Assignment III: Query Optimization

In this assignment, you will extend the code generator from Assignment II to include some query optimization techniques. The ultimate goal in this assignment is for your implementation to efficiently support table joins. This assignment is divided into the following steps:

1. Supporting more SELECT statements
2. Using indexes
3. Pushing \( \sigma \)'s
4. Simple heuristics

**Step 1: Supporting more SELECT statements**

In Assignment II, you were only required to support SELECT statement that selected “a list of columns (i.e., you don’t need to support SELECT * ) from a single table and with, at most, one condition in the WHERE clause. The condition will be of the form field op value”. Your code generator must now be able to support the following:

- Selecting a list of columns from multiple tables.
- Supporting multiple conditions in the WHERE clause (of the form field op value and field = field)

In both of the above you are allowed to assume that columns in the SELECT and WHERE clause are always specified in the table.column format. When selecting from multiple tables, you can use a brute-force algorithm that traverses every possible row of the cross product of the tables (and tests the conditions on every row). Note that this algorithm is not just a stopgap measure; the query optimizer might still have to use it if it determines that no optimizations are possible.

A complete implementation of this step should also include the following:

- SELECT *
- Remove the assumption that columns are always specified in the table.column format.

However, note that you will be able to implement the remaining steps without the above two features.

**Step 2: Pushing \( \sigma \)'s**

Refine the algorithm from Step 1 so it will "push \( \sigma \)'s". In other words, instead of testing all conditions on every possible row, test the conditions that affect a table when iterating through that table.

**Step 3: Using indexes**

Improve your code generator so that indexed fields (including the primary key) are accessed through the index, instead of doing a linear search of the table. This applies not just when testing conditions of the form field = value but also conditions of the form field = field when performing equijoins.
Step 4: Simple heuristics

When evaluating queries that involve equijoining multiple tables, we have thus far not specified in what order those tables are going to be inspected (or, in other words, how the loops iterating over each table are going to be nested). You must implement a simple heuristic that orders the nested loop by increasing cardinality of the tables (i.e., the table with the least number of rows is in the outermost loop, and the table with the largest number of rows is in the innermost loop). Note that table size is not stored anywhere in libchidb, so you must modify your in-memory schema to include this information.