Assignment

A Query Engine for Acyclic Conjunctive Queries

Data and Information Management

VRIJE UNIVERSITEIT BRUSSEL

Professor Bas Ketsman ⟨bas.ketsman@vub.be⟩
Assistant Tim Baccaert ⟨tim.baccaert@vub.be⟩

The deadline for this assignment is **Monday 18th, December 2023** at 23:59, Central European Time (GMT+1). You have to upload your solution to Canvas, as prescribed in the third section of this document. This assignment is designed to be made in **pairs**, if this poses a problem you may work **individually** as well.

I. Context

In this assignment, your goal is to implement a query engine for **acyclic conjunctive queries**, using the techniques and concepts we encounter during the lectures. This assignment is mandatory and constitutes your grade for the practical segment of this course, which is 40% of your overall grade.

We designed the assignment for **pairs** of students. You are free to assemble your group as you see fit, you do not need to notify us about its composition. In case you have trouble finding suitible colleagues for whatever reason, you may alternatively work on this assignment **individually**. We will, however, not grant you extra points if you do choose to work alone.

Programming Language

Since this is a course at the master's level, we have to accomodate students from a wide range of different universities. However, we assume you have advanced knowledge of at least one programming language, as can be reasonably expected from someone with a bachelor's degree in Computer Science. To facilitate this difference in background, we allow you to choose from the following selection of languages for this assignment:

- Java (`https://dev.java`)
- Rust (`https://www.rust-lang.org`)
- Scala (`https://scala-lang.org`)
- TypeScript (`https://www.typescriptlang.org`)

If you are not familiar with any of these languages, you may request an exemption by sending an email to tim.baccaert@vub.be. We strongly prefer the use of a statically typed language. Furthermore, the language must have a library with bindings to Apache Arrow (`https://arrow.apache.org`), as this assignment depends on it.

You are allowed to use libraries, provided they do not implement required algorithms for your assignment (*i.e.*, you may use libraries for logging, parsing, data structures, etc.; but not join algorithms, query engines, ...).

Note: We will not answer questions regarding the specifics of a programming language, or used libraries. You should be mature enough as a programmer to figure out these concerns yourself. Questions should exclusively relate to the concepts required for the assignment, or assignment practicalities.

Testing Dataset

Throughout this assignment we will make use of the dataset accessible from the Canvas space for testing purposes. It is comprised of five files in the comma-separated values (CSV) format, each file represents a single relation. A visual representation of the schema is given in Figure 1.

The relation **Beers** (`beers.csv`) has 8 attributes: `beer_id`, the artificial primary key of the beer; `brew_id`, the id of its brewery; `beer`, its name; `abv`, its alcohol by volume; `ibu`, its international bitterness unit; `ounces`, its typical volume measured in ounces; and finally `style` and `style2`, its brewing styles.

Beers has a foreign key relationship with the **Styles** (`styles.csv`) relation, which has 3 attributes: `style_id`, an artificial primary key (shared with `cat_id`) for the style; `cat_id`, the id of its category; and `style`, the textual label for this style. This relation is in a foreign key relationship with **Categories** (`categories.csv`), which has 2 attributes `cat_id` and `cat_name`, respectively its artifical primary key and category name.

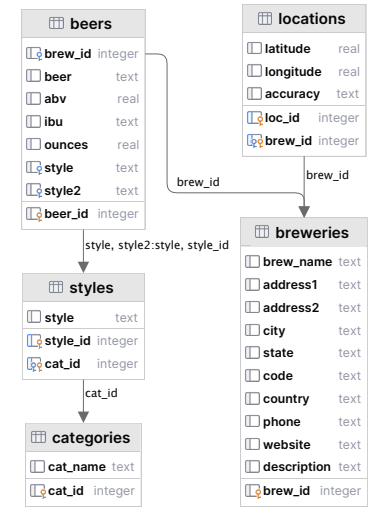

Fig 1. *visual schema representation.*

Next, we have the **Locations** (`locations.csv`) relation, with the 3 attributes: `loc_id` and `brew_id` forming a shared primary key, where the latter is the id of the brewery; `longitude` and `latitude` representing a brewery's geographic location; and `accuracy` representing the accuracy of its location.

Lastly, we have the **Breweries** (`breweries.csv`) relation, which has the following 11 attributes: `brew_id`, the artificial primary key of the brewery; `brew_name`, the name of the brewery; and the self-explanatory attributes `address1`, `address2`, `city`, `state`, `code`, `country`, `phone`, `website`, and finally `description`.

Apache Arrow

To store relations in memory, we expect you to make use of Apache Arrow (`https://arrow.apache.org`). This library defines a language-independent format for in-memory storage of relational data, optimised for analytic queries on modern hardware. Think of it as a portable in-memory database without a query engine and transaction support. Arrow supports loading data from many file formats, and we will use CSV files for this assignment, as mentioned previously.

The concrete API varies from language to language, hence we refer you to the corresponding websites for each of the languages: Java (`https://arrow.apache.org/docs/java/index.html`), Rust (`https://docs.rs/arrow/latest/arrow/`), Scala (see Java), and Typescript (`https://www.npmjs.com/package/@apache-arrow/ts`). Installation depends on your language environment, we suggest you use your language's standard package manager and repositories (*i.e.*, `sbt` for Scala, `cargo` for Rust, `npm` for Typescript).

II. Assignment

In the following subsections we briefly go over the features you are expected to implement. Each feature has a number of fixed requirements, and may include bonus requirements that you can use to distinguish yourself.

We only give points for bonus requirements if you implemented all the regular requirements.

Part I. Query Representation

In this assignment, we focus on conjunctive queries (CQs). This is a practically relevant fragment as most user queries are CQs. As seen in the lectures, we use datalog notation.

F1. Data Structure

First, we expect you to implement a data structure that syntactically represents a CQ. It has the following components and subcomponents:

- a **head** atom *e.g.*, Answer($\bar{y}$),
- a set of **body** atoms *e.g.*, $\mathrm{R}_1(\bar{u}_1), \ldots, \mathrm{R}_n(\bar{u}_n)$,
- each **atom** consists of a *relation name*, and a tuple of *terms e.g.*, $\mathrm{R}_1(u_1^1, \ldots, u_1^k)$, and
- each **term** u_j^i is either a *constant value* or a *variable*.

Here, a constant value is a value that has one of the Apache Arrow logical types[1]: `Utf8`, `FloatingPoint`, and `Int`. Depending on the language you are using, these may have a slightly different name; use the most similar ones. The precise data structures you use to represent these components are up to you, and will likely vary based on the rest of your implementation.

Example 1. The following query asks if there are beers that are categorised as Belgian and French ales:

Answer() :- Beers($beer\_id$, $brew\_id$, $beer$, abv, ibu, $ounces$, $style$, $style2$),
Styles($style\_id$, cat_id, $style4$),
Categories(cat_id, 'Belgian and French Ale').

The head atom is Answer() and the body atoms are Beers(...), Styles(...), and Categories(...). All lowercase symbols such as $beer$ and $style$ are variable terms, and 'Belgian and French' is a constant value of type `Utf8`.

Requirement 1 Implement a `Query` data structure.

As bonus functionality, you can implement a parser for the rule syntax.

(Bonus Functionality) Implement parsing functionality according to the Backus-Naur form (BNF) grammar below. It should produce an instance of your `Query` data structure (you may use a parsing library).

```
    <query> ::= <atom> ":-" <atom-list> "."
     <atom> ::= <relation> "(" <term-list> ")"
<atom-list> ::= <atom> | <atom> "," <atom-list> | ""
     <term> ::= <constant> | <variable>
<term-list> ::= <term> | <term> "," <term-list> | ""
 <constant> ::= <float> | <integer> | <string>
 <variable> ::= <letters>
 <relation> ::= <capital> <letters>
    <float> ::= <digit> "." <digits> | "-" <digit> "." <digits>
  <integer> ::= <digit> | "-" <digit> | <digits>
   <string> ::= "'" <letters> "'"
  <capital> ::= "A" | .. | "Z"
   <letter> ::= <capital> | "a" | .. | "z"
    <digit> ::= "0" | .. | "9"
  <letters> ::= <letter> <letters> | ""
   <digits> ::= <digit> <digits> | ""
```

Lst 1. *conjunctive query BNF grammar*

[1] `https://github.com/apache/arrow/blob/4fc2731a3ceaf7dd5b1dce6f29bf7cad0ab2f13e/format/Schema.fbs#L407-L430`

F2. Testing for Acyclicity

Since we will specifically focus on **acyclic** conjunctive queries, we need to make sure that the query we are given is *acyclic*. Recall from the lectures that a CQ is acyclic if the Graham-Yu-Ozsoyoglu (GYO) algorithm yields the empty query.

Example 2. Given the query from Example 1, the GYO algorithm should produce the empty query. A possible execution removes the following ears:

1. the ear Beers(...) with witness Styles(...),
2. the ear Styles(...) with witness Categories(...), and
3. the ear Categories(...), which has no shared variables, and is therefore its own witness.

After removing these ears, the query is empty, hence it is acyclic.

Requirement 2 Implement the GYO algorithm for your `Query` data structure.

F3. Join Trees

From the lectures, we know that a CQ is acyclic only if it has a join tree. Hence, for the next feature we ask you to implement a procedure that can construct such a join tree based on GYO's execution.

Example 3. A possible join tree based on the GYO execution in Example 2 is the following:

- $\{beer\_id, brew\_id, beer, abv, ibu, ounces, \mathbf{style}, style2\}$
- $\{style\_id, \mathbf{cat\_id}, \mathbf{style}\}$
- $\{\mathbf{cat\_id}\}$

Requirement 3 Implement an algorithm that constructs a join tree for your `Query` data structure.

Part II. Query Evaluation

We now move on to the evaluation of acyclic CQs. Here, we rely on the Yannakakis algorithm which was handled in the lectures. Note that in order for this algorithm to work, the query **must** be acyclic.

F4. Globally Consistent Databases

A globally consistent database is a database that does not contain any dangling tuples. We can use the join tree to build a **full reducer** for our database. When applied over the database, this yields us a globally consistent version.

Example 4. Given that the node with atom Beers(..) is chosen as the root. A possible application of a full reducer based on the join tree from Example 3 is given by:

$$\begin{aligned} \text{Styles'} &= \text{Styles} \ltimes \text{Categories} && \textit{(post-order)} \\ \text{Beers'} &= \text{Beers} \ltimes \text{Styles'} && \textit{(post-order)} \\ \text{Styles''} &= \text{Styles'} \ltimes \text{Beers'} && \textit{(pre-order)} \\ \text{Categories'} &= \text{Categories} \ltimes \text{Styles''} && \textit{(pre-order)} \end{aligned}$$

Requirement 4 Implement an algorithm that creates a globally consistent database given an input database and a join tree.

F5. The Yannakakis Algorithm

Finally, you will bring all components together into a single algorithm. First, verify whether the query given to you by the user is acyclic, and terminate with an error if it is cyclic. Then, create a join tree for the query, and apply the Yannakakis algorithm.

Example 5. Given the join tree from Example 3 and the globally consistent database obtained from applying the full reducer in Example 4, an evaluation of Yannakakis executes the following expression:

$$\pi_{()}(\text{Beers'} \bowtie (\text{Styles"} \bowtie \text{Categories'}))$$

The answer based on the test database should yield $\{()\}$ (*i.e.*, `true`) as the output.

Requirement 5 Implement the (generalized) Yannakakis algorithm for acyclic CQs.

(F6.) Additional Bonus Features

If you wish to distinguish yourself, you can extend your implementation with other algorithms regarding conjunctive queries. This could be a worst-case optimal join algorithm for cyclic CQs, it could include CQ minimisation, constant-time enumeration of CQ answers with linear preprocessing, ranked enumeration of CQ answers, CQ evaluation under inclusion or functional dependencies, and so on. You can look at the book for further inspiration.

III. Delivery

The deadline for the delivery of this project is on **Monday 18th, December 2023** at 23:59, Central European Time (GMT+1). Your report and implementation should be submitted in their respective Canvas locations.

Report You are expected to write a report that accompanies your assignment. Make use of the following ACM LaTeX-template for your report: `https://www.acm.org/publications/proceedings-template`, and use the `acmart` document class with the options `nonacm`, `sigconf`, and `screen`. The report is limited to a maximum of 4 pages excluding code listings, figures, and references.

Make sure to mention your own name (and that of your colleague, if there is one), including your student number and email. **Both students in a pair should submit the same assignment to Canvas**.

The report should be well-structured, and written in English. We expect you to discuss your implementation strategy for each of the requirements, and a short motivation for why you made your design decisions the way you did. **Name the file `<last-name-1>-<last-name-2>-report.pdf` or `<last-name>-report.pdf`.**

Implementation We will automate some aspects of the grading process. Hence, you should run your implementation on the following five conjunctive queries:

1. Answer() :- Locations(u_6, v, u_7, u_8, z), Breweries(v, y, u_7, u_8, u_9, u_{10}, u_{11}, u_{12}, u_{13}, u_{14}, u_{15}), Beers(u_1, v, x, '0.07', u_2, u_3, u_4, s, u_5), Styles(u_{16}, c, s), Categories(c, w).
2. Answer(x, y, z) :- Breweries(v, x, 'Westmalle', u_1, u_2, u_3, u_4, u_5, u_6, u_7, u_8), Locations(v, u_8, y, z, u_9).
3. Answer(x, y, z) :- Beers(u_1, u_2, x, u_3, u_4, u_5, u_6, x, z), Styles(u_7, y, x), Categories(y, z).
4. Answer(x, y, z, w) :- Beers(u_1, v, x, '0.05', '18', u_5, 'Vienna Lager', u_6), Locations(u_7, v, y, z, w).
5. Answer(x, y, z, w) :- Locations(u_6, v, u_7, u_8, z), Breweries(v, y, u_7, u_8, u_9, u_{10}, u_{11}, u_{12}, u_{13}, u_{14}, u_{15}), Beers(u_1, v, x, '0.06', u_2, u_3, u_4, s, u_5), Styles(u_{16}, c, s), Categories(c, w).

Your answers should be stored in a CSV file with the name `output.csv` which has the following columns:

1. **query_id** (`Int`): the identifier of the query (*e.g.*, 1, 2, 3, ...).
2. **is_acyclic** (`Int`): wether the query is acyclic (1) or cyclic (0) after executing your implementation of the GYO algorithm.
3. **bool_answer** (`Int`) (when the query is boolean) wether the query outputs true (1) or false (0) after executing your implementation of the Yannakakis algorithm. Leave this NULL when the query is not boolean.
4. **attr_x_answer** (`Int`) all constants which get mapped to by x by all valuations for the query. (i.e., the output of the x column)
5. **attr_y_answer** (`Int`) all constants which get mapped to by y by all valuations for the query. (i.e., the output of the y column)
6. **attr_z_answer** (`Int`) all constants which get mapped to by z by all valuations for the query. (i.e., the output of the z column)
7. **attr_w_answer** (`Int`) all constants which get mapped to by w by all valuations for the query. (i.e., the output of the w column)

The canvas space will include an example output file. **Leave columns NULL/empty if the query does not give output for that column, or if the query is cyclic**.

Store this file together with your source code (in a folder `src/`) in a zip archive.
Name the file `<last-name-1>-<last-name-2>-impl.zip` or `<last-name>-impl.zip`.

IV. Evaluation

This assignment accounts for 40% of your final grade. There is no oral defense. Remember that you must hand in this assignment in order to pass for the course as a whole. **A late submission counts as 0/20 for the assignment.** The evaluation of your assignment takes into account the following aspects:

- **Requirements.** The correctness of your implementation, and the completeness of each requirement will be evaluated. Do not forget to include your output file, as part of the grading process is automated. We will, however, manually check your code as well.
- **Design.** The assignment allows for some creative liberty with respect to the specific implementation strategy of each requirement. Hence, make sure to put some tought into which data structures and algorithms you use, and make sure to motivate your decisions in comments, or in the report.
- **Code Quality.** As is typical in most programming assignments, the quality of your code is also an important factor. Pay specific attention to the clarity of your code, that is, use clear but concise variable and function names, and provide meaningful comments to motivate your design decisions.

  Furthermore, make sure that your code is modularised and reusable. Use the abstraction facilities available in your language, and place care and attention into the prevention of needless code duplication.
- **Testing.** We expect that you provide some tests which showcase the functionality of your implementation in different scenarios.
- **Report.** The quality of your report will also be evaluated. Avoid paraphrasing or copy-pasting your code, as we have access to the source files; you can refer to your code instead. Make sure to focus on motivating your design.

Plagiarism

What is allowed? This assignment is to be made strictly individually and on your own. This means that you must create your assignment on an independent basis and you must be able to explain your work, reimplement it under supervision, and defend your solution. Copying work (code, text, etc.) from or sharing it with third parties (e.g., fellow students, websites, GitHub, etc.) is not allowed. Electronic tools will be used to compare all submissions with online resources and with each other, even across academic years.

Specifically, for this assignment:

- *It is allowed to use third-party code (e.g., logging libraries, testing libraries, etc.) with source attribution. If third-party code is used to fulfill any part of the requirements of the assignment (except the bonus feature on parsing CQs),* ***you will not receive any points for that part of the assignment****.*
- *We do not consider your language's standard library to be third-party code. Hence, you may use that without attribution.*

What is plagiarism? Any action by a student that deviates from the instructions given and does not comply with the examination regulations is considered an irregularity. Plagiarism is also an irregularity. Plagiarism means the use of other people's work, adapted or otherwise, without careful acknowledgement of sources. (cf. OER, Article 118§2). Plagiarism may relate to various forms of works including text, code, images, etc.

What happens if plagiarism is suspected? Any suspicion of plagiarism will be reported to the Dean of Faculty without delay. Both user and provider of such code will be reported and will be dealt with according to the plagiarism rules of the examination regulations (cf. OER, Article 118). The dean may decide on (a combination of) the disciplinary sanctions, ranging from 0 out of 20 on the course to a prohibition from (re-)enrolling for one or multiple academic years (cf. OER, Article 118§5).

Contact us if you are in doubt as to whether or not something would be considered plagiarism.