

COSC-4411(M) Midterm #2

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

- **Instructor:** Parke Godfrey
- **Exam Duration:** 75 minutes
- **Term:** Winter 2004

Answer the following questions to the best of your knowledge. 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 exam is out of 50 points.

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

Regrade Policy

- Regrading should only be requested in writing. Write what you would like to be reconsidered. Note, however, that an exam accepted for regrading will be reviewed and regraded in entirety (all questions).
-

Grading Box	
1.	
2.	
3.	
4.	
Total	

1. (15 points) **Join Algorithm Selection.** *Would you join me in choosing a join?*
[exercise / short answer]

Consider tables **R** with attributes A and B and **S** with attributes B and C. Column B is unique in **S**. Values of B in **R** are the same as the values of B in **S**. Assume that there are no indexes available.

Consider the sort-merge join to be the efficient *two-pass* version discussed in the textbook (in which the sort-merge's merge steps are integrated with the external sort's merge steps) and not the more general version. (In the more general version, the external sort is *entirely* done before the merge-join. Furthermore, assume that the sort-merge join algorithm produces runs of length $2B$ in "pass 0" of sorting.) Consider that there may be problems with skew.

For each of the following, choose which of the following join algorithms is likely to be best. In each case, there are 250 buffer frames allocated for the join.

- A. A block nested loops join with **R** as the outer and with **S** as the inner.
- B. A block nested loops join with **S** as the outer and with **R** as the inner.
- C. A (2-pass) sort-merge join with **R** as the outer and with **S** as the inner.
- D. A (2-pass) sort-merge join with **S** as the outer and with **R** as the inner.
- E. A hash join with **R** as the outer and with **S** as the inner.
- F. A hash join with **S** as the outer and with **R** as the inner.

Which would be the most cost effective method to evaluate $\mathbf{R} \bowtie_{\mathbf{B}} \mathbf{S}$? Briefly justify your answer in each case.

- a. (5 points) Table **R** is 200 pages with 80 records a page, and table **S** is 40,000 pages with 100 records a page.

A. *All of table **R**, 200 pages, fits within our 250-page allocation. So **R** can be read in as a single block. The **S** can be scanned once to accomplish the join. So that is 40,200 I/O's.*

*With **S** as the outer of a BNL join, we would have to read it as $\lceil 40,000/248 \rceil$, 162, blocks, scanning **R** each time. So $40,000 + 162 \times 200 = 72,400$ I/O's.*

We actually have enough room to do the 2-pass sort-merge join: $\sqrt{40,000} = 200 < 250$. And we have enough for the hash-join: $\lceil \sqrt{200} \rceil = 15 < 250$. Any of these though, will cost $3 \times (40000 + 200) = 120600$ I/O's.

- b. (5 points) Table **R** is 50,000 pages with 80 records a page, and table **S** is 40,000 pages with 100 records a page.

*C. The hash join or the 2-pass sort-merge, with either **R** or **S** as the outer, are each possible, and each will have a cost of $3 \times (50,000 + 40,000) = 270,000$ I/O's. A BNL join here would be quite expensive. Consider it with **S** as the outer: we would have to read it as $\lceil 40,000/248 \rceil, 162, 40,000 + \lceil 40,000/248 \rceil \times 50,000 = 8,140,000$ I/O's. Having **R** as the outer would be even more. Because data-skew is a problem here, we prefer a SMJ over HJ, given each works with the buffer allocation, and each costs the same (barring no problems at runtime). Which should be the outer table for our SMJ? Space-usage-wise, it makes no difference. Because looping-for-duplicates is usually implemented within SMJ on the inner table, it is safer to put **R** as the outer since each **R** tuple can match only one **S** tuple. (In truth, with a sophisticated implementation of SMJ, this would not matter in this case, since each **S.B** value occurs at most once.) If we did use a HJ, it is slightly safer to make **S** the outer. It is slightly smaller, so the HJ would be more immune to data skew; it is less likely that a partition of **S** made in the first pass will not fit in buffer pool in the second pass.*

- c. (5 points) Consider the case of $\mathbf{R} \bowtie_J \mathbf{S}$ when join column **J** has very few distinct values in **R** and in **S**. Would a hash join likely work well for $\mathbf{R} \bowtie_J \mathbf{S}$? Why or why not?

*No, it would not. HJ works by partitioning **R** and partitioning **S** initially each into B partitions (where we have, say, $B+2$ buffer pool-frames with which to work). in the number of distinct values of $J \ll B$, this is not going to work. HJ will only make $|J|$ number of partitions each (at most). Each partition of **R** (let this be our outer table) will be, on average, $|\mathbf{R}|/|J|$ long. This is likely to be much larger than B , so the second pass of the HJ will not work well. The textbook describes a "safety" mechanism that can be built into HJ: recursively partition any partition of the outertable that is too big ($> B$), using another hash function, of course. That won't work here! The more reasonable "safety" mechanism programmed in is just to read in as much of a partition of the outer as possible. Scan the matching partition of the inner to join. Repeat, reading in the next "block" of the too-long outer partition, until done. The extra expense is reading some inner partitions multiple times, instead of just once, as is the HJ ideal. Note that this makes HJ act like a BNL join in a way when partitions are too big. This implementation would keep HJ working here, but we would have lost efficiency.*

2. (10 points) **Indexes.** *To index or not to index. That is the question...* [analysis]

- a. (5 points) Is it possible to use *index intersection* as an access path that is also an *index-only* plan?

Explain briefly why or why not.

Sure.

An index-intersection access path is when we use two indexes (or more) in combination. All the data entries are fetched that match through the first index, and likewise, all the data entries are fetched that match through the second index. We then find the intersection of these data entry records by RID. Thus, this is equivalent to joining these on RID. One way to do this is to sort the two lists of data entry records by RID and then merge, as in a SMJ.

Finally, we fetch the data records for each data entry that survived the intersecting. We have the added advantage if we sorted them by RID that fetching the records may be more efficient now.

An index-only access path is one in which we need only fetch data entries and do not need to fetch the actual data records. This works when all the columns needed for the rest of the query are part of the index; that is, part of the index's search key.

There is no reason the two techniques cannot be used together. In fact, with the two indexes are involved, it is only necessary that all the columns needed show up in the union of the two indexes' search keys.

- b. (5 points) Consider table **R** with attributes A, B, and C (A is its primary key) and table **S** with attributes C, D, and E (C is its primary key)

There is a foreign key declared for **R** on C referencing **S** (on C). We know that queries which join **R** and **S** over C (that is, queries which involve $\mathbf{R} \bowtie_{\mathbf{C}} \mathbf{S}$) will be very common. Is it worthwhile to build an index on **R.C**? Under what conditions would having an index on **R.C** help? Provide an example scenario, or explain why it probably does not help under any reasonable circumstances.

By “condition”, we mean properties of **R** and **S** and of the types of queries asked that involve $\mathbf{R} \bowtie_{\mathbf{C}} \mathbf{S}$. For example,

- When **R** is small...
- When **S** is large...
- When the queries involve many conditions additional to the join condition ($\mathbf{R.C} = \mathbf{S.C}$)...
- When the query involves few conditions beyond the join condition ($\mathbf{R.C} = \mathbf{S.C}$)...

*We likely have an index on **S.C**, since that is unique. (The DBMS would build one.)
When would it be advantageous to have an index on **R.C**?*

- *When the index of **S.C** is not available because of choices in the query plan but **R.C** could be used. In particular, if INL join is a good candidate and we need to probe **R** as the inner table.*

*Assume there are other conditions on **S** on D and/or E that are highly selective. We may have other indexes that can be used here. Assume **R** is not big with respect to **S**, so an **S** tuple does not match many **R** tuples. So $\sigma_{D,E}(\mathbf{S})$ can be the outer, enabling an INLJ with **R** as the inner, using the index on **R.C** for the probe.*

- *This can enable an inexpensive SMJ.*
 - * *If the index on **R.C** is unclustered, this would be an option when no other fields of **R** are involved, as this would need to be an index-only plan.*
 - * *If the index on **R.C** is clustered, an SMJ can be used with **R** essentially already sorted.*
- *There are other conditions on C in the query that are highly selective. The **R.C** index could be used as an access path. This can be done before the join.*

This does not affect the join chosen, per se, but potentially would help still in such queries by providing another access path.

3. (10 points) **General.** *It costs how much?!* [multiple choice]

a. (2 points) *Pipelining* saves what?

A. Space in the buffer pool because only one page of each file being used in the current operation needs to be present at any given time.

B. The I/O's of writing out a temporary result table and reading it in again when those results can be immediately used by the next operation.

C. Disk space by packing variable-length records on a page more efficiently.

D. Time by prefetching pages that might be needed into the buffer pool ahead of time.

E. I/O cost by reading and writing sequences of pages arranged sequentially on disk much faster than reading and writing them one at a time.

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

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

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

b. (2 points)

```
select distinct S.sid, R.day
  from Sailor S, Reserves R
 where S.sid = R.sid and
        R.bid = 7 and S.level = 6;
```

Estimate the selectivity of the above query as the number of tuples it likely returns.

A. 2

B. 5

C. 20

D. 50

E. 100

F. 10,000

c. (2 points)

```
select distinct S.level
  from Sailor S, Reserves R
 where S.sid = R.sid and
        R.bid = 7 and S.level <= 7 and S.level >= 3;
```

Estimate the selectivity of the above query as the number of tuples it likely returns.

A. 1

B. 5

C. 20

D. 80

E. 100

F. 10,000

-
-
- d. (2 points) An efficient way intersect, $\mathbf{R} \cap \mathbf{S}$, might be implemented is
- A. via union: $\overline{\mathbf{R} \cup \mathbf{S}}$.
 - B. via except: \mathbf{R} except (\mathbf{R} except \mathbf{S}).
 - C. as a special case of project: project just those columns in \mathbf{R} that also appear in \mathbf{S} .
 - D. as a special case of join: join \mathbf{R} and \mathbf{S} using all the attributes as the join attributes.
 - E. via aggregation.
-

- e. (2 points) Which following heuristic tends to result often in a less expensive query tree?
- A. Pushing selects and projects below joins.
 - B. Pushing joins below selects and projects.
 - C. Preferring table scans to using an unclustered index.
 - D. Making the smaller table of the two in a join the outer table.
 - E. Making the larger table of the two in a join the outer table.
-

4. (15 points) **Query Plans.** *We've got the cheapest query plans in town!* [exercise]

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

- **Reserves:** on 2,500 pages
 - 100,000 records, primary key is `sid + bid + day`.
 - **Reserves.bid:** 1,000 values
 - **Reserves.day:** 5,000 values
- **Boat:** on 100 pages
 - **Boat.bid:** 1,000 values (primary key)
 - **Boat.size:** 10 values (3..12 meters)

Table **Reserves** has a foreign key on `bid` referencing **Boats** (on `bid`). All columns are *not null*.

Available indexes are as follows. Each follows alternative two (having data-entry pages).

- A unique, clustered hash index on `bid` for **Boat** (20 data-entry pages / buckets).
- A unique, clustered B+ tree index on `sid + bid + day` (in that order) for **Reserves** (750 data-entry pages / leaves).
- An unclustered hash index on `day` for **Reserves** (250 data-entry pages / buckets).

Consider the following query.

```
select R.sid, B.bid, B.bname
  from Reserves R, Boat B
 where R.bid = B.bid and
        B.size > 6 and
        R.day = '14 November 2003';
```

a. (5 points) What is the expected number of records returned by this query?

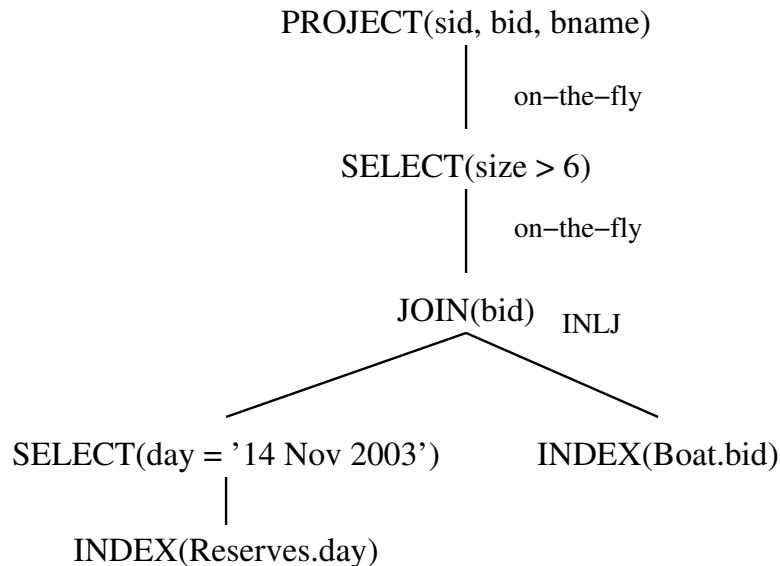
Make reasonable assumptions about reduction factors, fudge factors for accesses, etc., along the lines of Chapter 12 / System R.

*We can take a short-cut by realizing that the join from **R** to **B** is over a foreign key. So we are considering the 100,000 **Reserves** records (joined with the **Boat** information). This takes care of the join condition, **R.bid = B.bid**. The reduction factor of **B.size > 6** is 0.6 (60%), since 6...12 is 6 of 10 possible values. The reduction factor of **R.day = '14 November 2003'** is $\frac{1}{5,000} = 0.0002$ (0.02%). So the expected number of records resulting is $100,000 \times 0.6 \times 0.0002 = 12$.*

- b. (10 points) You have a buffer pool allocation of 15 frames for the query. Design the best query evaluation plan for the above SQL query that you can. Show it as a relational algebra tree. Annotate the tree with the chosen join methods, *on-the-fly*, and so forth. You should be able to design a query plan with an estimated cost better than 100 I/O's (but do the best you can).

Estimate the cost of your query plan.

Here is one query plan.



The select on **Reserves** can be done using the unclustered hash index **day**. The reduction factor for this was 0.0002, so we estimate that this would return $100,000 \times 0.0002 = 20$ records. These should fit on a single page.

The cost would be to access the directory page (1 I/O), access the data entry page (1.2 I/O's, the fudge factor for overflow pages), and then 20 data record pages (20 I/O's) since the index is unclustered. We write the single page out to a temp-file. So say 23 I/O's.

The join, an index-nested-loops join, reads in the temp-file as the outer (1 I/O). It probes the inner table, **Boat**, using the unique, clustered index on **Boat.bid**. Each probe will match just on **Boat** record since **bid** is key. Say only one directory page is involved (1 I/O), and stays in the buffer pool. Each of the 20 probes costs 1.2 I/O's, on average, to get the data entry bucket, then an I/O to get the record's page. (Clustered doesn't really help here.) So $1 + 24 + 20 = 45$ I/O's. So 46 I/O's.

The remaining select, **B.size = 6**, is done on-the-fly as is the final projections. So they add no (I/O) cost. Thus the total is around 69 I/O's.

(Scratch space.)