CS 222/122C – Fall 2016, Midterm Exam  
Principles of Data Management  
Department of Computer Science, UC Irvine  
Prof. Chen Li  
(Max. Points: 100)

Instructions:
- This exam has six (6) 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.
- The last two pages of this exam are blank; you can use them as scratch paper.

<table>
<thead>
<tr>
<th>QUESTION</th>
<th>POINTS</th>
<th>SCORE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>13</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>20</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>TOTAL</td>
<td>100</td>
<td>100</td>
</tr>
</tbody>
</table>
Question 1 (12 points) Short questions

a) (4 pts) Explain the purpose of the “pin()” and “unpin()” operations for a page in the buffer pool and how they are related to cache replacement.

A requester for a page needs to “pin” the page to indicate that the requester is using it. Internally we keep a “pin count” to know the number of requesters for this page. “pin()” increments this count by 1. The buffer manager can replace this page only if the pin count is 0. A requester should “unpin” the page after it finishes the operations on the page. “unpin()” decrements this count by 1.

b) (4 pts) The search performance of B+-tree depends on its length, which is related to the fanout of each node. For each of the “INT” type and “VARCHAR” type for an index key, give a technique to increase the fanout. Explain each technique using a simple example.

INT: since all the key values within a node are sorted, we can store the first key value and a sequence of deltas. These deltas tend to be small, and can be stored with fewer bytes using variable-length coding.

VARCHAR:

Suffix compression: “David Smith” can be shortened to “Dav”.

In general, as long as the condition “the right-most key in the left child < key <= the left-most key in the right child”, we have the opportunity to reduce the size of the “key” to save space.

Another solution is prefix compression. The main idea is the following. For those keys share some common prefix, we can store the prefix once then store the additional characters for each key.

c) (4 pts) Briefly explain the main idea behind a column store and at least one main advantage compared to a row store.
Store all the values of one attribute/column physically together.

Advantages: (1) If a query only retrieves a few columns, we can reduce the number of disk IOs by not reading the bytes of other irrelevant columns; (2) all the values with the same type can be compressed more easily.
Question 2 (13 points): Record Manager

a) (5 pts) Suppose we have a table with the following schema:
Player(pid INT, name VARCHAR(30), team VARCHAR(30), points_per_game 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 2-byte offsets to store the ending positions of the fields; (2) The schema is known in all record operations so there is no need to store the number of fields in each record; (3) Integers and floating point numbers are 4-byte long. (4) Each NULL value is represented using a special value in the corresponding pointer in the directory.

1) Please fill in the following byte array to explain how the following record is stored. Clearly indicate the number of bytes per segment in the array, and all their values.

(23, NULL, “Chicago”, 37.1)

| 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 |
| 12 | -12 | 19 | 23 | 23 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

Solution:

2) Explain how to use the directory to access the “team” field of this record. Make sure to include all the details in the calculation.

Solution: Since “team” is the 2nd field (starting from 0) of the record, we access the 2nd slot in the directory, which has a value of “19”. Since it’s a positive value, we know it’s not a NULL. Then we access its previous slot, which has a value “-12”, indicating the previous value is a NULL, and it ends at position 12. (If the design uses an arbitrary negative value to indicate the NULL value, we have to keep looking backwards until we see the first non-NUL value, which is still 12.) Then we know the length of this field is “19-12 = 7”. We read 7 bytes as the value for the “team” field.
b) (4 pts) Suppose we have an empty page with a size of 4,096 bytes. Use a diagram to show the layout of the page after we insert 3 new records to the page with a byte-array size of 30, 25, and 35, respectively. We assume each offset takes 2 bytes, and each array length also uses 2 bytes. Finish the following diagram by filling in the missing values, including the record offsets and lengths, number of bytes in free space, and number of slots.
Solution:

<table>
<thead>
<tr>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Record offset (2 bytes) | Free space in bytes (2 bytes)

55 | 30 | 0 | 3 | 3990

35 | 25 | 30 |

Record length (2 bytes) | # of slots (2 bytes)

c) (4 pts) In what operation do we need a tombstone for a record?

When a record is updated with a larger size and it no longer fits in the current page.

What information should be included in a tombstone?

The new RID where the moved record is stored.
Question 3 (20 points): B+ Tree

For sub-question a), b), and c), consider the following B+ tree (called “tree 1”):

![B+ Tree Diagram]

a) (4 pts) Suppose a data entry with key 90 is inserted to the B+ tree. Write down all the nodes that need to be read and all the nodes that need to be updated. You can mark “R” (for “read”) and “U” (for “Update”) on the nodes in the tree above directly.

Read: A, C, K
Write: K

b) (4 pts) Consider the original B+ tree 1 above. Draw the new B+ tree after inserting a data entry with key 42. If there is no ambiguity, you can just draw the part that is changed.

Solution:

![New B+ Tree Diagram]

c) (4 pts) Consider the original B+ tree 1 above. Draw the new B+ tree after inserting a data entry with key 79. If there is no ambiguity, you can just draw the part that is changed.
Solution:

For sub-questions d) and e), consider the following B+ tree (called “tree 2”):

**d) (4 pts) Consider the original B+ tree 2 above. Draw the new B+ tree after deleting the data entry with key 23. Assume we need to redistribute data entries if possible. If there is no ambiguity, you can just draw the part that is changed.**
e) (4 pts) Consider the original B+ tree above. Draw the new B+ tree after deleting the data entry with key 50. If there is no ambiguity, you can just draw the part that is changed.

Solution:

We also accept solution if the entry 50 in the root node is replaced by 59.
Question 4. Dynamic Hashing (20 Points)

We have a table: Flight (flightNo INT, aircraft INT, origin VARCHAR(10), destination VARCHAR(10), price INT). It has an extendible hashing index shown below. The directory consists of an array of 4-byte pointers, each of which points to a page/bucket.

The hash function $h$ uses the least significant bits of $h(K)$. Up to 4 records can fit into one page/bucket. For the following questions, we assume that we split a page whenever it has an overflow page.
a) *(5 Points)* We insert a data entry with a hash value 43. Show the file structure after the insertion. If there is no ambiguity, you can just draw the part that changed.
b) (5 Points) On the original hash table, we delete a record for an entry with hash value 6. Suppose the “merge policy” is that whenever we can merge a bucket with its “mirror bucket” without causing an overflow, we merge these two buckets. Draw the new index structure after the deletion. If there is no ambiguity, you can just draw the part that changed.
c) (10 Points) We have a linear hashing index (shown below) for the same table:

**Flight** (flightNo INT, aircraft INT, origin VARCHAR(10), destination VARCHAR(10), price INT)

Each bucket can hold up to 4 records. Our split policy is to split the page indicated by “Next” pointer whenever an overflow page is created due to an insertion.
Show the final file structure that will result if entries with hash values 15, 26, 57, and 25 are inserted to the linear hash table one by one. Make sure to include the details about the “Level” and “Next” pointer.

**Solution: structure after inserting “15”:**

```
Level = 0

- h1  h0
  000  00   32 8 24
  001  01   9 41 17
    Next = 2
  010  10   14 18 10 30
  011  11   31 35 7 11   15
  100  00   44 36
  101  01   21
```
The final structure:
Question 5. Query Performance (20 points)

Consider a table Flight (flightNo, aircraft, origin, destination, price) containing 100,000 records organized as a heap file.

- “flightNo” is an integer with values distributed uniformly over the range [0, ..., 99,999].
- “aircraft” is an integer with values distributed uniformly over the range [100, 400].
- “price” is a double with values distributed over the range [0, ..., 10,000].

We have the following secondary indexes on the table:

1) Unclustered B+ tree index on “aircraft”;
2) Unclustered B+ tree index on “price”;
3) Unclustered B+ tree index on composite key “<aircraft, price>”;
4) Unclustered B+ tree index on composite key “<price, aircraft>”.

a) (5 points) Draw a single diagram to illustrate the main idea of these four indexes and how their leaf nodes point to records in the heap file.
For each of the following queries, decide a most efficient access strategy (either a scan on the heap file or using one of the four indexes). Briefly explain the reason.

b) (5 points) SELECT * FROM Flight F WHERE F.aircraft = 393;

Index 1 (equality on aircraft)

c) (5 points) SELECT * FROM Flight F WHERE F.price > 5,000.

Heap-file scan or Index 2 (sorting record ids before accessing the heap file).

d) (5 points) SELECT F.aircraft, count (*) FROM Flight F WHERE F.price < 500 GROUP BY aircraft;

Index 3 (index-only plan); or Index 4 (Index-only plan) (Index 4 is accepted only when selectivity of price is explained)
Question 6. PAX Record Format (15 points)

In class we talked about a PAX format to store records within the page as multiple mini-pages corresponding to different attributes. The following diagram illustrates its main idea for a simple table

\[ \text{Emp(id INT, name VARCHAR(20), age DOUBLE).} \]

Notice the diagram is very abstract and conceptual, without giving the byte-level details.

![Diagram of PAX record format]

a) **(3 points)** What information do we need to put into the page header? Why?

- # of slots, which is # of mini-pages (optional);
- The offset and length of each minipage.

b) **(3 points)** For the directory of each mini-page, what optimization can we do to utilize space more efficiently?

For a mini-page of an attribute with a fixed length (e.g., “id” and “age”), there is no need to store the length of each value. We could even get rid of the slots in the directory.

c) **(3 points)** We want to allow the mini-pages to be able to “move” or “shift” within the page so that the space can be used more efficiently. Give a scenario where a mini-page needs to be shifted, and which part(s) of the page structure needs to be changed accordingly.
When we insert a new record, the mini-page for the “name” field is running out of space. At this moment, we can shrink the mini-pages of the other two attributes to allocate more space for this “name” field. Correspondingly, we need to modify the offsets and lengths in the header for these mini-pages.

d) (3 points) Suppose we want to implement this PAX format in our course projects. We would like to keep the same API of the record manager so that other modules are not affected by this change. Consider the following function:

```c
RC insertRecord(FileHandle &fileHandle, const vector<Attribute> &recordDescriptor, const void *data, RID &rid);
```

Notice that each RID still consists of a page number and an offset. Briefly explain how this function is internally implemented using the PAX format.

For each field, use the header of the page to locate the beginning of its mini-page. Go to the directory of the mini-page. Add a field value to the mini-page, and add its offset to its directory. If a mini-page is full, try to shrink other mini-pages to allocate more space for this mini-page.

e) (3 points) Give one query where this PAX format is more efficient than the traditional page format that is implemented in our projects.

```
SELECT age FROM emp;
```
Reason: We only need to access the values from the mini-page for the “age” attribute.