COSC 460 Databases
Midterm Exam 2 (Practice Version)
Fall 2018

This is was intended to be a practice exam... but it ended up being more of a study guide with a potpourri of questions. Nevertheless, you can expect the format of the second midterm to be similar to the format of the first (multiple choice, short answer, longer problems).

It is **different** from the real exam in several respects:

- The exam may have different questions and different parts per question. Nevertheless, this practice exam covers many of the topics you will see on the exam.
- There's no space to write your answers... the real exam will have space.
- I didn't assign points to questions (since each question has extra parts, the point totals wouldn't be accurate anyway).

Be sure to also review the recent homeworks. Similar questions may show up on the exam!

Question	Points	Score
1	0	
2	0	
3	0	
4	0	
5	0	
6	0	
7	0	
Total:	0	

1. Circle your answer.

   (a) Be sure to review questions from first practice exam, especially those about file organization and relational algebra.

   (b) With extendible hashing, overflow is impossible.

   ◯ True [✓ **False**]

   (c) The leaves of a B+tree are arranged sequentially on disk.

   ◯ True [✓ **False**]

   (d) Given a B+tree index on the Emp table with composite search key $\langle age, dno \rangle$, it is possible to evaluate the following query using only the index:

```
SELECT   E.dno, COUNT (*)
FROM     Emp E
WHERE    E.age=30
GROUP BY E.dno
```

   [✓ **True**] ◯ False

   (e) Suppose you have a unclustered hash index on the Emp table with composite search key $\langle age, dno \rangle$. Consider the following query

```
SELECT   *
FROM     Emp E
WHERE    E.age=30
```

   It would be more efficient to answer this query using the index than simply doing a file scan. ◯ True [✓ **False**]

   (f) To support range queries on two different search keys, the recommended practice is to construct a clustered index (data entry alternative 1) for each key.

   ◯ True [✓ **False**]

   (g) Circle the query descriptions below for which using a *clustered* B+tree index instead of an *unclustered* B+tree index would noticeably improve performance.

   A. A range query over a reasonably large range (e.g. $3.6 < Student.gpa < 4.0$**)**

   B. An equality selection on a key field (e.g. $Student.sid = 1001$)

   C. An equality selection on a field with many duplicate values (e.g. $Apply.major =' Econ'$**)**

2. For each question, provide a short (2-3 sentences) answer.

   (a) Suppose that you are building a DBMS and want to add a new aggregate operator called TOPK, which is a variation of the MAX operator, and it returns the k largest items where k is an input given "at runtime." Explain how you would implement this operator as efficiently as possible, assuming that no indexes are defined over the input relation. Your answer may depend on k (i.e., you may do different things depending the value of k). Describe the cost in terms of a relation with N pages and p tuples per page and M frames in buffer pool.

   Solution: Option 1: sort in descending order, then go to start of file, and iterate until k items. Cost is cost of sorting plus cost of partial scan (which is at most N pages).

   Option 2: scan once and keep top k in memory. Only an option if k is small and the corresponding tuples can fit in available memory (say, $k \leq p \cdot M$). Cost is N, much less than option 1.

   (b) Consider the same scenario as Part a but now assume you can create an unclustered B+Tree index. Explain how you would implement this operator.

   Solution: If B+Tree leaves form a doubly-linked list, then you could search for max value and iterate backwards over leaves until you've seen k tuples. Cost is cost of search plus roughly k additional page reads (because unclustered).

   If B+Tree leaves are a singly-linked list, then we need to insert records such that index stores them in descending order. (In Java, you would supply a Comparator that would be used by the B+Tree to sort them.)

   (c) Why are sequential page reads faster than random page reads?

   Solution: Seek and rotational delays are minimized. See early sections of Ch. 9 of Cow book.

   (d) What is pipelining and why is it important?

   Solution: p. 407 of Cow book.

   (e) Give a concrete example of a relational algebra query plan where pipelining results lowers overall cost. Briefly explain your answer.

   Solution: Materialization is when the output of each operator is produced in full into a temporary relation.

Pipelining is when operators are chained together and the output of one operator feeds directly into the next operator.

Consider this query: $\pi_{sid}(\sigma_{major='apply'}(A \bowtie S)$. With materialization, two temporary relations are produced. With pipelining, the tuples in the join feed directly into the select which feeds directly into the project – minimizing costly I/O of writing out temporary relations and reading them back in.

(f) Give a concrete example of a relational algebra query plan where materializing some results may lower overall cost. Briefly explain your answer.

Solution: Consider this query: $(R \bowtie S) \bowtie \sigma_P(T)$. Suppose we only have nested loops join. With materialization of the result of the select on T, the result might be really small and reduce the cost of scanning T repeatedly.

3. Hashing

   (a) Given an extendible hash index, if you insert a record and then delete it, the resulting index will be identical in structure and content as the index prior to the insertion.

   [✓ **True**] ◯ False

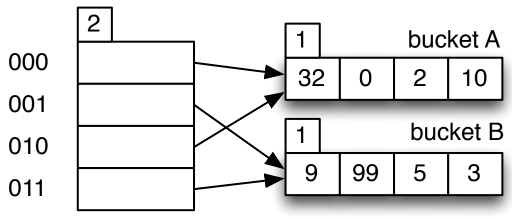
   (b) What are the advantages of an extendible hash index over a static hash index?

   Solution: See p. 373 of Cow book.

   Static requires choosing number of buckets. Too few: lots of overflow; too many: wasted space.

   Static hashing could be modified to periodically double number of buckets. This requires rehashing *all* of the data. With extendible, only the overflowing bucket needs to be rehashed.

   (c) Consider the following dynamic hash index. Suppose that the bucket capacity is 4 and the hashcode is $h(k) = k$.



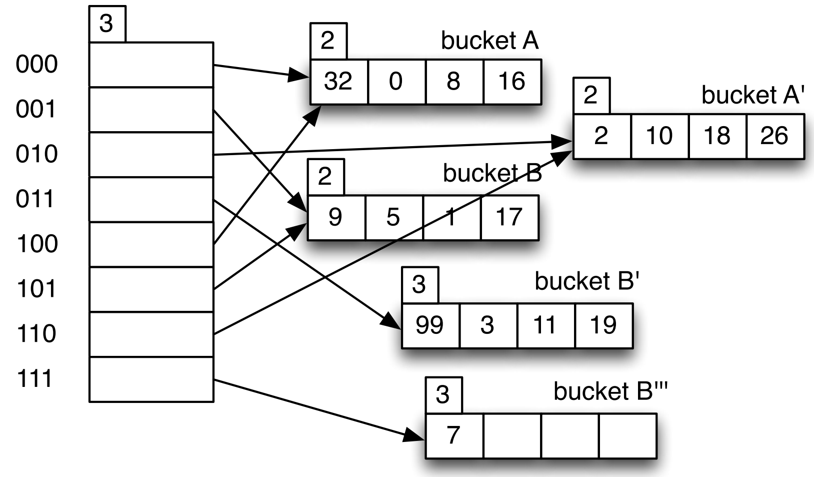
   What is the *maximum* number of entry insertions that will cause the directory in this index to double? Give one such possible sequence of insertions of maximum length.

   Solution: Insert 8, 16, 18, 26 to split first bucket and fill each new bucket.

   Insert 1, 17, 11, 19 to split second bucket and fill each new bucket.

   Insert anything else and one of the buckets will overflow and directory will need to be doubled. Suppose we insert 7.

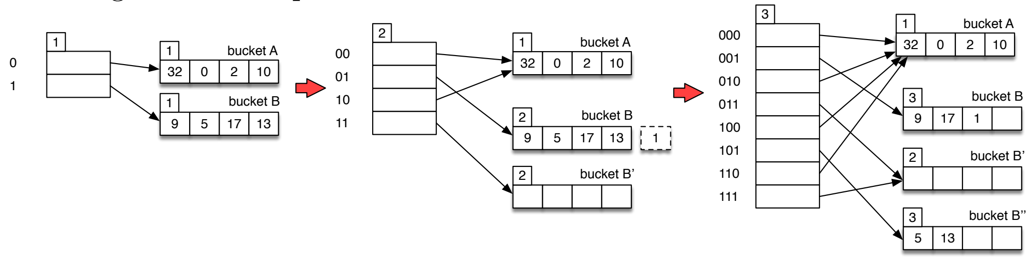
   (d) Show the resulting hash index.



Solution:

(e) Assume that when a bucket overflows, it continues to be split until it is no longer overflowing. Draw an extendible hashing index such that insertion of a data entry with hashed key value of 1 causes the directory to double *twice*. Show the local depth of each bucket and the global depth of the directory in your sketch.

Solution: All values in Bucket B and 1 have the same value for the two least significant bits: 01. Thus, doubling once won't eliminate overflow. Directory is doubled again. Now depth is 3 and the values differ in their last 3 bits.



4. SQL

Consider the following relational schema about a social network centered around college students and alumni. People can be friends, post messages, and like each other's messages. The year attribute in Users refers to the year of graduation.

```
Users(uid, name, school, year)
Friends(uid1, uid2)
Posts(pid, uid, text, datetime)
Likes(uid, pid, datetime)
```

If two users with ids 123 and 456 are friends, then (123, 456) and (456, 123) will both appear in the Friends relation.

For each of the following, write a **SQL query** that produces the desired information.

(a) For each post *author*, the name of the author along with the total number of likes they have received *across all of their posts*. Name this attribute `totalLikes`.

Solution:

```
select U.name, count(*) as totalLikes
from Users U, Posts P, Likes L
where U.uid = P.uid and P.pid = L.pid
group by U.uid, U.name
```

(b) For each post *author*, the maximum number of likes they have received *across all of their posts*. Name this attribute `maxLikes`.

Solution:

```
select U.name, max(likeCount) as maxLikes
from Users U, Posts P,
(select P.pid, count(*) as likeCount
from Likes L
group by P.pid) as X
where U.uid = P.uid and P.pid = X.pid
group by U.uid, U.name
```

(c) Find schools with at least 10 students. Avoid duplicates.

Solution:

```
SELECT school
FROM Users
GROUP BY school
```

```
HAVING count(*) >= 10;
```

(d) Find names of users that have no friends. Avoid duplicates.

Solution:

```
SELECT name FROM Users
where uid not in (select uid from Friends)
```

(e) Write a query that returns persons who have more distinct likers than friends. Alice is a "liker" of Bob if she likes one of Bob's posts.

Solution:

```
select A.uid
from
(select P.uid, count(distinct L.uid) as likers
from P, L where
P.pid = L.pid
group by P.uid) A,
(select uid1 as uid, count(distinct uid2) as friends
from F group by uid1) B
where A.uid = B.uid and A.likers > B.friends
```

(f) Write a query that returns the Colgate graduating class that has the most friends.

Solution:

```
WITH R AS
(select year, count(uid2) as friends
from F, U
where U.uid = F.uid1 AND U.school = "Colgate"
group by year)
select year
from R
where friends = (select max(friends) from R)
```

(g) Write a query that returns the names of "Colgate enemies": people who went to Cornell and have no Colgate friends (hint: they are not contained in set of friends of Colgate people).

Solution:

```
select name
from U
where school = "Cornell" and
uid not in (select uid2
            from U as U2, F
            where U2.school = "Colgate" and
            U2.uid = F.uid1)
```

(h) Write a query that returns the names of "Colgate friendlies": people who did not go to Colgate but have at least two friends that did.

Solution:

```
select name
from U
where school != "Colgate" and
2 <= (select count(*)
      from F, U2
      where F.uid1 = U.uid and
      F.uid2 = U2.uid and
      U2.school = "Colgate")
```

(i) Write a query that returns the names of friends of President Casey's friends.

Solution:

```
select distinct U2.name
from U as U1, F as F1, F as F2, U as U2
where
U1.name = "'GatePrez" AND
U1.uid = F1.uid1 AND
F1.uid2 = F2.uid1 AND
F2.uid2 = U2.uid
```

5. Comparing access methods. In this course you have learned about a number of structures for storing data: heapfiles, sorted files, clustered files, B-trees, dynamic hash indexes, static hash indexes, etc. No single structure is universally better than others: each structure is good for some access patterns and not so good for others. In the following, you are asked to compare two structures and describe a scenario in which one structure would be preferred to the other.

   For example: if asked when a heapfile would be preferred to a B+-tree, you might say that a heapfile would be preferred if the relation changes a lot (many inserts/deletes) but is not queried often (few searches).

   (a) You should be able to derive any of the expressions that characterize the the cost of various operations on the different types of files and indexes. (See the summary slide in lecture 9.)

   (b) Suppose you are interested in supporting range queries on a particular attribute. Describe a scenario in which a sequential scan on a heapfile could be faster than a search on an unclustered B+-tree.

   Solution: Look at performance analysis document on moodle. When b-tree unclustered, must pay a disk I/O *per record*. (You should know why that is.) If a large number of tuples fall within the range, this gets very expensive. It would be better to sequentially scan and pay one disk I/O per *page* of data.

   (c) When would a dynamic hash be preferred to a B+-tree?

   Solution: When you only need to support equality searches and you have a good hash function. Hashing is fast!

   (d) Suppose you want to support fast range queries on a particular attribute of relation R. In addition, R is *static* meaning that it does not change (no insertions or deletions). Explain why a sorted file might be preferred to an unclustered B+-tree.

   Solution: Similar idea as the answer for (b). Unclustered requires paying per matching record. With sorted file, matching records are all clustered into consecutive pages.

   (e) When would a static hash index be preferred to a dynamic hash index?

   Solution: When input is not dynamic... i.e., size of relation is known and does not change. We avoid the overhead of the directory table.

6. New data structures

(a) Suppose you want to efficiently support queries of the following form:

```
select sum(A) from R where v1 < R.B and R.B < v2
```

where $v1$ and $v2$ are values supplied at runtime – i.e., the same query is invoked repeatedly with different values for $v1$ and $v2$. Describe how one could store extra information in the internal nodes of a B+-tree index on $R.B$ to efficiently compute the above sum.

Solution: For each index entry, store sum of R.A values for tuples in that subtree. Requires indexing down to leaf for $v1$ and leaf for $v2$ but no need to scan all records in between... can use the sums stored in the trees instead!

(b) Our description of static hashing implicitly assumed that there was a way to immediately retrieve the first page of bucket i for any $i = 0 \dots N-1$. But how would this actually happen in practice? Briefly sketch how to implement a static hash table. You can assume that you can ask the disk manager for a contiguous sequence of up to C pages but your solution must work even if the number of pages you need exceeds C. Access to a block should still be efficient.

Solution: Use the directory idea from dynamic hashing. Create $M = \lceil N/C \rceil$ separate hash files each of size C. Then you need a directory containing M pointers to the M hash files. To search, first do $h_1(k) = \mathit{hashcode}(k) \mod M$ to find which entry in directory, then do $h_2(k) = \mathit{hashcode}_2(k) \mod C$ to get page in that directory.

7. Assume you have two relations R and S. Let N_R and N_S denote the number of pages in R and S respectively. Similarly, n_R and n_S denote the respective number of tuples. Let M denote the number of buffer pages.

(a) For each of the following join algorithms, give the most general expression for the cost of performing $R \bowtie S$ in terms of the variables denoted above. If the cost is based on any assumptions, state those explicitly.

i. Nested loops

ii. Block nested loops

iii. Index nested loops

iv. Hash join

v. Sort-merge join

Solution:

- Nested loops: $N_R + n_R * N_S$

- Page nested loops: $N_R + N_R * N_S$
- Index nested loops: $N_R + n_R * c$ where c is cost of probing index (use 1.2 for hash, less than 4 for B+-tree).
- Hash join: $3 * (N_R + N_S)$ assumes that hash is good and partition file of *smaller* relation can fit in memory. More general: $(1+2*rounds)*(N_R + N_S)$ where $rounds = \lceil \log_{M-1} N_R - 1 \rceil$ is the number of rounds of recursive hashing (see textbook, Ch. 12.5).
- Sort-merge join $sortCost(R)+sortCost(S)+N_R+N_S$ where $sortCost(R) = 2 * N_R * (1 + \lceil \log_{M-1} \lceil N_R/M \rceil \rceil)$. Assumes that groups in merge phase can fit in memory.

(b) Suppose R is much larger than S. How many buffer pages do you need to perform the sort-merge join with a cost of $3*(N_R+N_S))$? Give an expression for M in terms of the variables above. Explain your answer, including any assumptions made.

Solution: Merge takes a single pass, assuming the merge groups fit on a page. Must sort each relation in 2 passes. Since R is larger, need $M \leq M(M-1) \approx \sqrt{N_R}$.

(c) Describe an algorithm for eliminating duplicates from R.

(d) Describe an algorithm for performing $R \cup S$.

(e) Describe an algorithm for performing $R \cap S$.

(f) Describe an algorithm for performing $R - S$.

(g) Reconsider parts (d)-(f) under bag semantics (i.e., duplicates permitted). For instance $R - S$ under bag semantics is as follows: suppose R has x copies of some tuple t and S has y copies the relation $R - S$ has $\max(0, x - y)$ copies of t in it.

Solution: See the textbook for solutions to these extra problems. Bag semantics require only slight adaptation to the textbook descriptions.