Koko: A System for Scalable Semantic Querying of Text

Xiaolan Wang* Jiyu Komiya† Yoshihiko Suhara† Aaron Feng†
Behzad Golshan† Alon Halevy† Wang-Chiew Tan†
*University of Massachusetts †Megagon Labs
xlwang@umass.cs.edu {jiyu,yoshi,aaron,behzad,alon,wangchiew}@megagon.ai

ABSTRACT

KOKO is a declarative information extraction system that incorporates advances in natural language processing techniques in its extraction language. KOKO's extraction language supports simultaneous specification of conditions over the surface syntax and on the structure of the dependency parse tree of sentences, thereby allowing for more refined extractions. Furthermore, the KOKO extraction language allows for aggregating evidence from an input document and supports conditions that are tolerant of linguistic variation of expressing concepts.

In this demo, we outline the design of KOKO, a system for extracting information and understanding the results of the extraction. KOKO provides an interactive interface that allows participants to write queries, understand the input and results of the queries. In particular, the user can customize the input text, visualize the input text's dependency parse trees, and understand the correspondences between query components, dependency tree nodes, text tokens, and the computation and associated scores that led to an extraction.

PVLDB Reference Format:
Xiaolan Wang, Jiyu Komiya, Yoshihiko Suhara, Aaron Feng, Behzad Golshan, Alon Halevy, Wang-Chiew Tan. Koko: A System for Scalable Semantic Querying of Text. *PVLDB*, 11 (12): 2018-2021, 2018.
DOI: https://doi.org/10.14778/3229863.3236249

1. INTRODUCTION

Information extraction is the task of extracting structured data from text. Today, information extraction systems are either machine-learning based systems or rule-based systems. Machine-learning based systems often require a significant amount of training data to train a reasonable extraction model and they are more opaque to understand. In contrast, rule-based systems enjoy the advantage that they do not require training data and the results they produce are explainable [2]. KOKO is a declarative rule-based system that takes information extraction to a new level where advances in natural language processing techniques are incorporated into the extraction language.

Today, many extraction languages express extraction patterns through regular expressions, combined with conditions on the POS (part of speech) tags, over the surface text of a sentence. However, such regular expression patterns may not be sufficient for many extraction tasks.

EXAMPLE 1.1. *Suppose our goal is to extract food that are described as delicious in text. One approach would be to specify an extraction pattern that looks for the word* "delicious" *preceding a noun that is known to be in the category of foods. However, such extraction pattern may fail since the word* "delicious" *does not always immediately precede the noun of interest. For example, for the sentence* "I ate a delicious and salty pie with peanuts"*, the word* "delicious" *does not immediately precede the word* "pie"*, and it also precedes the word* "peanuts" *which were not deemed delicious. In the sentence* "I ate a chocolate ice cream, which was delicious, and also ate a pie" *the word* "delicious" *comes* after "ice cream" *which makes the extraction via regular expressions even more challenging.*

KOKO overcome the above challenges by supporting conditions that combines the surface-level patterns, e.g., regular expressions, with the semantic tree patterns over the text sentences. The semantic tree structure is represented as *dependency parse trees* (or *dependency trees* in short). The dependency tree of Example 1.1 is shown in Figure 1. The dependency tree shows that the word "*delicious*" is in the subtree of the direct object of the verb "*ate*", which is the "*chocolate ice cream*". As we explain later, this intuitively means that "*delicious*" refers to "*chocolate ice cream*". Hence, a pattern over the dependency tree that looks for the word "*delicious*" in the subtree of a noun in the food category could provide the correct extractions.

In addition to the expressive hybrid conditions as mentioned above, the KOKO language allows for extractions that accommodate variation in linguistic expression and aggregation of evidence. To exemplify, consider the task of extracting cafe names from blog posts. It is challenging to write rules that would extract them accurately since there is a variety of names that can be cafe names. However, it is possible to combine evidence from multiple mentions in the text to extract cafe names with high confidence. For example, if we see that an entity *employs baristas* and *serves espresso*, we might infer that it is a cafe. However, there are wide linguistic variations on how these properties are expressed in blogs. For example, a blog may express that a cafe *serves up delicious cappuccinos* or *hired the star barista* while another blog may mention that they *have world-champion baristas serving coffee*. KOKO includes a semantic similarity operator that retrieves phrases that are linguistically similar to the one specified in the rule. Semantic similarity can be determined using paraphrase-based word embeddings. KOKO attaches a confidence value to the phrases matched by the similarity operator, and these confidence values can be aggregated from multiple pieces of evidence in a document.

Proceedings of the VLDB Endowment, Vol. 11, No. 12
Copyright 2018 VLDB Endowment 2150-8097/18/8.
DOI: https://doi.org/10.14778/3229863.3236249

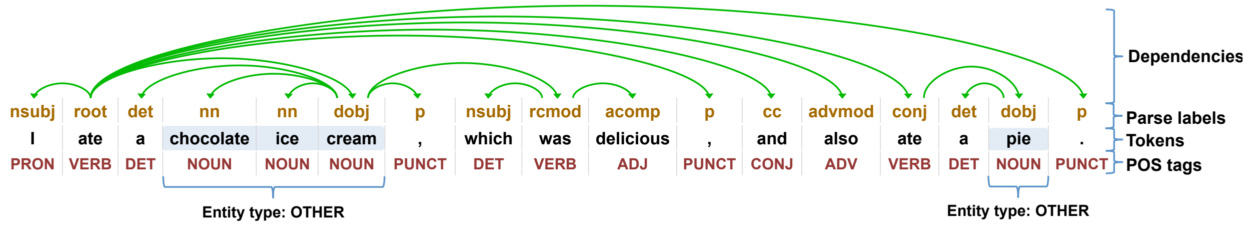

Dependencies
Parse labels
Tokens
POS tags
nsubj root det nn nn dobj p nsubj rcmod acomp p cc advmod conj det dobj p
I ate a chocolate ice cream , which was delicious , and also ate a pie .
PRON VERB DET NOUN NOUN NOUN PUNCT DET VERB ADJ PUNCT CONJ ADV VERB DET NOUN PUNCT
Entity type: OTHER
Entity type: OTHER

Figure 1: A sentence and its dependency tree annotated with parse labels [4], tokens, POS tags [5], and entity types. This dependency tree is generated from Google Cloud NL API [3].

Compare to existing extraction systems that also allow users to specify tree patterns, KOKO is novel in that it provides a single declarative language that combines the surface-level patterns with the tree patterns, accommodates variation in linguistic expression and aggregation of evidence, and uses novel indexing techniques to scale to large corpora. Our multi-indexing scheme is efficient in pruning irrelevant sentences, thus achieves at least 7× speedup compared to prior indexing techniques.

In this demonstration, we will showcase the KOKO system. Specifically, we will demonstrate the features of the KOKO language, including its ability to query over surface-level patterns in conjunction with tree patterns, and in combination with its ability to combine evidence from different parts of the text. Furthermore, we will demonstrate the ability of the KOKO system to explain its results. Given a selected result, KOKO can unfold the derivation by displaying the surface level patterns and dependency tree bindings, and the sentences (and relevant portions of it) where the evidence is combined from. We will also demonstrate KOKO's ability to scale by executing queries over 5 million Wikipedia articles. A preliminary version of KOKO is available at `https://github.com/biggorilla-gh/koko` and the full version of KOKO will soon be available in open source.

2. THE KOKO LANGUAGE

A basic KOKO query has the following form.

extract ⟨*output tuple*⟩ from ⟨*input.txt*⟩ if
  ⟨*variable declarations, conditions, and constraints*⟩
[satisfying⟨*output variable*⟩
  ⟨*conditions for aggregating evidence*⟩
with threshold α]
[excluding ⟨*conditions*⟩]

The KOKO query consists of essentially two parts: the extract clause which consists of conditions on the surface text with regular expressions and conditions on the hierarchical structure of the dependency tree. The other part, the satisfying clause contains linguistic similarity conditions whose results can be aggregated across the entire document. Note that there can be up to one satisfying clause for each output variable. Due to space limit, we omit some details of the KOKO language. The complete description can be found in [7].

2.1 Surface and hierarchy conditions

To support conditions on the surface text and the dependency tree, variables in a KOKO expression can be bound to *node terms* or *span terms*. *Node terms* refer to the nodes in the dependency tree of the sentence, and *span terms* refer to spans of text.

Node terms: Node terms are defined using XPath-like syntax [8]. A path is defined with the "/" (child) or "//" (descendant) axes and each axis is followed by a label (a parse label, POS tag, token, wildcard (*), or a pre-defined node variable). For example, the expression a = //verb/dobj binds a dobj node that is directly under a verb node. Each label can further be associated with conditions, which are specified in [...], such as a regular expression [@regex = ⟨regex⟩] or [@pos="⟨tag⟩"]. For example, //prep//*[@pos="noun"] denotes a node that is a decedent of the prep node with a noun POS tag. Multiple conditions stated within [...] are separated by ",". For example, [@pos="noun", etype="Date"] states that the POS tag is noun and the entity type is Date.

Span terms: A span term x is constructed with the syntax x = $\langle atom\rangle_1 + \ldots + \langle atom\rangle_k$, where $k \geq 1$ and $\langle atom\rangle_i$ is one of the following: a path expression as described above, a node variable, a sequence of tokens, x.subtree, or an elastic span ^ (which denotes zero or more tokens) or with conditions ^[...]. For ^, KOKO also allows the specification of regular expression over the span or the minimum and/or maximum number of tokens for the span. For example, x= //verb + a + b.subtree + ^[etype="Entity"] defines x to bind to a span that must begin with //verb, followed immediately by the span of a, the span given by b.subtree, and the span that defines an entity in this order. A span term has type Str (String). The output of a Koko query also serializes all entity types into strings.

Given a node x, x.subtree refers to the span that includes all the words in the subtree of x. Given a node x, we also use x to refer to the span that includes only the text of the word x.

Results: The output of a KOKO expression is a bag of tuples. The values in the tuples can be either nodes or spans. The tuples to be returned are defined by the extract clauses (that correspond roughly to the select and where clauses in SQL).

The Extract clause: The extract clause is where variables are defined. The defined variables also need to satisfy certain conditions: typed conditions, hierarchical structural conditions over the dependency tree and/or horizontally over the sequence of tokens. Users can specify constraints among the variables outside a block using the in or eq constructs. For example, "x in y" requires that the tokens of x are among the tokens of y, while "x eq y" requires that the two spans given by x and y are identical.

EXAMPLE 2.1. *The example query below extracts pairs (e,d) where e is an entity type and d is a string type. The variables a, b, c, and d are defined within the* block */ROOT:*{ ...} *where the paths are defined w.r.t. the root of the dependency tree. The variables a, b, and c are node terms while d and e (defined in the first line) are span terms. Outside the block, "(b)* in *(e)" is a constraint between b and e which asserts that the dobj token must be among the tokens that make up entity e.*

extract *e:Entity, d:Str* from *input.txt* if
(/ROOT:{
  a = //verb, b = a/dobj,
  c = b//"delicious", d = (b.subtree)
} *(b)* in *(e))*

For the sentence in Figure 1, there is only one possible set of bindings for the variables in this query: a = "ate", b = "cream", c = "delicious", d = "a chocolate ice cream, which was delicious", *and* e = "chocolate ice cream". *The query returns the pair (e,d).* □

2.2 Similarity and aggregation conditions

Additional constraints on the variables can be specified in the satisfying clause. The satisfying clause also contains conditions to aggregate evidence from different parts of the text. Some of these conditions are boolean and some are approximate and return a confidence value.

Boolean conditions A boolean condition is specified by a regular expression and evaluates to true or false. For example, x "*, a cafe*" requires that x is immediately followed by the string "*, a cafe*", and is a sufficient condition for determining that x is the name of a cafe. Other types of conditions using matches or contains (which we do not elaborate here) are also allowed.

Descriptor conditions A descriptor condition evaluates to a confidence value. There are two types of descriptor conditions. The first is of the form x similarTo ⟨descriptor⟩ which returns how similar x is to ⟨descriptor⟩. The second is of the form x [[*descriptor*]] (or [[*descriptor*]] x) and returns how similar *descriptor* is to the span *after* x (or before x). Note that the distance between x and the terms similar to *descriptor* affects the confidence returned for the similarity. In addition, every condition is associated with a weight between 0 and 1, which specifies how much emphasis to place on the condition when a match occurs. Upon receiving the query, each descriptor is expanded to words semantically close to it using a paraphrase embedding model[1]. For example, by simply specifying the condition (x [["serves coffee"]]), KOKO will automatically expands it to similar phrases such as "*sells espresso*" and "*sells coffee*". Although the expansion is not always perfect, descriptors enable users to be agnostic to linguistic variations to a large extent.[2]

Aggregation For every sentence where the extract clause is satisfied, KOKO will search the text corpus to compute a score for every satisfying clause. If the satisfying clauses are satisfied (i.e., passes the respective threshold), and the excluding clause is not satisfied, then the tuple as specified in the extract clause is returned.

The satisfying clause consists of a set of conditions each with a weight. The ith condition has weight w_i which corresponds to how important the condition is to determining whether the value in question should be extracted. The score of a value e for the variable under consideration is the weighted sum of confidences, computed as follows:

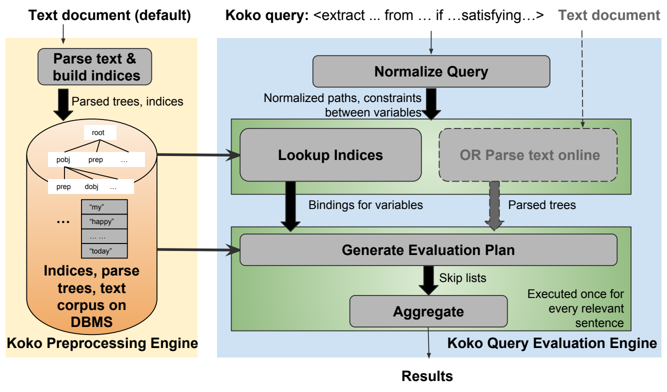
$$\text{score(e)} = w_1 * m_1(e) + \ldots + w_n * m_n(e)$$

where w_i denotes the weight of the ith condition and $m_i(e)$ denotes the degree of confidence for e based on condition i. The confidence for e is computed for each sentence in the text and aggregated together by their sums. By combining evidence from multiple mentions in a coherent article or blog post, KOKO allows users to perform extractions with high confidence. For every variable, if the aggregated score of the satisfying clause for that variable from the collective evidence passes the threshold stated, then the result is returned.

EXAMPLE 2.2. *The query below has an empty* extract *clause but a more complex* satisfying *clause. The intent of the query is to extract cafe names. It considers all entities as candidate cafe names. However, only those that pass the* satisfying x *clause will be returned as answers.*

[1] https://github.com/nmrksic/counter-fitting

[2] One can also supply a dictionary of different types of coffee to KOKO to guide the expansion.



Text document (default)
Koko query: <extract ... from ... if ...satisfying...>
Text document
Parse text & build indices
Parsed trees, indices
root
pobj
prep
prep
dobj
"my"
"happy"
"today"
Indices, parse trees, text corpus on DBMS
Koko Preprocessing Engine
Normalize Query
Normalized paths, constraints between variables
Lookup Indices
OR Parse text online
Bindings for variables
Parsed trees
Generate Evaluation Plan
Skip lists
Executed once for every relevant sentence
Aggregate
Koko Query Evaluation Engine
Results

Figure 2: KOKO **system workflow.**

The first two boolean conditions checks whether the name of the entity contains "Cafe" *or* "Roasters". *It also looks for evidence in input.txt that the name is followed by the string* ", a cafe". *In addition, it will search the text for evidence that the name is followed by a phrase that is similar to* "serves coffee" *or* "employs baristas".

```
extract x:Entity from "input.txt" if ()
satisfying x
        (str(x) contains "Cafe" {1}) or
        (str(x) contains "Roasters" {1}) or
        (x ", a cafe" {1}) or
        (x [["serves coffee"]] {0.5}) or
        (x [["employs baristas"]] {0.5})
with threshold 0.8
excluding (str(x) matches "[Ll]a Marzocco")
```

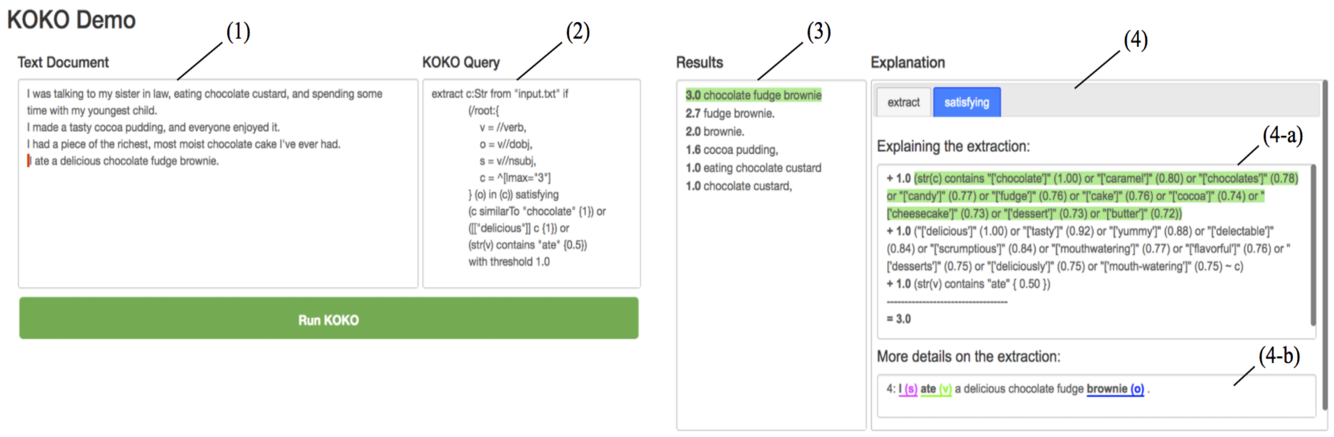
The first 3 conditions each has weight 1, and the last two have weight 0.5 each. When we run the above query over authoritative coffee sites where new cafes are described, it is highly unlikely that we will find exact matches of the phrase "serves coffee", *which is why the descriptor conditions play an important role. The* excluding *condition ensures that* x *does not match the string* "La (or la) Marzocco" *(an espresso machine manufacturer).*

3. THE KOKO SYSTEM

Figure 2 shows the basic workflow of KOKO. The input to a KOKO query is a text document that can either be processed by KOKO Preprocessing Engine or be parsed online. By default, KOKO will invoke its Preprocessing Engine to process the text document by first parsing it through a natural language parser (e.g., spaCy [6] or Google NL API [3]), and then create the indices for its parsed components. The parser transforms the text document into a sequence of sentences, each of which consists of a sequence of *tokens*. Aside from the original text, each token carries a number of annotations, such as the POS tag, parse label, and a reference to the parent in the dependency tree. We refer to a sequence of consecutive tokens in a sentence as a *span*.

With the preprocessed document, the KOKO Query Evaluation Engine evaluates a given query in 4 steps:

1. *Normalize query.* The expressions in the KOKO query are first normalized and conditions among variables are explicitly stated in preparation for subsequent steps.
2. *Decompose paths and lookup indices.* Every normalized components in the query are further decomposed into one or more paths so that each of them can be used to access an index. The results from all accesses to indices are then joined, as needed, to obtain a candidate set of sentences that should be considered next.



Text Document

I was talking to my sister in law, eating chocolate custard, and spending some time with my youngest child.
I made a tasty cocoa pudding, and everyone enjoyed it.
I had a piece of the richest, most moist chocolate cake I've ever had.
I ate a delicious chocolate fudge brownie.

KOKO Query

```
extract c:Str from "input.txt" if
        (/root:{
           v = //verb,
           o = v//dobj,
           s = v//nsubj,
           c = ^[lmax="3"]
        } (o) in (c)) satisfying
        (c similarTo "chocolate" {1}) or
        ([["delicious"]] c {1}) or
        (str(v) contains "ate" {0.5})
        with threshold 1.0
```

Results

3.0 chocolate fudge brownie
2.7 fudge brownie.
2.0 brownie.
1.6 cocoa pudding,
1.0 eating chocolate custard
1.0 chocolate custard,

Explanation

Explaining the extraction:

+ 1.0 (str(c) contains "['chocolate']" (1.00) or "['caramel']" (0.80) or "['chocolates']" (0.78) or "['candy']" (0.77) or "['fudge']" (0.76) or "['cake']" (0.76) or "['cocoa']" (0.74) or "['cheesecake']" (0.73) or "['dessert']" (0.73) or "['butter']" (0.72))
+ 1.0 ("['delicious']" (1.00) or "['tasty']" (0.92) or "['yummy']" (0.88) or "['delectable']" (0.84) or "['scrumptious']" (0.84) or "['mouthwatering']" (0.77) or "['flavorful']" (0.76) or "['desserts']" (0.75) or "['deliciously']" (0.75) or "['mouth-watering']" (0.75) ~ c)
+ 1.0 (str(v) contains "ate" { 0.50 })

= 3.0

More details on the extraction:

4: I (s) ate (v) a delicious chocolate fudge brownie (o) .

Figure 3: Screenshot of KOKO demonstration.

3. *Generate evaluation plan.* This module applies a heuristic on "horizontal" conditions to identify a set of variables whose evaluation can be first skipped. The bindings for the skipped variables are then derived and constraints are checked.
4. *Aggregate.* The conditions of satisfying clauses are evaluated across the text to obtain evidence and the final result is determined.

KOKO is also able to evaluate a query on the given text document without any preprocessing: in the 2nd step, instead of accessing the pre-built indices for relevant sentences, KOKO can also parse the given input text document on the fly and deliver the parsed results to the following step for evaluation.

4. DEMONSTRATION OUTLINE

We will demonstrate KOKO with an emphasis on three aspects of the system which makes it a powerful information extraction tool: *expressiveness*, *scalability* and *explainability*. To this end, we bundle KOKO with three (pre-indexed) text corpora, HappyDB [1], Wikipedia, and coffee blogs we scraped from Barista Magazine and Sprudge.com websites. KOKO has 4 panels (Figure 3) that allow users to specify the input and interactively explore the output. We detail the interactive process as below.

Step 1 (Define Text Input): The first panel ((1) Text document) in Figure 3 shows sentences in the input documents. KOKO allows users to define their own text input through the textbox. Meanwhile, users can also select one of the three text corpora as the text input.

Step 2 (Define KOKO Query): To explore the KOKO system, the second step is to define a KOKO query through panel ((2) KOKO query). Similar to Step 1, users have the flexibility to select one of the pre-defined queries, customize these queries, or define their own queries.

Step 3 (Run KOKO Query and Explore the Results): After the input text and Koko query is defined, the user needs to press "Run Koko", and the results are displayed on the third panel ((3) Results). To explore the results, the user can click the result and interact with the system through the rightmost panel ((4) Explanation). For example, if the user selects a result tuple "chocolate fudge brownie", the sentence from which the phrase "chocolate fudge brownie" is extracted, "I ate a delicious chocolate fudge brownie" in panel (1), will be highlighted as red. Simultaneously, panel ((4) Explanation) will display two tabs for explanation: one for the extract clause and one for each satisfying clause. The tab for the extract clause displays the dependency parse trees of every sentence that contributes to extracting the tuple. The dependency parse tree will be visualized in the same manner as Figure 1. The tree will also depict the bindings of the variables in the extract clause to the nodes of the tree. The tab for satisfying clause is composed by two components: the first component ((4-a) Explaining the extraction) shows the relevant conditions in the satisfying clause that contributed to the outcome. For example, panel (4-a) in Figure 3 shows three conditions and their scores to explain why "chocolate fudge brownie" was extracted. Users can further explore the conditions through panel ((4-b) More details on the extraction), which depicts the (spans of) sentences with the bound variables.

Through the three steps procedure, we demonstrate that KOKO is expressive, scalable, and explainable.

Expressiveness. In Step 2, we prepare several KOKO queries that present various features of KOKO including its ability to (1) query over the surface syntax, (2) KOKO's ability to query over the dependency parse tree and/or the surface syntax, and (3) KOKO's ability to do the above in conjunction with the aggregation of evidence.

Scalability. We showcase the scalability of KOKO by running our queries on the text corpora with different scales (Step 1). In particular, we demonstrate that KOKO is able to query millions of Wikipedia articles in minutes.

Explainability. In conjunction, we demonstrate KOKO's ability to explain results. Specifically, through Step 3, we demonstrate and explain why a result tuple is generated through our interactive user interface.

5. REFERENCES

[1] A. Asai, S. Evensen, B. Golshan, A. Halevy, V. Li, A. Lopatenko, D. Stepanov, Y. Suhara, W.-C. Tan, and Y. Xu. HappyDB: A corpus of 100,000 crowdsourced happy moments. In *Proc. LREC*, 2018 (to appear). https://rit-public.github.io/HappyDB/.
[2] L. Chiticariu, Y. Li, and F. R. Reiss. Rule-based information extraction is dead! long live rule-based information extraction systems! In *Proc. EMNLP*, pages 827–832, 2013.
[3] Google Cloud Natural Language API. https://cloud.google.com/natural-language/,
[4] R. T. McDonald, J. Nivre, Y. Quirmbach-Brundage, Y. Goldberg, D. Das, K. Ganchev, K. B. Hall, S. Petrov, H. Zhang, O. Täckström, C. Bedini, N. B. Castelló, and J. Lee. Universal dependency annotation for multilingual parsing. In *Proc. ACL*, pages 92–97, 2013.
[5] S. Petrov, D. Das, and R. T. McDonald. A universal part-of-speech tagset. In *Proc. LREC*, pages 2089–2096, 2012.
[6] spaCy: Industrial-Strength Natural Language Processing in Python. https://spacy.io/,
[7] X. Wang, A. Feng, B. Golshan, A. Halevy, G. Mihaila, H. Oiwa, and W.-C. Tan. Scalable semantic querying of text. *PVLDB*, 9(11):961–974, 2018.
[8] XML Path Language (XPath). Xml path language. https://www.w3.org/TR/xpath/.