University of Edinburgh
School of Informatics
INFR11199 - Advanced Database Systems (Spring 2022)

Coursework Assignment – Part 2

Due: **Thursday, 24 March 2022 at 4:00pm**

IMPORTANT:

- **Plagiarism:** Every student has to work **individually** on this project assignment.

  All of the code for this project must be your own. You may not copy source code from other students or other sources that you find on the web. You may not share your code with other students. You may not host your code on a public code repository.

  Each submission will be checked using plagiarism detection software.

  Plagiarism will be reported to School and College Academic Misconduct Officers. See the University's page on Academic Misconduct for additional information.

- Start early and proceed in steps. Read the assignment description carefully before you start programming.

- The assignment is out of 100 points and counts for 40% of your final mark.

This is the second part of the coursework assignment. If you have not read the first part, please do that now before reading this document any further.

4 Task 2: CQ Evaluation

In Task 2, you will implement a simple interpreter for conjunctive queries. You will build a program that takes in a database (a set of files with data) and an input file containing one query. The program will process and evaluate the query on the database and write the query result in the specified output file. The evaluation should be based on the set semantics, meaning no tuple duplicates should appear in the input and the output.

4.1 Query Language

For Tasks 2 & 3, we extend the query language from Task 1 to allow comparison atoms in the body of a query. A comparison atom has the form:

$$term_1 \ op \ term_2$$

where $term_1$ and $term_2$ are constants or variables, and $op \in \{=, !=, <, <=, >, >=\}$. For instance, $x = y$, $x < 4$, `'ADBS'` $>= z$, $3 > 12$ are valid comparison atoms. String comparison should be performed lexicographically.

Comparison atoms in the body of a query represent a conjunction of predicates that filter the query result, similarly to a conjunction of predicates in the `WHERE` clause in SQL.

For instance, the query

$$Q(name) \text{ :- } STUDENT(id, name, age),\ age = 23,\ name \texttt{ != 'Jones'}$$

returns the names of all students who are 23 years old and whose name is not `'Jones'`. The equivalent SQL query is (remember that this assignment assumes the set semantics):

```
SELECT DISTINCT name FROM STUDENT WHERE age = 23 AND name != 'Jones'
```

You can assume that both terms in a comparison atom are always of the same type. Furthermore, every variable appearing in a comparison atom must also appear in at least one relational atom. No need to perform such checks.

We make a few simplifying assumptions as below. When we say a query is *valid*, we mean it is a permitted input to your interpreter which you should be able to handle. When we talk about a *base relation*, we mean a real relation that exists in the database.

- You may assume all valid queries only refer to relations that exist in the database.
- You may assume there will be at least one relational atom in the body of the query. Two relational atoms with the same name may appear in the body of the query.
- The head atom contains at least one term. For instance, $Q()$:- $R(x, y)$ is a valid Boolean conjunctive query but will not be considered in Tasks 2 & 3.

4.2 Data and Output Formats

We have provided you some sample data and some sample queries. Take a look at the `data/evaluation` directory. It has `db`, `input`, and `expected_output` as subdirectories.

- The `input` directory contains files with some example queries. There is one query per file.
- The `db` directory contains `schema.txt` specifying the schema for your database as well as the `files` subdirectory, where the data itself is stored. The names `schema.txt` and `files` are hard-coded and must exist in a valid database directory. Your program should support an arbitrary schema defined in `schema.txt`, not just the schema of the sample data.

  The `schema.txt` file contains one line per relation (table) in the database. Every line contains several strings separated by spaces. The first string on each line is the relation name and all the remaining ones are attribute (column) types, in the order in which they appear in the relation. The possible names for attribute types are `int` and `string`.

  The `data` subdirectory contains one file per database relation, and the name of the file is the same as the name of the database relation with the added `.csv` extension. Every file contains zero or more tuples without duplicates; a tuple is a line in the file with field (attribute) values separated by commas. The type of each value is defined in the schema file. You do not have to handle null values, but you do need to handle empty relations.
- The `expected_output` directory contains the expected output files for the queries we provided. For example, `query1.csv` contains the expected output for the query in `query1.txt`. The format for the output is the same as the format for the data.

4.3 Compile and Run

We will compile and run your code from the command line as in Task 1:

```
$ mvn clean compile assembly:single
$ java -cp target/minibase-1.0.0-jar-with-dependencies.jar \
    ed.inf.adbs.minibase.Minibase \
      data/evaluation/db \
      data/evaluation/input/query1.txt \
      data/evaluation/output/query1.csv
  Entire query: Q(x, y, z) :- R(x, y, z)
  Head: Q(x, y, z)
  Body: [R(x, y, z)]
```

The `Minibase` class requires passing three mandatory arguments: the path to a database directory, the path to an input file, and the path to an output file. Your code should handle these arguments appropriately (i.e., do not hardcode any paths).

We will test your code using our own test queries and databases with potentially different schemas. The database directory will have the same structure as described above, with files in the `files` directory named according to the database schema with `.csv` as the file extension. You may assume that prior to execution, a given output file does not exist but the output directory does exist.

After we run your code, we will compare your output files with ours. The order of tuples in the output file does not matter for this assignment. As you can imagine, it is very important for you to respect the expected input and output format.

Note: We will test your code on a DICE machine with Ubuntu Linux. Remember that Linux/MacOS environments use '/' as path separator. The database directory will be provided with no final '/' symbol, as above. If you use Windows, make sure that when you form file paths, you use `File.separator` instead of '\' as path separator.

4.4 Operators and the Iterator Model

A key abstraction in this project will be the iterator model for relational operators. You will implement several operators:

- the set relational algebra *select*, *project* and (tuple nested loop) *join*.
- a *scan* operator which is the leaf operator for any query plan. This is really a physical operator rather than something you would add to the relational algebra, but for now we will put it in the same category as the above.

The standard way to implement all relational operators is to use an *iterator* API. You should create an abstract class `Operator`, and all your operators will extend that. Certain operators may have one or two child operators. A scan operator has no children, a join has two children, and the remaining operators have one child. Your end goal is to build a query plan that is a tree of operators.

Every operator must implement the methods `getNextTuple()` and `reset()` (put these in your abstract `Operator` class). The idea is that once you create a relational operator, you can call `getNextTuple()` repeatedly to get the next tuple of the operator's output. This is sometimes called "pulling tuples" from the operator. If the operator still has some available output, it will return the next tuple, otherwise it should return null.

The `reset()` method tells the operator to reset its state and start returning its output again from the beginning; that is, after calling `reset()` on an operator, a subsequent call to `getNextTuple()` will return the *first* tuple in that operator's output, even though the

tuple may have been returned before. This functionality is useful if you need to process an operator's output multiple times, e.g., for scanning the inner relation multiple times during a join.

For each of the above operators, you will implement both `getNextTuple()` and `reset()`. Remember that if your operator has a child operator, the `getNextTuple()` of your operator can - and probably will - call `getNextTuple()` on the child operator and do something useful with the output it receives from the child.

A big advantage of the iterator model, and one of the reasons it is popular, is that it supports *pipelined* evaluation of multi-operator plans, i.e., evaluation without materialising (writing to disk) intermediate results.

The bulk of this task involves implementing each of the above four operators, as well as writing code to translate a given query (i.e., a line of text) to a query plan (i.e., a suitable tree of operators). Once you have the query plan, you can actually compute the answer to the query by repeatedly calling `getNextTuple()` on the root operator and putting the tuples somewhere as they come out.

We suggest you add a `dump()` method to your abstract `Operator` class. This method repeatedly calls `getNextTuple()` until the next tuple is null (no more output) and writes each tuple to a suitable `PrintStream`. That way you can `dump()` the results of any operator – including the root of your query plan – to your favourite `PrintStream`, whether it leads to a file or whether it is `System.out` (for testing).

4.5 Implementation Instructions

We recommend that you implement and test one feature at a time. Our instructions below are given in suggested implementation order.

We also recommend (but do not require) you set up a test infrastructure early on. You should do two kinds of testing – unit tests for individual components and end-to-end tests where you run your interpreter on queries and look at the output files produced to see if they match a set of expected output files. As you add more features, rerun all your tests to check that you didn't introduce bugs that affect earlier functionality.

After you implement and test each feature, make a copy of your code and save it so if you mess up later you still have a version that works (and that you can submit for partial credit if all else fails!).

4.5.1 Implement Scan

Your first goal is to support queries that are full relation scans, e.g., $Q(x, y, z)$:- $R(x, y, z)$. To achieve this, you will need to implement your first operator – the scan operator.

Implement a `ScanOperator` that extends your `Operator` abstract class. Every instance of `ScanOperator` knows which base relation it is scanning. Upon initialisation, it opens a file scan on the appropriate data file; when `getNextTuple()` is called, it reads the next line from the file and returns the next tuple. You probably want to have a `Tuple` class to handle the tuples as objects.

The `ScanOperator` needs to know where to find the data file for its relation. It is recommended to handle this by implementing a *database catalog* in a separate class. The catalog can keep track of information such as where a file for a given relation is located, what the schema of different relations is, and so on. Because the catalog is a global entity that various components of your system may want to access, you should consider using the singleton pattern for the catalog; if unfamiliar with the singleton pattern, see many other online references.

Once you have written `ScanOperator`, test it thoroughly to be sure `getNextTuple()` and `reset()` both work as expected. Then, hook up your `ScanOperator` to your interpreter. Assuming that all your queries are of the form $Q(x, y, \ldots)$:- $RELATION(x, y, \ldots)$, write code that grabs *RELATION* from the body and constructs a `ScanOperator` for it.

In summary the top-level structure of your code at this point should be:

- parse the query from the input file
- construct a `ScanOperator` for the relation in the body of the query
- call `dump()` on your `ScanOperator` to send the results somewhere helpful, like a file or your console.

4.5.2 Implement Selection

The next order of business is single-relation selection. That is, you are aiming to support queries like $Q(x, y, z)$:- $R(x, y, z), y > 3$.

This means you need to implement a second `Operator`, which is a `SelectOperator`. Your query plan will now have two operators – the `SelectOperator` as the root and the `ScanOperator` as its child. During evaluation, the `SelectOperator`'s `getNextTuple()` method will grab the next tuple from its child (i.e., from the scan), check if that tuple passes the selection condition, and if so output it. If the tuple does not pass the selection condition, the selection operator will continue pulling tuples from the scan until either it finds one that passes or it receives `null` (i.e., the scan runs out of output).

The tricky part will be implementing the logic to check if a tuple passes the selection condition. The selection condition is a list (conjunction) of `ComparisonAtom`s from the body of your query. The `SelectOperator` needs to know that selection condition.

You will need to write a class to test whether a selection condition holds on a given

tuple. For example, if you have a relational atom $R(x, y, z)$, you may encounter a tuple $(1, 9, \texttt{'adbs'})$ and a selection condition consisting of two atoms, $x < y$ and $z! = \texttt{'adbs'}$, and you need to determine if the condition is true or false on this tuple.

Your code needs some way to resolve variable references; i.e., if our input tuple is $(1, 9, \texttt{'adbs'})$ coming from the atom $R(x, y, z)$, your code needs a way to determine that x is 1, y is 9, etc. So, it also needs to take in some *schema* information and allow mapping from variable references like x to their value.

Once you have written your class for evaluating selection conditions, unit-test it thoroughly. Start with simple expressions that have no variable references, like the conjunction of `1 < 2` and `3 = 17`. Then test it with variable references until you are 100% sure it works. Once your expression evaluation logic is solid, you can plug it into the `getNextTuple()` method of your `SelectOperator`.

Queries with constants in relational atoms are also valid, e.g., $Q(x, y)$:- $R(x, y, 4)$ is valid and equivalent to $Q(x, y)$:- $R(x, y, z), z = 4$. Support the latter form with explicit comparisons first and then do the former form, which also requires projection.

4.5.3 Implement Projection

Your next task is to implement projection, i.e., you will be able to handle queries of the form $Q(x)$:- $R(x, y, z), y > 10$. You get the projection variables from the query head. For this task, you may assume that every term in the head is a variable and that constants cannot appear in the head.

For implementing projection, you need a third `Operator` that is a `ProjectOperator`. Recall that projection must return only distinct tuples. When `getNextTuple()` is called, it grabs the next tuple from its child, extracts only the desired values into a new tuple, and returns that tuple if it has not been reported before; otherwise, the operator fetches the next tuple from its child. Note that the child could be either a `SelectOperator` or a `ScanOperator`, depending on whether your query has a selection condition.

To ensure that only distinct tuples are reported, the operator can maintain an internal buffer containing already reported (distinct) tuples. Buffering all these tuples in memory may not work for very large outputs; for this project assignment, this is fine.

Note that the variable order in the head atom does not have to match the attribute order in the relation. The queries $Q(x, y)$:- $R(x, y)$ and $Q(y, x)$:- $R(x, y)$ are both valid and produce different output results.

By this point you should have code that takes in a query and produces a query plan containing:

- an optional projection operator, having as a child

- an optional selection operator, having as a child
- a non-optional scan operator.

Thus, your query plan could have one, two or three operators. Make sure you are supporting all possibilities; try queries with/without a projection/selection. If the query does not project away any variables and does not reorder variables in the head, then do not create a projection operator, and if the query has no selection predicate, do not create a selection operator.

You are now producing relatively complex query plans; however, things are about to get much more exciting and messy as we add joins. This is a good time to pull out the logic for constructing the query plan into its own class, if you have not done so already. Thus, you should have a top-level interpreter class that reads the statement from the query file. You should also have a second class that knows how to construct a query plan for a `Query` object, and returns the query plan back to the interpreter so the interpreter can `dump()` the results of the query plan somewhere.

4.5.4 Implement Join

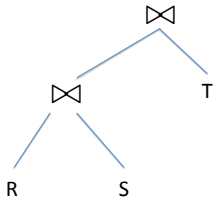
Next up, the star of the show: joins.

You need a `JoinOperator` that has both a `left` and `right` child `Operator`. It also has a selection predicate that captures the join condition. This selection predicate could be a single comparison such as $x = y$, a conjunction of comparisons, or it could be null if the join is a cross product.

Implement the simple (tuple) nested loop join algorithm: the join should scan the left (outer) child once, and for each tuple in the outer child, it should scan the inner child completely (finally a use for the `reset()` method!). Once the operator has obtained a tuple from the outer and a tuple from the inner, it glues them together. If there is a non-null join condition, the tuple is only returned if it matches the join condition (so you will be reusing your class for evaluating a selection condition from Section 4.5.2). If the join is a cross product, all pairs of tuples are returned.

Once you have implemented and unit-tested your `JoinOperator`, you need to figure out how to translate a query to a plan that includes joins.

For this project, we require that you construct a left-deep join tree that follows the order in the body of the query. That is, a query whose body includes relational atoms $R(x, y), S(y, z), T(x, y)$ produces a plan with the structure shown below:



The tricky part will be processing the body of the query to extract join conditions. The body may contain both selections on a single relation as well as join conditions linking multiple relations. For example,

$$Q(x) \text{ :- } R(x, y), S(z, u), x = y, z = 1, y > u$$

contains a selection condition on R, a selection condition on S, and a join condition on both R and S together. Obviously, it is most efficient to evaluate the selections as early as possible and to evaluate $y > u$ during computation of the join, rather than computing a cross product and having a selection later.

While the focus of this project is not optimisation, you do not want to compute cross products unless you have to as this is grossly inefficient. Therefore, we **require** that you have some strategy for extracting join conditions from the body part and evaluating them as part of the join. You do not need to be very clever about this, but you may not simply compute the cross products (unless of course the query actually asks for a cross product). You must explain your strategy in comments in your code and in the README that you submit with your code.

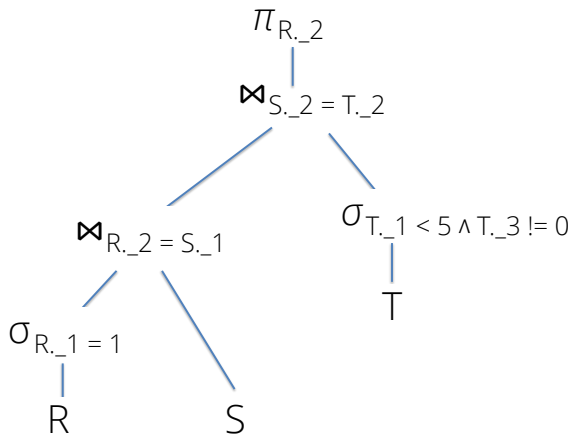
You should also be aware of implicit equi-join conditions arising when the same variable appears in more relational atoms, e.g., $Q(x)$:- $R(x, y), S(y, z)$ encodes the equality condition between the second variable of R and the first variable of S. An equivalent query would be $Q(x)$:- $R(x, y), S(yy, z), y = yy$.

A suggested way to do this is as follows. Identify all selection conditions (conjuncts) in the body, including both implicit and explicit conditions. For every condition, determine which relations are referenced and add the condition to an appropriate conjunction of conditions. If there are k relations being joined, there could be $2k - 1$ running conjunctions: $k - 1$ join conjunctions and k selection conjunctions on the individual relations. Once all the identified conditions are processed, the $2k - 1$ conjunctions of conditions can be integrated into the appropriate selection and join operators in the query plan. Of course some of these conjunctions may turn out to be empty, depending on the query.

For example, if we have

$$Q(b) \text{ :- } R(\texttt{1}, b), S(b, c), T(d, c, e), d < \texttt{5}, e \mathrel{!}= 0$$

the above approach would give the following query plan (in the figure below, $R.\_i$ refers to the i-th attribute of R):



You don't have to follow exactly this strategy. You don't need to worry about pushing projections past the joins, you may have one big projection at the root of your query plan.

5 Task 3: Group-By Aggregation

The final task is to extend the interpreter with the support for group-by aggregation. Your interpreter should support two aggregate functions – SUM and AVG – each taking one variable as argument and returning the sum and respectively the average of a set of values of that variable. Since the interpreter only supports integer values, the reported average values should be rounded to the nearest integer in the output.

An aggregate function is a term that may appear at most once in the head of a query, after all the head variables. The syntax and semantics of aggregate queries is explained next by means of examples.

The aggregate query

$$Q(\,\texttt{SUM}(x)\,) \text{ :- } R(x, y)$$

returns one integer value representing the sum of x-values in R. An equivalent SQL query would be `SELECT SUM(R.x) FROM R`. The output file for this query should contain just one integer number.

The group-by aggregate query

$$Q(x, \texttt{AVG}(z)) \text{ :- } R(x, y), S(y, z)$$

returns distinct x-values, each paired with the average of z-values computed from the body for the given x-value. An equivalent SQL query would be `SELECT R.x, AVG(S.z) FROM R JOIN S ON R.y = S.y GROUP BY R.x`. There could be multiple group-by variables in the head but at most one aggregate function, SUM or AVG, or none at all.

You will implement two aggregation operators, `SumOperator` and `AvgOperator`. Each operator reads all of the output from its child, extracts relevant values into tuples, organizes tuples into groups, and for each group computes an aggregate value. Each aggregation operator needs to see all of its input before producing any output. In other words, the aggregation operators are *blocking* operators. An aggregation operator, if present, should be the root of your query plan.

This second part of the assignment comes with the parser that supports `SUM` and `AVG` keywords. You should extend the code from the `QueryVisitor` class in `QueryParser.java` to construct Java objects for aggregate terms and include them to the `Query` object.

6 Grading

We strongly suggest that you follow the architecture we described for Tasks 2-3. However, we will not penalize you for making different architectural decisions, with a few exceptions:

- You must have relational `Operator`s that implement `getNextTuple()` and `reset()` methods as outlined above. This is the standard relational algebra evaluation model and you need to learn it. Do not hard-wire combinations of operators, e.g., projection should not assume selection as its child.
- You must construct a tree of `Operators` and then evaluate it by repeatedly calling `getNextTuple()` on the root operator.
- As explained in Section 4.5.4, you must build a left-deep join tree that follows the ordering of the relations in the body. Also, you must have a strategy to identify join conditions and evaluate them as part of the join rather than doing a selection after a cross product.

Disregarding any of the above three requirements will result in severe point deductions. Next we give the grading breakdown.

6.1 Code Style and Comments (10 points)

Follow standard guidelines for writing clean and understandable code; e.g., use standard naming conventions for classes and methods, break up large monolithic blocks of code into smaller logical pieces, avoid code duplication if possible – the "rule of three" says if your code is copied more than twice, then it probably needs to be abstracted out.

You must provide comments for every method you implement. At minimum, the comment must include one sentence about the purpose of the method, and `@params`/`@return`

annotations for every argument/return value respectively. In addition, every class must have a comment describing the class and the logic of any algorithm used in the class.

If you follow the above rules and write reasonably clean code that follows our overall architecture, you are likely to get the full 10 points for code style.

6.2 Automated Tests (90 points)

We will run your code on our own queries and data and award you points for every query that returns the correct output. You can expect that we will add additional relations to the database; of course the schema of these relations will be mentioned in `schema.txt` and the data files will be found in the `files` directory. We will test with basic queries as well as with complex queries that include any/all of the features you are to implement.

If you cannot implement one or more parts, that is fine, although obviously you won't get full points. In that case, you must clearly tell us in the README what you have not been able to implement.

7 Submission Instructions

Double-check that your code compiles and runs as described this document. You must keep `CQMinimizer` and `Minibase` as the top-level main classes of your code. Submit just one project directory for all three tasks.

Create a README text file containing the following information.

- An explanation of your logic for extracting join conditions from the body of a query. If this logic is fully explained in comments in your code, your README does not need to repeat that; however, it must mention exactly where in the code/comments the description is, so the grader can find it easily.
- Any other information you want the grader to know, such as known bugs.

Create a .zip archive containing a README file and your entire project directory so that we can compile and run your code on the command line. Do not include any `.class` files nor large `.csv` files. *Disable all debugging statements before submitting your code. Failure to do so may result in point deduction.*

Upload the zip archive to Learn: Assessment (the left panel) → Assignment Submission → Coursework: Minibase.

Make sure you start early! Good luck!