SpannerLib: Embedding Declarative Information Extraction in an Imperative Workflow

Dean Light
Technion
Haifa, Israel
dean.light92@gmail.com

Ahmad Aiashy
Technion
Haifa, Israel
ahmad-ai@campus.technion.ac.il

Mahmoud Diab
Technion
Haifa, Israel
mahmoud.diab@campus.technion.ac.il

Daniel Nachmias
Technion
Haifa, Israel
nach.daniel@gmail.com

Stijn Vansummeren
UHasselt, Data Science Institute
Diepenbeek, Belgium
stijn.vansummeren@uhasselt.be

Benny Kimelfeld
Technion
Haifa, Israel
bennyk@cs.technion.ac.il

ABSTRACT

Document spanners have been proposed as a formal framework for declarative Information Extraction (IE) from text, following IE products from the industry and academia. Over the past decade, the framework has been studied thoroughly in terms of expressive power, complexity, and the ability to naturally combine text analysis with relational querying. This demonstration presents SPANNER-LIB—a library for embedding document spanners in Python code. SPANNERLIB facilitates the development of IE programs by providing an implementation of Spannerlog (Datalog-based document spanners) that interacts with the Python code in two directions: rules can be embedded inside Python, and they can invoke custom Python code (e.g., calls to ML-based NLP models) via user-defined functions. The demonstration scenarios showcase IE programs, with increasing levels of complexity, within Jupyter Notebook.

1 INTRODUCTION

Historically, approaches to Information Extraction (IE) have been divided into two paradigms: *rule-based* IE, such as IBM's SystemT [1], and *ML-based* IE of which Transformers and LLMs are the predominant example. While the latter have shown impressive capabilities, it is increasingly recognized that to reach acceptable accuracy in practical scenarios one has to orchestrate LLM-based extractors

using scenario-specific logic. This logic is often implemented imperatively in languages such as python. In this paper, we present SpannerLib, a rule-based approach to IE that facilitates the orchestration of ML-based IE in a declarative framework.

Document spanners have been studied over the past decade as the theoretical core of industrial systems for Information Extraction (IE) from text [4] such as IBM's SystemT. The framework casts IE as relational querying: generic IE functions (such as regular expressions with capture variables) extract base relations that, in turn, are manipulated by the relational algebra. Thus, the framework allows to utilize the simplicity and popularity of query languages, such SQL, to greatly simplify the development of solutions for text analysis. The logical nature of document spanners does not preclude the potential of utilizing generic solutions for Natural Language Processing (NLP) that are nowadays dominated by statistical and numerical analysis, from basic classifiers to Large Language Models (LLMs). Extensions such as Spannerlog [8] showed how the theoretical framework can be elegantly combined with custom code as special user-defined functions called *IE functions*.

In this demonstration, we showcase SPANNERLIB: a library for integrating document spanners in Python, resulting in an IE development that enjoys the benefits of the *imperative* paradigm (arbitrary Python code, including calling ML models) and the *declarative* one (Datalog), thereby making rule-based IE development accessible to a wider developer audience. We describe three pillars of our system: *(1)* a full-fledged implementation of Spannerlog, *(2)*, an extension of Python with embedded Spannerlog via iPython's Magic system, where the two types of code communicate via Pandas DataFrames, *and (3)* a framework for constructing and registering IE functions to invoke Python code from within Spannerlog rules, which now serve as *callback* functions. Our demonstration scenarios start with basic Spannerlog solutions, then illustrate the power of SPANNERLIB by easily combining Spannerlog and LLMs towards a real task (code generation), and finally analyze an existing SPANNERLIB rewriting of an existing rule-based codebase where the code was simplified drastically (e.g., an order of magnitude fewer code lines).

2 THE SPANNERLOG FRAMEWORK

SPANNERLIB is a library for expressing document spanners. In particular, it provides a concrete implementation of Spannerlog [8], which is a variant of Datalog that operates on strings and spans in

addition to ordinary relational data. In this section, we review the basic concepts of spanners and Spannerlog.

Assume that we are given a set D of *text documents* (i.e., strings) and let Σ be the alphabet of all symbols used in these documents. A *span* is a triple of the form $\langle d, i, j \rangle$ where $d \in D$ and i and j are naturals numbers such that $1 \leq i \leq j \leq |d| + 1$, where $|d|$ is the length of d. The span $\langle d, i, j \rangle$ represents the position of the substring $d_i, \ldots, d_{j-1}$ of d.

By a *schema* we mean here a sequence of types, where each type is either "str" (string) or "span." An *IE function* f is a stateless function that takes a tuple over a fixed schema $in(f)$ and returns a relation over a fixed schema $out(f)$ derived from the input tuple. For example, the standard NLP tool part-of-speech (POS) can be associated with an IE function POS with $in(\text{POS}) = (\text{str})$ and $out(\text{POS}) = (\text{span}, \text{str})$. This IE function takes as input a string and returns the spans of every word over the input string and its part-of-speech tag as a string. IE functions that are widely studied in this context are regular expressions (regex formulas). If α is a regex formula, then rgx_α is a IE function with $in(\text{rgx}_\alpha) = (\text{str})$ and $out(\text{rgx}_\alpha) = (s_1, \ldots, s_n)$, where s_i = span and n is the number of capture groups in the α. Given an input document d, $\text{rgx}_\alpha(d)$ returns all matches of α in d where each match $(\text{span}_1, \ldots, \text{span}_n)$ is ordered by the order of the first appearance of each capture variable in α.

To illustrate, given $\alpha = \text{x}\{a+\}c+y\{b+\}$ and an input string $d = $ acb aacccbbb, $\text{rgx}_\alpha(d)$ will return the tuples $(\langle d, 0, 1 \rangle, \langle d, 2, 3 \rangle)$ and $(\langle d, 4, 6 \rangle, \langle d, 9, 12 \rangle)$ mapping to (a, b) and (aa, bbb), respectively. IE functions can be extended to handle other primitives (e.g., numbers), but we restrict to strings and spans for simplicity.

Assume a relational database schema. A *Datalog* program is collection of *rules* of the form

$$p(x_1, \ldots, x_n) \leftarrow q_1(\tau_{1,1}, \ldots, \tau_{m_1,1}), \ldots, q_k(\tau_{1,k}, \ldots, \tau_{m_k,k})$$

where $p, q_1, \ldots, q_k$ are predicate symbols, each x_i is a variable, and each $\tau_{i,j}$ can be either a variable or a constant. Rules can share the same left hand side predicate and can contain recursion.

Spannerlog [8] is essentially Datalog over relations on strings and spans, extended with *IE clauses* that refer to an execution of an IE function as a means of deriving facts using rules.

More precisely, an *IE predicate* is an expression of the form $f(x_1, \ldots, x_n) \mapsto (y_1, \ldots, y_m)$, where f is an IE function with an n-ary input schema and an m-ary output schema. Semantically, this predicate binds to $y_1, ..., y_m$ all facts that can be derived from an application of f over the values bound to $x_1, ..., x_n$. In this work, we use the $\mapsto$ notation to make the distinction between the input and the output of an IE clause more clear. A Spannerlog program is a set of Spannerlog rules, where a Spannerlog *rule* is defined similarly to a Datalog rule, except that rule bodies can include IE atoms. For illustration, consider the following rule.

$$R(\text{usr}, \text{dom}) \leftarrow \text{Texts}(d, t), \text{rgx}_\alpha(t) \mapsto (\text{usr}, \text{dom})$$

Here, Texts is a binary relation that contains document texts (t) and their dates of publication (d). α is a regex formula that extracts the user and domain of email addresses found in the given text t.

This extension of Datalog rules allows us to elegantly compose multiple IE functions as part of a rule in a readable way. For example, the following rule refers to the IE functions rgx_α and foo. It applies the regex extraction on the document produced by foo.

$$T(z, v, w) \leftarrow \text{Texts}(d, t), \text{foo}(d, t) \mapsto (z), \text{rgx}_\alpha(z) \mapsto (w, v)$$

3 MAIN LIBRARY COMPONENTS

In this section, we introduce the main components of SPANNER-LIB. We first discuss the basic implementation of Spannerlog (Section 3.1), then embedding of Spannerlog in Python (Section 3.2), and then the incorporation of IE (callback) functions in Spannerlog (Section 3.3).

3.1 Spannerlog Implementation

The bedrock of SPANNERLIB is a Python implementation of the Spannerlog language [8] which allows us to programmatically invoke the Spannerlog runtime from Python, and vice versa.

To build our Spannerlog implementation, we extended the *naive bottom up* evaluation method [5] to include evaluation of IE clauses. Since Spannerlog requires a more intricate definition of rule safety, which in turn determines IE function execution order within a rule, we also implemented a semantic safety checker according to the safety definitions in [8].

Additionally, we added aggregation functions to SPANNERLIB, and we will later illustrate their importance. For example, the next rule concatenates all extracted y in the same document x, in lexicographic order.

$$R(t, \text{lex\_concat}(\text{str}(y))) \leftarrow \text{Texts}(d, t), \text{rgx}_\alpha(t) \mapsto (y)$$

Note that the syntax and semantics are similar to previous Datalog formalisms for aggregation (e.g.,[9]).

3.2 Embedding Spannerlog in Python

To be useful in the setting of a Python program, we embedded our Spannerlog engine inside the Python runtime, via IPython's *Magic* system. The Magic system allows us to write and register Spannerlog rules directly inside a Jupyter Notebook Cell.

SPANNERLIB uses the markers %%python and %%log to separate between Python code and Spannerlog rules. The session object facilitates communication between the Python and Spannerlog runtimes. As an example, consider the following code.

```
%%python
from rgxlog import session
df = pd.Dataframe(... columns =['Date','Text'])
session.import(df,name='Texts')
%%log
R(usr,dom) ← Texts(d,t), rgx_alpha(t) ↦ (usr,dom)
%%python
Out = session.export('?R(usr,"gmail")')
```

This code constructs the Texts relation by taking a DataFrame and importing it to our engine, applies the Spannerlog rule to construct the relation R, and then populates the DataFrame Out by querying R (asking for users whose email domain is "gmail").

3.3 Embedding Python in Spannerlog

While Spannerlog is useful as an end-to-end text analysis engine, there are tasks that require calling external functionality, often

provided by machine-learning-based Natural Language Processing (NLP) models. These models are often accessible as software programmed or wrapped in Python. SpannerLib enables the user to empower Spannerlog with user-defined functions and integrate them within as novel callbacks to IE functions. Any stateless Python function can be added as an IE callback via the session object, as long as the function accepts a relation as input and returns a relation (and more generally sequence of relations) as output. Following is an example.

```
%%python
def foo(x:str,y:str):
…
session.register(foo, input=[str,str], output=[span])
%%log
T(z,v,w) ← R(x,y), S('bob',x), foo(x,y) ↦ (z),
    rgx_alpha(z) ↦ (w,v)
%%python
session.export('?T(z,w,"gmail")')
```

4 LIBRARY ILLUSTRATION

We next illustrate the utility of SpannerLib by describing an example of augmenting the context of an LLM as well as a concrete use case of rewriting an existing rule based NLP pipeline using SpannerLib.

4.1 Example: Code Documentation Task

We first illustrate how SpannerLib can help build LLM pipelines by concisely expressing programmer intent and, in particular, avoiding orchestration boilerplate and data management.

Consider the task of developing a code completion program focused on suggesting documentation for the function currently containing the user cursor in a code editor, similar to Github's Copilot.[1] One of the challenges of performing code completions is that the entire code base is too big to fit into the context window. Copilot's approach to solving this issue is to take the last k files that were accessed in the code editor (including the current file where the cursor is pointing to) and feed them all as context to the LLM model. This approach has two main weaknesses: (1) there would mostly be a lot of irrelevant context in some of these files, and (2) there could be very relevant instances of interacting with the function among the files that have not been touched recently.

We will illustrate the simplicity and elegance of implementing a more sophisticated context retrieval approach. We would like to provide the LLM with the following context that consists of two components: the code of the current function, and the code of all functions that call the current function. This task, like most LLM tasks and specifically Code LLM tasks, requires integration of basic LLM core functionalities (like a chat model API), can leverage basic human understanding of the text, and is unique in the orchestration of the specific solution while using very similar tooling as other solutions in the same field. For the illustration, let us assume we have the following relations:

- A binary relation Files(name, content) that holds all file names in the code project and their content as strings;

- A unary relation Cursor(pos) that holds a single span over the files with the position of our cursor.

Moreover, let us also assume some additional IE functions, which can be constructed by very thin wrappers around established libraries: the IE function LLM calls an LLM with a prompt and returns the answer as a string, and the IE function AST takes an XPath pattern and a document and returns all spans of code that fits the AST pattern. For example, the pattern .*.(FuncDecl|ClassDecl) returns all function and class definitions, nested in the AST. We also assume as IE functions some standard operations such as string concatenation, span containment, and a printf-like formatting.

Using these generic primitives, it is quite simple and elegant to convey our pipeline's logic via SpannerLib. We show here only parts of the code for brevity, the full SpannerLib implementation will be presented as part of the demonstration (see Section 5).

```
%%python
def llm(q):
…
session.register(llm, input=[str], output=[str])
def ast(pattern,doc):
…
session.register(ast,input=[str,str],output=[span])
%%log
scope_of(pos,s) ← Files(name,c) ,
    AST('.*.(FuncDecl|ClassDecl)',c) ↦(s) , contains(pos,s)
document(p,a)← scope_of(p,s) , get_name(p,name) ,
    mentions(name,m) , format(prompt,m,c)↦(q) ,
    LLM(q)↦ (a)
%%python
session.export("?document(pos,a)")
```

4.2 Case Study

To study the ability of SpannerLib to simplify text analysis tasks on a bigger scale, we took the source code for an existing rule-based NLP pipeline and rewrote it with SpannerLib. The original pipeline [2] deals with analyzing medical text documents to classify patients based on their COVID-19 risk. The pipeline consists of 4335 lines of Python code and is based on the open-source NLP library spacy [6]. The full rewriting process and the refactored code are available online.[2]

Table 1 shows a summary of the code-line comparison between the implementations. For a fair comparison, we kept library usage as close as possible to the original code. As can be seen, under 10% of the original code was either core computations or interfaces, which remained as Python code (203 lines). Over 90% of the code was either data management or pipeline orchestration (including specific constants supplied to the pipelines in different steps). Hence, we were able to reduce around 4000 lines of code into fewer than 400 lines of either Spannelog rules or tabular data.

5 DEMO SCENARIOS

All demonstration scenarios are interactive Jupyter notebooks that contain step by step walkthroughs of how to design and implement increasingly complex programs using SpannerLib.

Table 1: Code comparison between the original implementation [2] and the rewritten SPANNERLIB implementation.

Code Type	Original Code	SPANNERLIB
Native Python	4335	110
Python IE Functions		93
Spannerlog Code		107
Code As Data (txt/csv files)		286
Total Python	**4335**	**203**
Total Declarative Code		**393**
Total Lines	**4335**	**596**

Basic Task. In this scenario, we showcase the basic components of SPANNERLIB, in particular defining and using IE functions and composing them via Spannerlog code. This scenario will include simple tasks such as finding identical sentences in a corpus of documents and incorporating LLMs like ChatGPT in a question answering pipelines.

End-to-End Task. In this scenario, we show a full working implementation of the code documentation task explained in Section 4.1. We will use the LLM IE function from the previous scenario as well as wrap Python's AST library into an IE function AST. We will then show how we express our context augmentation pipeline's logic as Spannerlog rules and how we query our rules to receive the output of the LLM and export it to Python. We also analyze our code for readability and conciseness. This scenario demonstrates how complex data pipelines can easily be expressed using SPANNERLIB.

Extending SPANNERLIB Code. While good code is clear and concise, it should also make modifications and extensions easy for the programmer. In this scenario we demonstrate the ease with which SPANNERLIB code can be extended. To do so, we build upon the previous code documentation implementation and extend it to include: (1) augmentation of the LLM's context according to data sources that were not available to the LLM during training. (2) User feedback over previous executions of this task. These extensions will use two of the most widely used prompt augmentation techniques used in the industry today, namely Retrieval Augmented Generation (RAG) [7] and Few Shot Prompting [3].

This scenario will not only demonstrate the ease of extension of SPANNERLIB code but will also demonstrate that modern techniques from the field of NLP can be easily expressed and composed via SPANNERLIB to generate sophisticated pipelines that result in a very simple code base.

Real Code Base. In this scenario, we aim to quantify the benefits of integrating SPANNERLIB into a code base from a software engineering perspective. Concretely, we will demonstrate how to refactor an existing code base using SPANNERLIB, by means of the rule-based medical document classification pipeline of Section 4.2. We will show a side-by-side comparison of the original version and the SPANNERLIB rewriting. Using this comparison, we will explain and demonstrate best practices for decomposing existing code into SPANNERLIB, namely: (1) How to identify the scope of and implement IE functions? (2) Which code should turn into declarative code, either as Spannerlog rules and queries or as data? (3) What parts of the code base should be left as external Python code?

We will then analyze the size reduction ratio of different fragments of code (as seen in Table 1) as they are translated into SPANNERLIB code. We will also discuss how both implementations differ in various code quality measures such as conceptual decomposition, separation of concerns, bug surface area, and so on. This scenario will demonstrate that the paradigm introduced by SPANNERLIB is robust and powerful enough to significantly improve development even in real-world code bases.

6 DISCUSSION

While rule-based IE systems have been shown to increase processing performance and have been argued to increase developer productivity [1], they have not seen wide adoption. We believe this stems from a lack of accessible tooling and developer education. To remedy the former, SPANNERLIB (1) embeds itself within the python ecosystem, removing many of the barriers of writing IE functions and (2) leverages an extremely simple declarative language as its backbone. To remedy the latter, we discuss IE based programming from a software engineering perspective in our walkthroughs, using lines of code as a surrogate metric for ease of development. SPANNERLIB does not yet put an emphasis on processing performance, and is therefore not comparable to existing enterprise IE systems in terms of processing efficiency. We are addressing this topic in ongoing work.

REFERENCES

[1] Laura Chiticariu, Marina Danilevsky, Yunyao Li, Frederick Reiss, and Huaiyu Zhu. 2018. SystemT: Declarative Text Understanding for Enterprise. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 3 (Industry Papers)*. Association for Computational Linguistics, New Orleans - Louisiana, 76–83. https://doi.org/10.18653/v1/N18-3010

[2] Alec Chapman et al. 2020. A Natural Language Processing System for National COVID-19 Surveillance in the US Department of Veterans Affairs. In *Proceedings of the 1st Workshop on NLP for COVID-19 at ACL 2020*.

[3] Tom B. Brown et al. 2020. Language Models are Few-Shot Learners. In *Proceedings NIPS '20*. Article 159, 25 pages.

[4] Ronald Fagin, Benny Kimelfeld, Frederick Reiss, and Stijn Vansummeren. 2015. Document Spanners: A Formal Approach to Information Extraction. *J. ACM* 62, 2 (2015), 12.

[5] Todd J. Green, Shan Shan Huang, Boon Thau Loo, and Wenchao Zhou. 2013. Datalog and Recursive Query Processing. *Found. Trends Databases* 5, 2 (2013), 105–195.

[6] Matthew Honnibal, Ines Montani, Sofie Van Landeghem, and Adriane Boyd. 2020. spaCy: Industrial-strength Natural Language Processing in Python. (2020). https://doi.org/10.5281/zenodo.1212303

[7] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. In *Proceedings of NIPS' 20*. Article 793, 16 pages.

[8] Yoav Nahshon, Liat Peterfreund, and Stijn Vansummeren. 2016. Incorporating information extraction in the relational database model *(WebDB)*. Association for Computing Machinery, 1–7.

[9] Alexander Shkapsky, Mohan Yang, and Carlo Zaniolo. 2015. Optimizing recursive queries with monotonic aggregates in DeALS. In *ICDE*. IEEE Computer Society, 867–878.