Exploring Datalog for Modern Data Analysis
(Application paper)

Artem Pelenitsyn

Northeastern University

Abstract. The current approach to building data analysis applications is dominated by imperative languages armed with robust libraries for typical data processing tasks. Both, imperative features and APIs pose well-known challenges that can be avoided if a declarative domain-specific language for data manipulation is employed. In this paper, we showcase a commercial implementation of Datalog embedded in Julia, called Delve, that have a number of plummy features: it is declarative, has virtually zero amount of API vocabulary required to apply it, and can resort to the efficient JIT compiler of Julia for the tasks not amenable for declarative processing. We discuss two implementations of a data processing application, using Delve and R, and compare them along two axes: linguistics and performance.

Keywords: Datalog · Julia · R · embedded languages.

1 Introduction

The current approach to building data analysis applications is dominated by languages like Python, R, and, to some extent, Julia. All of these languages can be considered as general-purpose; e.g. it is easy to build a web server in Julia, and less so in R, but also possible. This generality calls for factorizing implementation of various tasks into library code, as well as gives vast freedom as to which language features to employ when building applications. This clearly has downsides.

General-purpose languages take the libraries-based approach to tackle data processing (e.g. Pandas in Python, `dplyr` in R, DataTables in Julia). Such libraries pose the challenge of learning the APIs with many possible caveats and subtle differences between each other. Another common issue with this approach is tracking changes in APIs, e.g. functions receive new parameters, become deprecated, etc.

Imperative features of the said languages, especially unchecked mutation of the global state, may hinder understanding of a script and the grounds of its validity. This often leads to subtle errors, and bugs in data science scripts are a topic of many recent studies (consider, for example, a recent study of bugs in COVID-19 related software [3]).

In the area of data analysis, both of the issues (the API-based approach and imperative features) can be avoided by employment of a declarative domain-specific language targeted at this particular kind of tasks. As a bonus, a significantly special-purposed DSL shall allow efficient compilation to a robust application — something that has been an issue with languages like Python and R for decades. On the the down side, a declarative interface would lack support for inherently imperative tasks faced by data analysts (e.g. file system manipulation). This can be resolved by embedding the DSL into a general-purpose language.

In this paper, we showcase a commercial implementation of Datalog embedded in Julia, called Delve, that checks all the above boxes: it is declarative, has virtually zero amount of API vocabulary needed to apply it, and it can resort to the efficient JIT compiler of Julia for the tasks not amenable for declarative processing. Delve's backend is build on solid foundations of database systems.

We consider a data analysis application we call Truck Factor. Two implementations are suggested, using Delve and R. We compare them along two axes. First, linguistics: how much of code is required, how much of "external" vocabulary (the one not concerned with the particular application) a programmer need to learn to build such implementation. Second, performance and scalability: we consider a spectrum from toy loads to the ones not fitting in computer RAM.

2 Truck Factor

For a case study, we aimed at a just-above-toy application having to do with data analysis. To this end, we picked a task of computing the truck factor (also known as bus factor) of a software repository roughly following the methodology of [1] including the degree-of-authorship formula [2] as the basis of the metric. We did not aim to reproduce results [1] especially because their dataset of 133 GitHub repositories is insufficient for our purposes: to make any conclusion performance-wise, we needed thousands of repositores. In this section we discuss data format we work with, the algorithm to compute the truck factor, and representative code snippets from both implementations, in R and in Delve.

2.1 Dataset

We use open source projects hosted on GitHub and using Git as their version control history. The history, or metadata, is what analysis uses as the input with the caveat that it works with CSV-formatted tables instead of binary Git-objects. In order to convert the latter to the former we use an auxiliary tool called GhGrabber[1]. We do not analyze the source code itself.

2.2 Algorithm

The purpose of the algorithm is to establish the number of a project (i.e. software repository) contributors who "authors" at least half of the source files. The

[1] https://github.com/PRL-PRG/ghgrabber

degree-of-authorship of a file f (represented by a filepath f_p) for a developer d (mapped to a GitHub identity m_d) is determined using the following formula [1]:

$$\begin{aligned}\mathrm{DOA}(m_d, f_p) = 3.293 + \\ 1.098 \times \mathrm{FA}(m_d, f_p) + 0.164 \times \mathrm{DL}(m_d, f_p) - \\ 0.321 \times \ln(1 + \mathrm{AC}(m_d, f_p)).\end{aligned} \quad (1)$$

Here, first authorship (FA) is 1 if m_d originally created f — otherwise it is 0; number of deliveries (DL): the number of changes in f made by m_d; finally, number of acceptances (AC): the number of contributions in f made by any developer, except m_d. The coefficients are picked up empirically [2].

The high-level structure of the algorithm is as follows: (i) for every project source file and every developer touched that file compute the three parameters mentioned above (FA, DL, AC) and the final metric (DOA); (ii) for every file, pick its developer with the maximum DOA as the file's author; (iii) sort developers in decreasing order by the number of files they author and count how many of the "top" developers author at least half of files.

2.3 R implementation

As the basis for the R implementation of the algorithm we use a popular high-level pipeline-oriented DSL `dplyr`[2]. It provides a set of combinators (called "verbs" in the documentation) familiar from SQL and some other relational-programming languages. Among those combinators: `select`, `filter`, `arrange`, `summarise`.

Here is an example of `dplyr` in action for computing the number of contributions (measured as the number of commits that touched the file, according to Git) by individual developers — this is useful for computing the AC metric mentioned above:

```
main %>%
  group_by(uid,author) %>%
  summarize(n=n()) ->
  contributions
```

The `main` table consists of records of contributions that various developers submitted to files in various projects. We generate unique global identifiers (`uid`'s) for every file in the dataset of projects in advance. Probably, the most confusing part of `dplyr`'s syntax here is the `n()` function: it counts the number of elements in the group, and the result is written down in the column conventionally named `n` too. The result of this pipeline is stored in a new table called `contributions`.

2.4 Delve implementation

Delve is a Datalog implemented as a Julia package. The frontend is standalone, so a Delve program has to be written in a separate file (or in a string literal in a

[2] https://dplyr.tidyverse.org/

Julia source file) and passed to the query_delve function in the Julia space. The backend, on the other hand, is tightly integrated with the Julia runtime, so the computed result comes out as a collection of plain Julia objects. Therefore, the outputs can be post-processed in a Julia program (e.g. printed to the console or to a CSV file).

The example from the previous subsection (computing the number of contributions) is expressed in Delve as follows:

```
def contributions = uid author contribution :
  main(uid,_,author,_,_,_) and
  (count[hash: main(uid,hash,author,_,_,_)])(contribution)
```

Here, def is a keyword signaling that we are about to define a new relation (contributions in this case). The relation will have three components listed between = and :, namely uid, author, and contribution. After the colon we provide a formula for the new relation. The uid and author components should participate in the main relation. For every such values we compute the value of contribution using the auxiliary count combinator. The input to count is a relation (in this case it is defined inline): it has one column hash and every row is the hash identifier of a commit authored by the author and touching the file identified by uid. Th output of count is again a relation but having just one row with one field indicating the number of rows in the input relation.

3 Evaluation

When trying to compare different implementations of the same algorithm, there are several possible view points. Two important ones are linguistic parameters and performance.

3.1 Linguistics

Table 1 compares linguistics of both implementations modulo data loading. The number of lines is similar, which is important because line count is one of the primary measures of code size.

What stands out in Table 1 is the word count for Delve: the corresponding implementation is far wordier than the one in R. But if we look at the number of unique words, it is a parity again —just as with the line count.

There is an important difference in the distribution of unique words though. In particular, among the top five most-frequent words in the Delve implementation three are the column names; the other two are def and and. In contrast, for the R implementation, among the five top words: two are the names specific to the dplyr vocabulary, (group_by, summarize), another two are application-specific (doa, uid), and the last one is somewhere in between: it is n — the name of a dplyr function, as well as the column name that we use to store results of calls to this function.

Table 1. Linguistic characteristics of both implementations.

	R	Delve
Lines	107	116
Words	277	450
Unique Words	77	79

Another interesting metric of vocabulary diversity is distribution of words frequencies. Occurrences of five most-frequent words in Delve implementation account for 42% of all words. For R, this metric is 21% —exactly two times less than for Delve! This shows that a Delve programmer most of the time thinks about the data representation, whereas an R programmer needs to care about various parts of the API simultaneously with application-specific identifiers.

3.2 Performance

Delve is a Datalog implementation with the focus on performance and large-scale applications, whereas it was never a strong suit for R. We performed several experiments that mostly proved this observation with the caveat that for tiny datasets R managed to outperform Delve a bit. This can be explained by overhead of a full-fledged database back-end employed by Delve.

In particular, on a set of 20 projects (20 Mb) R took 2.2 seconds for computations, while Delve required 4.7 seconds. In contrast, on a set of 1000 projects (254 Mb) R runs for 38 seconds, while Delve finishes in 14 seconds.

Delve authors advertise its ability to handle larger-than-RAM dataset transparently. R has several packages that attempt to provide the same ability but this is much less explored space for this language. We haven't studied performance of our implementations at this scale, although it could be an interesting future work.

4 Conclusion

Over-talkative API of `dplyr` and uncontrolled mutation available in R bit us several times during this work. For instance, certain `dplyr` functions changed interface slightly even during not-so-long period of time of this work, so we had to adjust. The ability to mutate tables served as a source of subtle bugs: an updated instance of a table was used sometimes when the original one was expected.

In comparison, working with Delve, although requiring a good amount of typing, is free from these issue and much easier to read after some break — most likely, because it speaks in application-specific terms as opposed to API-specific. With industrial-strength compilers such as Delve, we conclude that Datalog should be seriously considered as a tool for tasks in data analysis.

References

1. Avelino, G., Passos, L., Hora, A., Valente, M.T.: A novel approach for estimating truck factors. In: 2016 IEEE 24th International Conference on Program Comprehension (ICPC). pp. 1–10 (2016)
2. Fritz, T., Ou, J., Murphy, G.C., Murphy-Hill, E.: A Degree-of-Knowledge Model to Capture Source Code Familiarity, p. 385394. Association for Computing Machinery, New York, NY, USA (2010), https://doi.org/10.1145/1806799.1806856
3. Rahman, A., Farhana, E.: An exploratory characterization of bugs in covid-19 software projects (2020)