Technical Report: Answering Datalog Queries with Unix Shell Commands

Thomas Rebele, Thomas Pellissier Tanon, and Fabian M. Suchanek

Telecom ParisTech, Paris, France

June 22, 2018

Abstract

Dealing with large tabular datasets often requires extensive preprocessing. This preprocessing happens only once, so that loading and indexing the data in a database or triple store may be an overkill. In this paper, we present an approach that allows preprocessing large tabular data in Datalog – without indexing the data. The Datalog query is translated to Unix Bash and can be executed in a shell. Our experiments show that, for the use case of data preprocessing, our approach is competitive with state-of-the-art systems in terms of scalability and speed, while at the same time requiring only a Bash shell, and a Unix-compatible operating system. We also provide a basic SPARQL and OWL 2 to Datalog converter to make our system interoperable with semantic web standards.

1 Introduction

Many data analytics tasks work on tabular data. Such data can take the form of relational tables, TAB-separated files, or knowledge bases from the Semantic Web in the form of subject-predicate-object triples. Quite often, such data has to be preprocessed before the analysis can be made. In this paper, we focus on preprocessing in the form of select-project-join-union operations with recursion. This may include the removal of superfluous columns, the selection of rows of interest, or the amendment of certain rows by performing a join with another tabular dataset. In the case of knowledge bases, the preprocessing may involve extracting all instances of a certain class; in the case of graph data, the preprocessing may involve finding all nodes that are reachable from a certain node. These operations require recursion.

The defining characteristic of such pre-processing steps is that they are executed only once on the data in order to constitute the dataset of interest for the later analysis. It is only after such preprocessing that the actual data analysis task begins. This one-time pre-processing is the task that this paper is concerned with.

While there exist databases (or triple stores) to help with this preprocessing, loading large amounts of data into these systems may take hours or even days. Wikidata [41], for example, one of the largest knowledge bases on the Semantic Web, contains 267 GB of data. If only a small portion of the data is needed afterwards, then it is an overkill in terms of time and

space consumption to first load and index the entire dataset. After loading, purging the superfluous elements may again take several days, because indexes have to be rebuilt. All of this is frustratingly slow, as people who have worked with such data can confirm.

There are a number of systems that can work directly on the data, such as DLV [24] or RDFox [30]. However, these systems load the data into memory. While this works well for small datasets, it does not work for larger ones, such as Wikidata (as we show in our experiments). We are thus facing the problem of preprocessing large datasets that are not indexed, and that do not fit into main memory. There are tools to help with this (such as Spark [46], Flink [8], Dryad [20], Impala [22]), but these require the installation of particular software, the knowledge of particular programming languages, or even a particular distributed infrastructure.

In this paper, we develop a method to preprocess tabular file data without indexing it. We propose to express the preprocessing steps in Datalog [2]. Datalog is a particularly simple language, which has just a single syntactic construction, and no reserved keywords. Nevertheless, it is expressive enough to deal with joins, unions, projections, selections, negation, and in particular also with the recursivity that is required for preprocessing knowledge bases and graphs. We propose to compile this Datalog program automatically to Unix Bash Shell commands. We offer a Web page to this end: `https://www.thomasrebele.org/projects/bashlog`. The user can just enter the Datalog program, and click a button to obtain the Bash code. The Bash code can be copy-pasted into a Unix Shell, and executed without any prerequisites. Our method automatically optimizes the Datalog program with standard relational algebra optimization techniques, re-uses previously computed intermediate results, and produces a highly parallelized Shell script. For this purpose, our method employs pipes and process substitution. Our experiments on a variety of datasets show that this method is competitive in terms of runtime with state-of-the-art database systems, Datalog query answering engines, and triple stores.

More concisely, our approach allows the one-time preprocessing of tabular data in the form of select-project-join operations with negation and recursion

1. without any software installation beyond a Unix Bash shell on a POSIX compliant operating system
2. without any knowledge of programming or query languages other than Datalog
3. in a time that is competitive with conventional systems

The contributions of this paper are:

- a method that compiles Datalog to Unix Bash Shell commands
- the optimization of such programs
- extensive experiments on real datasets that show the viability of our method

This paper is structured as follows. We start with a motivating example in Section 2. Section 3 discusses related work, before Section 4 introduces preliminaries. Section 5 presents our approach, and Section 6 evaluates it. Section 7 shows how to use the Web interface, before Section 8 concludes.

2 Example

Setting. Since our approach may appear slightly unorthodox, let us illustrate our method by

a concrete example. Consider a knowledge base of the Semantic Web – for example BabelNet, DBpedia, YAGO, or Wikidata. These knowledge bases contain instances (such as *New York City>* or *USA>*), and these instances belong to certain classes (such as *City>* or *Country>*). The classes of a knowledge base form a hierarchy, where more specific classes (such as *President>*) are included in more general classes (such as *Politician>*). This data is typically stored in RDF. For simplicity and readability, assume that the data resides in a TAB-separated file *facts.tsv*:

```
Empire State Building     locatedIn     Manhattan
Manhattan                 locatedIn     New York City
New York City             locatedIn     USA
```

Figure 1: An excerpt from a knowledge base (facts.tsv)

Now consider a data engineer who wants to recursively extract all places located in the United States. Figure 2 shows how this query can be expressed in our Datalog dialect. The first line says that the predicate *fact* can be computed by printing out the file *facts.tsv*. Note the tilde, which signals that the body of the rule is a Unix command. The second line of the program says that the *locatedIn* predicate is obtained by selecting those facts with the predicate *locatedIn*. The third and fourth line say that we are interested in all places that are located in the United States.

```
fact(X, R, Y)  :~  cat facts.tsv
locatedIn(X, Y) :- fact(X, "locatedIn", Y) .
locatedIn(X, Y) :- locatedIn(X, Z), locatedIn(Z, Y) .
main(X) :- locatedIn(X, "USA") .
```

Figure 2: A Datalog program for finding places in the United States.

Translation. We propose to compile such Datalog programs automatically into Unix Bash Shell commands. For this purpose, the user can just visit our Web page and copy-paste the Datalog program there. She will then obtain a script similar to the code shown in Figure 3 (for readability, we have omitted a number of parameters, *sort* commands, and optimizations in this example). The code first extracts the *locatedIn>* facts from *facts.tsv*. From these facts, it extracts the places directly located in the *USA>*, and stores them in a file *delta.tmp* and in a file *full.tmp*. In the following *while* loop, the delta file is joined with all *locatedIn>* facts. The classes that had already been found previously are filtered out, and the remaining ones are added to *full.tmp* and put into the delta file. If that delta file is empty, a fixed point has been reached, and the loop stops.

This code can either be saved in a Shell script file, or else directly copy-pasted into the command-line prompt. When run, the code produces the list of places in the United States. This list is written to the standard output, and can be saved in a file.

```
awk '$2 == "locatedIn" {print $1 "\t" $3}' facts.tsv > li.tmp
awk '$2 == "USA" {print $0}' li.tmp | tee full.tmp > delta.tmp
while
    join li.tmp delta.tmp | comm -23 - full.tmp > new.tmp
    mv  new.tmp  delta.tmp
    sort -m -o full.tmp  full.tmp delta.tmp
    [ -s delta.tmp ];
do continue; done
cat full.tmp
```

Figure 3: The Datalog program in Bash (simplified).

Rationale. Such a solution has several advantages. First, it does not require any software installation. Installing and getting to run a complex system, such as BigDatalog [37], e.g., can take several hours. Our solution just requires a visit to a Web site. Second, the Bash shell has been around for several decades, and the commands are not just tried and tested, but actually continuously developed. Modern implementations of the *sort* command, e.g., can split the input into several portions that fit into memory, and sort them individually. Finally, the Bash shell allows executing several processes in parallel, and their communication is managed by the operating system.

3 Related Work

Relational Databases. Relational database management systems can handle data in the form of tables. Such systems include Oracle, IBM DB2, Postgres, and MySQL, as well as newer systems, such as MonetDB [6], and NoDB [5].

All of these systems (except NoDB) require loading the data and indexing it in its entirety. If the preprocessing is executed only once, then this time overhead may not pay off. We show in our experiments that just loading the data can take much longer than the entire preprocessing with our method. Furthermore, all of these systems (including NoDB) require the installation and setting up of software. Our approach, in contrast, can be run as a simple jar file, or even just as a service on the Web. The resulting Bash script then runs in a common shell console without any further prerequisites.

Triple Stores. Another class of systems target RDF knowledge bases. These are called *triple stores* and include OpenLink Virtuoso [13], Stardog, Jena [9], and others. Again, these require the loading and indexing of the data, and we will show that this is slower than our method for the purpose of preprocessing. Several approaches aim to speed up this loading: HDT [14] is a binary format for RDF, which can be used with Jena. Still, we find that this combination cannot deliver the speed of Bash. Linked Data Fragments [40] aim to strike a balance between downloading an RDF data dump and querying it on a server. The method thus addresses a slightly different problem. Apart from this, all of these approaches require the installation of software, while our approach works in a Bash shell.

NoSQL Databases. Several other data management systems target non-tabular data. These can be full-text indexing systems or key-value stores. Approaches, such as Cassandra, HBase, and Google's BigTable [10], target particularly large data. Our method, in contrast, aims at tabular data.

Distributed Processing. Distributed batch processing is a well known problem. The major paradigm used is Map-Reduce [12]. Dryad [20] provides a DAG dataflow system where the user can specify their own functions. These ideas have been implemented in systems, such as Apache Tez [33]. SCOPE [47], Impala [22], Apache Spark [46], and Apache Flink [8] provide advanced features, such as support for SQL or streams. While all these systems address our problem, they require the installation of particular software. What is more, they also require a distributed infrastructure. Our approach, in contrast, requires neither the installation of software nor a particular physical infrastructure. It just requires a Bash shell.

Datalog. The execution of Datalog is an active research topic, and the parallel processing of Datalog has been studied for over twenty years [44, 15, 16]. Several recent works have taken to improve the performance of Datalog execution by the use of modern data processing systems. For example, the work of [35] ports the usual semi-naive evaluation algorithm [2] to Hadoop. The work of [7] executes Datalog on top of both Map-Reduce [12] and Pregel [26]. Myria [42] provides a parallel distributed pipeline to evaluate Datalog programs and uses Postgres for storing facts. Recent works have used Apache Spark [46]. BigDatalog [37], in particular, tackles the problem of recursion in Spark. DatalogRA [32] deals with Datalog with data aggregation, and the work of [45] uses the naive evaluation strategy to evaluate Datalog programs and OWL ontologies. There is also recent work on recursive query evaluation on top of Spark [21]. The RDFox system [30] is specialized on Datalog queries on RDF data. There are also systems that can preprocess RDF datasets by filtering their content by SPARQL queries [27]. An example of such systems is RDFSlice [28]. It supports simple filtering and certain types of joins.

All of these systems address the same problem as us. Then again, all of these systems require the installation of software. The parallelized systems also require a distributed infrastructure. Our approach, in contrast, requires none of these. Nevertheless, we show in our experiments that the performance of our approach is competitive with the state of the art in the domain.

OWL Reasoners. OWL is an ontology language for Semantic Web data. Several systems can perform OWL reasoning. These include, e.g., Pellet [31], HermiT [36], RACER [18], and Fact++ [39]. Jena also supports OWL reasoning. These systems support negation, existential variables, and functional constraints. In this paper, we aim at a much simpler pre-processing language, Datalog. Datalog corresponds to a subset of the OWL 2 RL profile [29]. Thus, OWL reasoners are an overkill for our scenario. We make this point by comparing our approach with Jena and the Pellet successor Stardog.

4 Preliminaries

Datalog. We follow the definition of Datalog with negation from [1, 2]. In all of the following,

we assume 3 distinct sets of identifiers: predicates $\mathcal{P}$, variables $\mathcal{V}$, and constants $\mathcal{C}$. An n-ary *atom* is of the form $p(a_1, \ldots, a_n)$, with $p \in \mathcal{P}$ and $a_i \in \mathcal{C} \cup \mathcal{V}$ for $i = 1 \ldots n$. An atom is *grounded* if it does not contain variables. A *rule* takes the form

$$H :\!\!- B_1, \ldots, B_n, \neg N_1, \ldots, \neg N_m.$$

Here, H is the *head atom*, and $B_1, \ldots, B_n, N_1, \ldots, N_m$ are the *body atoms*. For $n = 0$, the rule simply takes the form "H.". We say that the body atoms $N_1, \ldots, N_m$ are *negated*. A rule is *safe* if each variable in the head or in a negated atom also appears in at least one positive body atom. We consider only safe rules in this work. A *Datalog program* is a set of rules. A set M of grounded atoms is a *model* of a program P, if the following holds: M contains an atom a if and only if P contains a rule $H :\!\!- B_1, \ldots, B_n, \neg N_1, \ldots, \neg N_m$, such that there exists a substitution $\sigma : \mathcal{V} \to \mathcal{C}$ with $\sigma(B_i) \in M$ for $i = 1 \ldots n$ and $\sigma(N_i) \notin M$ for $i = 1 \ldots m$ and $a = \sigma(H)$. A model is *minimal* if no proper subset is a model. In order to ensure the existence and the uniqueness of a minimal model for each given program with negation, we restrict ourselves to *stratified* Datalog programs [1, 2]. A Datalog program is stratified, if there exists a function σ from predicates to $\mathbb{N}$ such that for all rules of the form $H :\!\!- \ldots, B_i, \ldots,$ we have $\sigma(H) \geq \sigma(B_i)$, and for all rules of the form $H :\!\!- \ldots, \neg N_j, \ldots,$ we have $\sigma(H) > \sigma(N_j)$.

Relational Algebra. *Relational algebra* [11, 2] provides the semantics of relational database operations. There exist many different variants of relational algebra. Here, we want to use a variant that is equivalent to Datalog. A *table* is a set of tuples of the same arity. We write $arity(\cdot)$ for the arity of a tuple or the arity of the tuples in a set. We call *SPJAU unnamed relational algebra* the following set of operators on tables T and T' [2]:

Select (column equality): For $i, j \in [1, \ldots, arity(T)]$,
$\sigma_{i=j}(T) = \{t \in T \mid t(i) = t(j)\}$.

Select (column-value equality): For $i \in [1, \ldots, arity(T)]$,
$\sigma_{i=a}(T) = \{t \in T \mid t(i) = a\}$.

Project: For $i_1, \ldots, i_k \in [1, \ldots, arity(T)]$,
$\pi_{i_1,\ldots,i_k}(T) = \{\langle t(i_1), \ldots, t(i_k)\rangle \mid t \in T\}$

Constant Introduction: For $i \in [1, \ldots, arity(T) + 1]$,
$\pi_{i:a}(T) = \{\langle t(1), \ldots, t(i-1), a, t(i), \ldots, t(arity(T))\rangle \mid t \in T\}$.

Join: For $i_1, \ldots, i_k \in [1, \ldots, arity(T)]$, $i'_1, \ldots, i'_k \in [1, \ldots, arity(T')]$,
$T \bowtie_{i_1=i'_1,\ldots,i_k=i'_k} T' = \{\langle t, t'\rangle \mid t \in T \wedge t' \in T' \wedge t(i_1) = t'(i'_1) \wedge \cdots \wedge t(i_k) = t'(i'_k)\}$

Anti-join: For $n = arity(T')$ and $i_1, \ldots, i_n \in [1, \ldots, arity(T)]$,
$T \rhd_{i_1,\ldots,i_n} T' = \{t \mid t \in T \wedge \neg\exists t' \in T' : t(i_1) = t'(1) \wedge \cdots \wedge t(i_n) = t'(n)\}$

Union: $T \cup T'$ is the usual set union,
$T \cup T' = \{t \mid t \in T \vee t \in T'\}$

We often consider relational algebra expressions as syntax tree. The outermost operator represents the root. A node a is a child of another node b, if the output of a serves as input of

b. Descendants of a node a are all nodes, who are children of a or children of a descendant of a.

To map Datalog programs to relational algebra, we need an operator for recursive programs. For this, the work of [4] introduces a *least fixed point operator* (LFP).[1] For a function f from a table to a table, $\mu_x(f(x))$ is the least fixed point of f for the $\subseteq$ relation. The least fixed point can be computed with the semi-naive algorithm [2], as shown in Algorithm 1. This operator allows expressing all Datalog programs with stratified negation. We call *SPJAUR unnamed relational algebra* the SPJAU algebra extended with this operator. This algebra has the same expressivity as safe stratified Datalog programs [2].

Algorithm 1: Computation of $\mu_x(f(x))$ using the seminaive algorithm

```
1 Result ← ∅ ;
2 Δ ← ∅ ;
3 repeat
4 |   Δ ← f(Δ) \ Result ;
5 |   Result ← Result ∪ Δ ;
6 until Δ = ∅;
7 return Result;
```

> *Example (Relational Algebra):* Assume that there is a table *subclass* (which contains classes with their superclasses). Then the following expression computes the transitive closure of this table:
>
> $$\mu_x(\text{subclass} \cup \pi_{1,4}(x \bowtie_{2=1} x))$$
>
> This expression computes the least fixed point of a function. The function is given by a lambda expression. To compute the result of this expression, we execute the function first with the empty table, $x = \emptyset$. Then the function returns the *subclass* table. Then we execute the function again on this result. This time, the function joins *subclass* with itself, projects the resulting 4-column table on the first and last column, and adds in the original *subclass* table. We repeat this process until no more changes occur. This process terminates eventually, because the operators of our algebra are all monotonous – with the exception of the anti-join. Since our programs are stratified, x does not occur as the second argument of an anti-join, and thus the second argument does not change between iterations.

Unix. Unix is a family of multitasking computer operating systems. Unix and Unix-like systems are widely used on servers, on smartphones (e.g., Android OS), and on desktop computers (e.g., Apple's MacOS). One of the characteristics of Unix is that "Everything is a file", which means that files, pipes, the standard output, the standard input, and other resources can all be seen as streams of bytes[2]. For the present work, we are interested only in *TAB-separated*

[1] The work of [3] introduces a different relational algebra operator called α. However, α can express only transitive closures, and not arbitrary recursions.

[2] according to Linus Torvalds, the creator of Linux, `http://yarchive.net/comp/linux/everything_is_file.html`

byte streams, i.e., byte streams that consist of several *rows* (sequences of bytes separated by a newline character), which each consist of the same number of *columns* (sequences of bytes separated by a tabulator character). When printed, these byte streams look like a table.

The Bourne-again shell (Bash) is a command-line interface for Unix-like operating systems. It is the default interactive shell for users on most Linux and MacOS systems [43]. A *Bash command* is either a built-in keyword, or a small program. We are here concerned mainly with those commands of the POSIX standard that take one or several byte streams as input, and that produce one byte stream by printing to the standard output. We will use the following commands with the following parameters:

cat b_1 **...** b_n
: Prints the byte streams b_1 ... b_n one after the other.

sort -t $'\t' -k**$c_1$ **. . . -kc_n b
: Sorts the byte stream b on columns $c_1, \ldots, c_n$ and prints the result.

sort -u -m -o b_0 b_1 b_2
: Merges the sorted byte streams b_1 and b_2, eliminating duplicate lines, and prints the output to b_0.

comm -23 b_1 b_2
: Prints the lines that appear in the sorted byte stream b_1, but not in the sorted byte stream b_2.

join -t $'\t' -1**$c_1$ **-2c_2 **-o** d [**-v1**] b_1 b_2
: Joins the byte streams b_1 and b_2 on column c_1 of b_1 and column c_2 of b_2, and prints the output columns d of the result. For this, b_1 has to be sorted on column c_1, and b_2 has to be sorted on column c_2. The command supports joining on a single column only. With `-v1`, the command outputs those lines of b_1 that could not be joined.

echo -n > f
: Creates an empty file f, overwriting f if it exists.

mv f_1 f_2
: Renames file f_1 to f_2.

AWK. We will also use the command `awk`. It implements an interpreter for the AWK programming language. We use awk commands of the following form

```
awk -F$'\t'  'p' b
```

The `-F` option makes awk use the TAB character for column separation. This character can then be referred to as `FS`. b denotes the input byte stream, and p is an awk program of the following form:

c **{ print $**$i_1$ **FS** . . . **FS $**$i_k$ [>> "f"] **}**
: This AWK program prints out the columns $i_1, \ldots, i_k$ of the input byte stream, if a certain

condition c is fulfilled. A condition is either a column equality `$`i `==` `$`j, a column-value equality `$`i `== "value"`, or a combination of several conditions c_1 `&&` ... `&&` c_n. An empty condition always succeeds. If the optional `>> "`f`"` is given, the output is appended to file f.

{ print $0 FS $**$i_1$ s ... s **$**$i_k$ [`>> "`f`"`] **}
This AWK program prints a line of the input byte stream, and appends a single column to it. This single column is the concatenation of the columns $i_1, \ldots, i_k$, separated by the character s. We used the ASCII character 002 for this purpose, but another character can be used, as long as it does not appear in the Datalog program[3]. If a file f is given, the result is appended to f. We use this program to create a column on which we can run the `join` command.

Finally, we make use of the Bash control structure `while`, which we use as follows:

```
while c [ -s f ];
    do continue;
done
```

This code runs the sequence of commands c repeatedly until the file f is empty.

Pipes. When a command or control structure is executed, it becomes a *process*. In the Unix-like operating systems, processes can communicate through *pipes*. A pipe is a byte stream that can be filled by one process, and read by another process. If the producing process is faster than the receiving one, the pipe buffers the stream, or blocks the producing process if necessary. In Bash, pipes can be constructed as follows:

```
p1 | p2
```

This construction sends the output of process p_1 as input to process p_2. If p_2 is a command, the input byte stream no longer has to be specified explicitly. A process p can also send its output byte stream to two other byte streams b_1 and b_2 (including pipes or files), as follows:

```
p | tee b1 [b3 ...] > b2
```

A pipe can also be constructed "on the fly" by a so-called *process substitution*, as follows:

```
p1 <( p2 )
```

This construction runs the process p_2, and pipes its output stream into the first argument of the process p_1. Finally, it is possible to create a *named pipe* n with the command `mkfifo` n. Such a pipe can be closed with the Bash command `exec` d`>`n`; exec` d`>&-`, where d is an integer greater than 2, representing a not yet used file descriptor.
We will now see how these constructions can be used to execute Datalog programs.

[3] other excluded characters are ASCII characters 000, 001, and TAB

5 Approach

5.1 Datalog Dialect

In our concrete application of Datalog, we assume that the set $\mathcal{P}$ of predicates is the set of strings that consist of letters, and that start with a lower-case letter. The set of variables $\mathcal{V}$ is the set of strings that consist of letters, and that start with an upper-case letter. The set $\mathcal{C}$ of constants is the set of all strings that start and end with an ASCII double quotation mark. Constants may not contain ASCII double quotation marks other than the two delimiters. They may also not contain TAB characters, newline characters, or the separator character that we use in the AWK programs. Future versions of our compiler may relax these restrictions, but for the present work we stay with these conventions for readability.

For our purposes, the Datalog program has to refer to files or byte streams of data. For this reason, we introduce an additional type of rules, which we call *command rules*. A command rule takes the following form:

$$p(x_1, ..., x_n) :\sim c$$

Here, p is a predicate, $x_1, \ldots, x_n$ are variables, and c is a Bash command. Such a rule ends syntactically not with a dot (because Bash commands often contain dots), but with a new line. Notice the tilde in the place of the usual hyphen to distinguish command rules from ordinary rules. Semantically, this rule means that executing c produces a TAB-separated byte stream of n columns, which will be referred to by the predicate p in the Datalog program. In the simplest case, the command c just prints a file, as in `cat facts.tsv`. However, the command can also be any other Bash command, such as `ls -1`.

Our goal is to compute a certain output with the Datalog program. This output is designated by the distinguished head predicate `main`. An *answer* of the program is a grounded variant of the head atom of this rule that appears in the minimal model of the program. See again Figure 2 on page 3 for an example of a Datalog program in our dialect. We emphasize that our dialect is a generalization of standard Datalog, so that any normal Datalog program can be run directly in our system.

Our approach can also work in "RDF mode". In that mode, the input consists of a SPARQL query [19], a TBox in the form of OWL 2 RL [29], and an ABox in the form of an N-Triples file F. We convert the OWL ontology and the SPARQL query to Datalog rules, and we include the following AWK command in the Bash script to transform file F to a TSV file:

```
fact(S, P, O)  :~  awk '{ sub(" ", "\t");  sub(" ", "\t");      sub
    (/ \.$/, "");  print $0 }' F
```

The command replaces the spaces that separate the three parts by TAB characters, and removes the dot character at the end. If necessary, a similar AWK command can transform the output of the Bash script back to the N-Triples format.

The Datalog program contains predicate `main`, which returns the result of the SPARQL query on the file F, while having used the provided ontology for expansion. For example, we are able to produce the rule `hasParent`(X, Y) :— `hasFather`(X, Y) from the OWL axiom *subPropertyOf*(*hasFather*, *hasParent*). Like RDFox [30], we assume that all classes and properties axioms are provided by the ontology, and that they are not queried by the SPARQL query.

This assumption allows us to produce efficient programs. We do not yet support OWL axioms related to literals. Our SPARQL implementation supports basics graph patterns, property paths without negations, OPTIONAL, UNION and MINUS.

5.2 Loading Datalog

Algorithm 2: Translation from datalog to SPJAUR algebra

```
1  fn mapPred (p, cache, P) is
2      if p ∈ cache then
3          return x_p
4      end
5      plan ← ∅
6      newCache ← cache ∪ {p}
7      foreach rule
              p(H_1, ..., H_{n_h}) :— r_1(X^1_1, ..., X^1_{n_1}), ..., r_n(X^n_1, ..., X^n_{n_n}),
                                      ¬q_1(Y^1_1, ..., Y^1_{m_1}), ..., ¬q_m(Y^m_1, ..., Y^m_{m_m})
              in P do
8          bodyPlan ← {()}
9          foreach r_i(X^i_1, ..., X^i_{n_i}) do
10             atomPlan ← mapPred(r_i, newCache, P)
11             foreach (X^i_j, X^i_k) | X^i_j = X^i_k, j ≠ k do
12                 atomPlan ← σ_{X^i_j = X^i_k}(atomPlan)
13             end
14             bodyPlan ← bodyPlan ⋈ atomPlan
15         end
16         foreach ¬q_i(Y^i_1, ..., Y^i_{m_i}) do
17             atomPlan ← mapPred(q_i, ∅, P)
18             foreach (Y^i_j, Y^i_k) | Y^i_j = Y^i_k, j ≠ k do
19                 atomPlan ← σ_{Y^i_j = Y^i_k}(atomPlan)
20             end
21             bodyPlan ← bodyPlan ▷ atomPlan
22         end
23         plan ← plan ∪ π_{H_1,...,H_{n_h}}(bodyPlan)
24     end
25     foreach rule p(H_1, ..., H_{n_h}) :∼ c in P do
26         plan ← plan ∪ π_{H_1,...,H_{n_h}}(c)
27     end
28     return μ_{x_p}(plan)
29 end
```

Our approach takes as input a Datalog program, and produces as output a Bash Shell script. For this purpose, our approach first builds a relational algebra expression for the `main` predicate of the Datalog program with Algorithm 2. The algorithm takes as input a predicate p,

a cache, and a Datalog program P. The method is initially called with p=`main`, *cache*=$\emptyset$, and the Datalog program that we want to translate. The cache stores already computed relational algebra plans. In all of the following, we assume that p always appears with the same arity in P. If that is not the case, p can be replaced by different predicates, one for each arity. The full materialization of predicate p is the least fixed point of the union of the application of all the rules producing p.

Our algorithm first checks whether p appears in the cache (Line 2-4). In that case, p is currently being computed in a previous recursive call of the method, and the algorithm returns a variable x indexed by p (Line 3). This is the variable for which we compute the least fixed point.

Then, the algorithm traverses all rules with p in the head (Line 7). For every rule

$$p_h(H_1, \dots, H_{n_h}) :\!\!- r_1(X_1^1, \dots, X_{n_1}^1), \dots r_n(X_1^n, \dots, X_{n_n}^n), \\ \neg q_1(Y_1^1, \dots, Y_{m_1}^1), \dots, \neg q_m(Y_1^m, \dots, Y_{m_m}^m)$$

the algorithm recursively retrieves the plan for the r_i (Line 9-15), and the q_j (Line 16-22). It then adds a nested $\sigma_{j=k}$ if there are j, k such that $X_j^i = X_k^i$ (Line 11-13), and $Y_j^i = Y_k^i$ (Line 18-20) respectively. Then it combines these expressions pair-wise from left to right by adding the relevant join constraints between the r_i (Line 14). It also adds the anti-join constraints between the results of the combinations of the left elements and the q_j (Line 21). At the end of the for-loop, the algorithm puts the resulting formula into a project-node that extracts the relevant columns (Line 23). Then, the algorithm processes all command rules, wraps each command in a project-node, and adds it to the plan (Line 26). Finally, the algorithm wraps the plan in a least fixed point operator (Line 28). A subsequent optimization step removes this operator if it is not necessary.

We can add the anti-join constraints here, because the program is stratified. That means, applying `mapPred` on a negated atom q_j never reaches a rule with head p again. Furthermore, the program is safe, so all columns returned by the second parameter of the anti-join appear in the columns of the first parameter.

The implementation of the algorithm builds a directed acyclic graph (DAG) instead of a tree. When the function `mapPred` is called with the same arguments as in a previous call, it returns the result of the previous call. This implementation allows us to re-use the same sub-plan multiple times in the final query plan, thereby reducing its size. The technique also allows the Bash programs to re-use results that have already been computed.

Example (Datalog Translation): Assume that there is a two-column TAB-separated file `subclass.tsv`, which contains each class with its subclasses. Consider the following Datalog program P:

```
1 directSubclass(x,y)  :~  cat subclass.tsv
2 main(x,y) :- directSubclass(x,y).
3 main(x,z) :- directSubclass(x,y), main(y,z).
```

We call *mapPred*(`main`$, \emptyset, P$). Our algorithm goes through all rules with the head pred-

icate main. These are Rule 2 and Rule 3. For Rule 2, the algorithm recursively calls *mapPred*(directSubclass, {main}, P). This returns

$$\mu_{x_{\texttt{directSubclass}}}(\emptyset \cup [\texttt{cat subclass.tsv}]).$$

Since the lambda-expression does not contain the variable $x_{\texttt{directSubclass}}$, this is equivalent to [cat subclass.tsv].

For Rule 3, we call *mapPred*(directSubclass, {main}, P), which returns [cat subclass.tsv] just like before. Then we call *mapPred*(main, {main}, P), which returns $x_{\texttt{main}}$, because main is in the cache. Thus, Rule 3 yields

$$\pi_{1,4}(\texttt{[cat subclass.tsv]} \bowtie_{2=1} x_{\texttt{main}})).$$

Finally, the algorithm constructs the result

$$\mu_{\lambda x_{\texttt{main}}}(\texttt{[cat subclass.tsv]}\pi_{1,4}(\texttt{[cat subclass.tsv]} \bowtie_{2=1} x_{\texttt{main}}))$$

5.3 Producing Bash Commands

The previous step has translated the input Datalog program to a relational algebra expression. Now, we translate this expression to a Bash command by the function b, which is defined as follows:

$b([\![c]\!]) = c$
: An expression of the form $[c]$ is already a Bash command, and hence we can return directly c.

$b(e_1 \cup \ldots \cup e_n)$
: To remove possible duplicates, we translate a union into

```
sort -u <(b(e1)) ... <(b(en))
```

$b(e_1 \bowtie_{x=y} e_2)$
: A join of two expressions e_1 and e_2 on a single variable at position x and y, respectively, gives rise to the command

```
join -t$'\t'  -1x -2y \
    <(sort -t$'\t'  -kx <(b(e1))) \
    <(sort -t$'\t'  -ky <(b(e2)))
```

This command sorts the byte streams of $b(e_1)$ and $b(e_2)$, and then joins them on the common column.

$b(e_1 \bowtie_{x=y,\ldots} e_2)$
: The Bash join command can perform the join on only one column. If we want to join

on several columns, we have to add a new column to each of the byte streams. This new column concatenates the join columns into a single column. This can be achieved with the following AWK program, which we run on both $b(e_1)$ and $b(e_2)$:

```
{ print $0 FS $j1 s $j2 s ... s $jn }
```

Here, the indices $j_1, \ldots, j_n$ are the positions of the join columns in the input byte stream, and s is the separation character (see Section 4). Once we have done this with both byte streams, we can join them on this new column in the same way as described above for simple joins. This join also removes the additional column.

$b(e_1 \triangleright_x e_2)$

Just as a regular join, an anti-join becomes a `join` command. We use the parameter `-v1`, so that the command outputs only those tuples emerging from e_1 than cannot be joined with those from e_2. We deal with anti-joins on multiple columns in the same way as with multi-column joins.

$b(\pi_{i_1,\ldots i_n}(e))$

A projection becomes the following AWK program, which we run on $b(e)$:

```
{ print $i1 FS ... FS $in }
```

$b(\pi_{i:a}(e))$

A constant introduction becomes the following AWK program, which we run on $b(e)$:

```
{ print $1 FS ... $(i-1) FS a FS $i FS ... $n }
```

$b(\sigma_{i=v}(e))$

A selection node gives rise to the following AWK program, which we run on $b(e)$:

```
$i == "v" { print $0 }
```

This command can be generalized easily to a selection on several columns.

Note that several of these translations produce process substitutions. In such cases, Bash starts the parent process and the inner process in parallel. The parent process will block while it cannot read from the inner processes. Thus, only the innermost processes run in the beginning. Every process is run asynchronously as soon as input and CPU capacity is available. Thus, our Bash program is not subject to the forced synchronization that appears in Map-Reduce systems.

5.4 Recursion

We have just defined the function b that translates a relational algebra expression to a Bash command. We will now see how to define b for the case of recursion. A node $\mu_x(f(x))$ becomes

```
echo -n > delta.tmp;  echo -n > full.tmp
while
    sort b(f(delta.tmp)) | comm -23 - full.tmp > new.tmp;
    mv  new.tmp  delta.tmp;
    sort -u -m -o full.tmp  full.tmp <(sort delta.tmp);
    [ -s delta.tmp ];
do continue; done
cat full.tmp
```

This code uses 3 temporary files to compute the least fixed point of f: `full.tmp` contains all facts inferred until the current iteration. `delta.tmp` contains newly added facts of an iteration. `new.tmp` is used as swap file.

The code first creates `delta.tmp` and `full.tmp` as empty files. It then runs f on the delta file. The `comm` command compares the sorted outcome of f to the (initially empty) file `full.tmp`, and writes the new lines to the file `new.tmp`. This file is then renamed to `delta.tmp`. This procedure updates the file `delta.tmp` to contain the newly added facts. The `comm` command cannot write directly to `delta.tmp`, because this file also serves as input to the command produced by $b(f($`delta.tmp`$))$.

The following `sort` command merges the new lines into `full.tmp`, and writes the output to `full.tmp` (the `sort` command can write to a file that also serves as input). Now, all facts generated in this iteration have been added to `full.tmp`. The `[...]` part of the code lets the loop run while the file `delta.tmp` is not empty, i.e., while new lines are still being added. If no new lines were added, the code quits the loop, and prints all facts. Note that, due to the monotonicity of our relational algebra operators, and due to the stratification of our programs, we can afford to run f only on the newly added lines.

Our method generates such a loop for each recursion. Since such loops can run in several processes in parallel, we generate different temporary file names for each recursion. We also take care to delete the temporary files after the Bash program finishes.

5.5 Materialization

Materialization nodes. To avoid re-computing a relational algebra expression that has already been computed we materialize some intermediate computations. For this purpose, we introduce a new type of operator to the algebra, the materialization node. A materialization node ⧉$(m,(\lambda y : p_{m\rightarrow y}))$ has two sub-plans: m is the plan that is used multiple times, and that we will materialize. The function $(\lambda y : p_{m\rightarrow y})$ is the main plan, and takes the materialized plan as parameter. The plan $p_{m\rightarrow y}$ is the original plan p with all occurrences of m replaced by the variable y.

A sub-plan m will be materialized if one of the following applies:

- m is used by different nodes of the query plan, i.e., m has multiple parents in the relational algebra expression DAG.
- there exists a node $\mu_{x_q}(f)$ where f contains m, but m does not contain x_q.

We proceed in two phases: We first check whether to materialize a sub-plan, and then

decide where to materialize it. First, we discuss the simple case where no μ node is the ancestor of another μ node. We start with the deepest node m that fulfills one of these conditions. If m does not contain a node x of an LFP node, then and we replace the query plan p by ⧉$(m, (\lambda y : p_{m \to y})$. If m contains a node x_q of an $\mu x_q(q)$, and occurs at least twice in q, we materialize m within the μ node: $\mu_{x_q}($⧉$(m, (\lambda y : q_{m \to y})))$. We repeat this procedure until no more nodes can be materialized. Finally, if m contains variables x of several μ nodes, these LFP nodes are nested. We materialize m directly below the deepest μ node whose variable x is contained in m.

Translation. A node ⧉$(m,(\lambda y : p))$ gives rise to the following translation to Bash commands:

```
mkfifo lock_t
(
    b(m) > t
    mv lock_t done_t
    cat done_t &
    exec 3> done_t
    exec 3>&-
) &
b_{y→t}(p)
rm t
```

Here, t is a temporary file name. Each materialization node uses its own temporary file t, because ⧉-nodes can be nested. Our translation allows us to execute several materialization operations in parallel. The function b is the Bash translation function defined in Section 5.3. Commands that use t have to wait until $b(m)$ finishes. We ensure this by making these commands read from the named pipe `lock_`t. Since this pipe contains no data, the commands block. When $b(m)$ finishes, the two `exec` commands close the named pipe, thus unblocking the commands that need t. There can be a rare race condition: $b(m)$ may finish before any process that listens on the pipe was started. In that case, the two `exec` commands try to close a pipe that has no listeners. In such cases, the `exec` command would block. We solve this problem by reading from the pipe with a `cat` command that runs in the background. This way, the pipe has at least one listener, and the `exec` commands close the pipe. This, however, brings a second problem: If the processes that listen on the pipe were still not started, they would try to listen to a closed pipe. To avoid this problem, we rename the pipe from `lock_`t to `done_`t. Such a renaming does not affect any processes that already listen on the pipe, but it prevents any new processes from listening on the pipe under the old name.

Finally, we actually use the materialized plan. The function $b_{y \to t}$ extends b as follows: $b_{y \to t}(y)$ generates the bash code `cat` t; and all plan nodes p_i that have a child y generate the bash code

```
cat lock_t 2> /dev/null
b(p_i)
```

As explained above, the `cat` command blocks the execution until t is materialized. The part

"2> /dev/null" removes the error message in case `cat` is executed when the pipe was already renamed.

5.6 Optimization

Algebra Optimizations. We apply the usual optimizations on our relational algebra expressions: we push selection nodes as close to the source as possible; we merge unions; we merge projects; we apply a simple join re-ordering. Additionally, we remove LFP when there is no recursion; and we extract from an LFP node the non-recursive part of the inner plan (so that it is computed only once at the beginning of the fixed point computation).

Removing superfluous calls. In the Datalog program, a recursive call of a predicate occurs, if the predicate takes itself as input (eventually with mediating rules). In the algebra, a recursive call corresponds to the x of a LFP node. We remove recursive calls when they are not contributing new output. We call these calls "superfluous". The following example illustrates the concept of superfluous calls.

> *Example (Superfluous calls):* We want to obtain a list of professors. The rules of the knowledge base are as follows:
>
> ```
> Professor(X) :- Person(X), teachesCourse(X,Y).
> Professor(X) :- successorOf(X,Y), Professor(Y).
> Person(X) :- Employee(X).
> Person(X) :- Professor(X).
> ```
>
> The predicates `teachesCourse`, `successorOf`, and `Employee` represent input relations. The first and last rule combined are equivalent to the rule
>
> ```
> Professor(X) :- Professor(X), teachesCourse(X,Y)}.
> ```
>
> It is obvious that this rule does not add any new professors to the output. If X is not a professor, then the first atom of the body `Professor(X)` is not part of the model, and so the rule will not apply. If X is a professor, then the first atom is part of the model. However, in that case, applying the rule will infer `Professor(X)`, which is already in the model, so nothing changes.
>
> If a rule contains the head atom in the body (with the same variables in the same order), we can safely ignore the rule. We will apply this principle on the algebra plan. The algebra plan of this example is shown in Figure 4 on the next page. Only the recursive call x in the left subtree is superfluous.

First, we determine whether a recursive call x is superfluous, by tracing its columns until the LFP node. If they arrive completely, and in order, at the LFP node, all output computed in previous iterations would just pass trough the path from x to the LFP node.

In more detail, we detect superfluous recursive calls of a $\mu_x(f)$ node as follows. First, we define a function that returns all paths from an algebra tree node to its leaves, including the

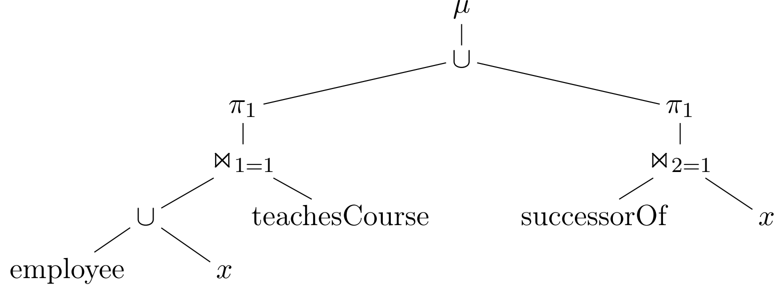

Figure 4: Algebra plan for removing superfluous calls example

position of the arguments:

- $paths(t) = \{t\}$, if t is an input table
- $paths(\Box(t)) = \{\Box/\, p \mid p \in paths(t)\}$, where $\Box$ is a relational algebra operator of the form $\sigma_{i=\dots}$, or $\pi_{i\dots}$
- $paths(\Box(t_1, t_2)) = \bigcup_i \{\Box\ i/\, p \mid p \in paths(t_i)\}$, where $\Box$ is an operator of the form $\bowtie_{\dots=\dots}$, $\triangleright_{\dots=\dots}$, or $\cup$
- $paths(\mu_x(f)) = \{\mu/p \mid p \in paths(f)\}$

> *Example (Paths):* We will list here all paths of the child node of the least fixed point in Figure 4: $\{\ \cup\ 1/\pi_1/\bowtie_{1=1}\ 1/\cup\ 1/\text{employee}$, $\cup\ 1/\pi_1/\bowtie_{1=1}\ 1/\cup\ 2/x$, $\cup\ 1/\pi_1/\bowtie_{1=1}\ 2/\text{teachesCourse}$, $\cup\ 2/\pi_1/\bowtie_{2=1}\ 1/\text{successorOf}$, $\cup\ 2/\pi_1/\bowtie_{2=1}\ 2/x\ \}$

Next, we define how the columns of a variable x of an $\mu_x(f)$ node propagate through the relational algebra expression. Let z be any constant, representing a column with an unknown value, and let $l = arity(\mu_x(f))$. The function *columns* maps a path to a word over the alphabet $\{z, c_1, \dots, c_l\}$. The character c_i represents the i-th column of x. Let n be the arity of the first relational operator of the argument of *columns*, and let $z^n = z \dots z$, the word that repeats the character z n-times. The function *columns* is defined recursively as follows:

- $columns(x) = c_1 \dots c_l$
- $columns(\sigma_{i=\dots}/p) = columns(p)$
- $columns(\pi_{i_1,\dots,i_k}/p) = w_{i_1} \dots w_{i_k}$, where $w_1 \dots w_j = columns(p)$
- $columns(\pi_{i:a}/p) = w_1 \dots w_{i-1}\, z\, w_i \dots w_j$, where $w_1 \dots w_j = columns(p)$
- $columns(\bowtie_{\dots=\dots}\ 1/p) = columns(p) \circ z^{n-j}$
- $columns(\bowtie_{\dots=\dots}\ 2/p) = z^{n-j} \circ columns(p)$
- $columns(\triangleright_{\dots}\ 1/p) = columns(p)$
- $columns(\cup\ i\,/p) = columns(p)$, for $i = 1$ and $i = 2$
- otherwise, $columns(p) = z^n$

> *Example (Columns):* We apply the function *columns* to the recursive calls x in Figure 4: In our example, $columns(x) = c_1$, as the query outputs only a single column. For the left, $columns(\cup\ 1/\pi_1/\bowtie_{1=1}\ 1/\cup\ \ 2/x) = c_1$, so the variables of the left recursive call are passed through the tree. For the right, $columns(\cup\ 2/\pi_1/\bowtie_{2=1}\ 2/x\ \}) = z$, which means that the column of x did not arrive at the μ node.

The superfluous variables x of $\mu_x(f)$ are the paths from the LFP node to a x node whose output columns are all columns of x in the right order:

$$superfluous_calls = \{p \mid p = p_1 \dots p_n \in paths(f)\ \wedge \\ p_n = x \wedge columns(p) = columns(x)\}$$

After detecting superfluous calls, we replace them with an empty table, and apply the usual optimizations.

> *Example (Superfluous calls):* The only path with $columns(p) = columns(x)$ is $\cup\ 1/\pi_1/\bowtie_{1=1}\ 1/\cup\ \ 2/x$. It is the only superfluous call. We therefore simplify the plan in Figure 4 by replacing the left x with the empty table $\varnothing$. Figure 5 shows the simplified result.

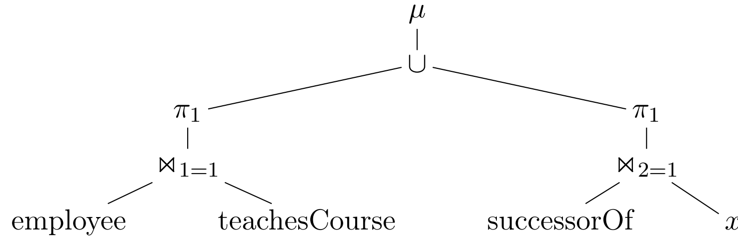

Figure 5: Algebra plan after removing superfluous calls

Semi-naive evaluation. We also optimize the fix point computation by using the same idea as the semi-naive Datalog evaluation [2]: For every expression of the form $p = \mu_{x_p}(f)$, we introduce a placeholder δ_p that represents the facts created by the last iteration of the fix-point computation. Then we apply the operation I_δ on f, which is defined as follows:

- $I_\delta(x) = \delta_p$
- $I_\delta(R_1 \bowtie_s R_2) = (I_\delta(R_1) \bowtie_s R_2) \cup (R_1 \bowtie_s I_\delta(R_2))$ if both R_1 and R_2 contain occurrences of x.
- in all other cases $I_\delta(\phi)$, we apply I_δ recursively on the children of ϕ.

It is easy to see that $\mu_x(f) = \mu_{x_p}(I_\delta(f))$.

If f represents a simple transitive closure, i.e.,

$$f = \pi_{1,4}(x_p \bowtie_{2=1} x_p) \text{ or } f = \pi_{2,3}(x_p \bowtie_{1=2} x_p),$$

we avoid the second join, and use the plan

$$\mu_{x_p}(\pi_{1,4}(\delta_p \bowtie_{2=1} x_p)).$$

Join reordering. Reordering join operations can greatly reduce the cardinality of intermediate results. There is an extensive corpus of work about different techniques to this end. For now, we apply only a simple join-reordering, and reserve more advanced reordering algorithms for future work. Let n be the top-most join-node in our relational algebra expression. We do a depth-first search that stops at every non-join node, and collect them into a list L, from left to right. From L, we construct a new ordered list L' as follows: We choose node p of $L \setminus L'$ that has the most join conditions with the nodes in L'. If there are no common join conditions, we choose the node that has the most join conditions overall. We add p to L', and repeat the procedure until L' contains all nodes of L.

We select the first two elements of L', and join them with all join conditions that can be applied to their columns. We join every following element in L' to the right of the previous join with all applicable join conditions. Finally, we wrap the so constructed join tree into a projection node, in order to retrieve the same columns that the original join n would have produced. We apply this method recursively to all nodes in L'.

Parallel file scanning. Preliminary experiments showed that the first phase of query execution was IO bound. It may happen that several commands read simultaneously from the same file, selecting on different conditions, or projecting different columns. If all commands access the file at the same position, the operating system can buffer the relevant block of the file. However, since our commands are not synchronized, they usually read from different blocks of the file, which makes the access very slow. To mitigate this problem, we collect different AWK commands that select or project on the same file into a single AWK command. This command runs only once through the file, and writes out all selections and projections into several files, one for each original AWK command.

Post-processing. Our Bash program may nest several `sort` commands. This can happen, e.g., if a union is the object of a join. We detect such cases, and remove redundant `sort` commands. To make sure that the final output of our program contains only unique results, we run `sort -u` on the final output.

Astonishingly, `sort` and `join` use a different character order. This means that `join` with input from `sort` warns about unsorted input. To mitigate this problem, we add the following as a first line to our program:

```
export LC_ALL=C
```

This construction forces all commands to use the default language for input and output, and to sort bytewise. This ensures, in particular, that our command works with UTF-8 encoded files. It also improves the processing speed.

6 Experiments

To show the viability of our approach, we ran our method on several datasets, and compared it to several competitors. All experiments were run on a laptop with Ubuntu 16.04, an Intel

Core i7-4610M 3.00 GHz CPU, 16 GB of memory, and 3.8 TB of disk space. We used GNU coreutils 8.25 for POSIX commands, and mawk 1.3.3 for AWK.

We emphasize that our goal is not to be faster than each and every system that currently exists. For this, the corpus of related work is simply too large (see Section 3). This is also not the purpose of Bash Datalog. The purpose of Bash Datalog is to provide a preprocessing tool that runs without installing any additional software besides a Bash shell. This is an advantage that no competing approach offers. Our experiments then serve mainly to show that our approach is generally comparable in terms of speed and scalability with the state of the art.

6.1 Lehigh University Benchmark

Dataset. Our first dataset is the Lehigh University Benchmark (LUBM) [17]. LUBM is a standard dataset for Semantic Web repositories. It models universities, their employees, students, and courses. The dataset is parameterized by the number of universities, and hence its size can be varied. LUBM comes with 14 queries, which test a variety of different usage patterns. These queries are expressed in SPARQL. For our purposes, we translated the queries to Datalog.

Competitors. We compare our approach to the following competitors:

DLV[4] is a disjunctive logic programming system. It can handle Datalog queries out of the box [24].

Souffle[5] is a translator for a rule based language to C++. It supports the parallel execution of Datalog programs [34].

RDFox[6] is an in-memory RDF triple store that supports shared memory parallel Datalog reasoning [30].

Jena[7] is an open-source RDF triple store written in Java. It can execute SPARQL queries [9]. We used the TDB implementation of Jena.

Jena+HDT[8] is a combination of Jena with the binary format HDT for triple data [14].

Stardog[9] is commercial knowledge graph platform that allows answering SPARQL queries on RDF data.

Virtuoso[10] is a commercial platform that also allows answering SPARQL queries on RDF data [13].

[4] `http://www.dlvsystem.com/dlv/`, v. Dec 17 2012
[5] `http://souffle-lang.org/`, v. 1.2.0
[6] `https://www.cs.ox.ac.uk/isg/tools/RDFox/`
[7] `https://jena.apache.org/`, v. 3.4.0
[8] `http://www.rdfhdt.org/`, v. 1.1.2
[9] `https://www.stardog.com/`, v. 5.2.0
[10] `https://virtuoso.openlinksw.com/`, v. 7.2.5 OS Edition

Query	Bash	DLV	Souffle	RDFox + TDB	Jena + HDT	Jena	Stardog	Virtuoso − I	Postgres* + I		NoDB* − I	MonetDB* + I	Slice*	RDF-
1	**0.7**	9.6	7.8	2.2	25.7	26.4	12.8	11.7	4.8	27.5	>600	1.8	2.6	12.6
2	**1.3**	9.3	119.4	2.2	281.3	>600	13.6	11.8	–	–	–	–	–	–
3	**0.9**	9.2	8.8	2.2	26.7	27.0	12.7	11.5	7.8	30.5	292.9	1.9	2.7	–
4	**1.9**	9.3	11.9	2.2	>600	>600	13.2	12.2	14.7	37.4	>600	2.3	3.1	–
5	**1.4**	9.3	10.3	2.2	>600	>600	12.9	–	28.9	51.6	–	2.2	3.0	–
6	**1.9**	9.4	11.1	2.4	>600	>600	17.6	–	21.2	43.9	–	3.9	4.7	–
7	2.4	9.5	56.3	**2.2**	>600	>600	13.4	–	21.6	44.3	>600	3.0	3.9	–
8	2.5	9.3	12.9	**2.3**	>600	>600	15.3	–	–	–	–	–	–	–
9	3.1	9.4	>600	**2.3**	>600	>600	13.4	–	71.1	93.8	>600	25.5	26.8	–
10	**2.0**	9.3	11.6	2.2	>600	>600	13.5	–	23.0	45.7	>600	5.8	7.1	–
11	**0.9**	9.3	7.8	2.2	25.3	35.7	13.0	11.8	–	–	–	–	–	–
12	**1.4**	9.2	10.9	2.2	>600	>600	13.1	–	–	–	–	–	–	–
13	**1.4**	9.2	10.1	2.2	>600	>600	12.9	–	8.4	31.1	>600	4.3	5.4	–
14	**0.8**	9.5	6.7	2.3	34.5	24.8	13.5	12.0	4.8	27.5	19.1	1.9	2.7	3.7
of which loading:				16.8	7.4	11.0	5.9	4.4	24.3		1.7	2.5		

Table 1: Runtime for the LUBM queries with 10 universities (155 MB), in seconds.
* = no support for querying with a TBox. We folded the TBox into the query.
+/−I = with/without indexes. A dash means that the query is not supported.

Postgres[11] is a relational database system that is developed as open-source. It supports SQL queries.

NoDB[12] is an extension of Postgres, which can execute SQL queries directly on TAB-separated files.

MonetDB[13] is an open source column-oriented database management system, which also supports SQL queries [6].

RDFSlice[14] is a tool for filtering RDF triples, implementing the Extract-Transform-Load paradigm for RDF data [28].

For the database systems (Postgres, NoDB, and MonetDB), we translated the queries to SQL. For this purpose, we used the relational algebra expression computed in Algorithm 2. Not all systems support all types of queries. MonetDB does not support recursive SQL queries. Postgres supports only certain types of recursive queries[15]. The same applies to NoDB. Virtuoso currently does not support intersections. RDFSlice aims at the slightly different problem of RDF-Slicing. It supports only a specific type of join. Also, it does not support recursion.

We ran every competitor on all queries that it supports, and averaged the runtime over 3 runs. Since most systems finished in a matter of seconds, we aborted systems that took longer than 10 minutes. The databases were run with and without indexing. NoDB comes with its own adaptive indexing mechanism that cannot be switched off.

LUBM10. Table 1 shows the runtimes of all queries for the different systems on LUBM with 10 universities. The runtimes include the loading and indexing times. For systems where we could determine these times explicitly, we noted them in the last row of the table. Since most systems finished in a matter of seconds, we aborted systems that took longer than 10 minutes. Among the 4 triple stores (Jena+TDB, Jena+HDT, Stardog, and Virtuoso), only Stardog can finish on all queries in less than 10 minutes. Jena is mostly too slow, no matter with which back-end. Virtuoso performs slightly faster than Stardog, but it cannot deal with all queries. RDFSlice can answer only 2 queries, and runs a bit faster than Stardog. The 5 database competitors (Postgres, NoDB, and MonetDB – with and without indexes) are generally faster. Among these, MonetDB is much faster than Postgres and NoDB. Postgres and MonetDB are fastest without indexes, which is to be expected when running the query only once.

Among the best performing systems are two Datalog systems (Bash Datalog, and RDFox). Not only can they answer all queries, but they are generally also faster than the other systems. Among the three, DLV is the slowest. RDFox shines with a very short and nearly constant

[11] `https://www.postgresql.org/`, v. 10.1

[12] `https://github.com/HBPMedical/PostgresRAW/tree/6ae475`, v. 9.6.5

[13] `https://www.monetdb.org/`, v. Jul2017-SP3

[14] `http://aksw.org/Projects/RDFSlice.html`, `https://bitbucket.org/emarx/rdfslice/src`, v. 2016-12-01

[15] For example, Datalog programs of the following shape cannot be translated into an SQL query supported by Postgres:

```
level(X,Y) :- level(X,Z), level(Y,Z).
level(Y,X) :- level(X,Z), level(Y,Z).
```

time for all queries. We suspect that this time is given by the loading time of the data, and that it dominates the answer computation time. Nevertheless, Bash Datalog is faster than RDFox on nearly all queries on LUBM 10.

LUBM100 to LUBM1000. Based on the previous experiment, we chose the fastest systems in each group as competitors: RDFox for the Datalog systems, Stardog and Virtuoso for the triple stores, and MonetDB with and without indexes for the databases. We then increased the number of universities in our LUBM dataset from 10 to 100, 200, 500, and 1000. Table 2 and 3 shows the sizes of the datasets and the runtimes of the systems. Across all datasets, our system performs best on more than half of the queries. The only system that can achieve a similar performance is RDFox. As before, RDFox always needs just a constant time to answer a query, because it loads the dataset into main memory. This makes the system very fast. However, this technique does not work if the dataset is too large, as we shall see next.

6.2 Reachability

Datasets. Our next datasets are graph datasets. We used the LiveJournal and com-orkut graphs from [25], and the friendster graph [23]. These datasets represent the graph structure of online social networks. They allow us to test the performance of our algorithm on real world data. Table 4 shows the number of nodes and edges of these datasets.

As our competitors, we chose again RDFox, Stardog, and Virtuoso. We could not use MonetDB, because the reachability query is recursive. As an additional competitor, we chose BigDatalog [37]. BigDatalog is a distributed Datalog implementation running on Apache Spark. BigDatalog was already run on the same LiveJournal and com-orkut graphs in the original paper [37].

Query. For all of these datasets, we used a single query: We asked for the set of nodes that can be reached from a given node *id*. We used the following Datalog program to this end, adapted from [37]:

```
reach(Y) :- arc(id, Y).
reach(Y) :- reach(X), arc(X, Y).
```

In order to avoid that RDFox materializes the entire transitive closure, we modified the program as follows for RDFox:

```
reach(id, Y) :- arc(id, Y).
reach(id, Y) :- reach(id, X), arc(X, Y).
```

For the experiment, we chose 3 random nodes (and thus generated 3 queries) for LiveJournal and com-orkut. We chose one random node for Friendster.

Results. Table 5 shows the runtime for each system (averaged over the 3 queries for LiveJournal and com-orkut). Virtuoso was the slowest system, and we aborted it after 25 min and 50 min,

Query	LUBM 100 (1.5 GB)									LUBM 200 (3.1 GB)								
	Bash	DLV	Souffle	RDFox	Stardog	Virtuoso	MonetDB (no indices)	MonetDB (indices)	RDFSlice	Bash	DLV	Souffle	RDFox	Stardog	Virtuoso	MonetDB (no indices)	MonetDB (indices)	RDFSlice
1	**8**	111	21	25	59	119	15	17	100	**9**	227	35	50	110	302	31	36	196
2	**11**	108	>1000	25	61	120				**21**	222	>1000	51	112	302			
3	**7**	108	21	25	59	119	16	18		**14**	223	36	50	110	302	32	37	
4	**17**	109	29	24	60	119	21	23		**37**	224	48	50	110	303	42	47	
5	**12**	108	26	25	59		19	21		**24**	224	45	50	110		59	64	
6	**17**	110	28	26	93		30	32		**41**	944	47	53	173		76	82	
7	**24**	110	>1000	25	61		28	31		**44**	227	>1000	51	112		74	80	
8	**22**	109	93	25	61					**45**	225	305	50	113				
9	28	111	>1000	**26**	62		34	36		61	227	>1000	**52**	113		138	138	
10	**18**	108	27	25	60		27	30		**44**	224	46	50	111		72	79	
11	**6**	108	20	24	60	119				**11**	222	34	50	110	302			
12	**10**	108	27	24	59					**21**	222	46	50	110				
13	**12**	107	26	25	59		19	21		**24**	222	44	50	110		58	64	
14	**5**	108	19	26	63	122	15	18	33	**11**	601	33	52	118	309	31	36	65
of which load:					57	118	14	16						108	301	28	33	

Table 2: Runtime for the LUBM queries, in seconds.

	LUBM 500 (7.8 GB)								LUBM 1000 (16 GB)						
Query	Bash	Souffle	RDFox	Stardog	Virtuoso	MonetDB (no indices)	MonetDB (indices)	RDFSlice	Bash	Souffle	RDFox	Stardog	MonetDB (no indices)	MonetDB (indices)	RDFSlice
1	**27**	82	131	582	1577	83	97	229	**75**	168	273	1955	185	210	1042
2	**53**	>3600	132	683	1580				**118**	>7200	278	2030			
3	**35**	95	131	609	1578	88	101		**89**	483	276	1955	186	217	
4	**95**	226	129	583	1579	118	131		307	>7200	**273**	1962	522	471	
5	**62**	283	131	498		290	364		**168**	>7200	278	1956	894	793	
6	**93**	113	137	1011		866	797		354	>7200	**287**	2361	2066	1934	
7	**122**	>3600	134	673		898	753		544	>7200	**279**	2005	1809	2016	
8	151	1818	**132**	768					447	>7200	**274**	1967			
9	250	>3600	**136**	749		2669	3064		712	>7200	**283**	2018	3275	3090	
10	**95**	337	132	678		587	491		334	>7200	**277**	1959	1845	1834	
11	**28**	156	130	498	1576				**64**	567	273	1957			
12	**56**	110	130	682					**164**	>7200	273	1959			
13	**63**	106	132	669		312	287		**174**	>7200	277	1969	908	955	
14	**28**	81	136	787	1595	85	99	74	**63**	167	284	2069	181	217	334
of which load:				489	1575	72	92					1946	160	194	

Table 3: Runtime for the LUBM queries, in seconds.

dataset	Nodes	Edges
LiveJournal [25]	4.8 M	69 M
orkut [25]	3.1 M	117 M
friendster [23]	68 M	2 586 M

Table 4: Statistics for the reachability datasets.

dataset	Bash	RDFox	BigDatalog	Stardog	Virtuoso
LiveJournal	117	**70**	532	941	>1500
orkut	225	**121**	1838	1123	>3000
friendster	**16306**	OOM	OOS	>36000	

Table 5: Runtime for the reachability query, in seconds (OOM=Out of memory; OOS=Out of space).

respectively. We did not run it on the Friendster dataset, because Friendster is 20 times larger than the other two datasets. Stardog performs better. Still, we had to abort it after 10 hours on the Friendster dataset. BigDatalog performs well, but fails with an out of space error on the Friendster dataset. The fastest system is RDFox. This is because it can load the entire data into memory. This approach, however, fails with the Friendster dataset. It does not fit into memory, and RDFox is unable to run. Bash Datalog runs 50% slower than RDFox. In return, it is the only system that can finish in reasonable time on the Friendster dataset (4:30h). We believe that this is because Bash Datalog can rely on the highly optimized implementations of the Bash commands, which can deal with large files even if they cannot be loaded into memory.

6.3 YAGO and Wikidata

Datasets. Our final series of experiments tests our system on knowledge bases. For this purpose, we used YAGO 3.1 [38], a knowledge base extracted from Wikipedia, and Wikidata [41]. The YAGO data comes in 3 different files, one with the 12 M facts (814 MB), one with the taxonomy with 1.8 M facts (154 MB), and a last one with the 24 M type relations (1.6 GB in size). The Wikidata simple dataset contains 2.1 billion triples and has an uncompressed size of 267 GB.

Queries. We designed 4 queries that are typical for such datasets (Table 6). Table 7 shows the TBox that we used. These queries and the TBox are slightly adapted to work with the different schema of the two knowledge bases. Query 1 asks for all subclasses of the class *Person>*. Query 2 asks for the parents of Louis XIV, and Query 3 asks recursively for the ancestors of Louis XIV. Query 4 asks for all people born in a place in Andorra. These queries are not difficult. The difficulty comes from the fact that the data is so large.

Results. Table 6 shows the results of RDFox and our system on both datasets. On YAGO, RDFox is much slower than our system, because it needs to instantiate all rules in order to answer queries. On Wikidata, the data does not fit into main memory, and hence RDFox

```
query1(X) :- subClassOf(X, "Person") .
query1(X) :- subClassOf(X, Y), query1(Y) .
query2(X) :- hasParent(X, "Louis XIV") .
query3(X) :- hasAncestor(X, "Louis XIV") .
query4(X) :- hasBirthPlace(X, Y),
             isLocatedIn(Y, "Andorra") .
```

Figure 6: Knowledge Base queries

```
hasParent(X,Y) :- hasChild(Y,X) .
hasAncestor(X,Y) :- hasParent(X,Y) .
hasAncestor(X,Z) :- hasAncestor(X,Y),
                    hasParent(Y,Z) .
isLocatedIn(X,Y) :- containsLocation(Y,X) .
containsLocation(X,Y) :- isLocatedIn(Y,X) .
isLocatedIn(X,Y) :- isLocatedIn(X,Y),
                    isLocatedIn(Y,Z) .
```

Figure 7: Knowledge Base rules

	YAGO		Wikidata	
query	Bash	RDFox	Bash	RDFox
1	**8**	483	**2259**	OOM
2	**5**	483	**2254**	OOM
3	**293**	483	**10171**	OOM
4	**5**	481	**2270**	OOM

Table 6: Runtime for the Wikidata/YAGO benchmark in seconds. (OOM = out of memory error)

cannot run at all. Our system, in contrast, scales effortlessly to the larger sizes of the data.

One may think that a database system, such as Postgres, may be better adapted for such large datasets. This is, however, not the case. Postgres took 104 seconds to load the YAGO dataset, and 190 seconds to build the indexes. In this time, our system has already answered nearly all the queries.

Discussion. All of our experiments evaluate only the setting that we consider in this paper, namely the setting where the user wants to execute a single query in order to preprocess the data. Our experiments show that Bash Datalog can preprocess tabular data without the need to install any particular software.

Our approach has some limitations. For example, we could not implement a disk-based

hash-join efficiently in Bash commands. Another limitation is the heuristic join reordering. It sometimes introduces large intermediate results, resulting in a less efficient query execution.

Overall, however, our approach is competitive in both speed and scalability to the state of the art. We attribute this to the highly optimized POSIX commands, and to our optimizations described in Section 5.6. Furthermore, the startup cost of our system is quite low, as it consists mainly of translating the query to a Bash script.

7 Web Interface

Our system can be used online at `https://www.thomasrebele.org/projects/bashlog`. Figure 8 shows a screenshot. Our interface provides three modes: a Datalog mode, a SPARQL mode, and an API.

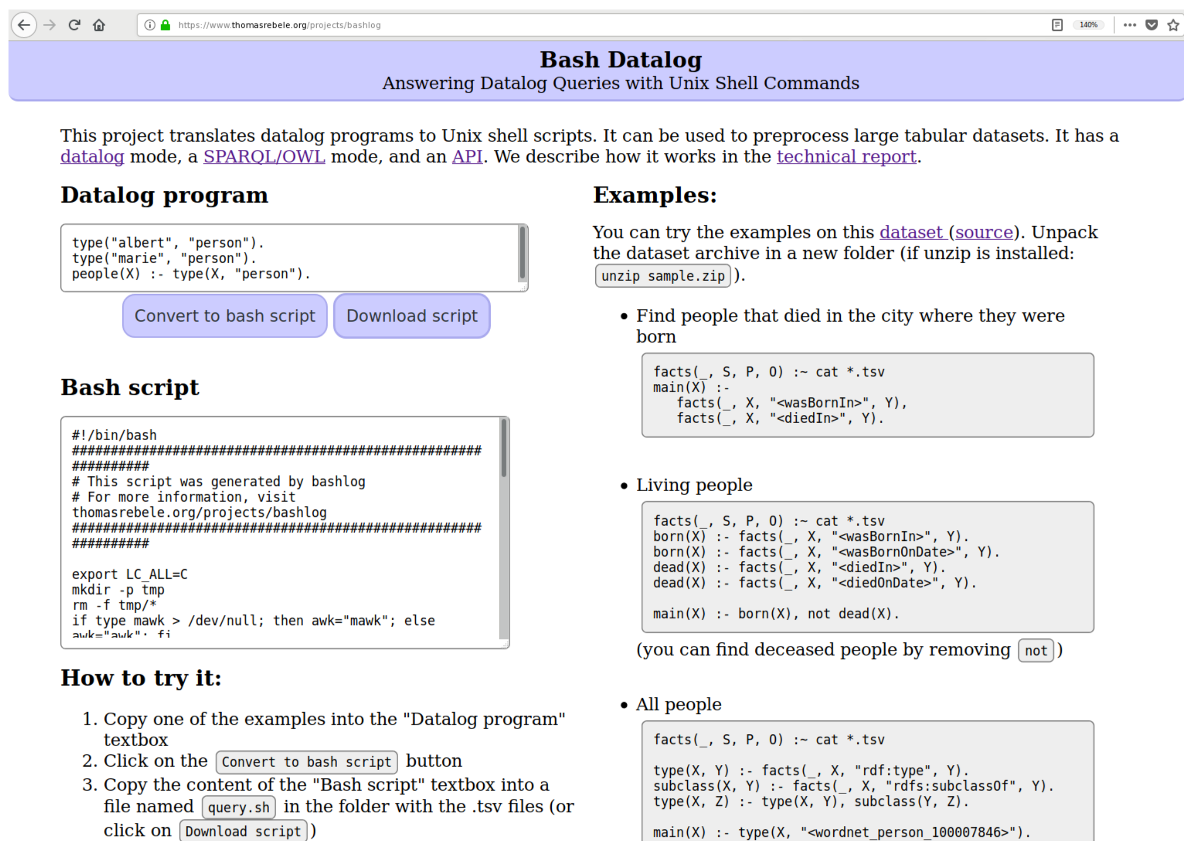

Figure 8: Screenshot of the web interface

Datalog mode. The user can enter her Datalog program in a text box. After she clicks on "Convert to Bash script", the Datalog program is transmitted to a server, which translates it to a Bash script. The Bash script then appears in the second text box. The user can copy and paste the script into a terminal and execute it. To help the user get started, we provide an example dataset based on YAGO, together with example queries.

Example (Datalog mode): Let us, e.g., walk through the query shown in Figure 9. It extracts all people that have an ancestor born in Italy from the knowledge base. Line 1 specifies the only command rule, which is responsible for reading the data from the disk. Lines 3-5 are shorthand predicates for the relations that we want to use in the query. Lines 7-9 define the TBox, which states that `hasAncestor` can be computed as a transitive closure. Lines 11-13 are the actual query. Once the user clicks on "Generate", our interface translates the query into the Bash script shown in Appendix A.1 on page 35.

```
1  fact(Id, S, P, O) :~ cat *.tsv
2
3  hasChild(X, Y) :- fact(_, X, "<hasChild>", Y).
4  wasBornIn(X,Y) :- fact(_, X, "<wasBornIn>", Y).
5  isLocatedIn(X,Y) :- fact(_, X, "<isLocatedIn>", Y).
6
7  hasParent(X,Y) :- hasChild(Y,X).
8  hasAncestor(X,Y) :- hasParent(X,Y).
9  hasAncestor(X,Z) :- hasAncestor(X,Y), hasParent(Y,Z).
10
11 main(X,Y) :- hasAncestor(X,Y),
12              wasBornIn(Y,Z),
13              isLocatedIn(Z, "<Italy>").
```

Figure 9: Example of a Datalog query that can be used with the Web interface. It finds all people in the YAGO knowledge base that have an ancestor that was born in Italy.

SPARQL/OWL mode. This mode allows preprocessing knowledge bases using Semantic Web standards. It takes a SPARQL query, and a TBox in the OWL 2 RL format. For the input of the ABox, we currently support RDF data in the form of N-Triples.

```
1  @prefix owl: <http://www.w3.org/2002/07/owl#> .
2  @prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
3  @prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
4  @prefix kb: <http://yago-knowledge.org/resource/> .
5
6  kb:hasParent
7    owl:inverseOf kb:hasChild;
8    rdfs:subPropertyOf kb:hasAncestor.
9
10 kb:hasAncestor
11   rdf:type owl:TransitiveProperty.
```

Figure 10: Example of an OWL TBox

```
PREFIX kb: <http://yago-knowledge.org/resource/>
SELECT ?X ?Y WHERE {
        ?X kb:hasAncestor ?Y .
        ?Y kb:wasBornIn ?Z .
        ?Z kb:isLocatedIn kb:Italy .
}
```

Figure 11: Example of a SPARQL query

> *Example (SPARQL/OWL mode):* Let us consider the same example as in Figure 9. Let us assume this time that the user formulates the query not in Datalog, but in SPARQL. Figure 11 shows the query. To add the semantics to the hasAncestor predicate, the user specifies the transitivity of the predicate in OWL (shown in Figure 10). When our system receives this input, it translates it to a Datalog program, and proceeds as before. The generated Bash script is shown in Appendix A.2 on page 36.

API. We also provide a way to use the Web interface from a command line interface, without opening a browser. Using the API requires a command which supports sending HTTP POST requests, such as `curl`. Unfortunately the POSIX standard does not include a command for HTTP requests. The command `curl` is widely spread and provides the right functionality for our purpose. Here we show the `curl` command for a SPARQL/OWL query:

```
curl --data-urlencode owl@ontology.owl \
     --data-urlencode sparql@query.sparql \
     --data-urlencode nTriples=kb.ntriples \
     https://www.thomasrebele.org/projects/bashlog/api/sparql
```

The command expects the TBox and the query as files `ontology.owl`, and `query.sparql`, respectively. The file `kb.ntriples` is the path to the knowledge base in N-Triples format. The "=" specifies that only the path, but not the file content, will be sent to the server. The command sends the content of the files `ontology.owl` and `query.sparql`, and the string `kb.ntriples` to the server. The server executes our algorithm, and sends the Bash script in the response of the HTTP request. The result of the command is a Bash script that can be saved in a file and executed locally.

8 Conclusion

In this paper, we have presented a method to compile Datalog programs into Unix Bash scripts. This allows executing Datalog queries on tabular datasets without installing any software. We show that our method is competitive in terms of speed with state-of-the-art systems. In particular, our method takes often less time to answer a query than a database system needs to load the data. Furthermore, our system can process datasets even if they do not fit in memory. This means that our approach is a good choice for preprocessing large tabular datasets.

Our system can be used online at `https://www.thomasrebele.org/projects/bashlog`.

The source code can be obtained at `https://github.com/thomasrebele/bashlog`. For future work, we aim to explore extensions of this work such as adding support of numerical comparisons to the Datalog language.

References

[1] S. Abiteboul and R. Hull. Data functions, datalog and negation (extended abstract). In *SIGMOD*, 1988.

[2] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[3] R. Agrawal. Alpha: An extension of relational algebra to express a class of recursive queries. *IEEE Transactions on Software Engineering*, 14(7), 1988.

[4] A. V. Aho and J. D. Ullman. The universality of data retrieval languages. In *ACM Symposium on Principles of Programming Languages*, 1979.

[5] I. Alagiannis, R. Borovica, M. Branco, S. Idreos, and A. Ailamaki. Nodb: efficient query execution on raw data files. In *SIGMOD*, 2012.

[6] P. A. Boncz, M. L. Kersten, and S. Manegold. Breaking the memory wall in monetdb. *Communications of the ACM*, 51(12), 2008.

[7] Y. Bu, V. R. Borkar, M. J. Carey, J. Rosen, N. Polyzotis, T. Condie, M. Weimer, and R. Ramakrishnan. Scaling datalog for machine learning on big data. *CoRR*, abs/1203.0160, 2012.

[8] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache flink™: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38(4), 2015.

[9] J. J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, and K. Wilkinson. Jena: implementing the semantic web recommendations. In *WWW Papers & Posters*, 2004.

[10] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems*, 26(2), 2008.

[11] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6), 1970.

[12] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1), 2008.

[13] O. Erling and I. Mikhailov. RDF support in the virtuoso DBMS. In *Networked Knowledge*, 2009.

[14] J. D. Fernández, M. A. Martínez-Prieto, C. Gutiérrez, A. Polleres, and M. Arias. Binary rdf representation for publication and exchange (hdt). *Web Semantics: Science, Services and Agents on the World Wide Web*, 19, 2013.

[15] S. Ganguly, A. Silberschatz, and S. Tsur. A framework for the parallel processing of datalog queries. In *SIGMOD*, 1990.

[16] S. Ganguly, A. Silberschatz, and S. Tsur. Parallel bottom-up processing of datalog queries. *J. Log. Program.*, 14(1&2), 1992.

[17] Y. Guo, Z. Pan, and J. Heflin. LUBM: A benchmark for OWL knowledge base systems. *J. of Web Semantics*, 3(2-3), 2005.

[18] V. Haarslev and R. Möller. RACER system description. In *IJCAR*, 2001.

[19] S. Harris, A. Seaborne, and E. Prud'hommeaux. SPARQL 1.1 query language. *W3C recommendation*, 3 2013.

[20] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys*, 2007.

[21] P. Katsogridakis, S. Papagiannaki, and P. Pratikakis. Execution of recursive queries in apache spark. In *Euro-Par*, 2017.

[22] M. Kornacker, A. Behm, V. Bittorf, T. Bobrovytsky, C. Ching, A. Choi, J. Erickson, M. Grund, D. Hecht, M. Jacobs, I. Joshi, L. Kuff, D. Kumar, A. Leblang, N. Li, I. Pandis, H. Robinson, D. Rorke, S. Rus, J. Russell, D. Tsirogiannis, S. Wanderman-Milne, and M. Yoder. Impala: A modern, open-source SQL engine for hadoop. In *CIDR*, 2015.

[23] J. Kunegis. Konect: the koblenz network collection. In *WWW*, 2013.

[24] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic*, 7(3), 2006.

[25] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[26] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, 2010.

[27] E. Marx, S. Shekarpour, S. Auer, and A. N. Ngomo. Large-scale RDF dataset slicing. In *ICSC*, 2013.

[28] E. Marx, S. Shekarpour, T. Soru, A. M. P. Brasoveanu, M. Saleem, C. Baron, A. Weichselbraun, J. Lehmann, A. N. Ngomo, and S. Auer. Torpedo: Improving the state-of-the-art RDF dataset slicing. In *ICSC*, 2017.

[29] B. Motik, B. C. Grau, I. Horrocks, Z. Wu, A. Fokoue, C. Lutz, et al. OWL 2 web ontology language profiles. *W3C recommendation*, 12 2012.

[30] B. Motik, Y. Nenov, R. Piro, I. Horrocks, and D. Olteanu. Parallel materialisation of datalog programs in centralised, main-memory RDF systems. In *AAAI*, 2014.

[31] B. Parsia and E. Sirin. Pellet: An owl dl reasoner. In *Third international semantic web conference-poster*, volume 18, 2004.

[32] M. Rogala, J. Hidders, and J. Sroka. Datalogra: datalog with recursive aggregation in the spark RDD model. In *Int. Workshop on Graph Data Management Experiences and Systems*, 2016.

[33] B. Saha, H. Shah, S. Seth, G. Vijayaraghavan, A. C. Murthy, and C. Curino. Apache tez: A unifying framework for modeling and building data processing applications. In *SIGMOD*, 2015.

[34] B. Scholz, H. Jordan, P. Subotic, and T. Westmann. On fast large-scale program analysis in datalog. In *International Conference on Compiler Construction*, 2016.

[35] M. Shaw, P. Koutris, B. Howe, and D. Suciu. Optimizing large-scale semi-naïve datalog evaluation in hadoop. In *Int. Workshop on Datalog in Academia and Industry*, 2012.

[36] R. Shearer, B. Motik, and I. Horrocks. Hermit: A highly-efficient owl reasoner. In *OWLED*, volume 432, 2008.

[37] A. Shkapsky, M. Yang, M. Interlandi, H. Chiu, T. Condie, and C. Zaniolo. Big data analytics with datalog queries on spark. In *SIGMOD*, 2016.

[38] F. M. Suchanek, G. Kasneci, and G. Weikum. Yago: a core of semantic knowledge. In *WWW*, 2007.

[39] D. Tsarkov and I. Horrocks. Fact++ description logic reasoner: System description. *Automated reasoning*, 2006.

[40] R. Verborgh, M. Vander Sande, O. Hartig, J. Van Herwegen, L. De Vocht, B. De Meester, G. Haesendonck, and P. Colpaert. Triple Pattern Fragments: a low-cost knowledge graph interface for the Web. *J. of Web Semantics*, 37–38, Mar. 2016.

[41] D. Vrandecic and M. Krötzsch. Wikidata: a free collaborative knowledgebase. *Communications of the ACM*, 57(10), 2014.

[42] J. Wang, M. Balazinska, and D. Halperin. Asynchronous and fault-tolerant recursive datalog evaluation in shared-nothing engines. *PVLDB*, 8(12), 2015.

[43] Wikipedia. Bash (unix shell) — wikipedia, the free encyclopedia, 2017.

[44] O. Wolfson and A. Silberschatz. Distributed processing of logic programs. In *SIGMOD*, 1988.

[45] H. Wu, J. Liu, T. Wang, D. Ye, J. Wei, and H. Zhong. Parallel materialization of datalog programs with spark for scalable reasoning. In *WISE*, 2016.

[46] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *USENIX Workshop on Hot Topics in Cloud Computing*, 2010.

[47] J. Zhou, P. Larson, and R. Chaiken. Incorporating partitioning and parallel plans into the SCOPE optimizer. In *ICDE*, 2010.

A Generated Unix Shell Script

This appendix shows the complete code listings of two Unix shell scripts that were created with our system.

A.1 Datalog Mode

The following script was translated from the Datalog query in Figure 9 on page 30 in Section 7. Lines 1-17 are the header. The header is included in all generated scripts, and checks for available optimizations. Lines 20-24 read the relevant facts from the input file, and saves them in three temporary files. Lines 27-32 sort one of the just created files, so that it can be reused in the `while` loop. Lines 34-58 show the actual query plan, and Line 60 cleans up the temporary files.

```
1  #!/bin/bash
2  ###############################################################
3  # This script was generated by bashlog
4  # For more information, visit thomasrebele.org/projects/bashlog
5  ###############################################################
6
7  export LC_ALL=C
8  mkdir -p tmp
9  rm -f tmp/*
10 if type mawk > /dev/null; then awk="mawk"; else awk="awk"; fi
11 sort="sort "
12 check() { grep -- $1 <(sort --help) > /dev/null; }
13 check "--buffer-size" && sort="$sort --buffer-size=20% "
14 check "--parallel"    && sort="$sort --parallel=2 "
15
16 read_ntriples() { $awk -F" " '{ sub(" ", "\t"); sub(" ", "\t"); sub(/ \.$/, "");
       print $0 }' "$@"; }
17 conv_ntriples() { $awk -F$'\t' '{ print $1 " " $2 " " $3 " ." }'; }
18
19
20 touch tmp/mat0 tmp/mat1 tmp/mat2
21 $awk -v FS=$'\t' ' ($4 == "<Italy>" && $3 == "<isLocatedIn>") { print $2 >> "tmp/
       mat0" }
22  ($3 == "<hasChild>") { print $4 FS $2 >> "tmp/mat1" }
23  ($3 == "<wasBornIn>") { print $2 FS $4 >> "tmp/mat2" }
24   ' *.tsv
25
26
27 mkfifo tmp/lock_mat3; (
28    $sort -t $'\t' -k 1 tmp/mat1 > tmp/mat3;
29    mv tmp/lock_mat3 tmp/done_mat3;
30    cat tmp/done_mat3 > /dev/null & exec 3> tmp/done_mat3;
31    exec 3>&-;
```

```
32  ) &
33
34  # plan
35  $sort -t $'\t' -k 1 -k 2 -u \
36  <(join -t $'\t' -1 1 -2 1 -o 1.2,1.3 \
37     <($sort -t $'\t' -k 1 \
38        <(join -t $'\t' -1 1 -2 2 -o 1.2,2.1,2.2 \
39           <($sort -t $'\t' -k 1 tmp/mat2) \
40           <($sort -t $'\t' -k 2 \
41              <($sort -t $'\t' -k 1 -k 2 -u tmp/mat1 \
42                     | tee tmp/full4 > tmp/delta4
43                 while
44
45                 $sort -t $'\t' -k 1 -k 2 -u \
46                    <(cat tmp/lock_mat3 1>&2 2>/dev/null ;  \
47                       join -t $'\t' -1 2 -2 1 -o 1.1,2.2 \
48                       <($sort -t $'\t' -k 2 tmp/delta4) tmp/mat3) \
49                     | comm -23 - tmp/full4 > tmp/new4;
50
51                 mv tmp/new4 tmp/delta4 ;
52                 $sort -u --merge -o tmp/full4 tmp/full4 tmp/delta4;
53                 [ -s tmp/delta4 ];
54                 do continue; done
55
56                 rm tmp/delta4
57                 cat tmp/full4)))) \
58     <($sort -t $'\t' -k 1 -u tmp/mat0))
59
60   rm -f tmp/*
```

A.2 SPARQL/OWL Mode

The following script was translated from the SPARQL query and OWL TBox from Figures 11 and 10 on page 30. Lines 1-17 are the header. Lines 20-33 store the relevant facts in temporary files. Lines 35-58 show the query plan, and Line 60 cleans up temporary files.

```
 1  #!/bin/bash
 2  ###############################################################
 3  # This script was generated by bashlog
 4  # For more information, visit thomasrebele.org/projects/bashlog
 5  ###############################################################
 6
 7  export LC_ALL=C
 8  mkdir -p tmp
 9  rm -f tmp/*
10  if type mawk > /dev/null; then awk="mawk"; else awk="awk"; fi
11  sort="sort "
12  check() { grep -- $1 <(sort --help) > /dev/null; }
13  check "--buffer-size" && sort="$sort --buffer-size=20% "
14  check "--parallel"    && sort="$sort --parallel=2 "
15
```

```
16 read_ntriples() { $awk -F" " '{ sub(" ", "\t"); sub(" ", "\t"); sub(/ \.$/, "");
       print $0 }' "$@"; }
17 conv_ntriples() { $awk -F$'\t' '{ print $1 " " $2 " " $3 " ." }'; }
18
19
20 touch tmp/mat0 tmp/mat1 tmp/mat2
21 $awk -v FS=$'\t' '
22   BEGIN {
23    mat0_out0c2_cond1["<http://yago-knowledge.org/resource/hasAncestor>"] = "1";
24    mat0_out0c2_cond1["<http://yago-knowledge.org/resource/hasParent>"] = "1";
25    mat0_out2c0_cond1["<http://yago-knowledge.org/resource/hasChild>"] = "1";
26   }
27
28  (($2) in mat0_out0c2_cond1){ print $1 FS $3 >> "tmp/mat0" }
29  (($2) in mat0_out2c0_cond1){ print $3 FS $1 >> "tmp/mat0" }
30  ($3 == "<http://yago-knowledge.org/resource/Italy>" && $2 == "<http://yago-
       knowledge.org/resource/isLocatedIn>") { print $1 >> "tmp/mat1" }
31  ($2 == "<http://yago-knowledge.org/resource/wasBornIn>") { print $1 FS $3 >> "tmp
       /mat2" }
32   ' \
33 <(read_ntriples yago-sample.ntriples)
34
35 $sort -t $'\t' -k 1 -k 2 -u \
36 <(join -t $'\t' -1 1 -2 1 -o 1.2,1.3 \
37     <($sort -t $'\t' -k 1 \
38         <(join -t $'\t' -1 1 -2 2 -o 1.2,2.1,2.2 \
39             <($sort -t $'\t' -k 1 tmp/mat2) \
40             <($sort -t $'\t' -k 2 \
41                 <($sort -t $'\t' -k 1 -k 2 -u tmp/mat0 \
42                         | tee tmp/full3 > tmp/delta3
43                     while
44
45                     $sort -t $'\t' -k 1 -k 2 -u \
46                         <(join -t $'\t' -1 2 -2 1 -o 1.1,2.2 \
47                             <($sort -t $'\t' -k 2 tmp/delta3) \
48                             <($sort -t $'\t' -k 1 tmp/full3)) \
49                         | comm -23 - tmp/full3 > tmp/new3;
50
51                     mv tmp/new3 tmp/delta3 ;
52                     $sort -u --merge -o tmp/full3 tmp/full3 tmp/delta3 ;
53                     [ -s tmp/delta3 ];
54                     do continue; done
55
56                     rm tmp/delta3
57                     cat tmp/full3)))) \
58     <($sort -t $'\t' -k 1 -u tmp/mat1))
59
60  rm -f tmp/*
```