Proceedings of the 12th Conference on Language Resources and Evaluation (LREC 2020), pages 2183–2191
Marseille, 11–16 May 2020
© European Language Resources Association (ELRA), licensed under CC-BY-NC

Odinson: A Fast Rule-based Information Extraction Framework

Marco A. Valenzuela-Escárcega, Gus Hahn-Powell, Dane Bell
LUM.AI, Tucson, Arizona, USA
{marco,ghp,dane}@lum.ai

Abstract

We present Odinson, a rule-based information extraction framework, which couples a simple yet powerful pattern language that can operate over multiple representations of text, with a runtime system that operates in near real time. In the Odinson query language, a single pattern may combine regular expressions over surface tokens with regular expressions over graphs such as syntactic dependencies. To guarantee the rapid matching of these patterns, our framework indexes most of the necessary information for matching patterns, including directed graphs such as syntactic dependencies, into a custom Lucene index. Indexing minimizes the amount of expensive pattern matching that must take place at runtime. As a result, the runtime system matches a syntax-based graph traversal in 2.8 seconds in a corpus of over 134 million sentences, nearly 150,000 times faster than its predecessor.

Keywords: information extraction, text mining, rule-based methods, information retrieval

1. Introduction

Information extraction (IE) forms the backbone of many important real-world applications, such as identifying fragments of protein signaling pathways from biomedical publications (Kim et al., 2013) and assembling complex models of real-world systems (Sharp et al., 2019). Efficient methods for scalable retrieval and extraction enable new forms of linguistic analysis leveraging large text corpora (Biber, 2009). At a high level, IE approaches can be grouped into two categories: (a) rule-based methods and (b) machine learning. Chiticariu et al. (2013) found that, while machine learning (ML) approaches have garnered great attention in academia, the opposite is true in industry, where rule-based approaches are preferred because of their interpretability and maintainability (Sculley et al., 2014). Our work focuses on the latter class of approaches.

Rule-based approaches make use of patterns that match over a variety of representations, from surface (de Does et al., 2017) to more complex syntactic and semantic structures (Chang and Manning, 2014). While matching patterns over structures like syntax can add resilience to linguistic variation, doing it naively can be very expensive. We argue that this is an important limitation for several reasons.

First, slow runtimes limit the applicability for large document collections. For example, PubMed[1], a repository of biomedical publications, has indexed more than one million publications each year, for the past seven years. Sequentially analyzing just one year's worth of these publications with an IE system that takes one minute per paper would require nearly two years. Second, it would be ideal for these systems to be interactive, especially in the rule development stage, and previous research has shown that a slowdown of even a few hundred milliseconds affects the way users interact with a system (Brutlag, 2009; Liu and Heer, 2014). Finally, efficiency is critical when incorporating IE systems as components of larger, iterative systems, as is done with active learning (Settles, 2009). In these complex systems, any speedups (or slowdowns) are magnified.

Here we propose an IE framework which couples a simple yet powerful pattern language that can operate over multiple representations of text in under 3 seconds over 134 million sentences.

Our work makes the following contributions:

1. We present *Odinson*, a rule-based IE framework with a runtime system that is five orders of magnitude faster than its predecessor, Odin (Valenzuela-Escárcega et al., 2016), and other IE systems. We achieve this speedup by indexing most of the information necessary for matching patterns, including directed graphs such as syntactic dependencies, into a custom Lucene[2] index. Indexing minimizes the amount of expensive pattern matching that must take place at runtime.

2. We introduce a novel pattern language that is inspired by Odin's language (Valenzuela-Escárcega et al., 2016). However, unlike Odin, Odinson's pattern language allows the mixture of multiple representations in the *same pattern*. For example, a pattern in this language may start as a regular expression over surface tokens and continue with a regular expression over syntactic dependencies. This provides additional flexibility in the way patterns can be written, especially useful in domains where syntactic parsers are unavailable or unreliable.

3. We release the entire IE framework as open-source software, under the Apache 2.0 license at https://github.com/lum-ai/odinson. In addition to the core software, an API and user interface are provided.

2. Related Work

Our work builds upon the vast amount of work on rule-based IE in the literature, which has resulted in many software frameworks designed for this purpose. However, most of these approaches have been developed to operate over a single representation of text, e.g., surface forms (Cunningham et al., 2002; Chang and Manning, 2014; Kluegl

[1] https://www.ncbi.nlm.nih.gov/pubmed

[2] https://lucene.apache.org

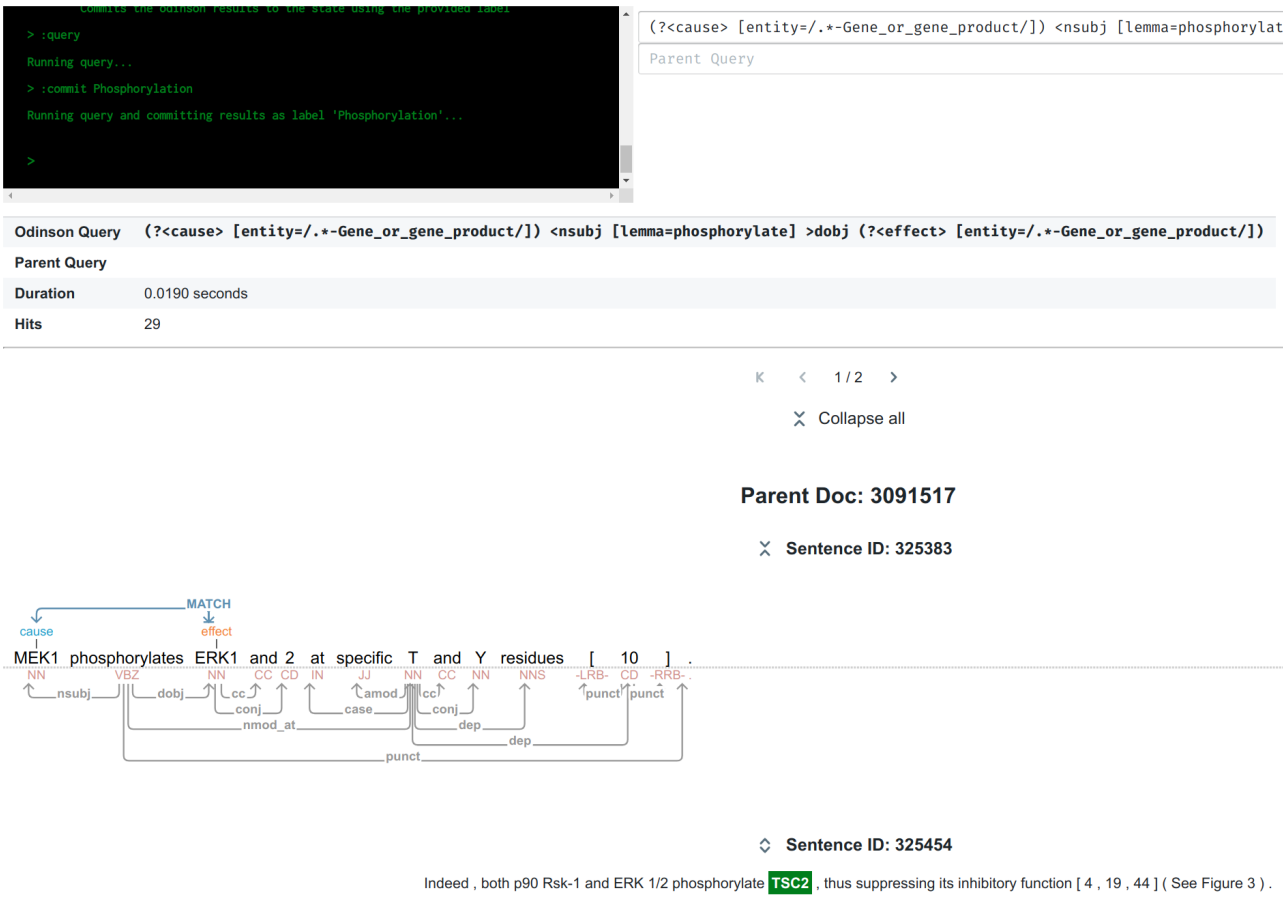


Figure 1: The browser-based user interface for Odinson provides for direct and rapid querying with rich visualization (Forbes et al., 2018) to support interactive exploration of large corpora.

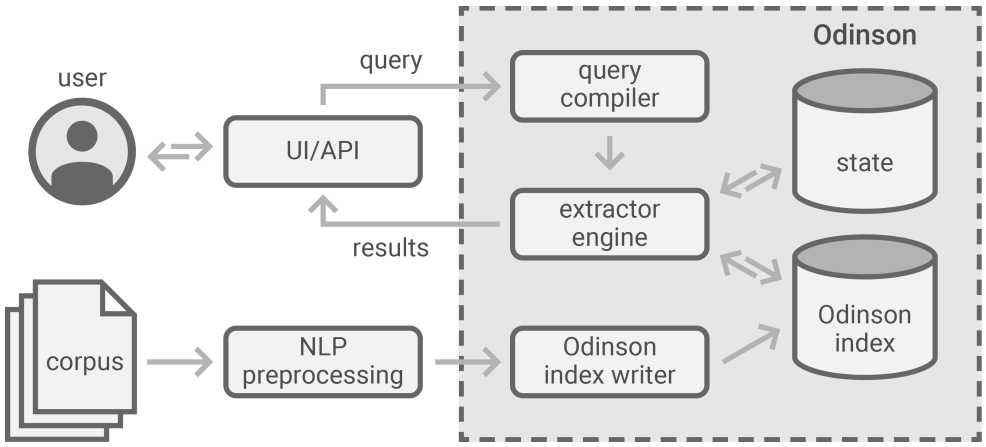


Figure 2: Architecture of our approach. Components that are a part of Odinson are contained inside the gray box. Odinson's index writer generates the inverted index. The extractor engine recieves queries from a user, either from the user interface or directly through the API, after they have been compiled by the query compiler. The queries are executed against the index and results are returned to the user and/or stored in a state for later use.

et al., 2016), or syntax (Levy and Andrew, 2006; Chambers et al., 2007), but not both. This limitation was addressed in Odin (Valenzuela-Escárcega et al., 2016), whose declarative rule language is capable of describing patterns over either surface or syntactic information in the same grammar. This flexibility was demonstrated to be useful in the biomedical domain, where syntactic parsers are brittle. Specifically, a state-of-the-art Odin grammar for this domain had to backoff to surface information for 38% of its patterns (Valenzuela-Escárcega et al., 2018). Here we extend this expressivity, allowing for both surface and syntactic information in the *same pattern*, i.e., a single Odinson pattern can include a regular expression over both surface tokens and dependency syntax.

Most existing systems function by attempting to match *all* patterns in a given grammar over *all* sentences in a given corpus. Odinson avoids this heavy computational overhead by indexing as much information as possible, including lexical and syntactic information, such that most sentences can be rapidly filtered out at runtime before more expensive pattern matching takes place. We show that this yields a speedup of four orders of magnitude over Odin.

3. Approach

We introduce Odinson, a novel rule-based information extraction (IE) framework, which efficiently indexes syntactic information to rapidly execute patterns over a large corpus. At a high level, the intuition is that the vast majority of sentences in a document collection *won't match* a given query, and so can be "discarded" without us needing to actually traverse the dependency graph. We use a custom inverted index to do this sentence elimination, and only the sentences that are likely to contain a match will be checked for the full pattern. The architecture of this approach is shown in Figure 2, and summarized below. We provide implementation details in the next section.

First, the Odinson index writer indexes a document collection which has been syntactically preprocessed[3]. Odinson defines a simple document format for data ingestion, allowing users to prepare their document collection in JavaScript Object Notation (JSON) using their own custom preprocessing pipeline[4] for sentence segmentation, tokenization, sequence tagging, and parsing. Each annotated sentence in the document is indexed individually, along with any desired metadata for the document itself to allow for filtering results by task-relevant criteria, e.g., author, date, geospatial context (detailed in Section 4.). The syntax graphs for the sentences are serialized and stored in the inverted index so that they are available at query time.

At runtime, a user provides a query to Odinson using the declarative rule syntax (Section 5.). The user's query is compiled into (a) an *index query* (a token-level pattern applied to the inverted index), and (b) a *graph traversal pattern* (a full description of the path which must exist in the dependency graph). The Odinson engine first tries to inexpensively match the index query against the inverted index. If this query succeeds for a sentence, then the corresponding syntax graph is deserialized and matched against the graph traversal pattern (Section 6.). This two-step matching process greatly improves runtime efficiency by avoiding unnecessary graph deserialization and traversal. Results are returned to the Odinson engine, and can be persisted to a state (i.e., a database with previous results, see Section 6.1.) and/or returned to the user.

4. Odinson Index Writer

The first step in the Odinson pipeline is the generation of the Odinson index. The index writer is responsible for creating the inverted index, which contains the full preprocessing for each sentence and document in the corpus and which enables the rapid execution of queries at runtime. Specifically, we index three types of information:

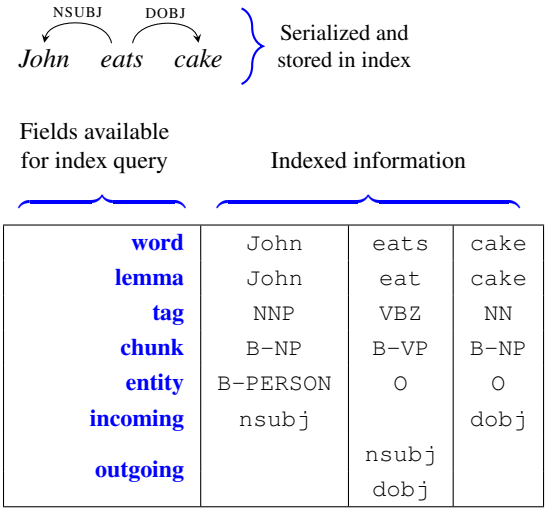


Figure 3: Example of indexed fields corresponding to a sentence. All fields are available for querying using the Odinson query language. Odinson also indexes other fields that we have omitted for brevity (e.g., a unicode-aware normalized version of the token). Note that both `nsubj` and `dobj` in the `outgoing` field are indexed in the position corresponding to *eats*, allowing either to be matched.

(a) The **token-level annotations** for each sentence are indexed as an independent Lucene document. To allow for query expressivity, the sentence-document fields are aligned, such that for each token position, Odinson has access to all of the available annotations for that token within a single query (shown in Figure 3).

Odinson was designed to have a flexible index schema, and so few fields are mandatory. In particular, the only mandatory field is the "raw" field, which encodes unmodified tokens. Typically, another field, called "word", is included. When the raw token is a symbol (e.g., the Greek letter β), this field will contain a text version of the token (here, "beta").

At indexing time, we automatically generate a field called "norm" that contains a normalized version of the tokens, designed to accommodate the fact that the same sequence of characters can be represented as different unicode sequences. We use the ICU library[5], and specifically the NFKC normalization[6] to enable matching tokens *independent of the actual unicode sequence*. We also perform Unicode-aware case-folding so that comparisons to the "norm" field are case insensitive. Furthermore, Odinson uses Lucene's `PositionIncrementAttribute` to index tokens from the "raw" and "word" fields in "norm" as synonyms in the same position, such that a user can query with any variant, e.g., β, beta, BETA, or BeTa, and receive the same results.

[3]While here we use examples which are *syntactically* processed, Odinson's modular design would allow a user to match patterns over any directed acyclic graph.

[4]In our experiments, we used v7.4.4 of `processors` (`https://git.io/Jv6PC`) for document preprocessing.

[5]`http://site.icu-project.org/home`

[6]`https://unicode.org/reports/tr15/`

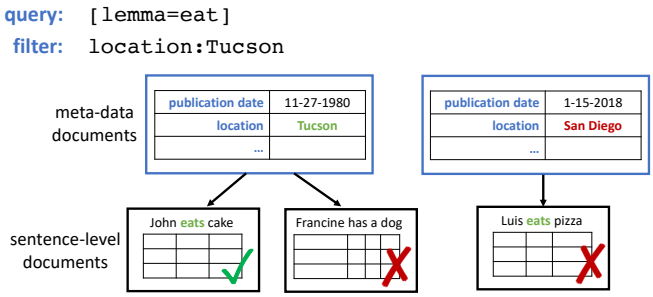


Figure 4: User-specified document metadata is indexed as a parent document of the sentence documents, enabling efficient filtering at query time. In the example, two sentences match the query, but only one of them satisfies the metadata filter.

To make use of syntax, two additional token-level fields are required: "incoming" and "outgoing". These fields encode what incoming and outgoing dependencies correspond to each token, e.g., if a given token is a head or dependent of some syntactic dependency. While these fields can be used directly by the user, they are critical for internal optimizations. For example, with the pattern `[lemma=city] >nmod_such_as []`, we reduce the number of dependency graphs that we actually have to check, i.e., only sentences whose dependency graph has `nmod_such_as` are considered, and furthermore, only sentences containing the lemma "city" and for which that specific word is the head of a `nmod_such_as` dependency relation.

Other fields are helpful to index but not required, such as lemma, chunk, entity, and part-of-speech tags.

(b) The **serialized syntax graph** is also stored in the index as a `DocValue`, for which a document-to-value mapping is built at index-time, allowing us to retrieve the dependency graph corresponding to a given sentence efficiently. The syntax graph is only retrieved and expanded when the sentence could potentially match the user's query, based on the token-level annotations, i.e., as determined with the "incoming" and "outgoing" fields described above.

Furthermore, to optimize the deserialization of the dependency graphs, we use a custom serialization specific to our data structure, which was benchmarked as being approximately 20 times faster at serialization than the built-in Java serializer on the same data structure, but more critically for runtime efficiency it deserializes more than 100 times faster.

(c) The **document metadata** is indexed in a nested manner, such that the metadata serves as a parent document, and its corresponding sentence-level documents are its children. This is illustrated in Figure 4. By using Lucene's index-time join capabilities, these filters are applied efficiently at query time. Odinson does not restrict metadata types, rather they are user-defined. If a filter is used, then Odinson returns *only* the matches that also satisfy the metadata filter.

5. Odinson Query Language

The declarative Odinson query syntax is based on that of Odin (Valenzuela-Escárcega et al., 2016), which includes token constraints, syntax queries, named captures, both greedy and lazy quantifiers, optional and required event arguments, and look-around assertions. Token constraints are boolean expressions over various attributes of a token (e.g., `[word=pretzel & tag=Noun]`). Syntax queries describe traversals over the syntax graph. For example, the query `John <nsubj eats >dobj []` would find *John*, traverse an incoming `nsubj` edge to *eats*, then traverse an outgoing `dobj` edge to any other token. By surrounding part of a query with `(?<argument_name>...)`, the tokens can be captured as named arguments. For example, modifying the previous query to `(?<agent> John) <nsubj eats >dobj []` would capture *John* as the `agent`.

In an important departure from the Odin language, Odinson also supports multiple representations in the same pattern. Consider this contextualized causal pattern: `(?<cause> []) leads to (?<effect> []) >nmod_during (?<context> [])`. Here, in the same pattern the user can specify sequences of lexical elements (i.e., `leads to`) along with syntactic graph traversals (i.e., `>nmod_during`).

5.1. Event Queries

Odinson also supports event queries, which make it easier to specify complex patterns involving many entities interacting through a trigger. In particular, each event query is a *structured* query that contain several sub-queries: one for the event *trigger*, and one for each of the required *arguments*. Additionally, the structure of the query provides opportunities for further optimization. If all these queries match, only then is the graph deserialized and checked to make sure that the required arguments are connected to the trigger in the specified way. Event queries can also include arguments that are not required to match, but which are included in the result if they do (i.e., *optional arguments*).

For example, consider this event query shown in Rule 1:

```
1 trigger = [lemma=eat]
2 theme:Food = >dobj
3 instrument:Tool+ = >nmod_with >conj?
```

Rule 1: Odinson event pattern that extracts a relation with two types of arguments: 1) a `theme` and 2) one or more `instruments` (indicated by the + quantifier following `Tool`). Unlike Rule 2, this formulation relies on `Food` and `Tool` mentions having already been found.

This pattern extracts an eating event, which has two types of arguments, a `theme` and one or more `instruments` (indicated by the +). When applied to the sentence *Audrey ate ramen with chopsticks and a spoon*, Odinson will extract an event with one theme and two instruments, as shown in Figure 5.

In the rule above, all of the referenced mentions (i.e., `Food` and `Tool`) must have been found in advance. However, for several use cases it is desirable to match patterns in a *top-down* manner, i.e., first identifying events of interest and

then creating relevant entities from the event arguments at runtime, rather than the bottom-up approach of first finding entities and then events that operate over them (Sharp et al., 2019). For this reason, in Odinson, we additionally support argument *promotion*, another important departure from Odin. That is, in an event with argument promotion, we try to match the trigger, follow the dependencies for the arguments, and then match the compatible resulting nodes *even if they are not pre-existing mentions*.
We do this by writing the previous event rule this way:

```
trigger = [lemma=eat]
theme:^Food = >dobj
  [chunk=B-NP][chunk=I-NP]*
instrument:^Tool+ = >nmod_with >conj?
  [chunk=/[BI]-NP/][chunk=I-NP]*
```

Rule 2: Alternative formulation of Rule 1 that allows `Food` and `Tool` mentions to be capture on-the-fly through use of the ^ symbol preceding argument mention labels. In this pattern, token sequences corresponding to NP chunks found at the end of either dependency path traversal will be assigned the argument labels `Food` and `Tool` respectively.

The ^ character in the argument labels tells Odinson to create the `Food` and `Tool` mentions if they don't already exist, rather than fail to match. This rule accepts any noun phrases at the end of the dependency paths and uses them as the `theme` and `instruments`. If the chunk pattern were omitted (i.e., `theme:^Food = >dobj`), then by default Odinson matches the single token at the end of the path.

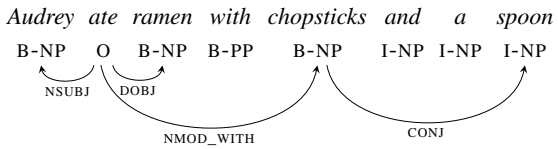
5.2. Compiler Modularization

While Odinson already provides a declarative language readily available for end users, the query compiler was designed to decouple the rule language from the specific low-level operations that need to be performed on the index through the use of an intermediate representation. This allows different languages to target Odinson's intermediate representations and take advantage of the compiler to generate highly efficient queries that can be used in the Odinson framework. Further, our compiler performs several optimizations on the query to greatly improve its efficiency.[7]

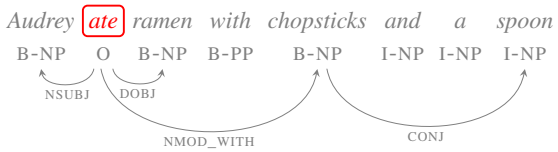
6. Query Compilation and Execution

Once a user has provided a query in our declarative language, Odinson compiles and executes it. For runtime optimization, Odinson aims to index as much as possible, but not everything can be indexed. For this reason, we use a multi-step, tiered execution of queries. Queries which do not involve syntax are compiled into an *index query* and matched against the inverted index only (recall Figure 3). Queries which do involve syntax are compiled into both an index query as well as a *graph traversal pattern*. However, the latter is only executed if the former matches. To help explain these representations, as well as the tiered query execution, consider the query and three sentences shown in Figure 6 alongside the detailed explanations below:

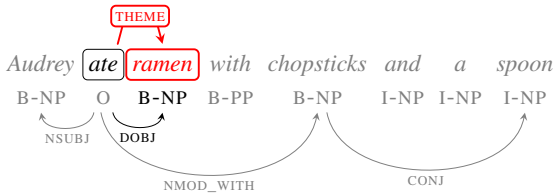
[7]Currently we focus on peephole optimizations, but we intend to extend these in the future.



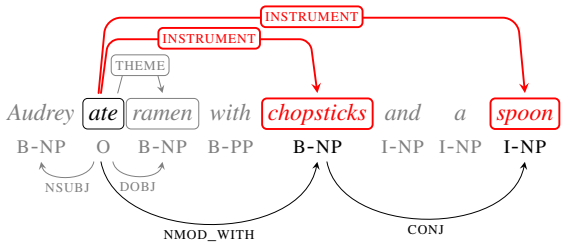
(a) A sentence preannotated with chunks and dependencies.



(b) A trigger is found.



(c) A `theme` argument is matched.



(d) Two `instrument` arguments are matched.

Figure 5: Example sentence illustrating the results of applying the Rules 1 & 2 given in Section 5.1. (a) The Odinson index contains many token and relation annotations as a basis for capture patterns. (b) The event trigger is a token pattern that, if matched, becomes the anchor for the argument patterns. The trigger rule matches here because the lemma of *ate* is *eat*. (c) The system matches the required argument `theme`, which asserts that there must be a direct object dependency link between the trigger and a noun phrase (here, as found by a chunker and encoded in the BIO notation). Both the trigger and the required arguments must match in order to extract the event as a whole. Optional arguments (marked with `?`) do not need to match, but if they are found they will be included as labeled arguments with the extracted event mention. (d) From the trigger, a `nmod_with` dependency leads to *chopsticks* with a `B-NP` chunk annotation, making a match. Because of the `+` marker on the `instrument` line, the rule also matches an additional `instrument` argument by continuing through the optional `conj` dependency.

Index query: For the user query shown in Figure 6, `pretzels <nsubj >xcomp thirsty`, the compiler builds an index query by collapsing the beginning of the graph traversal (i.e., `<nsubj`) to the adjacent word (*pretzels*), where the token field (i.e., `incoming` or `outgoing`) is chosen based on the direction of the dependency. This results in `[word=pretzels & incoming=nsubj]`.

Similarly, the last step of the graph traversal is collapsed, here resulting in a second index query, `[word=thirsty & incoming=xcomp]`.

Graph traversal pattern: For queries requiring syntax, the compiler also creates a graph traversal pattern – an automaton representation of the graph traversal, where the initial state is determined by the already known start position (i.e., the index query corresponding to the beginning of the path) and the valid terminal states are determined by the index query corresponding to the end of the path. The automaton will try (in parallel) all paths described by the pattern that are possible in the sentence. The fact that it can be in many positions (in the sentence) at once eliminates the need for backtracking. In order for the graph traversal to match, it needs to execute the entire pattern and end in a valid terminal state.

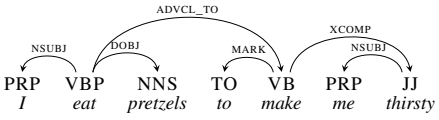
User query

```
pretzels <nsubj >xcomp thirsty
```

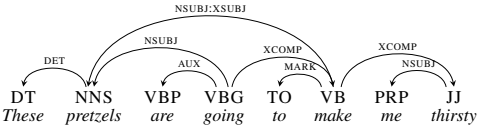
Compiled index query

```
[word=pretzels & incoming=nsubj]
[word=thirsty & incoming=xcomp]
```

(a) Example where the index query fails, so the sentence is skipped.



(b) Example where index query succeeds, but graph traversal fails.



(c) Example where both the index query and graph traversal succeed.

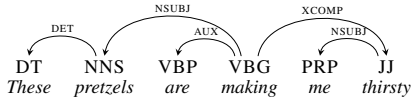


Figure 6: Example showing the compilation and execution of a user query involving syntax. If the index query succeeds (here, there is the word *pretzels* with an incoming `nsubj` relation and the word *thirsty* with an incoming `xcomp`), the full graph traversal pattern is checked to determine whether or not the user query does match the sentence. Here that is, starting at *prezels*, follow an incoming `nsubj`, then follow an outgoing `xcomp`, and land on the word *thirsty*. Full details are provided in Section 6..

Query execution: At execution, sentences that do not match all components of the index query can be discarded without inspecting their corresponding dependency graphs, as shown in Figure 6. This filtering step considerably reduces the runtime, and could be exploited even futher (e.g., by indexing dependency path n-grams). For the sentences which survive the index query, we deserialize the graph and perform the full traversal with the automaton.

In the example in Figure 6, the first sentence is not a match for the index query (because *pretzels* doesn't have an incoming `nsubj`), so it is discarded. The second sentence *does* match the index query, so the full traversal is performed. However, because there is an intervening `xcomp` relation, the traversal fails. In the third sentence, both the surface pattern and the graph traversal succeed and the result is returned to the user (and/or stored in the state as a labeled mention, see Section 6.1.).

6.1. Odinson Mentions and State

When a pattern successfully matches, the results are returned as a mention object. The simplest form of a mention consists of a contiguous span of tokens and optionally a label. When the result has named captures, these are also stored in the mention as arguments, each consisting of a name and a continuous span of tokens. Event mentions have a label, a trigger mention, and the found labeled argument mentions, which could be event mentions themselves, nesting arbitrarily deep.

The found mentions can be stored in a state so that they can be matched by subsequent patterns. For example, a pattern such as `John <nsubj eats >dobj @Food` tries to match against previously found (and stored) mentions of `Food`.

While there are many possible implementations of a state, currently the state is an SQL database that can either exist for a current session only (i.e., kept in memory), or it can be persisted to disk to be re-used in future sessions. Columns corresponding to the document id, the Lucene segment offset, and the mention label are indexed within the database to optimize retrieval time. We intend to explore other options for the state, to facilitate a wider variety of use cases, in the future.

7. Analysis

To evaluate our approach we compare the overall speed difference between Odin and Odinson.

We used Odin and Odinson to execute six queries (see Table 1) on the UMBC[8] WebBase corpus (Han and Finin, 2013; Han et al., 2013), a high-quality subset of WebBase (Hirai et al., 2000). The corpus contains approximately 3 billion words and 134 million sentences over 162,399 documents.

Because of the similarity between Odin and Odinson, it was possible to construct equivalent queries. An exception to this is the *eat* and *consult* queries, which use the argument promotion feature of Odinson (see Section 5.1.) for which there is no direct equivalent in Odin.

We executed each Odin query 5 times (due to time constraints) and each Odinson query 40 times and measured the time elapsed during extraction in each system. The

[8]University of Maryland, Baltimore County

Name	*Pattern*	*Example*	*Results*
president	(?<person> [entity=PERSON]+) [lemma=be] the? president of the? (?<organization> [entity=ORGANIZATION]+)	*Redesdale* was President of the *Royal Photographic Society*...	1,837
born	(?<person> [entity=PERSON]) </nsubj(pass)?/ born >/nmod_.*/* >nmod_in (?<year> /[0-9]{4}/)	*Pavel Andreievich Chekov* was born in *2241*...	4,443
system	[lemma=be] <cop [lemma=system] >nsubj (?<systemName> [tag=/NN.*/])	The *peerage* is a legal system of largely hereditary titles...	40,613
title	(?<person> [entity=PERSON]) <nsubjpass called >xcomp (?<title> [tag=/NNP.*/)	*Gioachino* was called "*Keno*" for short.	780
eat	trigger = [lemma=eat] theme:^Food = >dobj [chunk=B-NP][chunk=I-NP]* instrument:^Tool? = >nmod_with [chunk=B-NP][chunk=I-NP]* location:^Place? = >nmod_at [chunk=B-NP][chunk=I-NP]*	*Birds*, *carnivores* and *rodents* eat dropped *antlers*.	124,645
consult	trigger = [lemma=consult & tag=/VB[DZPGN]/] actor1:^Entity+ = <acl? >/nsubj(pass)?/ [entity=/PERSON|ORGANIZATION/]+ actor2:^Entity+ = <acl? (>nmod_with|>dobj)? >dep? [entity=/PERSON|ORGANIZATION/]+	*TSA* is consulting with the *National Maritime Security Advisory Committee*	601

Table 1: These queries were used to evaluate Odinson's speed. These include a token query (*president*), dependency queries (*born*, *system*, and *title*), and mixed queries with promoted entities (*eat* and *consult*).

maximum heap size was set to 1.5 GB for both systems, constraining them to modest parameters. Results are summarized in Table 2. Odinson is roughly *150,000 times faster* than Odin on a dataset of 134 million sentences. Moreover, Odinson loaded its index approximately 300,000 times faster than Odin loaded its serialized documents.

Recall from Section 4. that Odinson has one extra step that Odin doesn't, which is the indexing of the corpus. This indexing step is performed only once, and it only includes *static* information (e.g., POS tags, dependency graphs, etc). The query compositionality is handled at runtime, when the query is provided by the user, and therefore in Table 2, we provide the benchmark results for the online querying only, as this represents the real-world usage.

In order to measure the effect of corpus size on loading and extraction speed, we queried random subsamples of the overall corpus using the *born* query. A comparison of load and extraction times for each system is shown in Figure 7. The results indicate that the speed gains of Odinson are present at all corpus sizes and increase proportional to corpus size.

In the interest of reproducibility, we have made the code and data used for our benchmarks publicly available[9], so that the analysis can be extended to other rule-based information extraction systems such as KOKO (Wang et al., 2018). For this work, since we found that many of the language features described in Wang et al. (2018) are not

[9] https://github.com/lum-ai/ie-benchmarks

	Load time (s)		*Extract time (s)*	
Query	*Odin*	*Odinson*	*Odin*	*Odinson*
president	6.35×10^5	**1.50**	1.81×10^5	**1.93**
born	3.70×10^5	**1.53**	8.80×10^5	**2.56**
system	3.58×10^5	**1.20**	9.29×10^5	**4.58**
title	4.34×10^5	**1.50**	6.59×10^5	**1.85**
eat	N.A.	**1.57**	N.A.	**12.3**
consult	N.A.	**1.29**	N.A.	**2.68**
Mean	4.27×10^5	**1.42**	4.32×10^5	**2.83**

Table 2: The harmonic means of the loading and extraction times of each system. Load time is the time required to read in the corpus with its annotations. Annotations included static information such as POS tags and grammatical dependencies. In Odin's case, this is done from JSON format, but Odinson is loaded from a precompiled index. The Odinson index loading times are not averaged, since they were loaded only once per query.

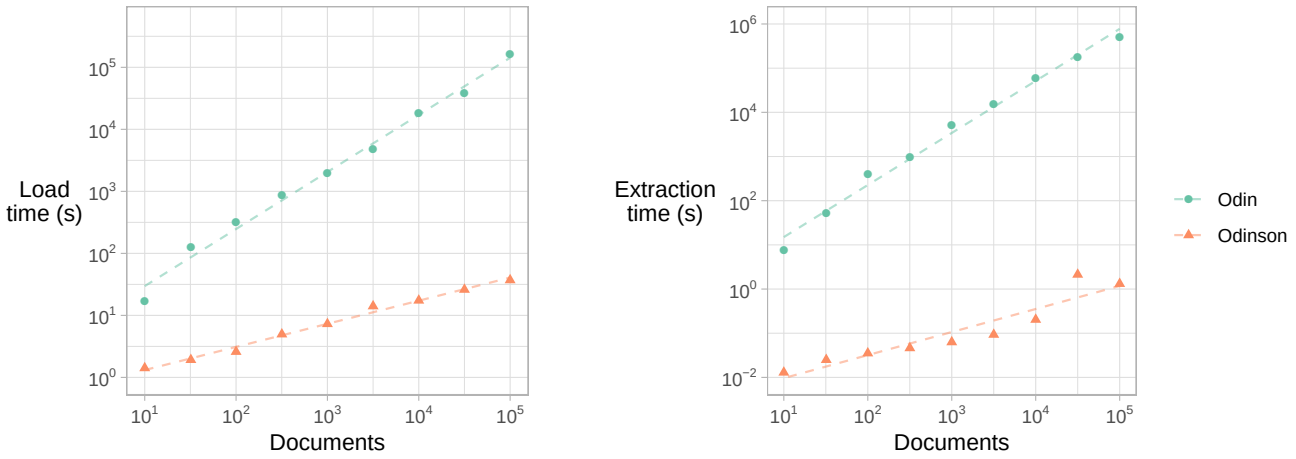


Figure 7: A comparison of load and extraction times for Odin and Odinson across different corpus sizes. Although both systems scale linearly, Odinson's optimizations allow for querying four orders of magnitude more documents in only two orders of magnitude greater time.

included in their public release, a direct comparison of syntactic patterns was impossible. However, Wang et al. (2018) report KOKO is 40 times faster than Odin. While we could not replicate their results, our work demonstrates that Odinson is 150,000 times faster than Odin, and thus faster than KOKO.

8. Conclusions

Here we introduce Odinson, a rule-based information extraction framework which is based on Odin but *highly optimized* to provide a runtime fast enough to be *interactive*, even when used at scale. This rapid runtime is critical when users are handling big data or developing rules. Odinson makes use of an inverted index for sentences and document metadata, indexing as much surface and syntactic information as possible to improve runtime. The dependency graph for each sentence is also serialized and stored in the index. At runtime, user queries (which can mix multiple representations of language) are compiled into index queries and graph traversal patterns. Only if the index queries succeed for a given sentence is its dependency graph deserialized and checked. This provides a rich filter that prevents unnecessary deserialization and traversal. As a result of all of these optimizations, Odinson is approximately 150,000 times faster than Odin.

9. Bibliographical References

Biber, D. (2009). *Corpus-Based and Corpus-driven Analyses of Language Variation and Use*. Oxford University Press, December.

Brutlag, J. (2009). Speed matters for google web search.

Chambers, N., Cer, D., Grenager, T., Hall, D., Kiddon, C., MacCartney, B., de Marneffe, M.-C., Ramage, D., Yeh, E., and Manning, C. D. (2007). Learning alignments and leveraging natural logic. In *Proceedings of the ACL-PASCAL Workshop on Textual Entailment and Paraphrasing*, pages 165–170, Prague, June. Association for Computational Linguistics.

Chang, A. X. and Manning, C. D. (2014). TokensRegex: Defining cascaded regular expressions over tokens. Technical Report CSTR 2014-02, Computer Science, Stanford.

Chiticariu, L., Li, Y., and Reiss, F. R. (2013). Rule-based information extraction is dead! Long live rule-based information extraction systems! In *Proceedings of the 2013 conference on empirical methods in natural language processing*, pages 827–832.

Cunningham, H., Maynard, D., Bontcheva, K., and Tablan, V. (2002). Gate: an architecture for development of robust hlt applications. In *Proceedings of 40th Annual Meeting of the Association for Computational Linguistics*, pages 168–175. Association for Computational Linguistics, July.

de Does, J., Niestadt, J., and Depuydt, K. (2017). Creating research environments with blacklab. In *CLARIN in the Low Countries*, chapter 20, pages 245–258. Ubiquity Press.

Forbes, A. G., Lee, K., Hahn-Powell, G., Valenzuela-Escárcega, M. A., and Surdeanu, M. (2018). Text annotation graphs: Annotating complex natural language phenomena. In *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC-2018)*. European Language Resources Association (ELRA).

Han, L. and Finin, T. (2013). Umbc webbase corpus.

Han, L., Kashyap, A. L., Finin, T., Mayfield, J., and Weese, J. (2013). UMBC_EBIQUITY-CORE: Semantic textual similarity systems. In *Proceedings of the Second Joint Conference on Lexical and Computational Semantics*. Association for Computational Linguistics, June.

Hirai, J., Raghavan, S., Garcia-Molina, H., and Paepcke, A. (2000). Webbase: A repository of web pages. *Computer Networks*, 33(1-6):277–293.

Kim, J.-D., Wang, Y., and Yasunori, Y. (2013). The Genia event extraction shared task, 2013 edition – Overview. In *Proceedings of the BioNLP Shared Task 2013 Workshop*, pages 8–15. Association for Computational Linguistics.

Kluegl, P., Toepfer, M., Beck, P.-D., Fette, G., and Puppe, F. (2016). Uima ruta: Rapid development of rule-based information extraction applications. *Natural Language Engineering*, 22:1–40, 1.

Levy, R. and Andrew, G. (2006). Tregex and tsurgeon: tools for querying and manipulating tree data structures. In *Proceedings of the Fifth International Conference on Language Resources and Evaluation (LREC'06)*.

Liu, Z. and Heer, J. (2014). The effects of interactive latency on exploratory visual analysis. *IEEE transactions on visualization and computer graphics*, 20(12):2122–2131.

Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., Chaudhary, V., and Young, M. (2014). Machine learning: The high interest credit card of technical debt. In *SE4ML: Software Engineering for Machine Learning (NIPS 2014 Workshop)*.

Settles, B. (2009). Active learning literature survey. Technical report, University of Wisconsin-Madison Department of Computer Sciences.

Sharp, R., Pyarelal, A., Gyori, B. M., Alcock, K., Laparra, E., Valenzuela-Escárcega, M. A., Nagesh, A., Yadav, V., Bachman, J. A., Tang, Z., Lent, H., Luo, F., Paul, M., Bethard, S., Barnard, K., Morrison, C., and Surdeanu, M. (2019). Eidos, INDRA, & Delphi: From free text to executable causal models. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics - Human Language Technologies (NAACL HLT): Software Demonstrations*. Association for Computational Linguistics.

Valenzuela-Escárcega, M. A., Hahn-Powell, G., and Surdeanu, M. (2016). Odin's runes: A rule language for information extraction. In Nicoletta Calzolari (Conference Chair), et al., editors, *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC 2016)*. European Language Resources Association (ELRA), may.

Valenzuela-Escárcega, M. A., Babur, Ö., Hahn-Powell, G., Bell, D., Hicks, T., Noriega-Atala, E., Wang, X., Surdeanu, M., Demir, E., and Morrison, C. T. (2018). Large-scale automated machine reading discovers new cancer driving mechanisms. *Database: The Journal of Biological Databases and Curation*.

Wang, X., Feng, A., Golshan, B., Halevy, A., Mihaila, G., Oiwa, H., and Tan, W.-C. (2018). Scalable semantic querying of text. *Proceedings of the VLDB Endowment*, 11(9):961–974.