best, total I/O = 5 or 17 (both are correct)
Either of the following will be counted as correct
If decomposing into several small queries for retweetCount = 1001, 1002, ..., 1005:
   # index entries accessed = 10M * (5 / 5,000) * (1 / 10) = 1,000
   Index I/O = 1,000 / 500 = 2
* Total I/O = \( \max(5, 2) = 5 \) (Each small query needs at least one page I/O)

If not decomposing:

\[
\text{# index entries accessed} = 10M \times \left(\frac{4}{5,000}\right) + 10M \times \left(\frac{1}{5000}\right) \times \left(\frac{1}{10}\right) = 8,200
\]

(Note: when retweetCount = 1005, and id scanned to 1M, then doesn’t need to continue scan the following leaf nodes)

\[
\text{# Index I/O} = \frac{8,200}{500} = 16.4 = 17
\]

* Total I/O = 17