University of California, Berkeley
College of Engineering
Computer Science Division – EECS

Spring 2006 Prof. Michael J. Franklin

MIDTERM II

CS 186 Introduction to Database Systems

NAME:___________________________ STUDENT ID:_________________

IMPORTANT: **Circle the last two letters of your class account:**

cs186 a b c d e f g h i j k l m n o p q r s t u v w x y z
a b c d e f g h i j k l m n o p q r s t u v w x y z

DISCUSSION SECTION DAY & TIME:_____________ **TA NAME:** ____________

This is a **closed book** examination – but you are allowed one 8.5" x 11" sheet of notes (double sided). You should answer as many questions as possible. Partial credit will be given where appropriate. There are 100 points in all. You should read **all** of the questions before starting the exam, as some of the questions are substantially more time-consuming than others.

Write all of your answers directly on this paper. **Be sure to clearly indicate your final answer** for each question. Also, be sure to state any assumptions that you are making in your answers.

GOOD LUCK!!!

Problem	Possible	Score
1. SQL	40	
2. Hash-based Operators	15	
3. Query Optimization	25	
4. More Query Optimization	20	
TOTAL	100	

Question 1 –SQL [6 parts, 40 points total]

Consider the following basketball schema with four relations (primary keys are underlined):

`Player` `(pid: integer, pname: varchar(50), team: varchar(30))`

Each player plays for only one team. The team field is a foreign key to Team.

`Team` `(tname: varchar(30), city: varchar(20))`

There can be multiple teams from the same city.

`Game` `(gameid: integer, homeTeam: varchar(30), awayTeam: varchar(30), homeScore: integer, awayScore: integer)`

homeTeam and awayTeam are foreign keys to Team. Two teams may play each other multiple times each season. The homeTeam and awayTeam are always different. Ties are not allowed.

`Stats` `(pid: integer, gameid: integer, points: integer, assists: integer, rebounds: integer)`

Stats records the performance statistics of a player within a game. If a player does not play in a particular game, they will not have any statistics recorded for that game. gameid is a foreign key to Games. pid is a foreign key to Player.

Write **SQL** queries for the following. **All of these are doable without creating views or temporary tables. You may do so if you feel it is necessary but we'll take off some points if you do. We will also take off points for doing unnecessary joins, distincts, etc.**:

a) [3 points] How many different players have scored more than 50 points in a game?

```
SELECT COUNT(DISTINCT pid)
FROM Stats
WHERE points > 50;
```

b) [5 points] For all the Players who have played in at least one game, list their pid and the total number of points they have scored. Return the list in *descending* order of the total number of points.

```
SELECT pid, SUM(points) AS Sum_Points
FROM Stats
GROUP BY pid
ORDER BY Sum_Points DESC;
```

Question 1 – SQL continued
Recall the schema:

```
      Player (pid, pname, team)
      Team (tname, city)
      Game (gameid, homeTeam, awayTeam, homeScore, awayScore)
      Stats (pid, gameid, points, assists, rebounds)
```

c) [5 points] A "triple double" is when a player's number of assists, rebounds, and points in a game are all 10 or more. For each triple double, list the name (pname) and team (tname) of the player who got it, and the city where the game was played.

```
SELECT P.pname, T.tname, T.city
FROM Stats S, Player P, Team T, Game G
WHERE S.pid = P.pid AND G.gameid = S.gameid AND
      Game.homeTeam = T.tname AND S.points >= 10 AND
      S.assists >= 10 AND S.rebounds >= 10;
```

d) [7 points] List the names of teams who never lost a game at home.

```
SELECT T.tname
FROM Team T
WHERE T.tname NOT IN (SELECT homeTeam
                      FROM Game
                      WHERE homeScore < awayScore);
```

Question 1 – SQL continued
Recall the schema:

```
      Player (pid, pname, team)
      Team (tname, city)
      Game (gameid, homeTeam, awayTeam, homeScore, awayScore)
      Stats (pid, gameid, points, assists, rebounds)
```

e) [10 points] List the name of each team that won more games at home than they lost at home.

```
SELECT DISTINCT T.tname
FROM Team T
WHERE (SELECT COUNT(*)
       FROM Game
       WHERE homeTeam = T.tname AND homeScore >= awayScore) >
      (SELECT COUNT(*)
       FROM Game
       WHERE homeTeam = T.tname AND homeScore < awayScore);
```

f) [10 points] List the name of each team and a count of the number of games the team has won overall (i.e., regardless of whether they were the home team or the away team).

```
SELECT T.tname, COUNT (*)
FROM Team T, Game G
WHERE (T.tname = G.homeTeam AND homeScore > awayScore) OR
      (T.tname = G.awayTeam AND homeScore < awayScore)
GROUP BY T.tname;
```

Question 2 –Hashing [4 parts, 15 points total]

Consider a relation containing information about university students:

```
Students (sid: integer, sname: varchar(50), street:
  varchar(50), city: varchar(30), age: integer)
```

consisting of 250,000 tuples, where there are 100 tuples per block (i.e., the students file is 2500 blocks long).

a) (2 points) Consider a simple, 2-pass disk-based hashing strategy as described in lecture, where in the first pass the file is partitioned and in the second pass each of the partitions is hashed (i.e., not hybrid hashing!). Ignoring any space overhead for building hash tables and assuming a perfect hash function, what is the minimum number of memory pages (same size as file blocks) required to hash the Students relation in 2 passes?

```
Answer: 51
Sqrt(pages(s))+ 1 = 51.
We need 1 extra buffer page for input!
```

b) (3 points) In part (a) above, how many I/Os will be required? Assume that the relation is originally on disk that the result of the hashing operation must also be written to disk.

```
Answer: 4*2500 = 10,000
There are 2500 pages.  Each pass requires 2500 IOs to read and
2500 IOs to write.  Since there are two passes, 2*2*2500 = 10k.
```

c) (5 points) Now, suppose that the hash function used for part (a) does not work very well. Describe briefly why 2 passes may not be sufficient and how a complete algorithm would solve this problem.

```
a.
  Previously we assumed that the hash function was perfect, so the
size of each partition is exactly pages(s)/# memory pages.  If the
hash function is faulty, a partition's size may be greater or
smaller than that.  If greater, the partition may not fit into
memory on the second pass!

b.
  We could run the partitioning function recursively while a
partition is too big.
```

d) (5 points) Now, consider a combined Hash/Aggregation operator using “hybrid hash”, such as the one you implemented in HW 2 and the query “**SELECT COUNT(*) FROM Students GROUP BY age**”. Given a relatively small amount of memory (say, 20 pages or so), would you expect the hybrid hash approach to perform better than the regular two-pass hashing approach on this query? Why or why not? (state any assumptions you are making and answer concisely)

```
Yes!

We know the maximum number of distinct values for age is ~100 for
the human population, and ~10 for the college population.  A hash
table with 100 buckets can easily fit into memory.
```

Question 3 – Query Optimization [4 parts, 25 points total]

Consider the following relational schema (with primary keys underlined):

STUDENTS (<u>sid</u>, s_name, street, city, age)
COURSES (<u>cid</u>, c_name, prof_name)
REGISTERED (<u>sid, cid</u>, credits)
where sid and cid are foreign keys referencing STUDENTS and COURSES respectively.

with the following characteristics:

- The **STUDENTS** relation has 10,000 tuples
- 20 tuples of **STUDENTS** fit in one disk block
- NKeys(city) for **STUDENTS** is 200 (i.e., there are 200 distinct values for city)
- NKeys(age) for **STUDENTS** is 50
- The **REGISTERED** relation has 40,000 tuples
- 50 tuples of **REGISTERED** fit in one disk block
- The **COURSES** relation has 500 tuples
- 40 tuples of **COURSES** fit in one disk block

and the following indexes:

- A hash index is defined on *sid* for **STUDENTS**
- A *clustered* B+Tree index is defined on the *city* attribute for **STUDENTS**
- An *unclustered* B+Tree index is defined on the *age* attribute for **STUDENTS**

For parts **a, b, and c,** state an *efficient method* for answering this query and state how many disk accesses will have to be made in order to answer the query using this method. Be sure to state which index(es) if any you are using. *State any assumptions you are making. Note, you cannot assume that any of the relations fit in memory.* Part of your grade will depend on how efficient a solution you choose.

a) [5 points] SELECT *
FROM STUDENTS
WHERE city = 'Berkeley' and age = 30;

```
Assume a uniform distribution of student in cities and students in
age groups
Assume that the ages are uniformly distributed across the
unclustered file.

The city qualification has higher selectivity (1/200) and since
the city index is clustered, we only have to read a very small
number of pages (500/200).

Plan: index scan on city with a filtering on the fly on age
```

Question 3 – Query Optimization (continued)

b) [5 points] SELECT *
FROM STUDENTS
WHERE city = 'Berkeley' and sid = 123456789;

```
Sid is a key, so we know there is exactly one student with the
given sid.
We have a hash index on sid, so we can find the page in 1.2 I/O's,
read the page, and filter on the city.  It takes 2.2 I/O's in
total.

Plan: hash index scan on sid followed by filtering on city on the
fly.
```

c) [7 points] Assuming there are five (5) pages of memory available for use in by the join, there are no indexes on **REGISTERED** or **COURSES,** and they are not currently sorted, choose an efficient method for computing the join of these two relations and estimate the number of disk I/Os it would incur. Be sure to state the join method you have chosen, and which relation is inner or outer if appropriate.

```
Block Nested Loop Join with COURSES as outer

Ceiling(13/4)*800 + 13 = 3213

Hash Index Join

3*(13+800) = 2439
```

d) [8 points] Consider the three way join between **STUDENTS, COURSES,** and **REGISTERED**. Show an efficient join ordering (no need to specify a join method for this question) and estimate the number of tuples that will be in the result of each of the joins.

```
Join COURSES and REGISTERED then with STUDENTS

1. Cannot join COURSES with STUDENTS as it will be a Cartesian
product
2. COURSES should be the inner-most relation because it's the
smallest relation

the resulting number of tuples is 40K
```

Question 4 – More Query Optimization [5 parts, 20 points total]

Recall the Basketball schema from Question 1:

```
      Player (pid, pname, team);  Team (tname, city)
      Game (gameid, homeTeam, awayTeam, homeScore, awayScore)
      Stats (pid, gameid, points, assists, rebounds)
```

a) [5 points] Consider the four-way join of these relations. Give one join ordering that a System-R optimizer would **NOT** consider, and briefly state why it would not consider it.

```
There are two types of valid answers.
a.
  Any tree that is not left deep
b.
  A tree containing a cross product, namely
  (((team x stats) x anything) x anything)
```

b) [3 points] Write a SQL SELECT statement that could take advantage of a composite B+tree index on Stats using the composite key (points, assists, rebounds) but could **not** take advantage of a composite B+tree index using (assists, rebounds, points).

```
Any query with predicates on points or points and assists.
Any query that uses all three attributes is incorrect.
```

c) [3 points] Write a SQL SELECT statement that could take advantage of a B+tree index on Stats using the key (points) but could **not** take advantage of a Hash index using the key (points).

```
Any range query or max/min predicate on points.

Ex:
  Select max(points) from stats;
  Select * from stats where points > 10;
```

d) [5 points] Assume that in the Stats relation, there are 10,000 tuples with 50 unique values for "points" and 20 unique values for "rebounds". What is the expected cardinality computed by a System R-style optimizer for the predicate "points = rebounds" on the Stats relation?

```
System R only considers the single best relation's selectivity.
So,
1/50 * 10,000 = 200.
```

e) [4 points] Consider the "points" attribute of the Stats relation. Why might a query optimizer that maintains histograms over values of points provide better query plans than one that uses the System R approach of keeping only the high-low values and the number of distinct values?

High-low values assume that points is evenly distributed. If points are non-uniformly distributed, a histogram will store more accurate information than high-low values.