CS222/CS122C – Principles of Data Management
Fall 2012, Department of Computer Science, UC Irvine
Final Exam

Max. Points: 100

Read the following instructions carefully.

- Total time for the exam is **120** minutes. No extra time will be awarded, so budget your time accordingly. The exam is closed books and closed notes.
- Please be succinct in your answers.
- If you find ambiguities in a question or in the schema, write down the interpretation you are taking, and then answer the question based on your interpretation.
- This exam contains **15** pages. You can use the last page as scratch paper.

NAME: **Student ID:**

Part	**Points**	**Score**
1	**15**	
2	**10**	
3	**15**	
4	**20**	
5	**20**	
6	**20**	
Total	**100**	

Problem 1: Short questions (3 points each)

1a) Briefly explain the format of a record id in a relational DBMS.

A record id is composed by a page id and a slot id.

1b) Briefly explain why we need to keep sibling pointers in the leaf nodes of a B+ tree.

It chains leaf nodes so as to speedup range search, as well as insertion/deletion.

1c) Briefly explain the benefits of B+ tree compared to hash tables.

B+ tree supports range search and equality search while hash tables only supports equality search.
B+tree is balanced while hash table has overflow pages.

1d) Briefly explain the benefits of linear hashing compared to extensible hashing.
Linear hashing grows slower than external hashing;
Linear hashing does not have a directory.

1e) Briefly explain why it is often better to push a selection condition down in a plan tree.
It will reduce the volume of intermediate data so as to speedup query execution.

Problem 2 (10 points): Histograms

Suppose we have a table Emp(eid, salary, ...) with 1 million records. We want to build a histogram for the Emp.salary attributes. Suppose we know the min and max values of the salary attribute, and the number of buckets in the histogram.

2a) (4 points) We want to build an equi-width histogram. Describe how to build the histogram by scanning the table once.

We can get buckets with max, min and #buckets.
We just need one scan to count the number of records in each bucket.

2b) (6 points) We want to build an equi-height histogram. Can you build the histogram by scanning the table once? If so, explain how. If not, describe an algorithm for building the histogram while minimizing the number of scans on the table.

No, one scan is not enough.

We can sort the data first, and then iterate the sorted data once to build the equi-height histogram since we already know the average number of tuples per bucket.

Problem 2 (15 points): Linear Hashing

Consider the Linear Hashing index shown in Figure 1. Assume the split policy is that we split one bucket and increment the “next” value whenever an overflow page is created. Answer the following questions about the index.

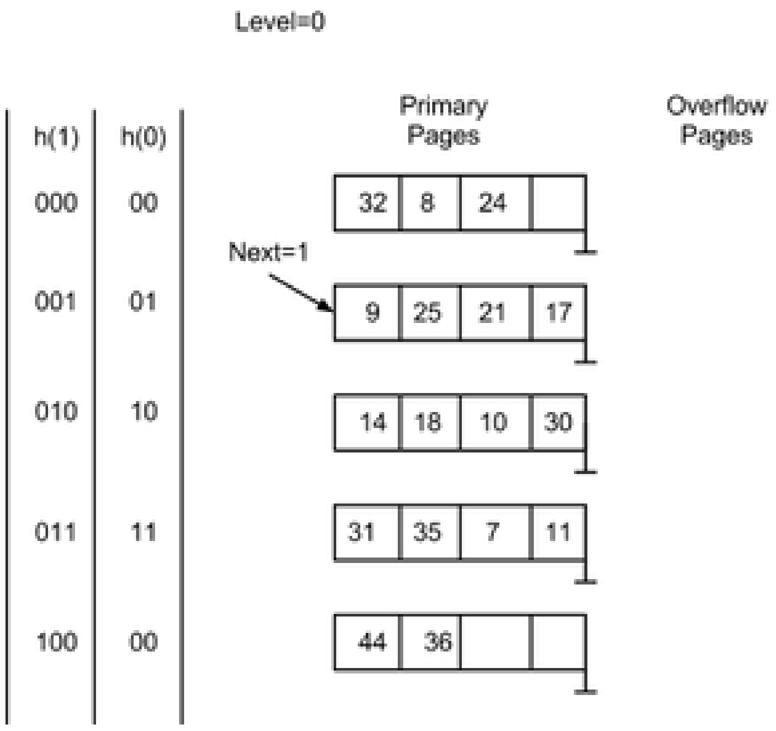
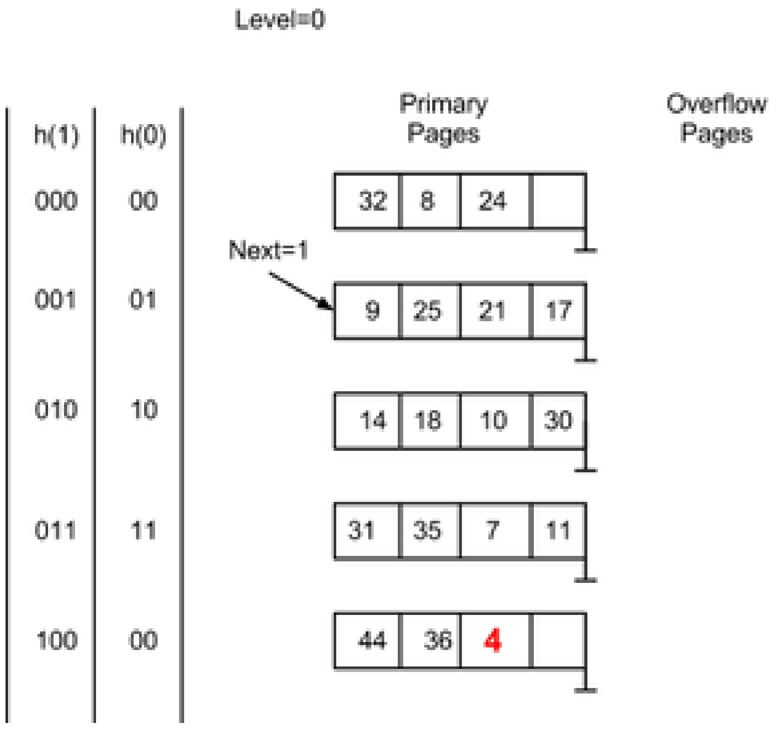


Figure 1

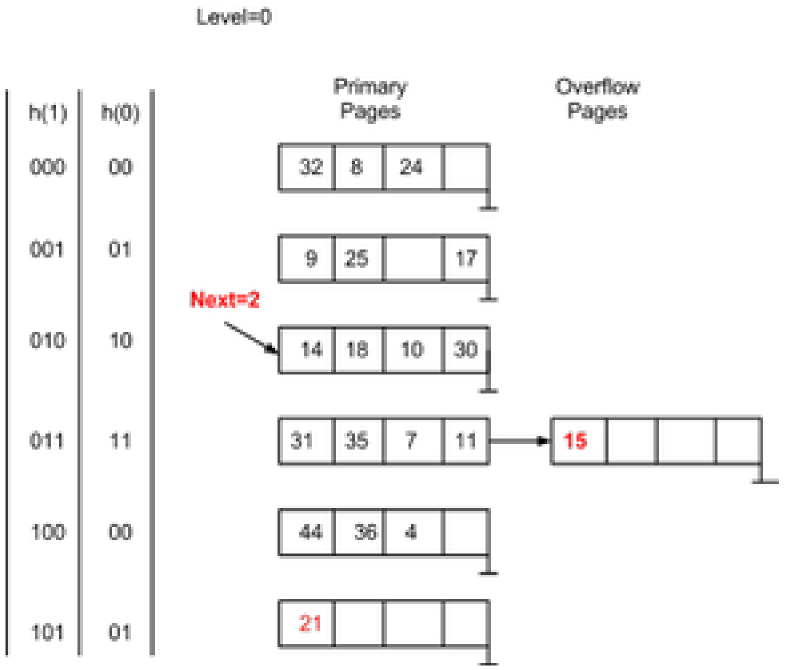
(3a) (**5 points**) Suppose you know that there has been no deletion from this index so far. Consider the last entry whose insertion into the index caused a split. What is a possible hash value of the key of this entry? Briefly explain your answer.

Since the “next” value is 1, and the split was triggered by an overflow page, so the previous “next” value was 0. The last bucket (with the values 44 and 36) should be the new bucket created. Its two values should come from the first bucket. It means before the split, the first bucket had 4 entries, while the insertion resulted in an overflow bucket, causing the split. Therefore, the hash value of the new entry must be one of the five values: 32, 8, 24, 44, and 36.

(3b) (**5 points**) Show the new index after inserting an entry (into the index in Figure 1) with a hash value 4.



(3c) (**5 points**) Show the new index after inserting an entry (into the original index in Figure 1) with a hash value 15.



Problem 4 (20 points): Query Evaluation - External Sort

Suppose you have a table *R* with |R| pages and you want to sort the table by a multi-way external merge sort algorithm which uses the quick sort algorithm for in-memory sorting. In all the following calculations, do **NOT** count the I/Os to write the **final** results.

Important: Please show your steps so that you are able to get partial scores.

(4a) (**8 points**) If you want to sort *R* within only **one** merge pass, what is the minimum number of pages in memory needed for the sort algorithm? Briefly explain your answers using a diagram to show your analysis.

Assume we have M in-memory pages. In the first phase, we generate |R|/M sorted run. In the merge phase, we need to have a page for each run, and a page for the output. So we have: |R|/M + 1 <= M. Therefore, we have: |R| <= M(M-1) < M^2. Hence, M > $\sqrt{|R|}$.

(4b) (**7 points**) If you want to sort *R* within **two** merge passes, what is the minimum number of memory buffer pages needed for the sort algorithm? Briefly explain your answers using a diagram to show your analysis.

In the first phase, we have |R|/M sorted run.

In the second phase, we merge every M-1 runs into one sorted run of M*(M-1) pages. We have |R|/(M*(M-1)) runs.

In the third phase, for each run, we keep a page in memory, and one page for the output. So |R|/(M*(M-1)) + 1 <= M.

Therefore, M >$\sqrt[3]{|R|}$**.**

(4c) (**5 points**) If you want to sort *R* within ***k*** merge passes, what is the minimum number of memory buffer pages needed for the sort algorithm? Briefly explain your answers using a diagram to show your analysis.

M > $\sqrt[k+1]{|R|}$**.**

Problem 5 (20 points): Query Evaluation - Join
Suppose you have a table *R* with |R| pages and a table *S* with |S| pages, and **4 < |R| < |S|**. We want to join the relations, and neither *R* nor *S* is ordered by the join key.

In this question, we have the following **assumptions**:

- The join keys in *R* and *S* are uniformly distributed and the hash function used by hash join can obtain a uniform hash value distribution.
- For simplicity, in hash join, we assume the fudge factor of hash table is 1.0.
- The GRACE hash join will recursively partition the input branch for "join build" until the hash table of single partition can fit in memory to be used by the in memory hash join algorithm.
- Hash join can automatically use the smaller input branch for "join build", while the nested loop join can automatically use the smaller input branch as the outer loop data.

In all the following calculations, do **NOT** count the I/Os to write the **final** results.

Important: Please show your steps so that you are able to get partial scores.

(5a) (**8 points**) Assuming there are in total four (4) memory buffer pages, how many disk I/Os are needed to join *R* and *S* by the **block nested-loop join** algorithm?

We use one page for reading an S page, one page for writing join results, and the remaining two pages to read R pages. So the number of iterations is |R|/2. The total number of disk IOs is |R| + (|R|/2) * |S|.

(5b) (**7 points**) If you want join *R* and *S* by the **GRACE hash join** with **only one partitioning pass**, what is the minimum number of memory pages needed for the join algorithm? Briefly explain your answers using a diagram to show your analysis.

|R|/ (M-1) + 2 <= M.

Thus, M at least has to be M > 1 + $\sqrt{|R|}$

(5c) (**5 points**) Assuming there are four (4) memory buffer pages, we want to join *R* and *S* by the **GRACE hash join** algorithm using multiple recursive partitioning passes. How many I/Os are needed?

Suppose we need k partitioning pass, we use one page as the input and three pages for three output partitions. So after k passes, the size of each R partition is |R|/3^k, which should be within 2 for the final build-probe phase. In the last phase, we need 2 pages for an R partition, one page for reading an S partition, and one page for the join output. So we have: |R|/3^k <= 2. The number of iterations is: $\log_3(|R|/2)$.

The total number of disk IOs is: (k * 2 + 1) * (|R| + |S|).

Problem 6 (20 points): Query Optimization

Consider the following relational schema and SQL query. The schema captures information about employees, departments, and company finances (organized on a per department basis).

> Emp(<u>eid: integer</u>, did: integer, sal: integer, hobby: char(20))
> Dept(<u>did: integer</u>, dname: char(20), floor: integer, phone: char(10))
> Finance(<u>did: integer</u>, budget: real, sales: real, expenses: real)

Consider the following query:

```
SELECT    D.dname, F.budget
FROM      EMP E, Dept D, Finance F
WHERE     E.did = D.did AND D.did = F.did AND D.floor = 1
          AND E.sal >= 59000 AND E.hobby = 'biking';
```

(6a) (**4 points**) Assume that the optimizer follows the heuristic of never considering plans that require the computation of cross-products. In addition, it only considers left-deep tree plans. List all the possible join orders (i.e., orders in which pairs of relations can be joined to compute the query result) that the optimizer will consider. Briefly explain your answer.

Since every two relations are can be joined without generating a cross product, we have 3! orders, such as
(E JOIN D) JOIN F
(E JOIN F) JOIN D
(D JOIN E) JOIN F
(D JOIN F) JOIN E
(F JOIN D) JOIN E
(F JOIN E) JOIN D.

Suppose that the following additional information is available: **unclusterted** B+ tree indexes exist on Emp.did, Emp.sal, Dept.floor, Dept.did and Finance.did. The system's statistics indicate that employee salaries range from 10,000 to 60,000, employees enjoy 200 different hobbies, and the company owns 2 floors in the building. There are a total of 50,000 employees and 5,000 departments (each with corresponding financial information) in the database. Some new hired employees may not have a department yet.

The DBMS used by the company has two join methods available: **merge join** and **hybrid hash join**. Note that the merge-join operator always assumes the inputs are ordered by the join key, and only does the join by merging two input branches. If an input branch to the merge-join operator is not sorted, one can place an extra **sort** operator on that input branch, right before the join operator.

We also have the following assumptions:

- The page size is 4KB.
- The DBMS has 10 memory buffer pages.
- Integer occupies 4 bytes and real occupies 4 bytes.
- The data (Emp, Dept, Finance) have uniform distributions on each attributes.
- The distributions of different attributes on the same relation are independent.
- The fudge factor of the hash table in hash join is 1.0.
- The R_0 in hybrid hash join can use **up to** 5 memory pages.

(6b) (**4 points**) For each of two base relations Emp and Dept estimate the number of tuples that would be initially selected from that relation if all of the non-join predicates on that relation were applied to it before any join processing begins.

Tuples from E: (1/50) * (1/200) * 50,000 = 5 tuples
Tuples from D: (1/2) * 5000 = 2500 tuples

(6c) (**4 points**) A “***data order”*** means on which attribute(s) the data is sorted, if the data is sorted. For example, an index scan on the index of Emp.did will produce the data ordered by did.

For the base relations, we want to know all possible access paths and show the **data order** (if there is any) that is delivered by the access path. The following are examples for the Finance table.

- Finance: access by the did index, order by did
- Finance: table scan, no order.

Give the answers for the other two relations.

E: access by did index -- order by did
E: access by sal index -- order by sal
E: table scan -- no order

D: access by did index -- order by did
D: access by floor index -- order by floor
D: table scan -- no order

(6d) (**8 points**) We want to produce an efficient physical plan for this query the lowest estimated cost (I/O cost + CPU cost). Make sure that you read those assumptions we described earlier. Notice that you cannot use nest-loop join methods. Draw the physical plan as an iterator tree, which includes the following:

r1. For each join operator, please specify which join algorithm (merge join or hybrid hash join) is used, as well as the join condition.
r2. If you use hybrid hash join in the plan, please annotate which input branch is for "join build" and which input branch is for "join probe".
r3. For each filter operator, please specify the condition(s).
r4. For each project operator, please specify the attribute(s).
r5. For each index scan operator, please specify the index and condition(s).
r6. For each table scan operator, please specify the table name.
r7. For each sort operator, please specify the sort keys.

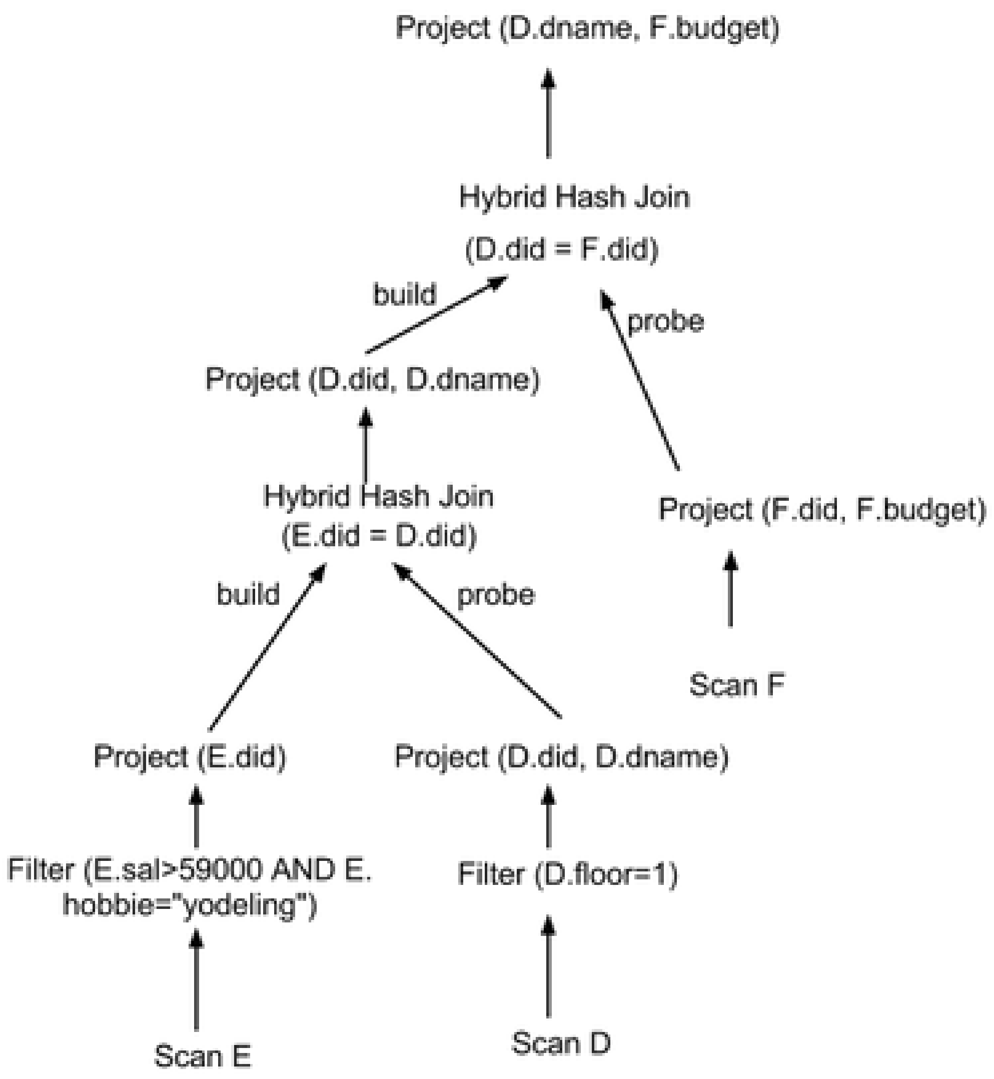
Your grade will be based on the expected efficiency of your plan and the details provided in the plan. Please show your reasoning steps so that you are able to get partial scores.

Size of table E: 50000 * 32 = 1600 KB = 400 pages
Size of table D: 5000 * 38 = 190 KB ~ 48 pages
Size of table F: 5000 * 16 = 80 KB ~ 20 pages

After filter and projection, input from E has 5 tuples (4 bytes each)~ 20 bytes.
After filter and projection, input from D has 2500 tuples (24 bytes each) ~ 15 pages
After projection, input from F has 2500 tuples (8 bytes each) ~ 5 pages

Reasoning:
1. None of the index access path can be adopted, since the indices are unclusterd, and will lead to a large number of page accesses (#page access ~ #tuples satisfying the condition on the index key).
2. If the memory size is enough, hybrid hash join becomes in memory hash join.
3. So, we first join E and D using hybrid hash join. It is better than using sort and merge join between E and F first. Because sorting incurs more CPU cost.

The optimal plan is:


Project (D.dname, F.budget)
Hybrid Hash Join
(D.did = F.did)
build
probe
Project (D.did, D.dname)
Hybrid Hash Join
(E.did = D.did)
build
probe
Project (F.did, F.budget)
Scan F
Project (E.did)
Project (D.did, D.dname)
Filter (E.sal>59000 AND E. hobbie="yodeling")
Filter (D.floor=1)
Scan E
Scan D