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

Coursework Assignment – Part 1

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.

1 Goals and Important Points

In this assignment, you will implement the minimization procedure for conjunctive queries and a lightweight database system for evaluating conjunctive queries called Minibase.

The goals of this assignment are threefold:

1. to understand the minimization procedure for conjunctive queries,
2. to teach you how to translate conjunctive queries into relational algebra query plans,
3. to learn the iterator model for relational operator evaluation and build naïve implementations of the most common operators (e.g., selection, projection, join).

The assignment consists of three tasks:

Task 1 (30%):	Implementation of the minimization procedure for conjunctive queries.
Task 2 (45%):	Implementation of the iterator model and common RA operators.
Task 3 (15%):	Implementation of aggregation operators (SUM, AVG, MEDIAN).

The remaining 10% of your mark is for code style and comments; more details later on. You will use the minimization procedure only in Task1 but not in Tasks 2 & 3; this is to ensure that any errors made in Task 1 are not propagated into the other tasks.

You should consider the *set semantics* for each task, meaning that every relation consists of distinct tuples and the input and output of each operator contain no duplicates.

You will start from a skeleton project consisting of two main classes:

- `CQMinimizer`, which should contain your code for Task 1
- `Minibase`, which should contain your code for Tasks 2 & 3.

Both classes define the expected command line interface. You are free to modify these classes but *must preserve* their command line interface as it is essential for marking.

The skeleton project also comes with a parser for our query language, so you do not have to write your own (unless you want to). The main classes show how to parse query strings into Java objects.

The rest of this document gives the instructions for Task 1. Read the document carefully, set up the development environment, and start with the implementation of Task 1.

Tasks 2 and 3 consider topics not yet covered in class. The instructions for Tasks 2 and 3 as well as how to submit your solution will be released on Wednesday, 15 February. The submission deadline is the same for all three tasks: 24 March at 4pm.

Coursework Assignment – Part 1 continues. . .

2 Setting Up Local Development Environment

You are free to use any text editor or IDE to complete the project. We will use Maven to compile your project. We recommend setting up a local development environment by installing Java 8 or later and using an IDE such as IntelliJ or Eclipse. To import the project into IntelliJ or Eclipse, make sure that you import as a Maven project (select the `pom.xml` file when importing).

3 Task 1: CQ Minimization

In Task 1, you will implement the minimization algorithm for conjunctive queries discussed in class. You will build a program that reads a query from the specified input file, minimizes the query, and writes a minimized query in the specified output file.

3.1 Query Language

Task 1 considers the language of conjunctive queries as presented in class, with queries expressed using the rule-based form:

$$\underbrace{Q(\mathbf{v})}_{head} \text{ :- } \underbrace{R_1(\mathbf{t}_1), \ldots, R_m(\mathbf{t}_m)}_{body}$$

where $R_1, \ldots R_m$ are relation names, $\mathbf{v}$ is a tuple of variables, $\mathbf{t_1}, \ldots, \mathbf{t_m}$ are tuples of terms, and each variable from $\mathbf{v}$ appears in the body. A term can be a positive integer constant (e.g., `42`), a string constant enclosed in single quotation marks (e.g., `'ADBS'`), or a variable. A valid variable name consists of lowercase letters, e.g., a, x, abc are valid variable names; note that this is different from the notation used in class where a, b, c denote constants. A valid relation name consists of uppercase letters, and the relation name in the head does not appear in the body. In our query language, `'ADBS'` is a string constant while $ADBS$ (without quotation marks) is a valid relation name.

Examples of valid conjunctive queries are:

$$\begin{aligned}
Q(x) &\text{ :- } R(x, \texttt{'This is a test string'}) \\
Q(x, z) &\text{ :- } R(\texttt{4}, z), S(x, y), T(\texttt{12}, \texttt{'x'}) \\
Q() &\text{ :- } R(y, z), R(z, x) \\
DBSTUDENTNAME(name) &\text{ :- } STUDENT(id, name), ENROLLED(id, \texttt{'ADBS'})
\end{aligned}$$

Examples of invalid conjunctive queries in our language are:

$Q(y)$:- $R(x, \texttt{'test'})$	– y does not appear in body
$Q()$:-	– empty body
$Q(x, \texttt{3})$:- $R(\texttt{4}, z), S(x, y)$	– constant in head
$Q(y)$:- $R(y), S()$	– empty relational atom in body

You can assume that only valid, correctly-typed conjunctive query will be provided as input. For instance, $Q(x)$:- $R(x, \texttt{4}), R(y, \texttt{'4'})$ is wrongly-typed as the second attribute of R does not have a unique type. Similarly, $Q(x)$:- $R(x, y), R(x, y, z)$ is also wrongly-typed as the arity of R is not unique. Such queries will never appear as input.

For simplicity, you can assume that every relational atom appearing in the body or the head has no repeated variables. For instance, the atom $R(x, x)$ is invalid according to this restriction, while $R(x, xx)$ and $R(3, 3)$ are valid; the query $Q(x, x)$:- $R(x)$ is also invalid. But the same variable can appear in different relational atoms in the body.

You do not need to perform any type or naming checks on the input query.

3.2 Query Parser

The provided code contains a parser for our query language. The parser was generated using ANTLR (`https://www.antlr.org/`) based on the grammar defined in `parser/Minibase.g4`. Have a look at the grammar to understand the structure of the language constructs. The `base` directory contains Java classes matching these constructs. You are free to modify and extend these classes.

The grammar also allows comparison atoms such as `x > 3` and `x != y` to appear in the body of a query. We will consider comparison atoms in Tasks 2 & 3. For Task 1, we will assume that the body consists of relational atoms only.

The `parser/generated` directory contains the parser classes generated by ANTLR. Do not modify these generated classes directly. The class from `parser/QueryParser.java` uses the visitor pattern[1] to construct a `Query` object from ANTLR objects.

To get you started, we provide a simple method in `CQMinimizer.java` that uses the parser to read a query from a file, print it out, and access fields of the `Query` object.

[1] `https://en.wikipedia.org/wiki/Visitor_pattern`

3.3 Input and Output

We provide a few sample conjunctive queries. The `data/minimization` directory contains `input` and `expected_output` as subdirectories. The `input` directory contains files with example conjunctive queries. There is one query per file. The `expected_output` directory contains the expected output files for the given sample queries. For example, the query in `expected_output/query1.txt` is the minimized version of the query in `input/query1.txt`.

`CQMinimizer.java` provides the command-line interface that expects two arguments: the path to an input file and the path to an output file. You are free to modify the provided code and include additional classes and packages, but you must preserve the existing command-line interface. You can run the `CQMinimizer` class from your IDE and provide the required arguments.

3.4 Compile and Run

We will compile your code from the command line as follows:

```
mvn clean compile assembly:single
```

This command will produce `target/minibase-1.0.0-jar-with-dependencies.jar`.

We will run `CQMinimizer` as follows:

```
$ java -cp target/minibase-1.0.0-jar-with-dependencies.jar \
    ed.inf.adbs.minibase.CQMinimizer
Usage: CQMinimizer input_file output_file
```

`CQMinimizer` requires passing two mandatory arguments.

```
$ java -cp target/minibase-1.0.0-jar-with-dependencies.jar \
    ed.inf.adbs.minibase.CQMinimizer \
      data/minimization/input/query1.txt \
      data/minimization/output/query1.txt
  Entire query: Q() :- R(x, 'z'), S(4, z, w)
  Head: Q()
  Body: [R(x, 'z'), S(4, z, w)]
```

The provided sample code parses the query from the input file into a Java object and prints different parts of the query. Your code should write the minimized query into the specified output file. You can assume that the output directory already exists.

After running your code, we will check whether the query in the output file is equivalent to ours up to variable renaming; e.g., $Q(x)$:- $R(x)$ would be equivalent to $Q(y)$:- $R(y)$. We will test your implementation on a set of conjunctive queries.