CS 564 Midterm Exam

Open book, open notes, open internet – closed friends and neighbors

If you find a question ambiguous or difficult, please do not ask other students in the class. Instead, please post your questions on Piazza. Please do not post answers to other students' questions – please leave that to the instructor and the TAs. Feel free to post follow-up questions if you find an answer ambiguous. Before you post a question, please check whether a similar question is already posted and perhaps even answered.

Please submit your exam as a single pdf file by **Friday, November 1, 12:00PM noon** (not midnight) on Canvas. Make sure to submit before the deadline-- late submissions will not be accepted.

Unless you have very readable handwriting, please type your answers. If we can't read it, we can't credit it. For some questions, please draw by hand and label by hand, but make sure it's all readable – then scan it or photograph it and include it in your one pdf file.
For questions 8-11, please be concise and use short, succinct sentences. 30-100 words should suffice. Do not copy from the book, from Wikipedia, or from other internet sources.

Starting on the next page, answer each of the following questions or assignments:

1. Draw an ER diagram for a (much simplified) database for a lending library. Map the following concepts to entities, relationships, and attributes: book, title, author(s), borrower, loan date (when the book was borrowed from the library), due date, patron (a person who may borrow books), name, address, phone (some attributes apply to both the library and to each patron). You may add a few attributes but keep that to a small number. For better or worse, there may be books with the same title, but perhaps by different authors. A patron may borrow the same book multiple times.

The important points here are to add identifiers to persons (borrower, patron) and to books; to model loans as a many-to-many relationship "loan"between entity "person" and entity "book;" and to model authors either as a many-valued complex attribute of books or as another many-to-many relationship to another entity "authors." Loan date and due date are attributes of the "loan." Correct author lists require a position attribute.

2. Map your ER diagram to a schema (table names, column names, primary and foreign keys). You can use the TPC-H schema as an example, but make sure you clearly indicate the primary and foreign keys.

Patron (<u>SSN/id</u>, name, address, phone)
Book (<u>ISBN</u>, title)
Loan (<u>SSN/id, ISBN</u>, loan date, due date)
Author (<u>SSN/id</u>, name, …)
Authorship (<u>SSN/id, ISBN, position</u>)

3. Write "create table" statements for these tables – include all integrity constraints that make sense. Try to include each kind of integrity constraint at least once in the database. Each table should have a primary key and each table should include a foreign key or be referenced by a foreign key.

```
create table Patron (identifier integer primary key, name varchar(30), address varchar(30), phone numeric (10))
create table Book (ISBN char(12) primary key, title varchar (50))
create table Loan (identifier integer, ISBN char(12), loan_date datetime, due_date datetime,
        primary key (identifier, ISBN),
        foreign key (identifier) references Patron, foreign key (ISBN) references Book)
```

```
create table Author (identifier integer primary key, name varchar(30), …)
create table Authorship (identifier integer, ISBN char(12), position integer,
        primary key (identifier, ISBN),
        foreign key (identifier) references Author, foreign key (ISBN) references Book)
```

4. Write the following queries over your tables. Feel free to use "with" clauses where convenient; use nested queries sparingly.
    a. List all book titles in ascending order – include the first author's name for each book, and include duplicate titles.

```
select b.title, a.name
from Book as b, Authorship as ba, Author as a
where b.ISBN = ba.ISBN and ba.identifier = a.identifier and ba.position = 1
order by b.title asc
```

    b. Count the number of books that the library owns.

```
select count (*) from Book
```

    c. For each book borrowed at least once, list the title, the number of times it has been borrowed, and the most recent due date (whether that due date is in the past or in the future).

```
select b.ISBN, b.title, count (*) as loan_count, max (l.due_date) as recent_or_future_due_date
from Book as b, Loan as l
where b.ISBN = l.ISBN
group by b.ISBN, b.title
```

5. For the last query in question 4, draw or write an expression in relational algebra. (As we did in class, assume the algebra includes a grouping/aggregation operation in addition to the usual select, project, and join operations).

```
grouping (by b.ISBN, b.title; compute loan_count := count (*), recent_or_future_due_date := max (l.due_date))
        join (b.ISBN = l.ISBN)
                Book as b
                Loan as l
```

6. Map your relational algebra expression to a query execution plan (draw or write).

```
hash agg (group by b.ISBN, b.title; compute … (as in relational algebra))
        hash join (b.ISBN = l.ISBN)
                scan Book as b
                scan Loan as l
```

7. Provide an alternative query execution plan (draw or write).

```
merge join (b.ISBN = l.ISBN)
        scan Book as b (using an index ordered on b.ISBN)
        in-stream agg (group by l.ISBN; compute … (as above))
                scan Loan as l (using an index ordered on l.ISBN)
```

8. Decide which one of your two alternatives is likely more efficient – state your choice and give reasons for your choice, or state that you can't decide and give reasons why either plan might be more efficient.

The second alternative seems more efficient in terms of CPU because data reduction (by aggregation) occurs early;

as well as in terms of memory because the plan exploits the sort order of the storage structures rather than building in-memory indexes (hash tables).

9. Explain (in your own words) the differences between an inner join, a (left) semi join, and a (left) outer join. Your explanation should address the sets of columns and the sets of rows in each operation's output.

All three operations consume two input tables and produce one output table. They all look for matching pairs of rows satisfying a join predicate. Inner join and outer join produce all input columns from both tables; left semi join produces all input columns from the left table only. A semi join selects rows from one table that have (one or more) matches in the other table; an inner join produces all matching pairs of rows; and a (left) outer join produces the same as the inner join plus non-matching left rows (also known as left anti semi join) with the output columns for the right join input filled with *null* values.

10. Explain (in your own words) the differences between index nested loops join, merge join, and hash join (all used to compute an inner join). Your explanation should include required properties of the join inputs and properties of the join output. For the hash join, you may assume that one join input fits in memory. For extra points, explain what happens if neither join input fits in memory.

This answer is much longer than was expected – but it attempts to list differences of various kinds.
All three algorithms compute join result from two inputs. Whereas naive nested loops join scans its entire inner input table for each row of its outer input, an index nested loops join searches its inner input with an index. Merge join exploits the sort order of both inputs for a join algorithm similar to a merge step in an external merge sort. Hash join builds an in-memory hash table on one input and probes this hash table with rows of the other input.
Index nested loops join supports any join predicate that can be mapped to an index search, whereas merge join and hash join require at least one equality predicate (to guide the merge logic or to compute hash values).
Index nested loops join requires an index on its inner input; this index could be temporary (for a single query only) but it usually is a permanent index. The performance of index nested loops join may benefit from sort order in its outer input, but sort order is not required for correct join results. A merge join requires that both inputs be sorted on join columns; one or two explicit sort operations may be required. A hash join can process inputs without sort order and without indexes.
Index nested loops join produces output sorted like the outer join input; merge join produces output sorted on the join keys just like its inputs; and hash join preserves the sort order of the outer input if (and only if!) the build input fit in memory and no overflow occurred.
Index nested loops join pipelines its outer input – the first output is produced as soon as the first match is found. Merge join pipelines both inputs, but a sort operation (if required) is a pipeline breaker. Hash join terminates the pipeline that produces the build input but in-memory hash join pipelines its probe input.
Merge join and hash join require very little extra logic for semi join, anti semi join, and all forms of outer joins; naive and index nested loops join can readily compute left semi join, left anti semi join, and left outer join, but the right and full variants require tracking matches of inner rows.
Index nested loops join works for any input size; it is superior to the other join algorithms if the outer input is so small that only a fraction of the inner input is ever required for the join.
Merge join works for inputs of any size (except for vast numbers of duplicate join key values in both inputs); a sort operation might require an external merge sort. It is superior to the other join algorithms if both inputs or at least the larger input is readily available in a suitable sort order.
Hash join works best if the build input fits in memory; if not, both inputs are partitioned to temporary storage. Each partition is a pair of overflow files (for build and probe input). Joining each pair of overflow partitions computes the entire join result. Extreme cases require recursive partitioning.

11. For extra points, explain (in your own words) the differences and similarities between hash aggregation and hash join. For the hash join, you may assume that one join input fits in memory. For extra extra points, explain differences and similarities if neither input nor output fit in memory.

The obvious difference is one vs two input tables. There is also a hash-based query execution algorithm that performs grouping (aggregation or duplicate removal) on its first input before joining with a second input – let's

ignore that variant here.
Hash aggregation and hash join consume unsorted input and require memory for an in-memory hash table.
The algorithms are quite similar; the candidate output rows of hash aggregation serve a role quite similar to the build input of hash join. A hash join requires build and hash phases; some outer joins and anti semi joins require a third phase scanning the hash table. Hash aggregation combines build and probe phase into one but requires a third phase (producing output from the hash table).
In case of memory overflow, hash join employs a pair of files for each partition; hash aggregation can employ a pair of overflow files (one for partially aggregated rows, e.g., with counts and sums, and one for input rows); or hash aggregation can employ a single overflow file per partition (after mapping input rows to partially aggregated rows with count = 1, sum = value, etc.).