UNIVERSITY OF CALIFORNIA
College of Engineering
Department of EECS, Computer Science Division

CS186
Spring 2010

J. Hellerstein
Midterm #2

Midterm Exam 2: Introduction to Database Systems: Solutions

1. Query Optimization [12 points]

Consider the following schema.

auctions (aid, minprice, description, seller, end_date).
members (mid, nickname, name, since).
bids (aid, buyerid, amount).

Assume there is an unclustered B-tree index on the key of each table. In answering questions, use the summary statistics functions that we learned about in class: NPages(), NTuples(), Low(), High(), NKeys(), IHeight(), INPages().

a. [3 points] Consider the query
```
SELECT 'found it!' FROM members WHERE mid = 98765;.
```
Given the information above, write a formula for the optimizer's lowest estimated cost for this query.

IHeight(Btree on members.mid)

b. [3 points] Consider the query
```
SELECT * FROM bids, members
 WHERE bids.buyerid = members.mid AND members.since < '2001';
```
Write the formula the optimizer would use for the selectivity of the entire WHERE clause.

[1/(MAX(Nvals(bids.buyerid), NVals(members.mid)]

*

[2001 - MIN(members.since)]
/ [MAX(members.since)-MIN(members.since)]

c. [2 points] Consider the same query as in part (b). Now suppose the optimizer knows that bids.buyerid is a "not null" foreign key referencing members. Write a simplified formula for the selectivity of the entire WHERE clause.

1/NTuples(members)

*

[2001 - MIN(members.since)]
/ [MAX(members.since)-MIN(members.since)]

d. [4 points] Consider the following query:

```
SELECT R.*
  FROM R, S, T
 WHERE R.a = S.b
   AND S.b = T.c
```

The following plans are generated during an intermediate pass of the Selinger (System R) optimizer algorithm. For each plan, write down the ordering column(s) of its output if any, and whether the plan would get pruned (P) or kept (K) at the end of the pass. If there is no clear ordering on the output, write "none".

PLAN	Cost in I/Os	Ordering Columns of Output	Prune or Keep
IndexNestedLoops (FileScan(R), IndexOnlyScan(Btree on S.b))	2010	None	P
SortMergeJoin(FileScan(R), FileScan(S))	3010	(R.a, S.b)	K
ChunkNestedLoops(FileScan(R), FileScan(S))	1010	None	K
IndexNestedLoops(IndexOnlyScan(Btree on (R.d, R.a)), IndexScan(Btree on S.b))	30000	(R.d, R.a)	P

2. Relational Algebra/Calculus [16 points]

Consider the following schema:

Store(<u>sid</u>, store_name, parent_company).

Branch(<u>sid</u>, <u>city</u>, open24).
- sid is a foreign key to store
- open24 is a boolean. If true, the store is open 24 hours a day. If false, it is open 7AM-10PM.

Has_Fruit(<u>sid</u>, <u>city</u>, <u>fid</u>, quantity, price).
- sid is a foreign key to store
- fid is a foreign key to fruit
- price is a floating point number representing the number of whole dollars and whole cents of the price (for example, 3.00)

Fruit(<u>fruit_id</u>, name, calories).

a. [3 points] Assume it is 2AM in Berkeley, you are craving some pineapple, and since you own stock in parent company "WFM", you want to buy your pineapple in a branch owned by them. Complete the relational calculus query below to return store_names of all stores with branches in Berkeley that are open at 2AM, have at least 5 pineapples, charge less than $3.00 per pineapple, and have parent company "WFM".

{ X | ∃ S ∃ B ∃ H ∃ F (S ∈ Store ∧ B ∈ Branch ∧ H ∈ Has_Fruit ∧ F ∈ Fruit ∧

s.sid = b.sid AND
s.sid = h.sid AND
b.city = h.city AND
h.fid = f.fruit_id AND
b.city = "berkeley" AND
b.open24 AND
<u>f.name</u> = "pineapple" AND
h.quantity >= 5 AND
h.price < 3.00 AND
s.parent_company = "wfm" AND
x.store_name = s.store_name

}

All 11 clauses correct: 3 points. 5 to 10 clauses: 2 points. 1 to 4 clauses: 1 point. Points were not deducted for switching </<= and >/>= or for missing quotes on strings or missing ".00" on floating-point values.

b. [4 points] Translate the query of part (a) to relational algebra, pushing selections in as far as possible, postponing projection until the very end, and using the join order: Store, Branch, Has_Fruit, Fruit.

π_{store_name} ($\sigma_{parent\_company='WFM'}$ (Store) $\bowtie$ $\sigma_{city='Berkeley' \text{ AND } open24}$ (Branch) $\bowtie$ $\sigma_{quantity >= 5 \text{ AND } price < 3}$ (Has_Fruit) $\bowtie_{fruit\_id=fid}$ $\sigma_{name='pineapple'}$ (Fruit))

1-2 points for correctness. If at least 1 point for correctness, then 1 point for pushing all selections down as far as possible, and 1 point for keeping all projections outside.

0 to 3 errors : 1 point off. >= 4 errors : 2 points off.

Points were deducted for:
- using a join order resulting in a cross product
- missing a selection condition
- missing or invalid join conditions (e.g. fid = fruit_id)
- missing tables in WHERE clause

c. [5 points] Assume you are part of LowCal, a student group that encourages students to eat low-calorie fruit. Your group is starting a new campaign to boycott parent companies that have at least one store branch with no "Has_Fruit" records for fruit with less than 300 calories. Complete the relational calculus expression below that lists the parent companies you should boycott.

{ X | ∃ S ∃ B (S ∈ Store ∧ B ∈ Branch ∧
    S.sid = B.sid ∧ X.parent_company = S.parent_company
∀ H ∈ Has_Fruit(
    (H.sid = B.sid ∧ H.city = B.city) =>
        ∃F(F ∈ Fruit ∧ F.fruit_id = H.fid ∧ F.calories >= 300)))
}

0.5 points for having "H.sid = B.sid ∧ H.city = B.city" in a context with correct quantifiers over H and B.
0.5 points for having "X.parent_company = S.parent_company" in a context with correct quantifiers over X and S.
0.5 points for having "S.sid = B.sid" in a context with correct quantifiers over S and B
0.5 point for having the equivalent of " F(F ∈ Fruit ∧ F.fruit_id = H.fid ∧ F.calories >= 300)" in a context with the correct quantifier over H.
3 points if the second part of your answer was logically equivalent to the solution above.

d. [4 points] Translate the expression from part (c) into a single SQL query (with a single "NOT EXISTS" sub-query) by completing the query below. Recall that SQL returns duplicates by default, whereas the relational calculus does not.

```
SELECT
        DISTINCT S.parent_company
FROM Store S, Branch B
WHERE
        S.sid = B.sid
AND NOT EXISTS (
        SELECT *
        FROM Has_Fruit H, Fruit F
        WHERE
                H.fid = F.fruit_id
                AND H.sid = S.sid
                AND F.calories >= 300
                AND H.city = B.city
);
```

The following six expressions should have been included:

1. DISTINCT S.parent_company
2. S.sid = B.sid
3. H.fid = F.fruit_id
4. H.sid = S.sid
5. F.calories >= 300
6. H.city = B.city

All 6 were included: 4 points
4 or 5: 3 points
2 to 3: 2 points
1: 1 point
0: 0 points

3. Parallel Query Processing [8 points]

a. [4 points] Skew can be a problem in certain query processing techniques. In the table below, please describe the problems that skew can cause for *single-table* sorting or hashing. In the first column please address the single-node algorithms that "spill" to disk; in the second column please address parallel algorithms and assume that the data that arrives at each nodes fits in RAM. In cases where skew does not cause problems, simply write "No Problem". Your answer should be very short, and fit within the box!

	SINGLE NODE, SPILLS TO DISK	PARALLEL, FITS IN RAM
SORTING	No Problem	Uneven distribution of tuples across machines will lead to stragglers
HASHING	Uneven distribution of tuples across partitions will lead to unnecessary recursive partitioning	Uneven distribution of tuples across machines will lead to stragglers

b. [4 points] Consider the query

```
SELECT key, AVG(val) FROM T GROUP BY key;
```

Both key and val are integer fields, and our machines use 4-byte integers and floats. Assume there are 10,000 key values, and 101 machines. Tuples are distributed randomly across all 101 machines to begin, and all 101 machines share the work of processing each query operator. How many bytes should this query send across the network, not including output? *Briefly* and *neatly* justify your answer by describing the data sent by each machine (your answer should fit easily below!)

Transvals have form (key,count,sum)	12	Bytes/Transval
# Transvals computed per node	10,000	Transvals/Node
Fraction of Transvals shipped	100/101	Nodes/Nodes
# nodes sending (partitioned ll-ism)	101	Nodes
	12,000,000	Bytes

4. SQL [13 points]

Consider the table created by the following SQL statement:

```
CREATE TABLE G (    -- G is short for Grades
   ssn INT,         -- ssn of student
   class CHAR(5),   -- department and number of course
   grade INT,       -- final grade between 0 and 100
   PRIMARY KEY(ssn, class)
)
```

Fill in the blanks to produce valid SQL queries to answer the following questions.

a. [3 points] Find the SSNs of students who received the maximum score in CS186 without using MAX.

```
SELECT ssn FROM G

WHERE class='CS186' AND _>= ALL_

    (SELECT grade FROM G);
```

One point was deducted for use of >=. No points were given for answers involving "100", since it was noted on the board during the exam that "maximum" refers to the highest score in the class, not the maximum possible score.

b. [3 points] Find the SSNs of all students who have received grades of 85 or above in at least 3 classes.

```
SELECT ssn FROM G


WHERE _grade >= 85_


GROUP BY _ssn_


HAVING _COUNT(*) >= 3_;
```

Any valid expression was accepted inside the COUNT() aggregate. Using ">" instead of ">=" in one place was forgiven. Using "> 84" or "> 2" is okay. Using > in both the WHERE and HAVING clause, using the wrong GROUP BY, or reversing the WHERE *and HAVING clauses resulted in lost points.*

c. [3 points] For each CS186 student, return three columns: their SSN, a boolean value indicating whether they have taken CS161, and the average of their grades in the two classes combined. If they have not taken CS161, produce NULL for their average grade.

```
CREATE VIEW T1 AS
    SELECT ssn, grade FROM g WHERE class='CS186';
CREATE VIEW T2 AS
    SELECT ssn, grade FROM g WHERE class='CS161';


SELECT  T1.ssn, T2.ssn IS NOT NULL, (T1.grade + T2.grade)/2

FROM T1  LEFT OUTER  JOIN T2

     ON  T1.ssn = T2.ssn ;
```

Points were not deducted for correct use of CASE, although this was unnecessary (only for CASE with wrong syntax or result). Points were lost for the syntax "ON ssn" (invalid ambiguous reference to ssn), for the syntax "T2.ssn <> NULL" (this is NULL when T2.ssn is NULL, not false), for the use of aggregates such as AVG (aggregates cannot aggregate over columns, only over rows, and there is no GROUP BY here), for the wrong type of JOIN, and for many other reasons.

d. [4 points] Find pairs of distinct students who took at least 3 classes together. Each unordered pair should be listed only once.

```
SELECT  G1.ssn, G2.ssn

FROM  FROM G AS G1, G AS G2

WHERE  WHERE G1.ssn < G2.ssn AND G1.class = G2.class

GROUP BY  G1.ssn, G2.ssn

HAVING  COUNT(*) >= 3 ;
```

Two points were deducted for not including G1.ssn < G2.ssn or G1.ssn > G2.ssn – this was the essential condition needed to ensure that duplicate unordered pairs do not occur (e.g. (2,4) and (4,2)). Alternate table names instead of G1, G2 were accepted, as were redundant conditions such as G1.ssn <> G2.ssn; and you could put anything valid inside the COUNT() aggregate. Points were also deducted for excluding the condition G1.class = G2.class, for selecting from only one copy of G, for grouping by class instead of ssn, for placing the HAVING condition in the WHERE clause, for not including both ssn columns in the output, and for many other reasons.