A framework for annotating CSV-like data

Marcelo Arenas, Francisco Maturana, Cristian Riveros, Domagoj Vrgoč
Pontificia Universidad Católica de Chile
marenas@ing.puc.cl, fjmaturana@uc.cl, cristian.riveros@uc.cl, dvrgoc@ing.puc.cl

ABSTRACT

In this paper, we propose a simple and expressive framework for adding metadata to CSV documents and their noisy variants. The framework is based on annotating parts of the document that can be later used to read, query, or exchange the data. The core of our framework is a language based on extended regular expressions that are used for selecting data. These expressions are then combined using a set of rules in order to annotate the data. We study the computational complexity of implementing our framework and present an efficient evaluation algorithm that runs in time proportional to its output and linear in its input. As a proof of concept, we test an implementation of our framework against a large number of real world datasets and show that it can be efficiently used in practice.

1. INTRODUCTION

Comma-separated values (CSV) are a simple format used to store and exchange data. In their essence, CSV documents are nothing more than data divided by separators that naturally structure the file content into cells, columns, lines, etc. An example of one such document is given in Figure 1. Although traditionally used to send data between organizations, these days CSV files also form a large percentage of data published on the Web under the "open data" initiative whose aim is to make the government data widely available. For example, the Office for National Statistics UK (ONS), makes its data available in CSV format [16], as do the Land Registry UK [12] and World Bank [29].

```
John   , 47 , 15/05/1968 , 3700
Arthur , 31 , 31/07/1983 , 3005
Eliot  , 30 , 21/08/1985 , 3600
```

Figure 1: An example of a CSV file.

CSV files are very intuitive and thus easily understood when presented to a human user. However, because of this simplicity CSV data lacks expressive power and at this point there is no standardised way for describing the content of a CSV file, like specifying the structure of the data, the data types used in different columns, or the way it should be displayed. In order to address this issue, the World Wide Web Consortium (W3C), has formed a CSV on the Web working group [27], whose main goal is to develop sound infrastructure for CSV data on the Web. Building this infrastructure is based on allowing CSV data to be enriched by versatile and easily usable metadata. The basic idea of metadata is to allow users to annotate their data with comments, additional meanings, or explanations that can be managed by users or programs processing the data. For example, specifying the meaning of some code that is often repeated in a file is an instance of metadata, as is providing the default value for an error code or a missing piece of information.

When trying to develop a metadata language for CSV one option is to concentrate on well-structured documents that have the same number of columns in each row, no unexpected line breaks, and always use the same delimiter (comma in most cases). In fact, the current recommendation by the W3C [26] treats CSV files as tables, thus allowing for a concise metadata syntax. However, CSV data can often be noisy, for instance when it is generated as a result of publishing a legacy database in CSV format, which can result in a skewed table with missing values, rows of different length, or non-standard data types. Similar issues occur when using CSV to store data coming from different sources, or when merging two CSV tables. In this case, what is the essence kept in every CSV file? Notice that the main feature of CSV data, which is preserved even in noisy files, is the fact that they use predefined landmarks to separate different data entries: rows are separated using the new line symbol, cells within each row are separated by commas, etc. Moreover, this idea is not exploited only by CSV files but by any sort of format that has predefined landmarks separating the stored data. This includes system and query logs [24], citation data [19], geometric vector files [17], chess game data [9] and many other formats. The main objective of this paper is to develop a metadata language for annotating this more general sort of CSV documents, that we call *CSV-like* documents, which do not necessarily conform to the tabular format prescribed by the W3C [26], but use the mechanism of landmarks to lazily structure the data.

In order to design such a language we must overcome several issues. First, we want the language to be applicable in practical use case scenarios, such as the ones proposed in [28]. Second, notice that our language is meant to provide metadata by annotating different elements of CSV-like files such as cells, rows, columns, or even entire documents. The question is then how can this be done in a uniform way? And lastly, we also need to make our language efficiently implementable in order for it to be applicable in practice. To overcome these issues we deploy several ideas from text processing [3] and information extraction [10, 18] that allow us to handle different elements of potentially noisy CSV files in a uniform and

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org.
Proceedings of the VLDB Endowment, Vol. 9, No. 11
Copyright 2016 VLDB Endowment 2150-8097/16/07.

efficient way. Our proposal is centred around the notion of a *span*, first introduced in [18]. Similarly to IBM's SystemT [18], we view documents as strings. A span is then simply an interval inside this string (i.e. it is defined by the starting point and the ending point of some substring). It is now easy to see that different elements of a CSV-like document, such as cells, rows, or columns, are nothing more than a collection of spans. In order to extract spans from a file we extend the work of [3, 10] to define a class of regular expressions enriched with variables called *span regular expressions*. The idea is to use regular expressions to position ourselves inside a document and then store spans into variables. To attach metadata to documents we allow combining such expressions using a datalog-like rule language into *annotation programs* that define the annotation attached to spans selected by the program. As we show, this results in an expressive language that can cover all the requirements put forward by the CSV on the Web working group [28] that specify meta data for the file, plus all their validation requirements. Since our language was designed for adding meta data, note that we cannot cover the transformation requirements from [28].

Although the base language we propose serves as a nice theoretical umbrella for the problem of extracting and annotating data, this framework does not internalize the landmark structure of CSV-like documents. This implies that annotation programs are too complicated for working with CSV-like documents, and their high expressive power also implies hard evaluation algorithms, thus hampering the language's usefulness in practice. To overcome these limitations, we further simplify the data extraction process by proposing the so called *navigation expressions* which allow simple navigation through documents by using item separators (e.g. commas, new line symbols) as landmarks. For instance, with the expression `next(,)` we specify that we want to move from the current position to the next comma in the file. Similarly, if we want to capture this part of the document into a variable we say `x:next(,)`, thus storing the resulting span into the variable `x`. The main contribution of this paper is then the language of *navigation programs* which is suitably designed to simplify the extraction process of annotation programs by using navigation expressions instead of span regular expressions. Navigation programs offer an intuitive syntax that is easy to use over CSV-like documents and, at the same time, still allows us to model all use cases put forward by the CSV on the Web working group [28].

To determine if our proposal has the potential to be used in practice we study the computational complexity of evaluating expressions and programs over documents. In general, all of these problems are NP-complete. However, if we restrict the structure of navigation expressions and require our programs to be *tree-shaped*, we end up with an expressive subfragment that can be evaluated in polynomial time. The main technical contribution of this paper is an efficient algorithm for evaluating navigation expressions, thus allowing them to extract spans from CSV-like documents. The algorithm is based on an indexing technique that allows us to use string matching algorithms [3] in order to process our expressions in time that is both linear in the size of the input document and the size of its output (the number of captured spans). This algorithm is also a constant-delay algorithm [20], which means it takes polynomial time to preprocess the input and constant time between producing consecutive outputs.

Finally, as a proof of concept, we also show how an implementation of our framework works in practice by designing a set of experiments that is based on the requirements from the W3C CSV on the Web working group [28] and running it against a wide range of real world CSV datasets. We also test the implementation in the context of annotating query logs and for this we use the data from public query endpoints of the British Museum [22] and DBpedia [8], two large providers of Semantic Web data. Based on this evaluation we conclude that our proposal results in a lightweight framework for annotating CSV-like data that can be efficiently implemented to run on an average laptop or a desktop machine, but is powerful enough to handle files one is likely to encounter in practice.

Related work. We categorise related work as follows.

CSV metadata languages: First, we clarify the connection of our approach with the proposal by the W3C's CSV on the Web working group for annotating CSV files [27]. The main difference is that the working group's proposal assumes the data to conform to a strict format where all rows have the same number of columns, the user comments appear only in the first row, etc. One of the main objectives of the language we propose is the ability to handle noisy data that does not necessarily conform to such strict specifications. Furthermore, the metadata proposed by the working group comes from a fixed vocabulary and does not allow arbitrary user defined annotations, as our approach does. Overall, our objective was to design a language that has a high level view of adding the metadata and can also be used to annotate files more general than CSV, such as e.g. query logs, scientific data, etc.

As far as we are aware, the only other body of work dealing with metadata in CSV files is Martens et. al [14]. However, note that [14] addresses only the problem of specifying schemas for CSV and does not deal with general metadata concepts such as assigning arbitrary annotations. Next, their approach assumes that the data is always tabular and is not designed to deal with noisy CSV files. Lastly, the biggest difference between the two approaches is the fact that [14] focuses on providing a conceptual framework for specifying CSV schemas, but does not deal with implementation issues. On the other hand, the main contribution of our work is a linear time algorithm for extracting data and annotating documents.

Spanners, AQL and SystemT: Annotation query language (AQL), which forms the basis for IBM's SystemT [18], together with its theoretical counterpart named spanners [10], had the biggest influence of the work we present here. In particular, we adopt their idea of viewing a document as a single string, and of using regular expressions with variables to extract data. The main difference between the two approaches is in the intended application domain and in the expressive power. Regarding the application domain, while SystemT is designed to deal with arbitrary text documents, the languages developed in this paper are optimized to work on files that have natural landmarks separating the data. This is also seen in the way that regular expressions are used. More precisely, while SystemT and AQL use regular expressions for extraction, we view them as a way of navigating between landmarks separating the data, and then extracting the portion of the file between two landmarks. In terms of expressive power, it is straightforward to see that the presented languages are incomparable (for instance, we allow full recursive rules which are not available in AQL, while overlap consolidation operators of SystemT are not expressible in our framework). However, since there are enough fragments considered both in [10] and our work to warrant a paper on its own, we leave such comparisons for future work.

Other information extraction tools: Here we compare our approach to standard stream editing tools such as AWK [5] and grep [25], which are used for matching regular expressions, or their extensions with variables [3], to a given string. The whole framework we propose properly subsumes all of these tools (as it allows the use of datalog-like rules), however, the classes of extraction expressions we use when defining our programs can be seen as variants of regular expressions with back referencing [3]. Having seman-

tics based on the notion of a span makes our expressions slightly weaker than regular expressions with back referencing (e.g. we can not express the language of all words of the form $w_1 w_1 \cdots w_k w_k$, for some k, which is definable using backreferences by the expression $(x\&x)^*$). On the other hand, sacrificing the expressive power allows us to define the semantics of our expressions in a much simpler way than it is done for regular expressions with back referencing [3]. We also do experimental evaluation of our framework against AWK in Section 7.

Relational tools: Finally, we would like to discuss the connection of our proposal with annotation systems for relational databases. After all, CSV tables can be viewed as relational data, so why not use the existing tools? One reason is that annotation systems for relational data are often row-based [11, 6], and thus do not support all the functionalities required from CSV metadata [28]. Another, perhaps more important reason, is that CSV was designed as a lightweight format that is widely available and independent of internal data storage mechanisms, while annotation tools for relational data [15, 6] rely heavily on the relational model and its indexing capabilities. Lastly, the approach we propose can naturally handle data that is not necessarily tabular, which is often problematic for relational systems as table based representation of such data is not straightforward.

Remark. Missing proofs, programs used in the experiments, and the source code of our implementation are available online at [1].

2. ANNOTATIONS IN ACTION

To develop an annotation language for CSV-like documents, we propose a framework where documents in general are modelled as strings. In this way, the process of extracting and annotating elements from a document can be simply viewed as a sequence of string operations. In what follows, we introduce the main notions used in our framework, which are then formalised in Section 3.

As a running example, assume that we are given the following document d_0 in the RIS format [19]:

TY - JOUR
AU - Simon
AU - Apt

This document stores information about a publication, indicating that the type (*TY*) of this publication is journal (*JOUR*), and that the authors (*AU*) of this publication are *Simon* and *Apt*. It is important to notice that every row in the RIS format starts with two letters followed by two spaces, a dash and a space. As mentioned before, document d_0 is viewed as a string in our framework:

$$d_0 \quad = \quad \vdash TY␣␣\text{-}␣JOUR↵AU␣␣\text{-}␣Simon↵AU␣␣\text{-}␣Apt\dashv$$

In this case, the string d_0 is defined over an alphabet consisting of the letters *a*, ..., *z*, *A*, ..., *Z*, the symbol dash (-), and the special symbols ␣, ↵, $\vdash$ and $\dashv$, which are used to indicate a space, the end of a line, the start of a document and the end of a document.

In our framework, the process of annotating some elements of a document d is reduced to the process of annotating some continuos regions of the string representation of d, which are called *spans* [10] in our setting. More precisely, a span p of d is a pair of the form (i, j) such that $1 \leqslant i \leqslant j \leqslant |d| + 1$, where $|d|$ is the length of the string d. Thus, p represents a continuos region of the document d, whose content is the infix of d between positions i and $j - 1$. For instance, the following figure includes d_0 and the index of each position in this string:

$$\vdash T\,Y␣␣\text{-}␣J\,O\,U\,R↵A\,U␣␣\text{-}␣S\,i\,m\,o\,n↵A\,U␣␣\text{-}␣A\,p\,t\dashv$$
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34

In this case, we have that the span $(2, 7)$ has as content the infix of d_0 between positions 2 and 6, that is, the string "*TY -*", while $(4, 4)$ is assumed to have as content the empty string ε (in general, the content of a span (i, i) is assumed to be ε). Moreover, we have that the span $(1, 35)$ corresponds to the entire document, as the infix of d_0 between positions 1 and 34 is d_0 itself.

The first key ingredient of our framework is the definition of expressions for extracting spans from a document. More precisely, we introduce in this paper the notion of a *span regular expression* (SRE), an extension of classical regular expressions that are designed for defining and extracting spans rather than strings. In a nutshell, SREs use the union (i.e. $+$), the concatenation (i.e. $/$) and the Kleene star of regular expressions to check patterns inside a document. For extracting spans, SREs are enhanced with variables (e.g. x, y, z) which store spans. For instance, suppose that in our running example we want to extract and annotate "*all spans in the first column of a RIS document d*", which have the information about the tags for the publications stored in d. This means that we want all spans that: (i) start after a symbol ↵ or $\vdash$, and end before a symbol ␣; and (ii) do not contain a symbol ␣ inside. We can define all spans that satisfy condition (i) with the SRE $R_0 = (↵ + \vdash) \,/\, \Sigma^* \,/\, ␣$, where Σ is the set of all symbols in the document d_0. Intuitively, R_0 is defining all spans of the form "↵w␣" or "$\vdash w$␣", where w is a substring of zero or more symbols in Σ. Since we are interested in capturing spans between two separators, we can refine R_0 and replace Σ^* with a variable x as follows: $R_1 = (↵ + \vdash) \,/\, x \,/\, ␣$. Thus, R_1 defines all spans that satisfy condition (i), and it stores in x the span between the symbols ↵ or $\vdash$ and the symbol ␣.

To define condition (ii) in the previous example, we need to introduce another ingredient to our framework. More precisely, to evaluate a SRE over a specific span (which can be an entire document or just a fragment of it), we use the so-called *extraction expressions*. For example, define SRE R_2 as $\Delta^*/R_1/\Delta^*$, where $\Delta = \Sigma \cup \{\vdash, \dashv\}$. Then $\texttt{doc}.R_2$, where doc is a reserved word, is an extraction expression that indicates that R_2 has to be evaluated over the span corresponding to an entire document. For instance, if $\texttt{doc}.R_2$ is evaluated over d_0, then we are indicating that R_2 has to be evaluated over the span $(1, 35)$ whose content is d_0. In particular, this expression indicates that d_0 has to match R_2, that is, d_0 should be of the form "u↵w␣v" or "$u\vdash w$␣v" where u and v are substrings of zero or more symbols in Δ, and it also indicates that the span whose content is w should be stored in the variable x (recall that the variable x occurs in R_1).

To indicate that a SRE R has to be evaluated over a span corresponding to a fragment of a document, we use an extraction expression of the form $z.R$ where z is a variable, which specifies that R has to be evaluated over the span stored in z. For example, assume that the extraction expression $\texttt{doc}.R_2$ has been evaluated over a RIS document d. Then to check that the span stored in the variable x satisfies some condition, we can use an extraction expression of the form $x.R$. More specifically, given that each span stored in x satisfies condition (i) in our running example, we can use another extraction expression to filter x with the condition (ii). In particular, we use the extraction expression $x.F$, with $F = (\Sigma - \{␣\})^*$, to say that x contains zero or more symbols excluding ␣.

The second key ingredient of our framework is a language for annotating spans, which uses rules to combine the results of several extraction expressions and annotations. For instance, in our running example we can use the following rule to extract from a RIS document d every span that corresponds to a row in this document:

$$\texttt{doc}.\Delta^*/(↵ + \vdash) \,/\, x \,/\, (↵ + \dashv)/\,\Delta^* \wedge x.G \;\rightarrow\; \text{Row}(x), \quad (*)$$

where $G = (\Sigma - \{\hookleftarrow\})^*$. That is, a row in d is a string w that appears right after a symbol $\hookleftarrow$ or $\vdash$, does not mention any symbol $\hookleftarrow$ and appears right before a symbol $\hookleftarrow$ or $\dashv$. Besides, we can reuse Row in other rules to define more specific annotations. For example, we know that if a row starts with the keyword *AU*, then it contains the name of an author of a publication. Thus, the following rule is used to extract the names of the authors in a RIS document:

$$\text{Row}(x) \wedge x.AU\text{␣␣-␣}/y \quad \rightarrow \quad \text{Author}(y).$$

The last key ingredient of our framework is a "landmark-oriented" language for extracting spans, called the *navigation language*. Although SREs are a nice theoretical umbrella for extracting spans, they are designed for general span extraction over any kind of document, thus not exploiting the natural structure of CSV-like documents. Namely, as CSV-like documents use separators to organize the data into fragments and subfragments, one would naturally use these separators as anchors to navigate the document and extract the desired data. To this purpose, we base the navigation language on two axis $\texttt{any}(S)$ and $\texttt{next}(S)$, which naturally suggest to navigate the document until *any* separator or to the *next* separator in the set S, respectively. For instance, in order to extract every span that corresponds to a row in a document, we can use the following navigation rule:

$$\texttt{doc.any}(\hookleftarrow + \vdash)/x : \texttt{next}(\hookleftarrow + \dashv) \rightarrow \text{Row}(x) \qquad (**)$$

When this rule is evaluated over a RIS document d, it says that one has to move until "any" place in d that has a symbol $\hookleftarrow$ or $\vdash$, and then move to the "next" symbol $\hookleftarrow$ or $\dashv$, while capturing the span into the variable x. Notice that this rule extracts and annotates the same spans as the rule (*), but it is much easier to define and understand.

As a final example, we consider again the task of extracting the names of the authors in a RIS document, which was carried out before by combining two rules. In what follows we provide a simple navigation rule that takes advantage of the separators used in RIS documents to carry out this task:

$$\texttt{doc.any}(AU\text{␣␣-␣})/x : \texttt{next}(\hookleftarrow + \dashv) \quad \rightarrow \quad \text{Author}(x).$$

In this case, the rule indicates that we have to move until "any" place in a RIS document that has the separator AU␣␣-␣, and then move to the "next" symbol $\hookleftarrow$ or $\dashv$, while capturing the span into the variable x.

3. ANNOTATING CSV-LIKE DATA

In this section, we formalise the different notions of extraction expressions and annotation programs that were discussed in the previous section, and which are studied in this paper.

From now on, assume that Σ is a finite alphabet that is used in documents and does not contain the reserved symbols $\vdash$ and $\dashv$. Then a document d is just a string in the language $\vdash \Sigma^* \dashv$, that is, d is a string of the form $\vdash w \dashv$ with $w \in \Sigma^*$. Notice that we can treat a document just as a string, so, for example, we define $|d|$ as the length of the string d. However, we will continue using the term document to emphasize the fact it is a string with starting symbol $\vdash$ and ending symbol $\dashv$.

As discussed in the previous section, a fundamental notion associated to a document d is the concept of a span, which corresponds to a continuos region in d. Formally, we define $\text{span}(d)$ as the set $\{(i,j) \mid i,j \in \{1,\ldots,|d|+1\} \text{ and } i \leqslant j\}$, where each element $(i,j) \in \text{span}(d)$ is called a span of d. Every span $p = (i,j)$ of d has an associated content, which is denoted by $d(p)$ or $d(i,j)$, and it is defined as the infix of d from position i to position $j-1$ (notice that if $i = j$, then $d(p) = d(i,j) = \varepsilon$).

3.1 Extraction expressions

As pointed out in Section 2, an extraction expression defines a way to retrieve spans from a document. The most general form of such statements considered in this article will be defined by using the notion of span regular expression. To define such notion, we first need to assume that $\mathcal{V}$ is a set of variables that is disjoint with Σ. Then a *span regular expression* (SRE) is defined by the following grammar:

$$R \;::=\; w,\ w \in (\Sigma \cup \{\vdash,\dashv\})^* \;\mid\; x,\ x \in \mathcal{V} \;\mid\; (R/R) \;\mid\; (R+R) \;\mid\; (R)^* \;\mid\; \langle R \rangle$$

We have already shown in Section 2 some examples of SREs, and how they are used to define a set of spans from a document. Besides, we have also briefly described in Section 2 how variables are used in SREs to store spans. To define the semantics of SREs, we consider variables in SREs as free variables and, thus, we need to consider a variable assignment when evaluating a SRE over a document d. More precisely, a variable assignment σ over d is a function from $\mathcal{V}$ to $\text{span}(d)$, that is, a function that assigns a span to every variable in $\mathcal{V}$. Then the semantics of SRE R, with respect to a document d and a variable assignment σ over d, denoted by $[\![R]\!]_{d,\sigma}$, is defined recursively as follows:

$$\begin{aligned}
[\![w]\!]_{d,\sigma} &= \{p \mid p \in \text{span}(d) \text{ and } d(p) = w\} \\
[\![x]\!]_{d,\sigma} &= \{\sigma(x)\} \\
[\![R_1/R_2]\!]_{d,\sigma} &= \{(i,j) \mid (i,j) \in \text{span}(d) \text{ and } \exists k : \\
&\qquad (i,k) \in [\![R_1]\!]_{d,\sigma} \text{ and } (k,j) \in [\![R_2]\!]_{d,\sigma}\} \\
[\![R_1 + R_2]\!]_{d,\sigma} &= [\![R_1]\!]_{d,\sigma} \cup [\![R_2]\!]_{d,\sigma} \\
[\![R^*]\!]_{d,\sigma} &= [\![\varepsilon]\!]_{d,\sigma} \cup [\![R]\!]_{d,\sigma} \cup [\![R^2]\!]_{d,\sigma} \cup [\![R^3]\!]_{d,\sigma} \cup \cdots \\
[\![\langle R \rangle]\!]_{d,\sigma} &= \{p \in \text{span}(d) \mid \exists p' \in [\![R]\!]_{d,\sigma} : d(p) = d(p')\},
\end{aligned}$$

where R^2 is a shorthand for R/R, similarly R^3 for $R/R/R$, etc.

EXAMPLE 3.1. Consider the following document d storing information about temperatures:

$$\vdash \textit{15.2}\,,\text{␣}\textit{20.3}\hookleftarrow \textit{14.3}\,,\text{␣}\textit{14.3}\hookleftarrow \textit{14.2}\,,\text{␣}\textit{18.9} \dashv$$
$$1\ \ 2\ \ 3\ \ 4\ \ 5\ \ 6\ \ 7\ \ 8\ \ 9\ \ 10\ 11\ \ 12\ \ 13\ 14\ 15\ 16\ 17\ 18\ 19\ 20\ 21\ 22\ \ 23\ \ 24\ 25\ 26\ 27\ 28\ 29\ 30\ 31\ 32\ 33\ \ 34$$

Notice that below the string d we have included the index of each position in it. Now consider $R_3 = (\textit{15.2} + \textit{14.3})$ that uses the union operator $+$ to define the set of spans whose content is either *15.2* or *14.3*. Then given an arbitrary variable assignment σ, we have that $[\![R_3]\!]_{d,\sigma} = \{(2,6),(13,17),(19,23)\}$ since $d(2,6) = \textit{15.2}$, $d(13,17) = \textit{14.3}$, $d(19,23) = \textit{14.3}$, and for every other $p \in \text{span}(d)$, it holds that $d(p) \neq \textit{15.2}$ and $d(p) \neq \textit{14.3}$. Notice that R_3 does not include any variable, so σ does not play any role when evaluating R_3 over d.

Several variables can be included in a SRE, such as in the case of $R_4 = \hookleftarrow/x/,/\text{␣}/y/\hookleftarrow$. To evaluate R_4 over d, we need to consider the values assigned to both x and y. For instance, assume that σ_2 is a variable assignment such that $\sigma_2(x) = (13,17)$ and $\sigma_2(y) = (19,23)$. It is then easy to check that $(12,24) \in [\![R_4]\!]_{d,\sigma_2}$.

If we replace R_4 by a SRE $R_5 = \hookleftarrow/x/,/\text{␣}/x/\hookleftarrow$, then it may look like we are defining the set of spans $p \in \text{span}(d)$ such that the content associated to p corresponds to a row having the same temperature value in both columns. But in this case we have that $[\![R_5]\!]_{d,\sigma}$ is empty for every variable assignment σ. The reason for this is that two spans (i,j) and (k,ℓ) are considered to be equal if and only if $i = k$ and $j = \ell$. To overcome this limitation, span regular expressions contain the operator $\langle\cdot\rangle$, which allows to compare the contents associated to two spans. Thus, if we consider the

SRE $R_6 = \text{↵}/x/, /\text{␣}/\langle x \rangle/\text{↵}$ instead of R_5, then we will be effectively defining the set of spans corresponding to rows with the same temperature value in both columns. In fact, if σ_3 is a variable assignment such that $\sigma_3(x) = (13, 17)$, then we have that $(12, 24) \in [\![R_6]\!]_{d,\sigma_3}$ as $(13, 17) \in [\![x]\!]_{d,\sigma_3}$, and $(19, 23) \in [\![\langle x \rangle]\!]_{d,\sigma_3}$ (given that $d(19, 23) = d(\sigma_3(x)) = d(13, 17) = \textit{14.3}$). □

Notice that the semantics of SREs is defined with respect to a valuation σ. However, we want to use SREs in order to extract spans from a document and store them into some variables, thus defining valuations. This is achieved by using the notion of extraction expressions. More precisely, an *extraction expression* is an expression of the form either $\texttt{doc}.R$ or $x.R$, where R is a SRE, doc is a reserved word and x is a variable. Moreover, the evaluation of these expressions over a document d is defined as follows:

$$[\![\texttt{doc}.R]\!]_d = \{\sigma \mid \sigma \text{ is a variable assignment over } d \text{ such that } (1, |d| + 1) \in [\![R]\!]_{d,\sigma}\}$$

$$[\![x.R]\!]_d = \{\sigma \mid \sigma \text{ is a variable assignment over } d \text{ such that } \sigma(x) \in [\![R]\!]_{d,\sigma}\}$$

Intuitively, an expression of the form $\texttt{doc}.R$ is used to extract valuations from a span $(1, |d| + 1)$ corresponding to an entire document d, therefore the use of the keyword doc, while an expression of the form $x.R$ is used to extract valuations from the spans assigned to the variable x. Note that an expression of the form $x.\ \vdash R \dashv$ is equivalent to $\texttt{doc}.R$, however, as the latter is used more often we keep it in the language.

EXAMPLE 3.2. Let d be the document in Example 3.1, and R be SRE $\Delta^*/\text{↵}/x/, /\text{␣}/\langle x \rangle/\text{↵}/\Delta^*$, where $\Delta = \Sigma \cup \{\vdash, \dashv\}$. This SRE can be used to extract spans from d whose content is a temperature that is repeated in a row. In fact, we have that $[\![\texttt{doc}.R]\!]_d$ is the set of all variable assignments σ such that $\sigma(x) = (13, 17)$. □

3.2 General annotation programs

The goal of this section is to introduce a general form of programs for annotating documents, which are based on the classic idea of combining extraction expressions using a rule-based language [21]. To this end, assume from now on that Γ is a finite alphabet of annotations that is disjoint with Σ and $\mathcal{V}$. Then an annotation program consists of a set of rules that indicate how to assign annotations from Γ to the spans in a document. These rules are constructed by combining extraction expressions with formulas of the form $A(x)$, where $A \in \Gamma$ and $x \in \mathcal{V}$. More precisely, a *rule* is an expression of the form:

$$\varphi_1 \wedge \cdots \wedge \varphi_m \wedge A_1(x_1) \wedge \cdots \wedge A_n(x_n) \rightarrow A(x), \qquad (\dagger)$$

where (a) $m \geqslant 0$ and $n \geqslant 0$; (b) φ_i is an extraction expression; (c) $A_j \in \Gamma$ and $x_j \in \mathcal{V}$ where x_j is not necessarily mentioned in $\varphi_1, \ldots, \varphi_m$; and (d) $A \in \Gamma$, $x \in \mathcal{V}$ and x occurs in the body of the rule. Moreover, an *annotation program* Π is a finite set of rules.

A ground annotation over a document d is an expression of the form $A(p)$, where $A \in \Gamma$ and $p \in \text{span}(d)$. Assume that Λ is a set of ground annotations over d. Then Λ is said to satisfy the rule (†) if for every variable assignment σ over d such that $\sigma \in [\![\varphi_i]\!]_d$ for every $i \in \{1, \ldots, m\}$, it holds that:

$$\text{if } A_i(\sigma(x_i)) \in \Lambda \text{ for all } i \in \{1, \ldots, n\}, \text{ then also } A(\sigma(x)) \in \Lambda.$$

Moreover, Λ is said to satisfy an annotation program Π if Λ satisfies each rule in Π.

To define the evaluation of an annotation program over a document, we need to consider all sets of ground annotations over the document that satisfy the program. More specifically, given a document d, an annotation program Π over d and a ground annotation $A(p)$ over d, we will consider $A(p)$ as a result of evaluating Π over d if we are certain about this annotation in the sense that $A(p)$ occurs in every set of ground annotations over d that satisfies Π. Formally, the evaluation of Π over d, denoted by $\text{ANN}(\Pi, d)$, is defined as follows:

$$\{A(p) \mid \text{for every set } \Lambda \text{ of ground annotations over } d \text{ such that } \Lambda \text{ satisfies } \Pi, \text{ it holds that } A(p) \in \Lambda\}$$

It is important to notice that this semantics is based on the usual semantics for (recursive) Datalog programs [2, 7]. In fact, $\text{ANN}(\Pi, d)$ is the result of taking the intersection of all possible sets Λ of ground annotations over d satisfying Π, which, given that the rules in Π do not include any negative atoms, is equivalent to computing the minimum such set Λ under set inclusion. This corresponds to the usual least fixed point semantics for Datalog programs without negation.

EXAMPLE 3.3. Assume that we want to indicate that if a document d starts with the string *temp*, then d stores information about temperatures. To do this, we use the following program to annotate d with the tag TempDoc if it satisfies the previous condition:

$$x.\vdash /temp/\Sigma^*/\dashv \quad \rightarrow \quad \text{TempDoc}(x).$$

Notice that $\sigma \in [\![x.\vdash /temp/\Sigma^*/\dashv]\!]_d$ if, and only if, the content associated to the span $\sigma(x)$ is a string of the form $\vdash tempw \dashv$ with $w \in \Sigma^*$, that is, if and only if $\sigma(x)$ corresponds to the document d, which starts with the string *temp*. Second, assume that in every document storing information about temperatures, we need to annotate every value with the tag TempVal. If $D = \{0, 1, \ldots, 9\}$ and $N = (\Sigma \setminus D)$, then this is done by using an annotation program that combines the rule defining TempDoc with the following rule:

$$\text{TempDoc}(x) \wedge x.\vdash /\Sigma^*/N/y/N/\Sigma^*/\dashv \wedge\ y.R \rightarrow \text{TempVal}(y),$$

In the body of the rule, the annotation $\text{TempDoc}(x)$ indicates that x stores a span whose content is a document storing information about temperatures, while the expression $x.\vdash /\Sigma^*/N/y/N/\Sigma^*/\dashv$ is used to retrieve from x and store in y a span that could represent a temperature value. Notice that we do not want to extract a part of a temperature value, for example we do not want to extract 12 if the stored temperature is 12.3; thus, we check that the symbols right before and after y are not digits (they belong to N). Finally, assuming that $P = \{1, \ldots, 9\}$, we have that R is the SRE $(0 + PD^*).D^*$. Hence, the expression $y.R$ is used to check that the content of y is a decimal number. □

3.3 Annotation programs for CSV-like data

Annotation programs are designed to deal with general documents, relying on span regular expressions to extract spans from them. In this sense, annotation programs do not take advantage of the structure of CSV-like documents where separators play a key role. In fact, in many practical scenarios users only need a simple language that is oriented towards navigating CSV-like documents using these separators as landmarks. In these scenarios span regular expressions are, in general, too expressive, so we propose in this section a simple language that separates the navigational features needed in CSV-like documents from the use of regular expressions.

We start by defining a navigation language, which uses span regular expressions in a very restricted form but can express most of

the span-directed extraction used in practice. A *navigation expression* (NE) ψ is defined by the following grammar:

$$\begin{array}{rcl} \psi & ::= & \psi/\psi \mid \texttt{any}(S) \mid \texttt{next}(S) \mid \\ & & x : \texttt{next}(S),\, x \in \mathcal{V} \mid \langle x \rangle : \texttt{next}(S),\, x \in \mathcal{V} \\ S & ::= & w,\, w \in \Delta^+ \mid S + S \end{array}$$

where S is assumed to be prefix free, that is, every expression S is of the form $w_1 + \cdots + w_n$, where (a) $\Delta = \Sigma \cup \{\vdash, \dashv\}$, (b) every $w_i \in \Delta^+$ $(1 \leqslant i \leqslant n)$, and (c) w_i is not a prefix of w_j $(1 \leqslant i, j \leqslant n$ and $i \neq j)$.

An NE is constructed as a sequence of expressions using either `any` or `next`. The axis $\texttt{any}(S)$ is used to move forward in a document reading any sequence of symbols ending with a word in S, while the axis $\texttt{next}(S)$ is used to move forward to the next occurrence of a word in S. Moreover, $x : \texttt{next}(S)$ and $\langle x \rangle : \texttt{next}(S)$ perform the same form of navigation as $\texttt{next}(S)$, but in the former case the traversed span is stored in the variable x, while in the latter case it is checked whether the content of the traversed span coincides with the content of the span stored in x. Thus, the expressions $\texttt{any}(S)$, $\texttt{next}(S)$, $x : \texttt{next}(S)$ and $\langle x \rangle : \texttt{next}(S)$ are useful to restrict the navigation between two or more separators, which is a very common operation on CSV-like documents. Notice that we assume that S is prefix free, as the set of separators used in practice usually satisfies this restriction (e.g comma and semicolon).

We define the semantics of NEs in the same way as for span regular expressions. More precisely, given a document d and a variable assignment σ over d, the base case $[\![S]\!]_{d,\sigma}$ and the recursive case $[\![\psi_1/\psi_2]\!]_{d,\sigma}$ are defined as for the case of span regular expressions. Moreover, $[\![\texttt{any}(S)]\!]_{d,\sigma}$ is defined as $[\![(\Sigma \cup \{\vdash, \dashv\})^*/S]\!]_{d,\sigma}$. Finally, the evaluation of $\texttt{next}(S)$, $x : \texttt{next}(S)$ and $\langle x \rangle : \texttt{next}(S)$ are defined as follows assuming that $S = w_1 + \cdots + w_n$ and r_S is the SRE $\Delta^* w_1 \Delta^* + \cdots + \Delta^* w_n \Delta^*$ with $\Delta = \Sigma \cup \{\vdash, \dashv\}$:

$$\begin{array}{l} [\![\texttt{next}(S)]\!]_{d,\sigma} = \{(i,j) \in \text{span}(d) \mid \\ \qquad \exists k \geqslant i : (k,j) \in [\![S]\!]_{d,\sigma} \text{ and } (i, j-1) \notin [\![r_S]\!]_{d,\sigma}\} \\ [\![x : \texttt{next}(S)]\!]_{d,\sigma} = \{(i,j) \in \text{span}(d) \mid \exists k \geqslant i : \sigma(x) = (i,k), \\ \qquad (k,j) \in [\![S]\!]_{d,\sigma} \text{ and } (i, j-1) \notin [\![r_S]\!]_{d,\sigma}\} \\ [\![\langle x \rangle : \texttt{next}(S)]\!]_{d,\sigma} = \{(i,j) \in \text{span}(d) \mid \exists k \geqslant i : (i,k) \in \\ \qquad [\![\langle x \rangle]\!]_{d,\sigma}, (k,j) \in [\![S]\!]_{d,\sigma} \text{ and } (i, j-1) \notin [\![r_S]\!]_{d,\sigma}\} \end{array}$$

Notice that a span p belongs to $[\![r_S]\!]_{d,\sigma}$ if the content associated to p is a string of the form uw_iv with $u, v \in \Delta^*$ and $1 \leqslant i \leqslant n$, that is, if one of the separators in S occurs in the content associated to p. Thus, if $(i,j) \in [\![\texttt{next}(S)]\!]_{d,\sigma}$, then we know that there exists a position k such that $i \leqslant k < j$, the content of (k,j) is a word in S and no separator in S occurs between positions i and $j-1$. Hence, in this case we know that k is the next position from i where a separator from S occurs.

Extraction expressions based on NEs are defined exactly as in Section 3.1, that is, if ψ is an NE, then $\texttt{doc}.\psi$ and $x.\psi$ are considered to be extraction expressions if $x \in \mathcal{V}$. However, the semantics of these formulae are defined in a slightly different way:

$$\begin{array}{rcl} [\![\texttt{doc}.\psi]\!]_d & = & \{\sigma \mid \sigma \text{ is a variable assignment over } d \text{ such that} \\ & & \qquad (1,k) \in [\![\psi]\!]_{d,\sigma} \text{ for some } k, 1 \leqslant k \leqslant |d|+1\} \\ [\![x.\psi]\!]_d & = & \{\sigma \mid \sigma \text{ is a variable assignment over } d \text{ such that} \\ & & \qquad \sigma(x) = (i,j) \text{ and } (i,k) \in [\![\psi]\!]_{d,\sigma} \text{ for some } k, i \leqslant k \leqslant j\} \end{array}$$

This definition formalises the fact that NEs are intended to be used to navigate forward in a document until we find a separator, and without taking into consideration the symbols after this separator. Thus, $\sigma \in [\![\psi]\!]_d$ if there exists a prefix $(1,k)$ of the span $(1, |d|+1)$ representing the entire document d such that $(1,k)$ conforms to the conditions encoded in ψ, thus without taking into account the symbols in the positions $k+1, \ldots, |d|$.

We are ready to define our simplified notion of annotation program. A *navigation program* Ω is a finite set of rules of the form (†) where (a) $m \geqslant 0$ and $n \geqslant 0$; (b) φ_i $(1 \leqslant i \leqslant m)$ is an expression of the form either $\texttt{doc}.\psi$ or $x.\psi$ or $x.R$, where $x \in \mathcal{V}$, ψ is an NE and R is a SRE without variables; (c) $A_j \in \Gamma$ and $x_j \in \mathcal{V}$ $(1 \leqslant j \leqslant n)$; and (d) $A \in \Gamma$, $x \in \mathcal{V}$ and x occurs in the body of the rule. The evaluation of Ω over a document d, denoted by $\text{ANN}(\Omega, d)$, is defined as for the case of annotation programs.

EXAMPLE 3.4. Assume that d is a document where values are separated by commas. Moreover, suppose that the elements of the first column of d should form a primary key, so that two different rows of d should have distinct values in their first columns. We can annotate with the tag Error any violation to this constraint:

$$\texttt{doc.any}(\vdash + \hookleftarrow)/x : \texttt{next}(,)/\texttt{any}(\hookleftarrow)/\langle x \rangle : \texttt{next}(,) \rightarrow \text{Error}(x)$$

Thus, we can check if the first column of d is a primary key by verifying whether any span has been annotated with the tag Error. □

Recall that the navigation program (**) defined in Section 2 is equivalent to annotation program (*) from the same section. In fact, it can be proved that each navigation program can be transformed into an equivalent annotation program, thus showing that navigation programs are a simplification of annotation programs.

4. COMPLEXITY OF ANNOTATING CSV

The goal of this section is to study the complexity of evaluating an annotation program over a CSV-like document, for which we also have to study the evaluation problem for extraction expressions. This study allows us to identify the features of our framework that need more resources, and the fragments that can be evaluated efficiently.

We start by analysing the computational complexity of evaluating an extraction expression defined by either a span regular expression or a navigation expression. Given a document d and an extraction expression of the form $\texttt{doc}.\psi$, where $\{x_1, \ldots, x_k\}$ is the set of variables occurring in ψ, we are interested in computing the set $\{(\sigma(x_1), \ldots, \sigma(x_k)) \mid \sigma \in [\![\texttt{doc}.\psi]\!]_d\}$, that is, we are interested in enumerating all the variable assignments in $[\![\texttt{doc}.\psi]\!]_d$. To provide a lower bound for the complexity of this problem, and also to simplify the analysis, we consider the decision problem of verifying whether $[\![\texttt{doc}.\psi]\!]_d$ is not empty. Formally, if $\mathcal{L}$ is either the class $\mathcal{SRE}$ of span regular expressions or the class $\mathcal{NE}$ of navigation expressions, then we consider the following problem:

Problem:	NONEMP($\mathcal{L}$)
Input:	A document d and an expression $\psi \in \mathcal{L}$.
Question:	Is $[\![\texttt{doc}.\psi]\!]_d$ not empty?

Unfortunately, we can show that the problem is intractable.

THEOREM 4.1. NONEMP($\mathcal{SRE}$) *and* NONEMP($\mathcal{NE}$) *are both* NP*-complete.*

The proof of Theorem 4.1 gives a PTIME bound in the case of *data complexity*, where the expression $\psi \in \mathcal{L}$ is considered to be fixed. Unfortunately, the algorithm would require to check tuples of spans (of fixed arity) one by one, which depends exponentially in the number of (fixed) variables: a bound often not feasible in

practice. The proof also reveals the real source of intractability in NONEMP($\mathcal{L}$): the overuse of variables and the content operator $\langle\cdot\rangle$ in extraction expressions. In practice, a restriction on the use of the content operator is reasonable, as checking whether two spans have the same content is not used very often (see [28]). The following theorem shows that under this assumption, the complexity decreases for the case of navigation expressions.

THEOREM 4.2. *If the content operator is not allowed, then* NONEMP($\mathcal{SRE}$) *is* NP*-complete and* NONEMP($\mathcal{NE}$) *is in* PTIME.

The membership of NONEMP($\mathcal{NE}$) in PTIME opens a window to find an efficient algorithm for our initial enumeration problem for navigation expressions without the content operator. In fact, we provide such an algorithm in Section 6.

Now we move our attention to the evaluation of annotation programs. Similarly to the case of extraction expressions, we are interested in enumerating all the A-annotations in ANN(Π, d) for an annotation program Π, a document d, and an annotation label $A \in \Gamma$. In terms of complexity analysis, we define the natural decision problem associated to this computation problem, assuming that $\mathcal{P}$ is either the class $\mathcal{AP}$ of annotation programs or the class $\mathcal{NVP}$ of navigation programs:

Problem:	MEMB($\mathcal{P}$)
Input:	A document d, a program $\Pi \in \mathcal{P}$, and a ground annotation $A(p)$.
Question:	Is $A(p) \in$ ANN(Π, d)?

We know from Theorem 4.1 that MEMB($\mathcal{AP}$) and MEMB($\mathcal{NVP}$) are both intractable problems (i.e. NP-hard). The following result shows that these problems are indeed NP-complete.

THEOREM 4.3. MEMB($\mathcal{AP}$) *and* MEMB($\mathcal{NVP}$) *are both* NP*-complete.*

A consequence of Theorems 4.2 and 4.3 is that evaluating annotation programs without the content operator is still NP-complete. We do not know if the same happens for navigation programs without the content operator. On the other hand, evaluating navigation programs without the content operator and with a tree-like structure of each rule can be done efficiently. Formally, let $\theta = \ell_1.\psi_1 \wedge \cdots \wedge \ell_m.\psi_m$ be a conjunction of extraction expressions, where each ψ_i is a navigation expression and $\ell_i \in \mathcal{V} \cup \{\text{doc}\}$ $(1 \leqslant i \leqslant m)$, and assume that G_θ is a graph obtained from θ as follows. The set of nodes of G_θ is the set of variables occurring in θ, and there exists an edge (x, y) in G_φ if and only if $x = \ell_i$ and y occurs in ψ_i, for some $i \in \{1, \ldots, m\}$. Then we say that θ is *tree-like* if (1) $\ell_i \neq \ell_j$ for every $i, j \in \{1, \ldots, m\}$ such that $i \neq j$; and (2) G_θ is a rooted-forest (i.e. a set of rooted-trees). For example, the following conjunction of extraction expressions is tree-like:

$$\texttt{doc.any(}\hookleftarrow\texttt{)}/x\texttt{:next(}\hookleftarrow\texttt{)} \wedge x.y\texttt{:next(,)}/z\texttt{:next(,)} \wedge z.0^*$$

We say that a navigation program Ω is tree-like if for every rule in Ω of the form (†) (see Section 3.2), it holds that $\varphi_1 \wedge \cdots \wedge \varphi_m$ is tree-like.

THEOREM 4.4. *If we restrict to tree-like navigation programs not using the content operator, then* MEMB($\mathcal{NVP}$) *is in* PTIME.

The tree-like restriction just defined is a reasonable assumption over navigation programs.

5. USABILITY OF THE FRAMEWORK

Next, we analyse the usability of our framework in practice by considering the use cases put forward by the CSV on the Web working group [28]. Here we concentrate on the requirements from [28] which deal with metadata specification, and do not consider the ones prescribing how CSV files should be transformed into other formats. The use cases presented in [28] are very diverse and are mostly designed for tabular data like the file in Figure 1. By simple inspection, one can easily check that navigation programs can fulfil most of the requirements in [28], but not all of them. In particular, navigation programs cannot define *Foreign Key References* and *Association of Code Values With External Definitions* from [28]. To satisfy the missing requirements, one can extend our framework with two simple features: (1) managing multiple documents and (2) allowing binary relations in the consequents (heads) of rules. For the former extension, one can easily add the use of multiple documents to our framework by allowing extraction expressions of the form $d.\psi$, where d is the name of a document. For the latter, we can enrich our Datalog-like rules with binary predicates in their consequents, and then define their semantics in the same ways as in Section 3. Instead of giving the formal definitions, we illustrate these extensions with an example.

EXAMPLE 5.1. *The requirement Foreign Key References asks to cross-reference data between different files [28]. For example, suppose that we have two CSV documents, called d_1 and d_2, and we want to say that a row in d_2 is related with a row in d_1 if they have the same value in the first cell. We can define this as follows:*

$$d_1\texttt{.any(}\hookleftarrow\texttt{)}/x\texttt{:next(}\hookleftarrow\texttt{)} \wedge d_2\texttt{.any(}\hookleftarrow\texttt{)}/y\texttt{:next(}\hookleftarrow\texttt{)} \wedge x.z\texttt{:next(,)} \wedge y.\langle z\rangle\texttt{:next(,)} \rightarrow \mathit{FKey}(x, y)$$

Notice that the head of this rule is a binary relation FKey(x, y), which establishes an association between the spans in x and y.

Navigation programs satisfy all the use cases in [28] if extended with features (1) and (2). Besides, the inclusion of these features does not change the complexity analysis in Section 4; in particular, all the positive and negative results still hold for expressions and programs with multiple documents and binary predicates in the consequents of rules. Thus, we are convinced that navigation programs are a natural "sweet spot" between the expressiveness and efficiency needed to deal with metadata for CSV-like files.

At this point it is natural to ask whether one could allow the use of binary predicates anywhere in annotation programs, and also extend this to n-ary predicates. To answer this, one has to consider that our approach is lightweight and allows an efficient implementation, while implementing Datalog with n-ary predicates requires an extensive use of indexes and optimization techniques, which are not guaranteed to work efficiently in all cases. Also, one has to consider that evaluating Datalog programs with n-ary (intentional) predicates is EXPTIME-complete [2, 7] and, thus, all results in Section 4 will not hold in this extended framework. Thus, the inclusion of n-ary predicates is a very interesting but delicate issue that we have left for future work, where we plan to consider integration within a system like DLV [13].

6. EFFICIENT EVALUATION OF NEs

The use of our framework requires an efficient algorithm for enumerating all the variable assignments satisfying a navigation expression. The complexity results provided in Section 4 show that such a procedure does not exist if the content operator is allowed (unless PTIME = NP). On the other hand, we provide in this section such a procedure for navigation expressions

without the content operator. This algorithm has running time $O(|\psi| \cdot |d| + |\text{Output}|)$, where ψ is a navigation expression, d is a document and |Output| is the size of the output. Furthermore, this algorithm belongs to the class of constant-delay algorithms [20], namely, enumeration algorithms that take polynomial time in preprocessing the input (i.e. ψ and d), and constant time between two consecutive outputs (i.e. variable assignments).

6.1 A normal form for navigation expressions

The initial step for evaluating a navigation expression is to remove unnecessary any-operators from the input navigation expression. For this purpose, we introduce a normal form for navigation expressions and show that every formula can be transformed into this normal form. Specifically, we say that an NE φ is a *next-formula* if it is the concatenation of next-operators and at least one variable occurs in φ. Then a navigation expression is in *next normal form* (NNF) if it is of the form $\varphi_0/\texttt{any}(S_1)/\varphi_1/\ldots/\texttt{any}(S_k)/\varphi_k$, where $k \geqslant 0$, φ_0 is a sequence of next-operators and $\varphi_1, \ldots, \varphi_k$ are next-formulas. Thus, between any pair of contiguous any-operators there must exist at least one variable that captures a span.

The next step is to show that every NE φ can be efficiently converted into an equivalent NE ψ in NNF. Here we say that φ and ψ are equivalent if for every document d and every variable assignment σ over d, it holds that $[\![\varphi]\!]_{d,\sigma} = [\![\psi]\!]_{d,\sigma}$. Then we consider the following rewriting rules to meet our goal:

$$\begin{aligned} \psi_1/\texttt{any}(S)/\varphi/\texttt{any}(S')/\psi_2 &\rightarrow \psi_1/\texttt{next}(S)/\varphi/\texttt{any}(S')/\psi_2 \\ \psi_1/\texttt{any}(S)/\varphi &\rightarrow \psi_1/\texttt{next}(S)/\varphi \end{aligned}$$

where φ is a sequence of zero or more next-operators without variables, and ψ_1, ψ_2 are arbitrary NEs. The following example illustrates how these two rules can be used to convert a navigation expression into an equivalent one in NNF.

EXAMPLE 6.1. *Consider the following navigation expression:*

$$\varphi \;=\; \texttt{any(;)/next(;)/any(}\hookleftarrow\texttt{)/}x\texttt{:next(}\hookleftarrow\texttt{)/any(;)}$$

This NE is not in NNF as it starts with two any*-operations without variables in between. This can be solved by applying the first rewriting rule, giving us:*

$$\varphi_1 \;=\; \texttt{next(;)/next(;)/any(}\hookleftarrow\texttt{)/}x\texttt{:next(}\hookleftarrow\texttt{)/any(;)}$$

Notice that φ and φ_1 are equivalent NEs. Now φ_1 is not in NNF as it ends with an any*-expression. This can be solved by applying the second rewriting rule, resulting in:*

$$\varphi_2 \;=\; \texttt{next(;)/next(;)/any(}\hookleftarrow\texttt{)/}x\texttt{:next(}\hookleftarrow\texttt{)/next(;)}$$

Finally, we have that φ_2 is in NNF and φ_2 is equivalent to φ.

It can be proved that every NE φ can be transformed into an equivalent NE ψ in NNF by using the previous rewriting rules. Moreover, this transformation can be performed in time $O(|\varphi|)$.

6.2 An efficient algorithm for evaluating NEs

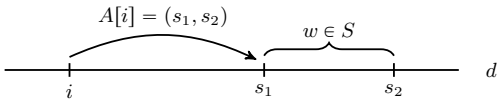
We divide the evaluation of an NE into four steps. The input of this process is an NE ψ in NNF and a document d, and then the output is the set of variable assignments σ such that $\sigma \in [\![\texttt{doc}.\psi]\!]_d$. We assume that ψ has no repeated variables, as if it had then the evaluation of ψ would be empty (recall that the content operator is not used, and two spans (i, j) and (k, ℓ) are assumed to be equal if $i = k$ and $j = \ell$). Besides, it can be easily checked whether ψ has repeated variables.

The first procedure takes as input a document d and a prefix-free set of words S (recall that in an NE of the form $\texttt{any}(S)$ or $\texttt{next}(S)$, the set S is assumed to be prefix-free). The procedure then runs the Aho-Corasick algorithm [4] to produce an array A that is of the length of the input document, and such that $A[i]$ stores the next span in d that matches a word from S for every $i \in \{1, \ldots |d|\}$. This idea is illustrated in the following figure.

Data: A document d and a prefix-free set of words S
Result: An array $A[1..|d|+1]$ of span(d)

Function separators_match(d, S)
  aho_corasick.init(d, S)
  $m := 1$
  while $(s_1, s_2) :=$ aho_corasick.next() **do**
    for $i = m$ **to** s_1 **do**
      $A[i] := (s_1, s_2)$
    $m := s_1 + 1$
  return A

Figure 2: Finding all matches for a set of separators.



In this figure, and others illustrating how the evaluation works, the straight line represents the input document d, while the markings i, s_1, s_2 denote positions inside the document. Recall that the content of a span (i, j) is the infix of d between position i and $j - 1$.

The algorithm itself (called separators_match) is given in Figure 2. To analyse the algorithm, observe that we repeatedly run the Aho-Corasick string matching procedure. The iterator m starts at the begining of the document and after we find a match (s_1, s_2) for some string in S we store this span into $A[m]$ through $A[s_1]$. After this the iterator m is set to $s_1 + 1$, as this is the position of the next possible match. It is important to stress that there are no matches for strings in S beginning between positions m and s_1, therefore (s_1, s_2) is the first possible match. Besides, due to the fact that S is prefix-free, this is also the only possible match starting at s_1. As the running time of the Aho-Corasick algorithm is $O(|S| + |d|)$ (since S is prefix-free), and the only overhead we have is assigning spans to the array A, the total time of the algorithm separators_match is still $O(|S| + |d|)$.

The next part of the algorithm, presented in Figure 3, deals with computing the possible valuations for a *context*, that is, a subformula of the expression that is of the form $\texttt{any}(S)/v_1\texttt{:next}(S_1)/\ldots/v_n\texttt{:next}(S_n)$, where v_i is either a variable, or a placeholder $\perp$, specifying that next is used without a variable. Note that contexts are the building blocks of any expression in NNF since any NNF-expression is of the form $\varphi_0/E_1/\ldots/E_n$ where φ_0 is a sequence of next-operators and each subformula E_i is a context. Similarly to the previous algorithm, here we will again return an array whose i-th position will contain the information about the next possible match that occurs after the position i.

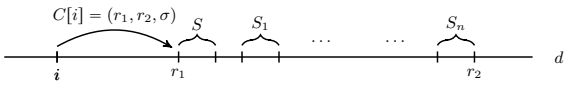
To start the computation, the algorithm context_match in Figure 3 calls the function separators_match from Figure 2 for each of the input sets of words S, S_1, …, S_n. The information for the set S is stored in an array A, while the information about each S_i is stored in an array $B[i]$. Therefore, A and $B[i]$, for $i = 1 \ldots n$, are all arrays of size $|d|$. This means that we can also refer to B as a matrix whose entry $B[i][j]$ contains the information about the next span matching a word from the set S_i after the position j of the input document d.

Data: A document d, a prefix-free set of words S, and a sequence $(v_1, S_1), \ldots (v_n, S_n)$ where each S_i is a prefix-free set of words and $v_i \in \mathcal{V} \cup \bot$.
Result: An array $C[1..|d|+1]$ of triples (r_1, r_2, σ), where $(r_1, r_2) \in \text{span}(d)$ and $\sigma : \mathcal{V} \to \text{span}(d)$ is a partial function.

Function context_match$(d, S, (v_1, S_1), \ldots (v_n, S_n))$
- $A :=$ separators_match(d, S)
- **for** $i = 1$ **to** n **do**
  - $B[i] :=$ separators_match(d, S_i)
- $m := 1$
- **while** $A[m] \neq$ `null` **do**
  - $(s_1, s_2) := A[m]$
  - $r_1 := s_1$
  - $\sigma := \varnothing$
  - **for** $i = 1$ **to** n **do**
    - **if** $B[i][s_2] =$ `null` **then break**
    - $(t_1, t_2) := B[i][s_2]$
    - **if** $v_i \neq \bot$ **then** $\sigma(v_i) := (s_2, t_1)$
    - $(s_1, s_2) := (t_1, t_2)$
  - **if** $i = n + 1$ **then**
    - $r_2 := s_2$
    - **for** $i = m$ **to** r_1 **do**
      - $C[i] := (r_1, r_2, \sigma)$
    - $m := r_1 + 1$
  - **else if** $A[r_1 + 1] \sqsubseteq A[r_1]$ **then**
    - $A[m] := A[r_1 + 1]$
  - **else break**
- **return** C

Figure 3: Finding all matches for a context.

The main loop of the algorithm now proceeds to try and match the context expression $\texttt{any}(S)/v_1 : \texttt{next}(S_1)/\ldots/v_n : \texttt{next}(S_n)$ to the input document d to the right of the position m (beginning with $m = 1$). As long as the algorithm can keep on matching S (the condition $A[m] \neq \texttt{null}$), it stores the information about the next possible match and tries to match S_1 through S_n (now using the matrix B in the for loop). If the matching was successful we store the information about the used-up portion of the document d into the (context) array C (note that this means that all positions from m to the start of the match for S, namely r_1, will have this information), and then we start matching again from the position $r_1 + 1$. In the case we did not manage to match all of the context expression, the only other possibility of a successful match is when we have that $A[r_1 + 1]$ is contained in $A[r_1]$, where a span (k_1, k_2) is contained in an span (ℓ_1, ℓ_2), denoted by $(k_1, k_2) \sqsubseteq (\ell_1, \ell_2)$, if $\ell_1 \leqslant k_1$ and $k_2 \leqslant \ell_2$. To clarify why this is so consider the following illustration of what the algorithm does.



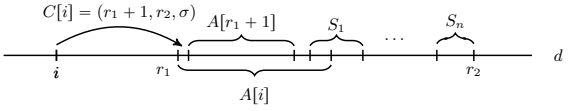
If there is a match for S that starts after the position r_1, ends after the ending position of the current match for S, and allows to successfully match all the sets S_1 through S_n, then so does the current match of S (after all our expression only asks for the next position that matches each S_i and nothing more). Therefore, if the match starting at r_1 fails, so does one starting after r_1 that is longer than it.

Data: A document d and a sequence $C_1, \ldots, C_n$ such that each C_i is an array $C_i[1..|d|+1]$ of triples (r_1, r_2, σ), where $(r_1, r_2) \in \text{span}(d)$ and $\sigma : \mathcal{V} \to \text{span}(d)$ is a partial function.
Result: An array $R[1..|d|+1]$ over $\{1, \ldots, n+1\}$.

Function forward_index$(d, C_1, \ldots, C_n)$
- $k := n$
- **for** $i = |d| + 1$ **to** 1 **do**
  - **if** $k = 0$ **then**
    - $R[i] := 1$
  - **else if** $C_k[i] =$ `null` **then**
    - $R[i] := k + 1$
  - **else**
    - $(r_1, r_2, \sigma) := C_k[i]$
    - **if** $R[r_2] = k + 1$ **then** $k := k - 1$
    - $R[i] := k + 1$
- **return** R

Figure 4: Computing the forward index.

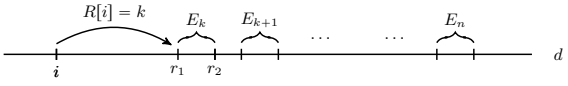
On the other hand, if the match at r_1 fails, but there is one starting at or after $r_1 + 1$ that is contained in it, there is a possibility for this match to be extended to cover all the sets S_1 to S_n. This possibility is illustrated in the figure below.



Notice that if all of the possibilities fail, we have exhausted our options and the algorithm finishes. To analyse the running time of the algorithm context_match, we first notice that running separately the function separators_match for the sets S, $S_1, \ldots, S_n$ takes total time $O(|S| + \sum_{i=1}^{n} |S_i| + (n+1) \cdot |d|)$. Moreover, the while loop of the algorithm takes time $O(n \cdot |d|)$, as we have to loop over every set S_i, for $i = 1 \ldots n$. We can thus conclude that running the procedure context_match from Figure 3 takes time $O(|S| + \sum_{i=1}^{n} |S_i| + n \cdot |d|)$, which is indeed $O(|\psi| \cdot |d|)$ where $|\psi|$ is the size of the input navigation expression.

With the two procedures presented before, we are able to find all matchings for a context of the form $\texttt{any}(S)/v_1 : \texttt{next}(S_1)/\ldots/v_n : \texttt{next}(S_n)$. Recall that every expression in NNF is simply a concatenation of such contexts, plus an easily evaluable initial segment. In fact, the last two steps of the algorithm assume that we have partitioned our input navigation expression ψ into contexts $E_1, \ldots, E_n$ of such subexpressions, and we have computed the corresponding arrays $C_1, \ldots, C_n$ using the algorithm context_match from Figure 3.

The next step of our procedure is a quick index building algorithm (called forward_index) that will allow us to discard positions not leading to a match in an efficient manner. This step, presented in Figure 4, computes an array R such that for each position i in the input file (e.g. CSV-like document), $R[i]$ contains the least number k such that, starting from position i, it is possible to find a match for the subexpression $E_k/E_{k+1}/\cdots/E_n$ of the input expression. This idea can be depicted as follows:



As expected, the algorithm forward_index traverses the input document d backwards and at each position tries to match the context E_k, starting with $k = n$. If a match is possible (the second else clause in Figure 4), we take note of that and reduce the index k by one in order to move to the previous subexpression E_{k-1}. Since all of the information is already stored in arrays $C_1, \ldots, C_n$, the cost of this part of the algorithm is simply the cost of traversing the input document d once, or in other words $O(|d|)$.

The final part of the algorithm, presented in Figure 5, computes all the possible valuations that make an expression $\psi = E_1/E_2/\ldots/E_n$ true, where each E_i is a context of the form $\texttt{any}(S)/v_1 : \texttt{next}(S_1)/\ldots/v_n : \texttt{next}(S_n)$. As input, all_matches takes all of the information computed by the previous algorithms. In particular it has at its disposal the arrays C_i corresponding to the context E_i (computed by the function context_match), as well as the array R from forward_index. With this information the algorithm proceeds to compute the output in time that is proportional to the number of valuations that allow for a successful match of ψ.

The idea of the evaluation is to simulate the typical recursive approach that tries all possible combinations of matchings for E_1 through E_n and backtracks as necessary. To do this, we use the array T whose i-th position stores the next possible starting point for a match of the context E_i. Intuitively, T acts as a stack where we store the current position of a match for the context E_i in $T[i]$. Whenever $T[1]$ to $T[i]$ contains a match of E_1 to E_i, respectively, we try to match the contexts E_{i+1} to E_n (and compute all the successful matches for them) before we move the starting point of the next match for E_i one position to the right. The procedure is then repeated until we have exhausted the search space. What makes our approach efficient is the fact that we only explore a branch of the search tree that is guaranteed to lead to a match. Specifically, we reduce the search space by using the index R from forward_index, which tell us if a match from the current position is possible. Next we describe this process in more detail.

The algorithm all_matches uses the iterator m to denote which context E_m is currently processing. It starts with $m = 1$ and with $T[1] = 1$, thus assuming that it will be possible to match the entire expression ψ to the input document d. In each step the algorithm then checks if it can match the part of the expression starting from E_m, namely, the subexpression $E_m/\ldots/E_n$. If this is not possible (the condition $R[T[m]] > m$ is true), then we simply move to the previous subexpression and try to match it from the next position to the right. Note that the use of array R allows us to terminate the evaluation of a branch that will not lead to a successful match at the first possible occasion. If a match is possible, we take note of that (σ_m stores the valuation for E_m using the information precomputed in array C_m) and move to the next context. This step is executed in the else clause of the algorithm all_matches. Finally, if m reaches $n + 1$, then we manage to match the entire expression (the final if clause), so we take the union of σ_1, ..., σ_n to produce a variable assignment that makes ψ true (this union is well defined as ψ does not have repeated variables). Then we try to match E_n again, but this time starting one position to the right of the previous match. The algorithm then moves downwards to find the next match for E_{n-1} and so on until it found all the matches for ψ.

To analyse the running time of the algorithm, first notice that the total running time needed to precompute arrays C_i and R is bounded by $O(|\psi| \cdot |d|)$. As discussed above, the final part of the algorithm simply outputs all valid matchings, and does so knowing in advance if a branch in the tree of all possible matches will result in a valid match. Therefore the running time of that part of the algorithm is equal to the number of valid matchings for the expression, or size of the output. Summing up, the total running time of the algorithm is $O(|\psi| \cdot |d| + |\text{Output}|)$. The algorithm is also constant-delay [20], as it takes $O(|\psi| \cdot |d|)$ time to preprocess the input, and constant time between two consecutive outputs.

Data: A document d, an array $R[1..|d|+1]$ over $\{1, \ldots, n+1\}$, and a sequence $C_1, \ldots, C_n$ such that each C_i is an array $C_i[1..|d|+1]$ of triples (r_1, r_2, σ), where $(r_1, r_2) \in \text{span}(d)$ and $\sigma : \mathcal{V} \to \text{span}(d)$ is a partial function.
Result: A set O of partial functions $\sigma : \mathcal{V} \to \text{span}(d)$.

Function $\text{all\_matches}(d, R, C_1, \ldots, C_n)$
- **Let** $T[0..n+1]$ be an array of integers
- **Let** $\sigma_1, \ldots, \sigma_n$ be partial functions from $\mathcal{V}$ to span(d)
- $T[1] := 1$
- $m := 1$
- **while** $m \geqslant 1$ **do**
  - **if** $R[T[m]] > m$ **then**
    - $m := m - 1$
    - $T[m] := T[m] + 1$
  - **else**
    - $(r_1, r_2, \sigma) := C_m[T[m]]$
    - $T[m] := r_1$
    - $\sigma_m := \sigma$
    - $m := m + 1$
    - $T[m] := r_2$
  - **if** $m > n$ **then**
    - $O := O \cup \{\sigma_1 \uplus \cdots \uplus \sigma_n\}$
    - $m := m - 1$
    - $T[m] := T[m] + 1$
- **return** O

Figure 5: Computing all possible matches.

7. EXPERIMENTAL EVALUATION

To illustrate that the algorithm from Section 6 does not only have good theoretical complexity, in this section we describe how a system based on its implementation performs over real world datasets. Here we describe the datasets and the experiments used to test the efficiency of this algorithm, and compare it with the stream editing tool AWK. Due to the lack of space programs used in the experiments have been omitted, but are made available at [1], where we also provide the complete source code and documentation of our implementation and more detailed results of the experiments.

Implementation details. Our prototype implements a restricted but functional version of navigation programs that covers all use cases in [28]. To simplify the implementation, we restrict navigational expressions to use at most one content operator, rules to be tree-like (see Theorem 4.4), and navigational programs to be non-recursive. It is important to add that to cope with the requirement *ForeignKeyReferences* we allow binary relations in the head of rules for storing results. For the evaluation of these programs, we use the algorithm discussed in Section 6 to evaluate navigational expressions. Furthermore, to efficiently evaluate the content operator we use an extension of the algorithm where matches under the content operator are sent to a Hash table in order to easily check content equivalence. Finally, for the evaluation of non-recursive programs we compute each rule in order, evaluating its navigational expressions separately, and intersecting their results with the precomputed annotations mentioned in the rule.

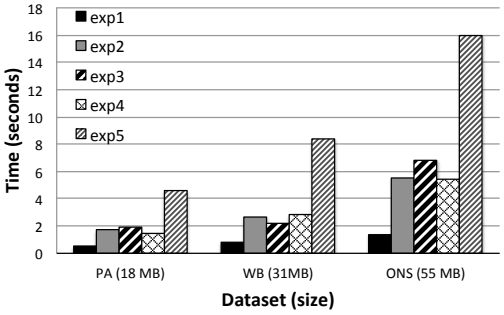


Figure 6: Running time on medium sized CSV files.

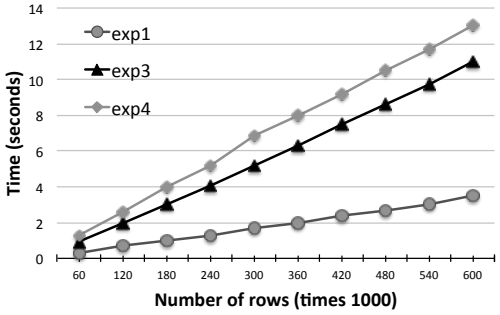


Figure 7: Scaling based on file size.

Datasets. We test our implementation on a number of CSV datasets and query logs. CSV files come from the use cases considered by W3C's CSV on the Web working group [28] and use data from: The World Bank (WB) [29], Office for National Statistics UK (ONS) [16] and the City of Palo Alto tree data (PA) [23]. While the Palo Alto tree data provides us with only one CSV file, for other two datasets we used many files of different sizes. In the end we tested our implementation on a total of thirteen different CSV files with sizes ranging from 6 to 183 MB. Note that most of the files published by these organisations are of much smaller size, however, we decided to use larger files in order to show how the implementation works in extreme circumstances. As all of the CSV files we obtained were well structured, we also tested how our implementation behaves on noisy data by modifying the existing files by including additional empty rows, changing the expected values in one column in 5% of the rows, and adding an extra column to 5% of the rows. For each CSV document we created its noisy variant and used it for testing. The query log files we use were collected by the Linked SPARQL Queries Dataset team [24] and come from public SPARQL endpoints of the British Museum [22] and DBPedia [8]. For the experiments we used the raw log files provided by [24] and tested our implementation on a total of 19 files, their sizes ranging between 2 and 190 MB. Due to a large number of files,and since files of similar size show same trends in evaluation times, we will present the evaluation result for only a handful of them. We provide the complete results for all the files in [1].

Experiments for CSV files. These experiments were motivated by the requirements for CSV metadata proposed by the W3C CSV on the Web working group [28]. To test our implementation we will use five different experiments that run navigation programs which annotate CSV files, or check if some constraint is violated. Our first experiment (`exp1`) annotates the entire file. This type of annotation is used when we want to specify that the file is in particular language, or that it is to be displayed in a particular way. Next, in `exp2` we annotate a single column within a file, which is used when one wants to specify that this column contains values that are of a certain type, or that it forms a primary key. In `exp3` we check if a primary key constraint is violated in our file. We continue with `exp4` where conformance of a column to a datatype specification is tested. Finally, in `exp5` we use a more complex navigation program consisting of three rules, each of which annotates a different column and checks that its values are of a certain datatype.

Experiments for query logs. Here we were annotating the data one might naturally want to obtain when managing log files, such as the actual text of the query, the time it was executed, the endpoint used, etc. Over these datasets we also use five experiments. The first experiment `exp1` annotates all the queries which use the OPTIONAL operator of the SPARQL query language. Similarly, in `exp2` we find all the queries which have more than one occurrence of this operator. In `exp3` we annotate all the queries using the FILTER operator that were executed between noon and one o'clock. Next, in `exp4` we find the queries where the operator OPTIONAL appears at least twice, and using a binary output relation store, such queries together with the time when they were executed. The final experiment, `exp5`, works similarly, but stores queries with two occurrences of the operator FILTER, together with the details about the endpoint used to execute them.

Results. The testing was done using a Laptop with an Intel Core i7-4510u processor and 8 GB of main memory, running Arch Linux x86_64, kernel 4.2.2. Each experiment was ran three times and the average score was reported (we also note that there were no significant deviations from the average).

The first set of results, presented in Figure 6, shows the running times when CSV files of reasonable size are used. Here we test on three files: the Palo Alto trees file is 18 MB in size, while the World Bank file weights 35 MB and the ONS one 55 MB. As expected, the running times scale according to the size of the file, but as all of the times were really fast (less than sixteen seconds), we can conclude that on average sized files our implementation runs well considering that we use raw data with no precomputed indices. One can notice that times are much longer for experiment 5, however, this is to be expected, as the program `exp5` does about three times more work than any of the other programs, as discussed above.

Next we wanted to see how the results scale when similar files of increasing sizes are used. To test this we selected one large CSV file from the World Bank dataset of size 130 MB and containing around 600,000 lines. From this we generated ten different files, each containing first $k \times 60{,}000$ lines of the original file, with k between one and ten. The experiments were then ran against each of these files. As we can see from Figure 7, the results do scale as expected from the formal analysis in Section 6. Here we selected experiments 1, 3 and 4 as they are the most representative. The other two experiments behave similarly (see [1] and Table 1).

We also considered three large files (one obtained from the World Bank and two from the ONS). Although these files are in no way representative of the average size appearing in practice, we wanted to see if our implementation could still be used in these cases. The file from the World Bank dataset was 160 MB in size, while the two from the ONS weighed 175 MB and 183 MB. The test results are show in Table 1. As we can see, for files less than 200 MB in size there are no significant problems in terms of the evaluation. Since in practice one is likely to work with files that fall on the smaller end of the spectrum, we believe that our experiments serve to illustrate that navigation programs can be used efficiently.

	exp1	exp2	exp3	exp4	exp5
WB (160 MB)	4	13	12	13	39
ONS (175 MB)	5	16	17	16	47
ONS (183 MB)	5	17	18	17	50

Table 1: Running times (in seconds) on large files.

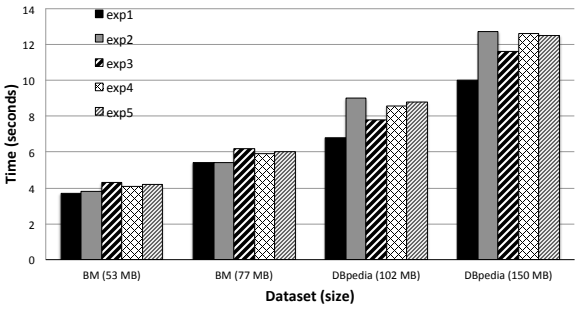


Figure 8: Running times on query logs.

Our final round of experiments for CSV files was conducted using noisy documents which do not conform to the tabular format as described in [26]. The theoretical analysis of our programs in Section 6 showed that, whether a file is noisy or not, this should not have much impact on the performance of the evaluation and our experiments on CSV files with synthetically created noise show this. In particular, since the amount of noise we added did not significantly change the size of the files, the performance for each experiment was essentially the same as on original data (in fact the results on noisy files are generally slightly faster as less data passes the filters and gets stored). The details on noisy files and the precise performance of the experiments over them can be found at [1].

As far as the experimental results on query logs are concerned they show similar performance as the ones on CSV files. In Figure 8 we present evaluation times of our five programs for four different query log files. From the logs of the British Museum we selected the smallest and the largest file available, and from DBPedia we selected two files on the larger end of the spectrum. The sizes were selected so that they further illustrate the fact that the performance scales as the size of the document increases.

Comparison with stream editing tools. We also compare our implementation with the standard stream editing tool AWK [5]. Although AWK is less expressive than the framework we propose, in some cases AWK programs which simulate navigation expressions can be constructed. To carry out the comparison we created AWK programs equivalent to navigation expressions used in `exp1` through `exp4` over query logs. Of course, when allowed to match just a single pattern line by line, AWK performs better than our implementation; however, when asked to produce the same set of annotations as our experiments over the files being treated as a single line, the majority of AWK programs took more than a minute, or ran out of memory, while processing the DBPedia and British Museum query logs. In comparison, our implementation computed the answers in less than 14 seconds (Figure 8). Complete programs and running times can be found at [1]. We can conclude that when annotations spanning multiple lines, or with noisy files missing many new line symbols, AWK might not be the best choice.

Acknowledgments. We would like to thank the anonymous reviewers for many helpful comments. This work was funded by the Millennium Nucleus Center for Semantic Web Research under Grant NC120004.

8. REFERENCES

[1] Online Appendix. `http://dvrgoc.ing.puc.cl/CSV`, 2016.
[2] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
[3] A. V. Aho. Algorithms for finding patterns in strings. In *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity (A)*, pages 255–300. 1990.
[4] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6), 1975.
[5] A. V. Aho, B. W. Kernighan, and P. J. Weinberger. *The AWK Programming Language*. Addison-Wesley, 1988.
[6] D. Bhagwat, L. Chiticariu, W.-C. Tan, and G. Vijayvargiya. An annotation management system for relational databases. *The VLDB Journal*, 14(4):373–396, 2005.
[7] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. Complexity and expressive power of logic programming. *ACM Comput. Surv.*, 33(3):374–425, 2001.
[8] DBpedia team. DBpedia. `http://dbpedia.org`, 2015.
[9] S. J. Edwards. Portable game notation. `https://en.wikipedia.org/wiki/Portable_Game_Notation`.
[10] R. Fagin, B. Kimelfeld, F. Reiss, and S. Vansummeren. Document spanners: A formal approach to information extraction. *Journal of the ACM*, 62(2), 2015.
[11] F. Geerts and J. V. den Bussche. Relational completeness of query languages for annotated databases. *J. Comput. Syst. Sci.*, 77(3), 2011.
[12] HM Government Land Registry. Price Paid Data. `https://www.gov.uk/government/collections/price-paid-data`.
[13] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV system for knowledge representation and reasoning. *ACM Trans. Comput. Log.*, 7(3):499–562, 2006.
[14] W. Martens, F. Neven, and S. Vansummeren. SCULPT: A Schema Language for Tabular Data on the Web. In *WWW*, pages 702–720, 2015.
[15] S. Melnik, E. Rahm, and P. A. Bernstein. Rondo: A programming platform for generic model management. In *ACM SIGMOD Conference*, pages 193–204, 2003.
[16] Office for National Statistics. ONS Data Explorer. `http://www.ons.gov.uk/ons/data/web/explorer`.
[17] Open Geospatial Consortium. Well-known text. `https://en.wikipedia.org/wiki/Well-known_text`.
[18] F. Reiss, S. Raghavan, R. Krishnamurthy, H. Zhu, and S. Vaithyanathan. An algebraic approach to rule-based information extraction. In *ICDE 2008*, pages 933–942, 2008.
[19] Research Information systems. RIS (file format). `https://en.wikipedia.org/wiki/RIS_(file_format)`, 2015.
[20] L. Segoufin. A glimpse on constant delay enumeration. In *STACS 2014*, pages 13–27, 2014.
[21] W. Shen, A. Doan, J. F. Naughton, and R. Ramakrishnan. Declarative information extraction using datalog with embedded extraction predicates. In *VLDB*, pages 1033–1044, 2007.
[22] The British Museum. British Museum public endpoint. `http://bm.rkbexplorer.com`, 2015.
[23] The City of Palo Alto, California Urban Forest Section. Palo Alto tree data. `http://w3c.github.io/csvw/use-cases-and-requirements/Palo_Alto_Trees.csv`, 2015.
[24] The LSQ team. The Linked SPARQL Queries Dataset. `http://aksw.github.io/LSQ/`, 2015.
[25] Unix.com. Man page for grep. `http://www.unix.com/man-page/linux/1/grep/`.
[26] W3C. Model for Tabular Data and Metadata on the Web. `http://www.w3.org/TR/tabular-data-model/`.
[27] W3C. CSV on the Web Working Group. `http://www.w3.org/2013/csvw/`, 2013.
[28] W3C. CSV on the Web: Use Cases and Requirements, Editor's Draft. `http://w3c.github.io/csvw/use-cases-and-requirements/`, December 2014.
[29] World Bank. Data Records. `http://data.worldbank.org/`.