CSE-4411A Test #2

Sur / Last Name:
Given / First Name:
Student ID:

- **Instructor:** Parke Godfrey
- **Exam Duration:** 80 minutes
- **Term:** Fall 2009

Answer the following questions to the best of your knowledge. Your answers may be brief, but be precise and be careful. The exam is open-book and open-notes. Calculators, etc., are fine to use. Write any assumptions you need to make along with your answers, whenever necessary.

There are four major questions. Points for each question and sub-question are as indicated. In total, the test is out of 50 points.

If you need additional space for an answer, just indicate clearly where you are continuing.

Grading Box	
1.	/15
2.	/10
3.	/15
4.	/10
Total	/50

1. (15 points) **Indexes and Access Paths.** *Forest for the trees.* [Short Answer]

Table **R** has a *clustered* tree index of type alternative #2 on A, B, C, D (in that order, all ascending). The index pages contain 99 *index records* (encompassing 99 pointers, plus a 100*th* pointer), on average;[1] data-entry pages contain 50 data entries each, on average; and data-record pages contain 20 data records each, on average.

A's values range over 1..10,000; B's over 1..1,000; C's over 1..100; and D's over 1..10.

a. (3 points) An index record contains effectively the information A, B, C, D, and a pointer (an address to another page). A data entry contains effectively A, B, C, D, and an RID (which is an address to another page *and* a slot number). A data entry is *slightly* larger than an index record—by a slot number—but only slightly. So explain how it is possible there are 99 index records per page, on average, but *only* 50 data entries per page, on average.

Key prefix compression.
Some gave different answers. Namely, the index pages could be completely filled, and the data-entry pages only about half filled. But that would be quite unusual!

b. (3 points) Say that table **R** has 1,000,000 records. How deep is the index tree?

20K data-entry pages. Need 200 index pages to index these, with the 100 pointers $(99 + 1)$ *on each. Need two index pages one level up to index these 200. So then need a root index page.*
Thus, 4 deep: three layers of index pages, one layer of data-entry pages.
Or, we can do it by formula: $\lceil \log_{100}(\#\textit{entries}/\#\textit{entries-per-page}) \rceil + 1$.

[1] This accounts for the fill factor. A page could hold more than 99 index records.

Consider the query

select * from R where A $=$ 3141 and B $>$ 500 and D $>$ 2;

c. (3 points) What is the cardinality estimation of the query?
What assumption, often faulty, are you having to make to estimate the cardinality?

$1m\ records \cdot \frac{1}{10000} \cdot \frac{500}{1000} \cdot \frac{8}{10} = 40$

d. (3 points) Estimate the I/O cost of the query using the index as the access path.

Probe to left-most entry where $A = 3141$ *and* $B > 500$*: 3 I/O's (not counting that first data-entry page). Scan over 2 data-entry pages for the 50 matches: 2 I/O's. (50 D.E.'s per page, on average). Fetch the matching records themselves. Clustered, so likely the 50 records at 20 per page span 3 pages: 3 I/O's. 8 I/O's total.*

e. (3 points) Name a *functional dependency* that, if true for the table **R**, would guarantee that use of the index would satisfy *order by* A, B, D.

$A, B \mapsto C$. *This way, there is a unique C value for every pair of values for A, B. Therefore, if we are ordered by A, B, we are ordered by A, B, C.So, ordered by A, B, C, D is then the same as ordered by A, B, D.*

2. (10 points) **Query Optimization.** *One is not like the others.* [Multiple Choice]

   Choose *one* best answer for each of the following. Each is worth one point. There is no negative penalty for a wrong answer.

---

a. An iterator, pull-based architecture for the query processor
   - **A.** eliminates the need for a buffer pool manager.
   - **B.** eliminates the need for a query optimizer.
   - **C.** is good for object-oriented database systems, but not for relational systems.
   - [✓ **D.**] implements pipelining.
   - **E.** does not allow for on-the-fly operations.

---

b. A System R style optimizer—the type of optimizer the textbook presents—preserves sub-plans that generate *interesting orders* while enumerating possible query plans because
   - **A.** such plans are *always* less expensive than plans that have no interesting order.
   - **B.** pipelining is impossible without them.
   - [✓ **C.**] such plans may allow certain subsequent operations to be done on-the-fly.
   - **D.** hash joins are impossible without them.
   - **E.** this prunes the search space of possible query plans.

---

c. All are access path options *except*
   - [✓ **A.**] block-nested-loop join
   - **B.** file scan
   - **C.** index-only scan
   - **D.** index intersection
   - **E.** index probe

---

d. Consider the two-pass hash join and the two-pass sort-merge join. Which of the following is not true?
   - **A.** The hash join can be used whenever the sort-merge join can be used.
   - [✓ **B.**] The sort-merge join can be used whenever the hash join can be used.
   - **C.** Both are blocking operations that break pipelining.
   - **D.** Sort-merge join is often preferred as it is immune to skew.
   - **E.** Sort-merge join is often preferred as it produces an interesting order.

---

e. Which of the following is true?
   - **A.** Anywhere a (2-pass) hash join can be used, a (2-pass) sort-merge join could be used instead, at the same cost.
   - **B.** A block-nested-loops join is never better than a hash join or a sort-merge join. It is part of the repertoire simply for the cases when hash join and sort-merge join cannot be applied.
   - **C.** Select conditions can only be evaluated by use of an index.
   - [✓ **D.**] Index-nested-loops join will rarely be the best choice unless there are other highly selective—returning few results—conditions in the query.
   - **E.** Hash joins and sort-merge joins cannot be used together in the same query plan.

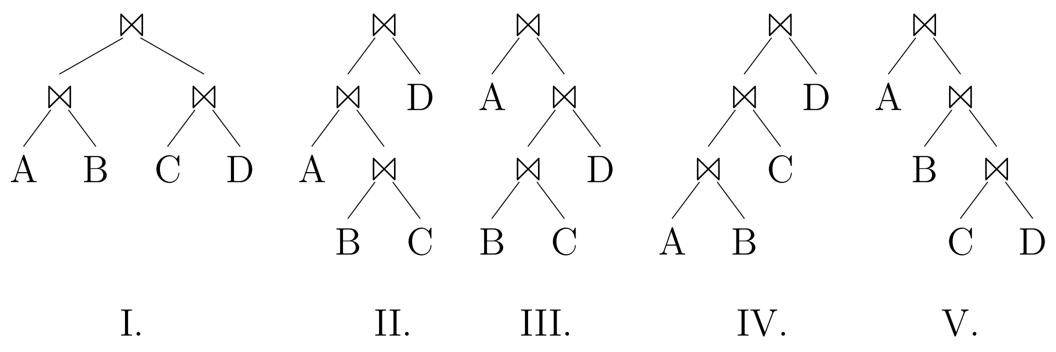


Figure 1: Join Trees.

f. Which of the trees in Figure 1 is a left linear (left-deep) join tree?
- **A.** I.
- **B.** II.
- **C.** III.
- [✓ **D.**] IV.
- **E.** V.
- **F.** They *all* are left linear join trees.

g. Which of the trees in Figure 1 is not a linear join tree?
- [✓ **A.**] I.
- **B.** II.
- **C.** III.
- **D.** IV.
- **E.** V.
- **F.** They *all* are linear join trees.

The following information is available on tables **Sailors** and **Reserves**.

- **Reserves**: 10,000 records
  - bid: 50 values (1..50)
- **Sailors**: 1000 records
  - level: 10 values (1..10)
  - fav_colour: 5 values
- **Boats**: 50 records
  - colour: 5 values

The primary key of **Sailors** is sid, of **Reserves** is sid + bid + day, and of **Boats** is bid. Table **Reserves** reserves has a foreign key on sid referencing **Sailors** (on sid), and a foreign key on bid referencing **Boats** (on bid). All columns are *not null*.

For each of the following queries, estimate the selectivity of the query as the number of tuples it likely returns, as System R would. (The estimation techniques discussed in Chapter 12 of the textbook are from System R.)

---

h.
```
select S.sid, B.bid
    from Sailor S, Boats B
    where B.colour = S.fav_colour;
```
A. 1
B. 50
C. 100
D. 5,000
$\boxed{\textbf{E.}}$ 10,000
F. 500,000

---

i.
```
select S.sid, R.day
    from Sailor S, Reserves R
    where S.sid = R.sid and
          R.bid = 37;
```
A. 1
B. 20
C. 50
D. 100
$\boxed{\textbf{E.}}$ 200
F. 500

---

j.
```
select S.sid, S.sname, R.bid, B.bname, R.day
    from Sailor S, Reserves R, Boats B
    where S.sid = R.sid and R.bid = B.bid and
          R.bid = 37 and S.level = 7;
```
A. 1
B. 5
$\boxed{\textbf{C.}}$ 20
D. 80
E. 100
F. 10,000

3. (15 points) **Operator Algorithms.** *Try a left-handed integral join...* [Exercise]

For the following, the joins considered are equi-joins; that is, the join predicates are equalities.

a. (3 points) Consider that table **R** has a foreign key referencing table **T**, on **T**'s primary key A. The foreign key is mandatory. **T** has a clustered tree index on A. Consider a query with an equi-join on A between **R** and **T**. The query has no other predicates, selections or otherwise.

In what scenario—sizes of tables, whether the join columns are *distinct* for the outer or the inner, and so forth—would an index nested loop join be significantly less expensive than a two-pass sort-merge join or hash join?

Explain why.

> **R** *is quite small,* **T** *is quite large. Each* **R** *record references a* **R** *record. So most of* **T** *is never accessed. SMJ or HJ pays the cost of* sorting *all of* **T**. *We ony pay* $|\mathbf{R}| \cdot$ *cost-of-probe with INJ, though.*

b. (3 points) Name a scenario—sizes of tables, etc.—when having the outer sorted *or* partitioned with respect to the join columns would benefit an index-nested-loop join.

Explain why.

> *The outer can have duplicates with respect to the join columns. If the outer stream is grouped by the join columns, consecutive probes with the same values do not incur new I/O's, as the pages for the probes are already in the buffer pool.*

Schema:

> **Student**(<u>sid</u>, sname, startdate, major, advisor)
> FK (advisor) refs **Prof** (pid)
> **Class**(<u>cid</u>, dept, number, section, term, year, room, time, pid, ta)
> FK (pid) refs **Prof**
> FK (ta) refs **Student** (sid)
> **Enrol**(<u>sid</u>, <u>cid</u>, date, grade)
> FK (sid) refs **Student**
> FK (cid) refs **Class**
> **Prof**(<u>pid</u>, pname, pdept, office)

Assume no attribute is nullable. The attribute pid in **Class** refers to the the professor / instructor for the class. The attribute ta in **Class** refers to the teaching assistant for the class. The attribute advisor in **Student** refers to the student's academic advisor.

Statistics:

- **Student**: 50,000 records on 1,000 pages
  - advisor: 2,500 distinct values
- **Enrol**: 2,000,000 records on 20,000 pages
  - sid: 50,000 distinct values
  - cid: 80,000 distinct values
- **Class**: 80,000 records on 1,600 pages
  - pid: 4,000 distinct values
  - ta: 5,000 distinct values
- **Prof**: 4,000 records on 40 pages

Indexes:

- **Student**:
  - clustered tree index on sid (200 data entries per page)
- **Enrol**:
  - clustered tree index on cid, sid (167 data entries per page)
  - unclustered tree index on sid, cid (167 data entries per page)
- **Class**:
  - clustered tree index on cid (200 data entries per page)
- **Prof**:
  - clustered tree index on pid (200 data entries per page)

All indexes are of alternative #2. For each tree index, the index pages are 3 deep, except for the index on **Prof**.pid which is 2 deep.

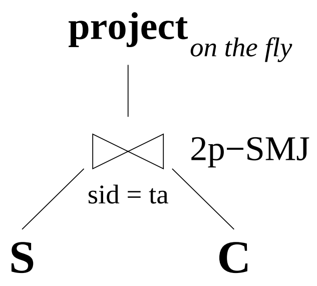
Query:

```
select sid, sname, cid, dept, number, section, term, year
    from Student S, Class C
    where S.sid = C.ta;
```

c. (3 points) What is the cardinality of the query?

$$\frac{50k \cdot 80k}{max(50k, 5k)} = 80k$$

d. (6 points) Devise a least-cost query plan for the query. Assume you have a buffer-pool allocation of 50 frames. Show your plan's cost.



7,800 I/Os

We have enough space for the two-pass sort-merge join.

4. (10 points) **Join Mechanics.** *Maybe it's the carburetor?* [Analysis]

a. (3 points) Dr. Datta Bas has an idea for how to share partitions with two-pass hash joins. Say we have a query joining tables **R**, **S**, and **T** with **R**.A = **S**.A and **S**.A = **T**.A. He claims we can reuse partitions of one hash join in the next hash join.

Is he right? Will this save I/O's? If so, explain. If not, explain why he is wrong.

Hey! He is right!
When we join **R** *and* **S***, say, on partition* i*, the results (***RS***) are of partition* i*. So save these.*
Then, when we do **RS** $\bowtie_A$ **T***, we only need to make partitions for* **T***. We already have the partitions for* **RS***. So it saves all of pass 0 of the HJ for the outer, in the second join.*

b. (2 points) Say you have a fixed buffer pool allocation of B frames for the query in Question 4a, and will use Dr. Bas's shared-partition idea with two two-pass hash joins. What would have to be true so that it could be used?

These paritions of the outer for **RS** *in the second HJ should each fit in the buffer pool.*
Of course, we could "swap" and use **T***'s partitions as the "outer" in the second HJ, instead. In that case, two of the three tables should have partitions that are small enough for the buffer-pool allocation, or the partition results of the first join themselves are small enough.*

c. (3 points) We have a query that joins tables **R**, **S**, and **T**, with join predicates **R**.A = **S**.A, **R**.B = **S**.B, and **S**.A = **T**.A.

- **R** is 50,000 tuples on 1,000 pages.
- **S** is 1,000,000 tuples on 20,000 pages.
- **T** is 200,000 tuples on 2,500 pages.
- **R** $\bowtie_{A,B}$ **S** is 5,000,000 tuples on 143,000 pages.
- **R** $\bowtie_{A}$ **T** is 300,000 tuples on 10,000 pages.
- **S** $\bowtie_{A}$ **T** is 10,000,000 tuples on 335,000 pages.
- The result of (**R** $\bowtie_{A,B}$ **S**) $\bowtie_{A}$ **T** is 10,000 tuples on 500 pages.

Using two-pass sort-merge joins, show an efficient plan to evaluate the query. There are no indexes. You have a buffer pool allocation of 150 frames.

What is its I/O cost, in total?

We cannot use two-pass sort-merge join first to join **R** *and* **S**, *or to join* **S** *and* **T**. *The results of either—***RS** *or* **ST***—are too big to do a two-pass sort-merge join as the second join.*

We can join **R** *and* **T**, *and then join the results (***RT***) with* **S**. *Unfortunately, the* order *of the stream* **RT** *is on A, but the join predicates for* **RT** *and* **S** *are A* and *B. So, we have to store* **RS** *temporarily on disk, and the second join has to sort it.*

- *Join #1.* **R** $\bowtie_{A}$ **S** *by 2p-SMJ: 10,500 I/Os.*
- *Write out temp* **RS***: 10,000 I/Os.*
- *Join #2.* **RS** $\bowtie_{A,B}$ **T** *by 2p-SMJ: 90,000 I/Os.*
- *Total: 110,500 I/Os.*

d. (2 points) Dr. Dogfurry thinks that an even better plan can be found for the query in Question 4c using two-pass sort-merge joins if *over sorting* is allowed; that is, the first sort-merge join sorts the stream *more* than it needs to for the benefit of future operations.

Is he right? If so, what efficient plan can you find using over-sorting, and what is its I/O cost, in total?

Yes.

Over-sort by A, B for Join #1 above. Then, in Join #2, the outer stream is sorted already as needed. So the second SMJ only needs to sort the inner, and read the streams of the inner to join-merge. That is 60,000 I/O's.

So the cost now is the cost of the first join, 10,500 I/O's, and this. The outer from the first is pipelined to the second, so it does not have to be written to disk. Thus, the total is 70,500 I/Os.

Note we save in two *places: the results of the first join do not have to be sorted; and the stream from the first join are* pipeleined *to the second, not needing to be written temporarily to disk.*

(Scratch space.)

Relax. Turn in your exam. Go home.