CS222P – Fall 2017, Final Exam
Principles of Data Management
Department of Computer Science, UC Irvine
Prof. Chen Li
(Max. Points: 100 + 15)

Instructions:
● This exam has seven (7) questions and one (1) additional extra-credit question.
● This exam is closed book. However, you can use one cheat sheet (A4 size).
● The total time for the exam is 120 minutes. Budget your time accordingly.
● 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 (reasonable) interpretation you are taking and then answer the question accordingly.

<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>10</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>8 (extra credit)</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>TOTAL</td>
<td>100 + 15</td>
<td></td>
</tr>
</tbody>
</table>
Question 1: Short questions (12 points)

a) (3 pts) Give two reasons why many DBMS systems choose to B+ tree over hashtable.

(1) B+ tree supports both range predicates and point predicates, while hash table doesn’t support range predicates.
(2) The complexity of B+ a tree search is \( \log_{\text{fanout}}(\text{record number}) \), which grows very slowly. Even though it is worse than the \( O(1) \) complexity of a hash table lookup, it’s not much worse.
(3) A scan of a B+ tree can return the records in order by the key.
(4) A B+ tree is balanced, while a hash table may need overflow pages if the values are not well distributed (skewed) or the hash function is not good.

Grading rubric: Any 2 from the 4 is good. 1.5 points for each reason

b) (3 pts) In class we discussed two query execution models: push-based and pull-based. Our course projects used the pull-based model. Briefly explain its idea.

The root of the tree plan calls `open()` then `getNext()` iteratively, which will retrieve information from its descendant operators.

c) (3 pts) Consider the final merge phase of external sort as discussed in class. Suppose the number of runs is \( A \), the number of pages per run is \( B \), and the number of records for each page is \( C \). In class we covered an algorithm to merge the runs in memory using a priority queue (heap). Write a formula for the cost of this algorithm, and briefly explain the answer. (Notice that we are NOT asking for the number of disk IOs.)

\[ 2 \times \log(A) \times (A \times B \times C). \]

Each record needs to be pushed into and popped out of the heap (priority queue), and each push/pop operation has a cost of \( \log(A) \). If we combine the pop of the previous top element and the push of the new element into one step, we may get rid of the “2 *” in the formula.

d) (3 pts) Explain two main reasons why the System-R optimizer only considers left-deep trees.

(1) A left-deep plan has a better chance to allow the output of each operator to be pipelined into the next operator without storing it in a temporary relation.
(2) Reduce search space.
Question 2: B+Tree / Record Manager (10 points)

a) (3 pts) How does key compression benefit B+ tree? Name a main advantage.

Increase fanout.

b) (4 pts) Suppose a non-leaf node of a B+ tree has a key entry “gdb” as shown in the following diagram. Draw an example B+ tree where compressing this entry to “gd” would make a search in this B+ tree fail. You don’t need to draw a complete tree. Just show the part that can illustrate the reason. Briefly explain why this compression would invalidate this B+ tree.

The compression will break the total order of B+ tree.

c) (3 pts) Briefly describe how tombstones are used in the record manager in our projects.

When updating a record, if the record is larger than the original record, we need to leave a tombstone to point to the new location.
Question 3: Sort (16 points)

Suppose we have $N = 8,400$ pages of fixed-length records in a heap file. We have $B = 21$ available pages in memory to sort the file using the external sort algorithm covered in the lectures.

a) (6 pts) For each pass (including pass 0 of generating the initial runs), write down the number of sorted runs, and the size of a single sorted output run.

Pass 0: 400 sorted runs of 21 pages each
Pass 1: 20 sorted runs of 420 pages each
Pass 2: 1 sorted file of 8400 pages

b) (4 pts) Suppose the number of initial runs is $X$. Write down a formula to calculate the number of I/Os required to sort the entire file, excluding the writes in the last pass.

Total number of IO, including pass 0: $((\log_2(X) + 1) \times 2 - 1) \times N$
Written in all symbols: $2N \times (\log_B-1(X) + 1)$

c) (6 pts) In the algorithm of generating the initial runs in pass 0 as covered in our lectures, we implicitly assumed that the records are fixed length, so that we can do in-place swap of two records in memory. Now consider the case where the records are variable-length, which does not support in-place swap since different records have different lengths. Describe a method to sort them in memory efficiently.

Full credit solution 1: Build an array of pointers of the records. For each comparison, compare the original record and swap the pointers instead of the actual records. Finally write the records to disk based on the order of the pointers.

Full credit solution 2: Build an array of pointers of the records plus the prefix of sorting keys. Then sort the arrays using some sort algorithm. If a swap is needed, we swap the two pointers and the prefixes (we cannot swap original records since they have different lengths, but the prefixes have the same length). If comparison is needed, we first compare the prefix of sorting keys (hopefully this can answer most of comparisons), and check the original record if there is a tie (this will require two random memory accesses).
Common partial credit answers:

3 (5 points): Use a temporary output buffer page. For each iteration, find the smallest record and move it to the output buffer. Flush the output buffer to the disk. This is an $O(N^2)$ algorithm and is less efficient compared to potential $O(N \log(N))$ algorithm of solution 1.

4. (5 points) Build an array of pointers and the sorting key (not the prefix). This solution is different from solution 2 because the sorting keys may still be variable length. Therefore the in-place swap problem is still not solved, unless you explicitly mention that you only swap the pointers and you do not swap the keys.

5. (3 points) Insertion sort: For each record, find its position in the all the records. Shift all other records so there’s enough space to put it there. This algorithm needs to do a mem-move of the lots of records for each iteration, which is very inefficient.

Common incorrect answers:
Use a buffer to temporarily store the swapped values.

Only mentioning we can use B-1 pages instead of B pages without giving any details about how to do the sort.

Only mentioning “replacement sort” or “selection sort” (name of solution 3) without giving any details.
Question 4: Join (16 points)
Suppose we have the following two tables.

- **School**(school_id (primary key), school_name, avg_GPA...)
  - 5,000 records, 250 pages;
  - An unclustered B+ tree on “school_id”;

- **Applicant**(name, school_id, GPA...)
  - 40,000 records, 2,000 pages;
  - “School_id” is a foreign key to “School.school_id”;
  - An unclustered B+ tree on “school_id”;
  - The “school_id” values are uniformly distributed.

We want to join them on their “school_id” attributes, i.e., School.school_id = Applicant.school_id.

a) (2 pts) For each of the following join algorithms, circle whether it supports theta (non-equi) join.

1. (Yes / No) Block Nested Loop Join
2. (Yes / No) Index Nested Loop Join
3. (Yes / No) Sort Merge Join
4. (Yes / No) Grace Hash Join

b) (4 pts) Estimate the IO cost of the **Index Nested Loop Join**, by scanning “School” heap file as the outer table and using the B+ tree of Applicant.school_id/. Assume all non-leaf pages of the B+ tree are cached in memory.

Explanation: For the B+ tree search, we only need 1 IO for the leaf page because the non-leaf pages are cached in memory. For Applicant table, school_id is foreign key and uniformly distributed, so each school_id will match 40000/5000 = 8 records. We assume they are in different pages. Therefore for each record in School, we need 1 B+ tree leaf page IO + 8 IOs to fetch the records

\[
250 + 5000 \times (1 + 40000/5000) = 45,250
\]

c) (4 pts) Estimate the IO cost of the **Index Nested Loop Join**, by scanning “Applicant” heap file as the outer table and using the B+ tree of School.school_id.

Explanation: The each record in the Applicant table will only match 1 school_id in School Table. Therefore, each record needs 1 B+ tree leaf page IO + 1 page IO to fetch the record

\[
2000 + 40000 \times 2 = 82,000
\]
d) (6 pts) Consider the case where we want to use **Grace Hash Join** by treating the smaller table as the outer table, with \( B = 11 \) in-memory buffer pages. Calculate the number of disk I/Os in the **partitioning** phase, and the number of disk I/Os in the **building/probing phase**, excluding the final writes.

Need 2 partition passes.
Partitioning: \( 2 \times 2 \times (250 + 2000) = 9000 \)
Building/probing: \( 250 + 2000 = 2250 \)

**Question 5: Rule-Based Query Rewriting (13 points)**

In the System-R query optimizer, suppose we want to implement a technique to push selection down with the hope to make a query plan more efficient. In this question, we want to go through the details of this technique starting with an example.

**a) (3 pts)** Consider a join operator on two tables R and S, and a selection predicate \( a_1 \ op \ a_2 \) where \( a_1 \) is an attribute, and \( a_2 \) is either a constant or an attribute. Discuss when the following push-down is correct \( \sigma_{a_1 \ op \ a_2}(R \bowtie S) = (\sigma_{a_1 \ op \ a_2}(R)) \bowtie S \) when none of \( a_1 \) and \( a_2 \) are from table S.

**b) (5 pts)** Now consider three tables with the following schemas:

\[
R(A, B, C, D), \quad S(D, E, F, G), \quad T(G, H, I, J, K).
\]

We have the following query:

\[
\begin{align*}
&\text{SELECT A, G, SUM(B) as sum} \\
&\text{FROM R, S, T} \\
&\text{WHERE R.D = S.D AND S.G = T.G AND R.A='a' AND R.C=S.E+S.F AND T.I = 'i'} \\
&\text{GROUP BY A, G;}
\end{align*}
\]

Draw a tree diagram to show an unoptimized logical plan for this query, which contains only one selection operator.
c) (5 pts) Transform the logical plan in b) to another logical plan by introducing selection operators as deep as possible, i.e., close to leaf nodes. Show the new logical plan.
Question 6: Cost Estimation (13 points)

Consider a table with following schema:

```
CEO_SALARY(name, salary, company).
```

It has 300 records with the following distribution of the “salary” values of double type.

![Histogram of CEO salaries](histogram.png)

(a) (3 pts) Draw a 4-bucketed equi-width histogram.

```
0-10 10-20 20-30 30-40
100 173 21 6
```

(b) (3 pts) Consider the following query:

```
SELECT * FROM CEO_SALARY WHERE salary >= X;
```

What is the X value that can make 270 as its estimated number of records based on the equi-width histogram in a)?
c) (3 pts) Draw a 3-bucketed equi-height histogram.

```
100  100  100
0-10 10-14 14-40
```

d) (4 pts) What is the estimated number of results from the following query based on the equi-height histogram in c)? You can leave your answer as a formula.

```
SELECT * FROM CEO_SALARY WHERE salary BETWEEN 18 AND 22;
```

\[
\frac{4}{26} \times 100
\]

Question 7: System-R Optimizer (20 points)

Consider the following relations about movies, actors, studios, and their relationships, together with available B+ tree indexes:

- **Movies(mid, title, year, revenue)**
  - Clustered index on “mid” (meaning “movie id”);
  - Unclustered index on “year”
- **Actors(aid, name, gender, date-of-birth)**
  - Clustered index on “aid” (meaning “actor id”);
  - Unclustered index on “name”;
- **ActorInMovies(aid, mid)**
  - Clustered index on “mid”;
  - Unclustered index on “aid”;
- **Studios(sid, name, address, telephone)**
  - Clustered index on “sid” (meaning “studio id”);
  - Unclustered index on “name”;
- **MovieByStudios(mid, sid)**
  - Clustered index on “mid”;
  - Unclustered index on “sid”.

The meanings of the tables and attributes are self explanatory. Consider the following query:

```
SELECT aid, SUM(M.revenue)
FROM Movies M, ActorInMovies AM, Actors A, MovieByStudios MS, Studios S
WHERE M.mid=AM.mid AND AM.aid=A.aid AND M.mid=MS.mid AND MS.sid=S.sid
AND M.year='2016' AND S.name='Disney'
```
We want to use the techniques in the System-R optimizer to generate an efficient physical plan.

a) (2 pts) Write the meaning of the query in plain English.
For each actor, for all his/her movies in 2016 and made by Disney, compute their total revenue.

b) (6 pts) For each of the following tables, write down all the access methods considered by the optimizer. Specify which of them will be considered for the next phase and explain why. Write down all the available interesting orders for each relation.

- Movies
  1) B+ tree scan on mid, interesting order on pid; kept;
  2) B+ tree search on year for the condition “year = 2016”; kept if its cost is less than the access method 1) and 4).
  3) B+ tree search on mid for a constant, kept for a later index-based join
  4) Full scan on table; kept if its cost is less than 1) and 3)

- Actors
  1) B+ tree scan on aid, interesting order on aid; kept;
  2) B+ tree search on aid for a constant, kept for a later index-based join

c) (4 pts) Explain how the optimizer generates efficient access methods for joining the tables Movies and ActorInMovies. No need to show all the enumerations.

  1) Movies join ActorInMovies: For each access method on Movies kept from the previous step, for each access method on ActorInMovies kept from the previous step, consider all possible valid join methods: block nested loop join, index-based join, sort merge join, hash join, etc. Estimate the cost of each subplan. Use the interesting order from the previous method, if any, when estimating the cost of a sort-merge join. For each interesting order, select the join method with the smallest cost, and remove those subplans that are dominated by another subplan in terms of both cost and interesting order(s).
  2) Repeat the same step for ActorInMovies join Movies;

d) (4 pts) Write down all possible 3-relation subsets considered by the optimizer by combining the subplans from previous passes. The join order of each subset doesn’t matter.

  1) (A, AM, M)
  2) (AM, M, MS)
  3) (M, MS, S)
Notice that AM and MS can still join their “nid” attributes. If your answer assumes AM and MS do not join when M is not included, we can also accept an answer that does not include “4)” and “5”).

The System-R optimizer does not consider cross products. So a subset that produces a cross product is wrong.

e) (4 pts) Use this example to briefly explain at least four main ideas in the System-R optimizer as covered in our lectures.

- Interesting orders
- Left-deep trees only
- Avoid Cartesian products
- Dynamic programming to do plan enumeration
- Selection push down
- Deal with group by at the end
- Deal with nested subqueries as blocks, and optimize them separately (not shown in the example)

Question 8: Extra Credit Question (15 points)

Suppose you’re a DBA for an e-commerce company. Suppose your company has two separate database systems, one for online transaction processing (OLTP), and another one for online analytical processing (OLAP). You’ve set up an incremental migration utility that moves new data from the OLTP system to the OLAP system on a daily basis. Suppose the OLAP system already has a B+ tree with 1,000,000 leaf-node pages, each of which can hold 1,000 records.

a) (5 pts) Suppose we want to insert a large number (5 million) records into the B+ tree. A simple method (“method 1”) is to insert them to the index one by one, but this method is not
efficient. We want to use another method ("method 2") based on bulk loading covered in class. A main problem is that bulk loading assumes an empty B+ tree.

Come up with this "method 2" by extending the bulk loading idea, and briefly explain why it’s more efficient than “method 1”.

Sort the new records. Perform a full scan over the original B+ tree. Merge the results of sorted new records with the results from scanning B+ tree to bulk load a new B+ tree.

b) (5 pts) Method 2 still does not solve all your problems. After the system is running for a few months, it starts taking longer and longer time to migrate daily data. There is another method that stores new records of each day into a separate B+ tree, which only takes minutes to finish! In this way, you get more and more B+ trees over time.

Discuss how to deal with “deleteRecord()” in this design without modifying the existing B+ trees.
Use a tombstone key to indicate that a key has been deleted. During the query time, if we found a tombstone key, we’ll realize it has been deleted.
Alternatively, you can store deleted keys into a separate index structure, e.g., a separate small B+ tree.

c) (5 pts) Continuing question b), present how to do a search on the list of B+ trees to find records within a range such as “20 <= price <= 56”.
Two approaches can be used.
1) Search multiple B+ trees at the same time. Use a priority queue to merge the search results from the B+ trees. Specifically, for each primary key, we only output the record once from the newest B+ tree. In case of the tombstone record, we do not output it.
2) Searching B+ trees from newest to oldest one by one. During search, maintain a hash set to store encountered primary keys. Only output a key when it’s not found in the set.

No matter which approach you’re using, you need to be careful about two things:
1. a key can be deleted
2. a key can be updated (it can appear multiple times in B+ trees)