Instructions:
- This exam is closed book and closed notes but open "cheat sheet.
- The total time for the exam is 80 minutes, so budget your time accordingly.
- Be sure to answer each part of each question after reading the whole question carefully.
- If you don’t understand something, you may ask for clarification.
- If you still find ambiguities in a question, write down the interpretation you are taking and then answer the question based on that interpretation.
- Be sure to briefly show your work, in addition to the final answer, for any problem(s) on the exam that you might want partial credit consideration for.
- You are expected to provide numerical answers to questions that ask for costs; you are welcome to use a calculator if you would like to.
- The last page of this exam is blank; you can use it as scratch paper.

<table>
<thead>
<tr>
<th>QUESTION</th>
<th>POINTS</th>
<th>SCORE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>4</td>
<td>40</td>
<td>40</td>
</tr>
<tr>
<td>TOTAL</td>
<td>100</td>
<td>100</td>
</tr>
</tbody>
</table>

STUDENT NAME: I. M. WRIGHT  STUDENT ID: #1
Question 1: External Sorting (20 points)

Consider a relation R that contains |R|=2,000,000 fixed-length data records. Suppose that each disk page is 16KB long and that the size of a data record is |r|=800 bytes. Finally, suppose that the average disk I/O costs about 10 msec per page. (We will ignore the random vs. sequential I/O distinction to simplify our analysis here.) You are to analyze the I/O cost (in msec) for efficiently sorting R using external merge-sort given an allotment of W=100 pages of working memory. (Note: Don’t forget to account for input and output buffers in your analysis.)

(a) (4 points) What is the initial file size of R, on disk, in pages?

\[
\frac{16,384}{800} = 20 \text{ RECS/PG} \Rightarrow 100,000 \text{ PGS}
\]

(b) (4 points) After the initial sorting pass of the external sort operator has generated its initial set of sorted subfiles for R:

(i) How many sorted subfiles will there be? (Show your work.)

**First pass reads & sorts data in 100-PGS CHunks, writing each chunk out to a sorted subfile.**

\[
\frac{100,000 \text{ PGs}}{100 \text{ PGs}} = 1,000 \text{ subfiles}
\]

(ii) What will each of the subfiles’ sizes (in disk pages) be? Try to be exact; remember that files on disk can only contain integral numbers of pages.

(See above.)

100 PGS

(integral # handled by Int(above, n) (a).)

SCORE: 8
(c) (4 points) Next, after the first merge pass of the external sort has finished its work:

(i) How many sorted subfiles will there be now? (Show your work.)

\[ \frac{1,000 \text{ subfiles}}{99} = 11 \text{ subfiles} \]

(ii) What will each of the subfiles' sizes (in disk pages) be now? Again, be exact.

All but one will be \( 99 \times 100 = 9,900 \) pages long, and the last will have what's left or \( 100,000 - 99,000 = 1,000 \) pages, so:

\[ 10 \text{ subfiles of 9,900 pages} + 1 \text{ subfile of 1,000 pages} \]

(d) (8 points) The external sort will continue until \( R \) is completely sorted.

(i) How many more passes (i.e., in addition to the two that you've already analyzed) will it take to finish merging \( R \) down into a single sorted result file?

We can merge <= 99 ways at a time and only have an 11-way merge left, so:

\[ 1 \text{ pass required} \]

(ii) What will the overall I/O cost be for starting with the unsorted relation \( R \) on disk and producing a sorted tuple stream for \( R \)? (Don’t include the cost to write the final result to disk; assume that the sort operator is feeding the next operator downstream in an iterator-based query execution pipeline. (Be sure to show your work as well as giving the final answer in msec.)

Pass 1 (Sort): \( (R + U) \text{ 100,000 pages} \)
Pass 2 (Merge): \( (R + U) \text{ 100,000 pages} \)
Pass 3 (Final Merge): \( R \text{ 100,000 pages (left to merge)} \)

Total I/O count: \( 200,000 \text{ R + 200,000 U} = 500,000 \text{ I/O's} \) (each 10 msec)

\[ \therefore \text{ Total I/O cost in msec} = \frac{5,000,000 \text{ msec}}{5 \text{ sec}} = 1000 \text{ sec} \]

SCORE: 12
Question 2: Hash Joins (20 points)

Consider again relation R with |R| = 2,000,000 records. A second relation S contains |S| = 10,000 records. The size of a disk page is 16KB and the average size of the relations' records is |r| = |s| = 800 bytes. A disk I/O costs 10 msec per page on average. (We will again ignore random vs. sequential I/O for simplicity.) Compute the I/O cost (in msec) for efficiently joining R and S using the Grace hash join algorithm (from lecture or the Shapiro paper) given a working memory allotment of W = 100 pages. Recall that the Grace algorithm first partitions each relation and then joins the results. You may assume that the hash function is perfectly random, spreading each tables' data evenly across the partitions, and you can use F = 1.0 for the Shapiro “fudge factor” for simplicity. (Note: Don’t forget to account for input and output buffers in your analysis.)

(a) (4 points) What is the initial file size of S, on disk, in pages? (You already answered this for R.)

\[
\frac{10,000 \text{ RECS}}{20 \text{ RECS/PG.}} = 500 \text{ PGS.}
\]

(b) (4 points) At the end of the partitioning phase for R:

(i) How many partitions of R will there be? (Explain or show your work.)

**With one input buffer and 99 output buffers, the initial partitioning will yield:**

**99 partitions**

(ii) What will each of the R partitions' sizes be (in disk pages)? Be exact, and recall that a file on disk can only contain an integral number of pages (some of which might be only partly filled).

Since R has 100,000 PAGES, we'll have:

\[
\frac{100,000 \text{ PAGES}}{99} = 1,011 \text{ PAGES}
\]

(Their sizes will vary slightly but we'll assume perfectly normal and ignore that.)

**Note:** R's partitions won't fit in memory but if S's will - and they will - we've done partitions R.

**Score:** 8
(c) (4 points) At the end of the partitioning phase for S:

(i) How many partitions of S will there be? (Show your work.)

\[ S \text{ must have the same number as } R \text{ does! (THAT IS HOW HASH JOINS WORK.) So:} 99 \text{ partitions} \]

(ii) What will each of the S partitions' sizes be (in disk pages)? Be exact, and recall that a file on disk can only contain an integral number of pages (some of which might be only partly filled).

\[ \left\lfloor \frac{500 \text{ pages}}{99} \right\rfloor = 6 \text{ pages} \quad \text{(which easily fit in memory)} \]

(d) (8 points) Once R and S have each been partitioned, the joining phase can proceed.

(i) Which one of tables R and S should be the build table (the one whose partitions are loaded into memory) versus being the probe table during the joining phase?

\[ \text{Build table} = S \quad \text{(which fits)} \quad \text{Probe table} = R \quad \text{(which does not)} \]

(ii) What will the overall I/O cost be for the join if you are starting with R and S as tuple streams, not on disk, and it is producing a joined result stream? I.e., don't count: the cost of initially reading R or S or the cost to write the final result tuples to disk – assume that the join is being fed its inputs from other relational operators in a bigger query plan and that it is similarly feeding its results into the next operator in an iterator-based execution pipeline. (Be sure to show your work as well as giving the final answer in msec.)

\[
\begin{align*}
\text{Partitioning Phase for S:} & \quad 6,000 \text{ pages} \\
\text{Partitioning Phase for R:} & \quad 100,000 \text{ pages} \\
\text{Build Phase for S:} & \quad 500 \text{ pages} \\
\text{Build Phase for R:} & \quad 100,000 \text{ pages} \\
\text{Total I/O count:} & \quad 100,500 \text{ R + 100,500 W = 201,000 I/O's (each 10 msec)} \\
\therefore \text{Total I/O cost in msec =} & \quad 2,010,000 \text{ msec or} \\
& \quad 2,010 \text{ sec}
\end{align*}
\]

Score: 18
**Question 3: Query Optimization (20 points)**

This class has been so much fun that you can’t bear to see it stop! You have decided to write a query optimizer for your project code base that uses a System-R (*a.k.a* Selinger) style query optimization algorithm to efficiently explore the space of possible query plans. Not wanting to be accused of being stuck in the 70’s, however, you are basing your cost estimation approach on equi-height histograms. Suppose you have the following tables and statistical information:

- **Table Emps** (ssnum, name, age, deptid, jobid, gender): primary key ssnum, foreign keys deptid (referencing Depts) and jobid (referencing Jobs), unclustered indexes on ssnum, name, age, deptid, and jobid. There are 10,000 employees with 9,900 names, a 50-50 mix of males and females, and 2,000 employees each in the age ranges: 1-25, 26-30, 31-40, 41-45, and 51-100.
- **Table Depts** (deptid, dname, floor, budget): primary key deptid, clustered index on deptid, unclustered index on dname, unclustered index on floor.
- **Table Jobs** (jobid, descrip, lowsal, highsal): primary key jobid, unclustered index on jobid, clustered index on jobid.

Consider the following SQL query:

```sql
select E.ssnum, E.name, E.age, D.floor, J.descrip
from Emps E, Depts D, Jobs J
where E.deptid = D.deptid and E.jobid = J.jobid
  and E.age > 20 and E.age <= 35 and E.gender = 'female'
order by D.floor;
```

(a) (5 points) List the interesting orders for each of the tables involved in this query:

- **Emps**: deptid, jobid
- **Depts**: deptid, floor
- **Jobs**: jobid

(b) (10 points) Estimate the number of employee tuples that will satisfy the employee selection criteria. (Briefly show how you arrived at your estimate underneath your numerical answer.)

```
| Emps where E.age > 20 and E.age <= 35 and E.gender = 'female' | = 1700
```

(c) (5 points) Which pair of tables will your new optimizer **not** consider joining? Why?

**Score: 20**
Question 4: Query Execution Choices (40 points)

You still can’t get enough of this project! Your plan is to continue working on your CS122c/CS222 relational DBMS until it has more features than you know what to do with. Another quarter goes by, but luckily one of your friends intervenes and stops you just before you reach that point – so your system now has the perfect number of features, and in this final problem, you will show that you do know what to do with them. Given the following query execution strategies and a database with the indicated tables and other properties, you are to indicate – for each of the queries below – and given 90 pages of buffer space for processing them – which query execution strategy would likely be best for the given query and why.

For storage, assume that the tables are stored as heap files, as they are in your project now, but that you have added support for both unclustered and (best-effort) clustered B+ tree indexes. (Try to answer each question using only the information that is given here, but if something is missing that you absolutely need to assume, just be sure to state that assumption.)

Database information:

- Table Listings (lid, uid, pid, price, descrip, photo, ...): 2M pages of data, 10 tuples/page, primary key lid, unclustered index on lid, clustered index on uid (foreign key for Users), unclustered index on pid (foreign key for Products), unclustered index on price (ranging from $0-$20M).
- Table Users (uid, uname, email, rating, gender, ...): 200K pages of data, 10 tuples/page, primary key uid, clustered index on uid, unclustered index on uname, unclustered index on gender.
- Table Categories (cid, cname, descrip, ...): 100 pages of data, 50 tuples/page, primary key cid, unclustered index on cid, clustered index on ctime.
- Table Products (pid, cid, pname, descrip, manuf, ...): 10K pages of data, 10 tuples/page, primary key pid, unclustered index on pid, clustered index on cid (foreign key for Categories), unclustered index on pname, unclustered index on manuf.

Available query execution strategies:

1. Table scan (say of which table)
2. Clustered index scan (say of which index) and then fetch data
3. Unclustered index scan (say of which index) and then fetch data
4. Index-only scan (say of which index)
5. Grace hash join
6. Simple hash join
7. Sort-merge join
8. Basic (tuple) nested-loop join
9. Block nested-loop join
10. Nested-loop index join

The following are the queries of interest. For each one, indicate the best choice of strategy for executing the given query and briefly note (in 1-2 sentences) why you think that strategy will be best. (For a join query, indicate the choice of strategies for accessing each of the tables as well as the choice of strategy for performing the join itself.)

SCORE: 


(a) (4 points) select * from Listings L where L.lid = 411;
Strategy: UNCLUSTERED INDEX SCAN (Listings.lid) + ACCESS DATA
Reason: FAST FOR SINGLE-RECORD (PREY) LOOKUP

(b) (7 points) select * from Products P, Categories C where P.cid = C.cid
Strategy: (i) SIMPLE HASH JOIN OF Categories + Products TABLE SCANS
(ii) INDEX-NULL JOIN OF Categories TABLE SCAN PROBING Products, cid
     CLUSTERED INDEX
(iii) SM-JOIN USING SORT (Categories TABLE SCAN) AND MERGING
     WITH PRE-SORTED Products, cid CLUSTERED INDEX SCAN.
Reason:
   (i) Categories ALWAYS FIT IN MEMORY => SIMPLE HASH JOIN Good.
   (ii) CLUSTERED Products, cid INDEX => EFFICIENT Products ACCESS.
   (iii) Categories SMALL, Products,cid Gives PRE-SORTED ACCESS + SORT, INDEX (Good)

(c) (5 points) select * from Users U where U.gender = 'female';
Strategy: TABLE SCAN of Users
Reason: GIVEN 99% FEMALES AND NO CLUSTERING IT WOULD BE
BAD TO USE AN UNCLUSTERED INDEX TO ACCESS WHAT WILL
TURNS OUT TO BE ALL USERS AS ANYWAY!

(d) (7 points) select * from Users U, Listings L where U.uid = L.uid;
Strategy: SORT-MERGE JOIN OF Users + Listings USING THEIR
     CLUSTERED UID INDEXES TO ACCESS TABLES IN SORTED ORDER
Reason: ONLY HAVE TO MERGE USERS + LISTINGS SINCE WE'VE
     GOT CLUSTERED/SORTED ACCESS VIA THEIR INDEXES.
(e) (5 points) \textbf{select * from Categories C where C.cname like 'Music%';}

\textbf{Strategy:} \texttt{CLUSTERED INDEX \texttt{CNAME} (Categories, C.cname)} + \texttt{ACCESS \texttt{LUST}}.

\textbf{Reason:} Since "Music" is a prefix, this is very much like a smallish range query on \texttt{Categories.cname} - so the index is the way to go to minimize I/O.

(f) (5 points) \textbf{select avg(L.price) from Listings L;}

\textbf{Strategy:} \texttt{INDEX-ONLY \texttt{SOB} OF \texttt{LISTINGS.price INDEX}}.

\textbf{Reason:} Everything needed to answer the query is right in the index - no data access needed. (And the index has many fewer rows than the data file.)

(g) (7 points) \textbf{select * from Listings L, Users U where U.uid = L.uid and U.uname = 'Ripoff, Robbie';}

\textbf{Strategy:} \texttt{NESTED LOOP INDEX JOIN using the result of doing an index scan on \texttt{INDEX-SO} of \texttt{LISTINGS and joining on \texttt{LISTINGS.uid}} \texttt{AND \texttt{USERS.uname}} to \texttt{find Robbie's listings}}

\textbf{Reason:} There will presumably be few outer tuples, and we'll fetch only the matching inner tuples this way. (We avoid reading most of both tables!)

\textbf{SCORE: 17}