CSE177/EECS277 – DATABASE SYSTEMS IMPLEMENTATION

Project 2: Query Compiler

Due date: March 1 & 2 (in the lab)

This project requires the implementation of the query compiler. A database query compiler takes as input a SQL query and generates as output an execution tree consisting of relational algebra operators. To accomplish this task, it makes heavy use of the database catalog in order to identify table and attribute properties. There are two main components in the query compiler: the query parser and the query optimizer.

Query Parser

The query parser transforms the text of a SQL query into an abstract representation that extracts the defining elements required for the execution of the query. Such elements are represented by the tables appearing in the query, the attributes, the predicates, the aggregates, etc. As with any programming language parser, there are two components in a SQL query parser: the lexer and the parser. The lexer employs lexical analysis over the SQL text in order to identify the language tokens, such as keywords, e.g., SELECT, FROM, WHERE, etc., names of tables and attributes, and constants. The parser enforces that the SQL query obeys the syntax or structure of the language, e.g., SELECT-FROM-WHERE is a correct query, while SELECT-WHERE is not correct.

You do not have to implement a SQL query parser from scratch because we have already implemented one for you. The query parser is implemented using the well-known packages lex (flex) and yacc (bison). This parser supports a subset of SQL, given by the following rules:

- SELECT SelectAtts FROM Tables WHERE AndList
- SELECT SelectAtts FROM Tables WHERE AndList GROUP BY Atts

where only AND conditions involving at least one attribute are allowed in the WHERE clause. The only operators supported in WHERE predicates are <, >, and =. As in standard SQL, Tables is a list of tables (no aliases are allowed, thus a table can appear only a single time) and Atts is a list of grouping attributes. SelectAtts is defined as follows:

- SelectAtts := Function ',' Atts | Function | Atts | DISTINCT Atts

where Function is SUM of any arithmetic expression involving attributes and constants and Atts is a list of attributes. While three types of attributes are supported in the system (Integer, Float, String), Function can operate only over Integer and Float.

The exhaustive list of tokens supported by the language is given in file QueryLexer.l, while all the syntactical constructs are included in file QueryParser.y. It is recommended that you familiarize with these files and not change them for the time being. There is plenty of documentation available online if you search for *"lex"* and *"yacc"* in Google.

The output of the parser is composed of multiple lists populated with the principal elements of the query. They are defined in QueryParser.y (the type of these lists is defined in file ParseTree.h) and included below for reference:

```
struct FuncOperator* finalFunction; // the aggregate function
struct TableList* tables; // the list of tables in the query
struct AndList* predicate; // the predicate in WHERE
struct NameList* groupingAtts; // grouping attributes
struct NameList* attsToSelect; // the attributes in SELECT
int distinctAtts; // 1 if there is a DISTINCT in a non-aggregate query
```

For this phase of the project, these will represent your only interaction with the query parser. You do not need to worry about the details of how everything is implemented and how it works. Nonetheless, feel free to get your hands dirty and figure out all the details.

Query Compiler

The query compiler takes as input the data structures produced by the parser and transforms them into a query execution tree of relational algebra operators. While it is the job of the compiler to figure out and configure the necessary operators appropriately, the query optimizer is responsible for determining the structure of the tree, specifically the order of the joins. File `RelOp.h` contains the definition of all the relational operators considered in this project, together with their member attributes. The main part of this project stage consists in the creation of instances of these operators from the data structures produced by the query parser. Essentially, you have to implement the constructor of each operator class and configure it with the corresponding elements, extracted from the data structures produced by the parser. In the following, we describe the elements of each type of operator:

- `Scan (Schema& _schema, DBFile& _file)` manages the access to the data file. There is a `Scan` instance for every base table in the `SQL` query. It has to be initialized with the schema of the records in the file and a `DBFile` object that provides access to the records. Both these can be instantiated from the metadata in the catalog.

- `Select (Schema& _schema, CNF& _predicate, Record& _constants, RelationalOp* _producer)` implements the selection relational algebra operator. It has to be initialized with the schema of the records taken as input, the selection predicate, the constants appearing in the predicate, and the operator producing the tuples. Class `CNF` in file `Comparison.h` implements functionality to handle predicates from `WHERE`. A `CNF` object consists of one or multiple `Comparison` objects connected by `AND`, where a `Comparison` is a predicate of the form A `op` B. `op` can be one of <, >, and =. At least one of A and B has to be an attribute in `_schema`. The `Record _constants` contains all the constants appearing in comparisons in the `CNF`. Method `ExtractCNF (AndList& parseTree, Schema& schema, Record& literal)` in class `CNF` extracts the `CNF` and constants corresponding to a given schema from the representation of the `WHERE` clause produced by the parser.

- `Project (Schema& _schemaIn, Schema& _schemaOut, int _numAttsInput, int _numAttsOutput, int* _keepMe, RelationalOp* _producer)` implements the projection operator. It has to be configured with the schema of the input records, the schema of the output records, the attributes to be kept, and the producer operator. The projected attributes are specified by their position in the input schema. They are grouped in the array `_keepMe` of size `_numAttsOutput`.

- `Join (Schema& _schemaLeft, Schema& _schemaRight, Schema& _schemaOut, CNF& _predicate, RelationalOp* _left, RelationalOp* _right)` implements the join operator. It has to be configured with the schemas and the producing operators, the output schema (which is just the union of the input schemas), and the join predicate (represented as a `CNF`). In this case, each `Comparison` in the `CNF` has to contain one attribute from the left schema and one attribute from the right schema. Method `ExtractCNF (AndList& parseTree, Schema& leftSchema, Schema& rightSchema)` from class `CNF` extracts the join predicate corresponding to two schemas from the `WHERE` clause.

- `DuplicateRemoval (Schema& _schema, RelationalOp* _producer)` implements the `DISTINCT` operator. It has to be initialized with the schema of the input records and the producer operator.

- `Sum (Schema& _schemaIn, Schema& _schemaOut, Function& _compute, RelationalOp* _producer)` implements the aggregate operator. The schema of the input records, the schema of the output record, the function applied to attributes in the input records, and the producer operator have to be supplied. The output schema consists of a single attribute `sum`. Class `Function` in file `Function.h` implements the functions that can appear in the `SELECT` clause. Method `GrowFromParseTree (FuncOperator* parseTree, Schema& mySchema)` creates a `Function` object from the representation produced by the query parser.

- `GroupBy (Schema& _schemaIn, Schema& _schemaOut, OrderMaker& _groupingAtts, Function& _compute, RelationalOp* _producer)` implements the `GROUP BY AGGREGATE` operator. The output schema contains the aggregate attribute sum on the first position, followed by the grouping attributes. sum is computed based on Function. Class OrderMaker in file Comparison.h defines the grouping attributes. OrderMaker objects can be built from a schema and the positions of the attributes to group on (see the constructor OrderMaker(Schema& schema, int* atts, int atts_no)).

- WriteOut (Schema& _schema, string& _outFile, RelationalOp* _producer) is an operator for printing the output to a file, passed as parameter. It has to know the schema of the records and the output file (which can be an arbitrary file). Class Record contains a method print that you are supposed to use.

How to build the query execution tree? We follow a bottom-up approach, starting from the Scan operators and going up to the root of the tree. Class QueryExecutionTree in file RelOp.h is in charge of managing the query execution tree. This is done exclusively through the root of the tree. A Scan operator is created for every table in the FROM clause. Next, a Select operator is created for every Scan operator for which there exists a non-empty CNF, i.e., a selection predicate. This procedure is known as *push-down selection*. Next comes the most difficult part: determining the order in which to join the tables. This is the job of the query optimizer, discussed in a separate section. Once the order is determined, the corresponding Join operators are created and the tree is completed. Based on the type of the query, multiple paths can be taken from here:

- If the query is a standard SELECT-FROM-WHERE, a Project operator is appended at the root. In the case of DISTINCT, a DuplicateRemoval operator is further inserted at the root.

- If the query is an aggregate query, a Sum operator is inserted at the root.

- If the query is a standard SELECT-FROM-WHERE-GROUPBY, a GroupBy operator is appended at the root.

The final step is to append a WriteOut operator at the root of the query execution tree. All our execution trees will have a WriteOut at the root.

Query Optimizer

The query optimizer is in charge to determine the optimal query execution plan. There are two aspects it has to consider: the join order and the specific implementation for each operator. In our case, we consider only the join order, since we have not discussed about possible operator implementations yet. The first thing to settle upon when we talk about optimality is the cost function. The standard metric used in query optimization is the total size of the intermediate relations produced during query execution. Remember that every relational operator takes as input one or more tables, i.e., relations, and generates as output another table. We simplify the process and define as our cost function **the total number of tuples produced by the intermediate operators in a query execution tree**. The job of the optimizer is then to find that query execution tree that processes the least number of tuples, without actually executing the query. The optimizer only estimates this number for all the considered plans and chooses the query plan with the minimal cost. It uses statistics on table and attribute cardinality and number of distinct elements in order to compute the estimates. These statistics are stored in the database catalog and updated by a separate statistics maintenance process executed at certain time intervals. The details of how these statistics are propagated through the different relational operators are discussed in class and presented in detail in the textbook (Section 16.4). You can use them as given or come up with your own formulae.

How to compute the optimal join order? The query optimizer considers all possible trees that can be generated for a given number of tables. The number of tuples in these tables is obtained by applying the eventual selection predicates to the corresponding base tables. Starting with a node for each table, the optimal join order is computed by considering all possible joins of 2, 3, etc. tables. The cost of a join of k

tables can be easily computed from the cost of joining $k-1$ tables. If only the best cost for joining a set of k tables is maintained, a dynamic programming algorithm can be devised. Such an algorithm is presented in Section 16.6.4 (and thoroughly discussed in class). Your job is to implement it. Once the optimal join order is computed, you create an instance of `struct OptimizationTree` defined in file `QueryOptimizer.h` and return it to the query compiler, which will then generate the corresponding `Join` operators.

Requirements

- You have to get familiarized with all the code provided for this project phase. This includes the lexer and parser, relational algebra operators, and all their components.
- You have to implement all the methods in file `RelOp.cc`. They include the constructors/destructors for all the relational algebra operators and the `print` methods. You also have to implement the `operator<<` in class `QueryExecutionTree` for printing a complete query execution tree. This calls the `print` method for every operator.
- You have to implement all the methods in file `QueryCompiler.cc` and `QueryOptimizer.cc`, respectively.
- File `main.cc` already implements the basic functionality of an application that uses the query compiler. It reads a `SQL` query and generates the optimal query execution tree. It does not do anything at this time because the components are not implemented yet. That is your job. The prompt expects you to input a query in the `SQL` subset supported by the parser. Finish the query with `CTRL+D`. Alternatively, you can redirect the query from an input file with <. The parser checks the lexical and syntactic correctness of the input query and prints a message on the screen. Try this executable multiple times to familiarize yourself with the syntax of the supported query language. You are required to extend `main.cc` such that it does not finish after a single query. It should work as follows. It asks the user for a query. When a query is input, the compiler is called on it and the optimal query execution tree is printed. Then, the prompt is returned to the user. A new query can be inserted. If the user inputs *“exit”*, the program finishes.
- File `catalog.txt` contains the statistics to be used in the query optimizer. These values are from a TPC-H instance scale 1 (1 GB). Add them to your `SQLite` database catalog.
- Folder `Queries` contains a series of 10 `SQL` queries to test your compiler/optimizer. They are syntactically correct.
- Required packages: `flex`, `flex-doc`, `bison`, `bison-doc`.

Resources

- `http://epaperpress.com/lexandyacc/index.html`
- `http://www.cplusplus.com/`
- *Thinking in C++*: `http://mindview.net/Books/DownloadSites`