COSC 420 - Group Project 2
Parallel Database Query Processing System

Dr. Yaping Jing

Either from your first database class or in our daily lives, we learn how to query a database in more or less technical ways, if a database query processing system is provided. For this project, instead of assuming you're given such a system, you are asked to design and implement a simplified version of database query processing system using the parallel technology that you learned in this class. To develop such a system, you need the following two components for the bare minimum:

1. **Database Construction:**

   For simplicity, you are given a table schema with all the attributes and `ID` as primary key (see the `CarInventory` table next page). Unless you want to design your own, you can use this single table, ideally with at least a million tuples of data when it comes to performance test. You should write a separate C program (save it as `dataGen.c`) such that:

   - It can generate n tuples of data, where $n > 0$ and n can be as big as possible.
   - The table should have at least six attributes (as we're not going to implement complex operations such as join or nested query).
   - It must have a `primary key` that can "uniquely identify a tuple". In the `CarInventory` example, ID is the primary key, which means for the same values of ID, there can be no two different tuples. Your program must ensure this property when generating all the tuples.
   - For convenience, you must implement a function that can print all the tuples, including all the attributes (i.e., the column headers) onto the console (not the file). This function will be used when debugging your data.

   Normally, for a structured database, you would need to create a schema, then populate the table with all the data, and rely on the system to ensure that the primary key's property is preserved. In this case, you must make sure that the property of the primary key is preserved when generating all the tuples.

CarInventory

ID	Model	YearMake	Color	Price	Dealer
1000	Accord	2008	Gray	$ 10, 000	Pohanka
1001	Corolla	2015	White	$ 18, 000	AutoNation
1012	Civic	2015	Blue	$ 18, 000	Mitsubishi
1013	Accord	2015	Green	$ 12, 000	Sonic
1014	Accord	2020	Gray	$ 20, 000	Suburban
1015	Maxima	2016	Red	$ 17, 000	Atlantic
1016	Civic	2021	Red	$ 21, 000	Ganley
1017	Focus	2015	Red	$ 12, 000	Victory
1018	Camry	2015	White	$ 18, 000	Atlantic
1019	Maxima	2018	Gray	$ 15, 000	Pohanka
1020	Accord	2015	Blue	$ 16, 000	Pohanka
1021	Accord	2020	Blue	$ 22, 000	GM

2. **Query Processing Engine:**

For this component, you'll implement a data structure (try Struct) to store query details, and "functions" to parse SQL-like queries. Again, for simplicity, your functions are only required to interpret and process the basic type of SQL queries as illustrated below:

```
SELECT [attribute1, attribute2, ...]
FROM [table name]
WHERE [conjunctive or disjunctive conditions];
```

That is, the `SELECT` clause is followed by a list of attributes, each is separated by comma; `FROM` clause is to be followed by a table name, and `WHERE` is a conditional clause that can contain `AND`, `OR`, =, ! =, and all the inequality comparison operators `>, >=, <, <=`. At the minimum, your query processing system shall handle the following query examples.

```
SELECT ID, Model, YearMake, Color
FROM CarInventory
WHERE Model="Accord" AND YearMake="2015"
         AND (Color="blue" OR Color="gray");


SELECT ID, Model, YearMake, Color
FROM CarInventory
WHERE Model="Civic" AND YearMake="2021" AND (Color="blue" OR Color="red");
```

Given the above table of data, the query results should be

ID	Model	YearMake	Color
1020	Accord	2015	Blue

ID	Model	YearMake	Color
1016	Civic	2021	Red

To summarize, you need two input files: one is a big table of data in plain text (say `db.txt`) that is generated from your C program; another is your SQL queries also in plain text (say `sql.txt`). Your program flow should be:

(a) Load the table (`db.txt`) from the text file into the memory;

(b) Load the SQL queries from the text file into the memory;

(c) Parse each of those queries (Hint: write a function to parse the query). For this project, you can assume the same query type as the example given above, since a generic parser is not the focus of this project;

(d) Execute all the queries and print the corresponding results (Hint: write a function to process the query)

(e) Print the run time.

Note that you're certainly not supposed to connect to any sql server, nor should you use any sql library, since you're implementing a piece of sql processing engine. Programming Hint: you need one structure (such as `struct CarInventory`) to define the table attributes and another structure (such as `struct Query`) to define the query. To parse a query, it's easier just to use `sscanf`.

- **Thoughts of Parallelization and Data Structure:** Ideally, you should have two dimension of parallelism. The first dimension is to have multiple queries running in parallel. The second dimension is that for each query, you can apply the parallel technique, which we'll discuss in class. Further, in real database, it uses `B-tree` (`https://github.com/tidwall/btree.c`) as data structure for faster performance. In this case, you may try to construct a BST or AVL (balanced BST) for all the generated data (each node contains a row of data), before processing those queries. If you do use B-tree to implement the program, I may consider some extra credits depending on your overall program results (correctness, robustness, and efficiency).

- **Required Implementations and Documentation for the Query Processing Engine (QPE):**

  1. Implement a Serial version `QPESeq.c`.
  2. Implement a Parallel version using OpenMP `QPEOMP.c`.
  3. Implement a Parallel version using OpenMPI `QPEMPI.c`.
  4. For each of the parallel versions (OMP and MPI, respectively), discuss runtime analysis, and answer the following questions (save it as `Proj2Doc.pdf`):
     (a) What's the speed up and efficiency based on Amdahl's law for the different number of processors (for MPI) and the number of threads (for OMP), respectively? (You may give tables as well as plots)
     (b) Based on the above data, what's your conclusion regarding the optimal number of processors (for MPI), and the optimal number of threads (for OMP) based on the Amdalh's law?
     (c) If you increase your program size proportionally to the increase of the number of processors (threads respectively), how does your program scale in terms of runtimes? You may give a table that lists problem size, number of processors (threads, respectively), and runtime.
  5. Please provide a **readme** in text file that instructs:
     - How to compile and run your programs (including how to generate the data).
     - Specific files submitted;
     - Each member's contribution (who did what tasks);
     - Give a brief discussion how you ensured your program's correctness while keeping performance on the top.

  Submission: please submit all the files, one copy each group on MyClass. To summarize, the file list should include:

  1. `dataGen.c`
  2. `QPESeq.c`
  3. `QPEOMP.c`.
  4. `QPEMPI.c`.
  5. `Proj2.pdf`
  6. A sample data file `db.txt`.
  7. A sample query file `sql.txt`.
  8. `Readme.txt`

  Presentation Requirements: Each group will present in terms of the following points:

1. Live Demo (smaller problem size for correctness, larger problem size for performance, streamlined execution, user-friendly output, etc.)
2. What have been implemented successfully, what have been tried but not successful (if any).
3. What performance results achieved relative to the problem size compared among serial, distributed parallel, and shared memory parallel systems.
4. What did you learn.
5. Discuss team work (each member shall discuss their contribution)