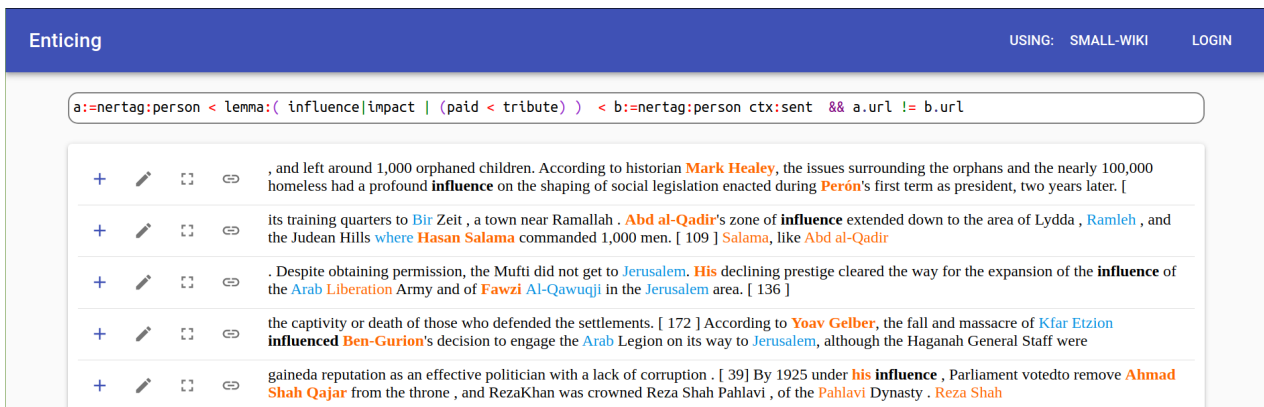


Enticing – Semantic Search Engine

David Kozák*



The screenshot shows the Enticing search engine interface. At the top, it says "Enticing" and "USING: SMALL-WIKI LOGIN". Below that is a search bar containing the query: `a:=nertag:person < lemma:(influence|impact | (paid < tribute)) < b:=nertag:person ctx:sent && a.url != b.url`. Below the search bar, there are five search results, each with a plus icon, a pencil icon, a square icon, and a link icon. The results show various sentences where the word "influence" is used, with some words highlighted in orange or red.

Abstract

The topic of this paper is semantic searching over big textual data. It describes the design and implementation of a search engine called Enticing that queries semantically enhanced documents efficiently and has a user friendly interface for working with the results. First, state of the art solutions along with their strengths and shortcomings are analyzed. Then a design for new search engine is presented along with a specialized query language EQL. The system consists of components for indexing and searching the documents, management server, compiler for the query language and two clients, web based and command line. The engine has been successfully designed, developed and deployed and is available via Internet. As a result of that, the possibility to use semantic searching is available to a wide audience.

Keywords: search engine — semantic enhancement — MG4J — compiler — indexation — searching — annotation — big data

Supplementary Material: [Web Interface](#)

*xkozak15@stud.fit.vutbr.cz, Faculty of Information Technology, Brno University of Technology

1. Introduction

The topic of this paper is semantic searching over big textual data. The Knowledge Technology Research Group (KNOT)¹ at FIT BUT has a Natural language processing (NLP) pipeline which can analyse documents written in natural languages and add additional meta information to them. Such information can be *syntactic*, such as lemma of the word or its position within sentences and paragraphs, or *semantic*, such as entities like people and places. The output of this pipeline is a big volume of textual data. It is already

a great piece of work on it's own, but without the ability to query these semantically enhanced documents, their usage is limited. The goal of this project is to design and develop a search engine that would query the documents efficiently while allowing to use all the meta information in the queries. The engine will be called Enticing.

A couple of search engines with support for semantic search such as Mimir or Sketch Engine have been implemented before. One such engine has even been created internally within KNOT². However, none of

¹<https://www.fit.vut.cz/research/group/knot/>

²<http://knot.fit.vutbr.cz/projects.html>

those matched the requirements for the new engine. Its goal is to allow the user to query all the meta information including entities with attributes and relationships between them. At the same time, the queries should be easy to read even for non-programmers. The engines mentioned above either did not provide support for entities with attributes or were way too complex for our use case. The previous engine was an attempt to create such a system, but unfortunately, it was not written in a maintainable way, which made any further extensions or bugfixes very time consuming. That's why the decision was made to create a new engine and design it with stability and maintainability in mind.

In order to query the semantic metadata, a special query language has to be used. As a part of this project, such query language called EQL has been designed and its compiler was integrated into the search engine infrastructure.

The text is structured as follows. Basic definitions and techniques used inside search engines are defined in section 2. Semantic enhancement and semantic search are introduced in 3. Section 4 describes state of the art semantic search engines along with their strengths and weaknesses. The design of Enticing is presented in 5. Section 6 describes Enticing Query Language. Section 7 discusses testing and evaluation of the platform. In the end, a conclusion is given.

2. Indexing and searching inside search engines

This section describes the problem of indexing and searching in the context of search engines. First subsection defines basic terms and typical techniques used within search engines. Afterwards a search engine called MG4J is presented, as it is used internally within Enticing.

Basic definitions

Since this project is a follow-up work of [1, 2], the terminology will be mostly identical as it was defined in [1] with some modifications and extensions.

Index The Oxford dictionary defines it as an alphabetical list of names, subjects, etc. with reference to the pages on which they are mentioned³.

In [1], they defined index as a structure allowing a faster access to a certain piece of information without the need to process all the data. This definition suits our needs perfectly, therefore we will follow it.

Inverted Index An inverted index over a collection of documents contains, for each term of the collection, the set of documents in which the term appears and additional information such as the number of occurrences of the term within each document, and possibly their positions [3]. This data structure is used inside search engines to efficiently determine which documents contain the specified words.

Query expansion Modern web search engines rely on query expansion, an automatic or semi automatic mechanism that aims at rewriting the user intent (i.e., a set of keywords, maybe with additional context such as geographical location, past search history, etc.) as a structured query built upon a number of operators [4].

Tree based evaluation of the query Search queries can be efficiently represented as trees. The leaves represent keywords from the query and the intermediary nodes represent operators. The searching algorithm can then proceed in the bottom up way as follows. First, all the leaves of the tree are evaluated and the results are stored within them. Then their parents are evaluated, combining results from their children. The execution proceeds all the way to the root, which represents the whole query [1].

Semantic models of searching The semantics of the search query is given by the semantic model. The simplest one is the boolean model, where only conjunctions, disjunctions, negations and keywords are allowed. Unfortunately, this model does not provide any information regarding the fact how the document was matched by the query. MG4J uses a semantic model called Minimal Interval Semantics. It uses intervals of natural numbers that are incomparable towards inclusion. Each interval is a witness of the satisfiability of the query, and defines a region of the document that satisfies it [4].

Snippet If a query matched a document, the result has to be presented to the user. One of the most common forms of presenting search results are snippets. A snippet is a part of a document that matched given query, possibly extended with some additional information given by the searching algorithm.

MG4J – Managing Gigabytes for Java

This subsection describes MG4J, a free full-text search engine for large documents written in Java [5]. It is developed under the GNU Lesser General Public

³<https://www.lexico.com/en/definition/index>

License⁴ at the University degli Studi di Milano⁵.

MG4J has a query language with very expressive set of operators allowing to build complex queries. These operators are implemented using new very efficient search algorithms [6]. It supports searching over multiple indexes and combining the results. On top of that, MG4J is open source, so it is possible to dive into the source code when the answers cannot be found in the documentation. Unfortunately, it has no support for entities with attributes and relationships between them. Nevertheless, the aforementioned properties make it a very suitable backend for our semantic search engine.

3. Semantic enhancement of natural languages

This section explains the topic of semantic enhancement. First it provides basic definitions. Then it describes how semantically enhanced documents are created within KNOT.

Basic definitions

The key definition in semantic enhancement is **Semantic annotation**. Semantic annotations are metadata assigned to other data in order to increase their context and semantics [1]. These annotations are usually derived from unstructured content using Natural language processing and afterwards they are encoded in a structured format suitable for semantic search [7].

Semantic search over documents aims to find information that is not based just on the presence of words, but also on their meaning. It is gradually establishing itself as the next generation search paradigm, which can satisfy better a wider range of information needs, as compared to traditional full-text search. In the case of semantic search, what is being indexed is typically a combination of words, formal knowledge typically expressed in an ontology, and semantic annotations mentioning ontological concepts in the text [7].

Corpora processing tools

Any search engine would be useless without large enough amount of data for searching. This subsection briefly introduces the corpora processing pipeline⁶ which is used within KNOT to create semantically enhanced documents. These documents are then used in Enticing. The input of this pipeline are html pages and the output are tsv files, where each line represents a

single word in the document along with its metadata. The pipeline consists of the following stages.

1. **Verticalization** – HTML pages are parsed into tsv files.
2. **Deduplication** – Filters out duplicate pages and paragraphs within them.
3. **Tagging** – Identifies parts of speech.
4. **Parsing** – Syntax analysis.
5. **SEC** – Semantic enhancement.

4. State of the art solutions

This section covers three state of the art search engines which support semantic search to a various extent along with their strengths and shortcomings.

Mimir

Mimir is a semantic search engine developed at the The University of Sheffield. It was developed as a part of the Gate infrastructure for language engineering⁷. Using index federation and cloud-based deployment, it can scale up to 150 millions documents. It also supports hybrid queries that arbitrarily mix full-text, structural, linguistic and semantic constraints [7]. It internally combines different indexing technologies. The full-text search is done using MG4J and a triple store queried using SPARQL is then used for accessing Linked Open Data resources.

Mimir is a very powerful engine, even too powerful for our use case. Most of its functionality is not necessary, therefore using it would be too heavyweight.

Sketch Engine

Sketch Engine is a tool for analyzing how languages work. It analyses large text corpora in order to find out what is typical and what is rare for a given language. It houses more than 500 corpora in more than 90 languages [8] and it supports searching in them. It is an interesting piece of work that can be used for studying languages, but it was not designed to be a publicly available search engine.

Previous system at FIT BUT

One semantic search engine was developed within KNOT. It's original author was Jan Kouril⁸ and it was extended by S. Panov in [1] and K. Gresova in [2]. Its architecture was the following. There were two main components, Webserver and IndexDaemon. IndexDaemon was able to index documents and query them. Webserver provided user interface for the system.

⁴<https://www.gnu.org/licenses/lgpl-3.0.en.html>

⁵<http://www.unimi.it/>

⁶<http://knot.fit.vutbr.cz/corpproc/corpproc.en.html>

⁷<https://gate.ac.uk/>

⁸<https://www.fit.vut.cz/person/ikouril/>

For querying the documents, a special package Query along with a query language mg4j-eql was developed in [1]. The Query package was created so that it could be used in various clients. It was later integrated into the old webserver in [2]. K. Gresova also made several changes to the webserver to make it more flexible. However, the original code was not written in a maintainable way and, as it typically happens with software products, it's quality got worse and worse over time. And because of the maintainability issues, it was very hard to extend its functionality or provide bug fixes.

5. Design of the platform

This section covers the design of Enticing. It is a distributed system consisting of several components. The components and the relationships between them are visualized in the Figure 1.

The system is designed to handle very large text corpuses. The components for indexing and searching are meant to be run in parallel on multiple machines, each of them working with a piece of the corpus. Since it is quite easy to split the set of documents into parts of similar length, this design allows to scale up or down very easily simply by adding or removing servers.

There is quite a lot of metadata available, but users sometimes need just a part of it. The system allows users to customize what metadata should be returned, which can reduce the size of the result only to the information that is really wanted.

Each of the components of the system will now be described in more details.

Webserver

Webserver is the first and only component common users will interact with. It is also the only one that is meant to be publicly accessible. All requests should pass through it before being forwarded further into the system. It exposes an API that can be used by any third-party service to submit a search query. It is bundled with a single page JavaScript application that serves as a GUI of the system.

Apart from being the entry point for queries, Webserver's responsibilities also include user management and search settings management.

ConsoleClient

Sometimes, especially for research purposes and for testing, it is useful to submit a query and collect all the results into a file or to execute a group of queries in a batch. ConsoleClient is a component that handles this use case.

IndexServer

IndexServer is a component that handles a set of documents and exposes an API for querying them. This API should be accessible only internally, that is from the Webserver or the ConsoleClient, it is not meant to be directly reachable for the users of the system.

Internally, the set of documents is divided into collections. Each of these collections is handled concurrently. The documents are queried using MG4J.

This component is meant to be deployed multiple times on multiple machines to handle bigger text corpuses.

IndexBuilder

This component is a command-line tool responsible for preprocessing documents using MG4J and creating indexes that are later used by IndexServers. The process of indexing is both time and resource consuming, that's why it is handled separately and not directly from the IndexServers.

6. EQL – Enticing Query Language

EQL is a language which can be used to query semantically enhanced documents on the Enticing platform. The queries can be as simple as a few words, but also very complex, containing logical operators, subqueries over multiple indexes or constraints further limiting the results.

Operators

The operators can be divided into four categories.

Basic operators

- **Implicit and** – $A B$ – If no operator is specified, **and** is chosen implicitly. That is all mentioned words have to be in the document, in any order.
- **Order** – $A < B$ – A should appear before B , but they do **not** have to be next to each other.
- **Sequence** – " $A B C$ " – A , B and C have to appear in this order next to each other.
- **And** – $A \& B$ – Both A and B have to be in the document, in any order.
- **Or** – $A | B$ – At least one of A , B has to be in the document.
- **Not** – $!B$ – B should not be in the document.
- **Parenthesis** – $(A | B) \& C$ – Parenthesis can be used to build more complex logic expressions.
- **Proximity** – $A B \sim 5$ – A and B should appear at most 5 positions apart.

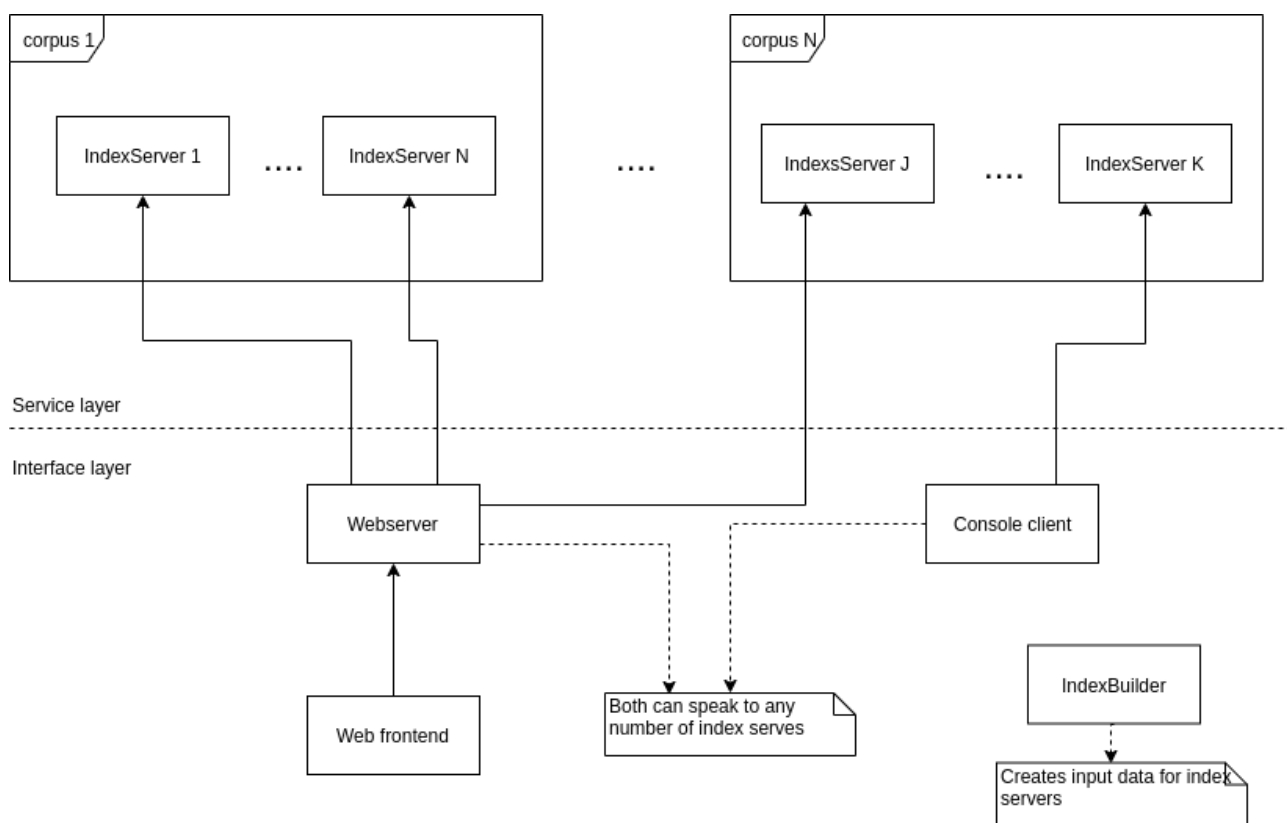


Figure 1. Components Components of the system

Index operators

To work with meta information, one has to specify index for querying. That can be done using the index operator.

- **Index** – `index:A`

This query will look for documents, where value A is present at given index. For example, `lemma:work` will match any document in which any word, whose lemma is work, appears, e.g. works, working, worked, etc.

It is also possible to ask more complex queries such as `index:A|B|C`. Apart from another index operator or entity operator, a query of any form can be used.

- **Attribute** – `person.name:A`

When working with entities, it is possible to query their attributes as well. The query above will return documents with people whose name is A.

- **Align** – `index1:A ^ index2:B`

The align operator allows to express multiple requirements over one word. For example, it is possible to look for a noun, whose lemma is do. This query can be written as `pos:noun ^ lemma:do`.

Context constraints

The default context for searching is the whole document. For more granular queries, it is possible to add the following limitations to the end of the query.

- **Paragraph** – `ctx:par` – Limits the query to one paragraph only.
- **Sentence** – `ctx:sent` – Limits the query to one sentence only.

Global constraints

Sometimes it is necessary to specify a relationship between multiple entities that can't be expressed using the previous operators. One example might be searching for documents talking about two artists influencing each other. A simple query for that would be the following.

```
nertag:artist < lemma:influence
< nertag:artist
```

But there is a problem with this query. It might return irrelevant snippets, because there is no requirement that the two artists should be different. This is where the global constraints come into play. The global constraint is a predicate which is separated from the query by symbols `&&`. The constraint consists of one or more equalities and inequalities connected using logical operators `and`, `or`, `not` and parenthesis, if necessary.

In order to use the global constraints, relevant parts of the query have to be identified first.

- **Assignment** – $x := A$ – Assign an identifier to a certain part of the query.

Afterwards, it is possible to write queries with global constraints.

```
a:=nertag:artist < lemma:influence
  < b:=nertag:artist
  && a.nerid != b.nerid
```

Searching algorithms

As mentioned earlier, MG4J has no support for entities with attributes and relationships between them. It also does not have support for identifiers. Therefore, the query has to be rewritten before being passed into MG4J and the results have to be post-processed. This step computes which parts of the documents were matched by which nodes in the Abstract Syntax Tree (AST) of the query. Using this information, global constraints can be evaluated. It can also be used to provide very precise highlighting. Unfortunately the computation is both time and space consuming. It essentially requires us to re-implement the searching functionality of MG4J on our own, which is not an easy task. On the other hand, it gives us a chance to create our own semantics for the query, which can be different from the underlying technology.

Minimal-Interval Semantics used by MG4J can significantly reduce the number of returned snippets and it allows them to use very efficient searching algorithms [4]. However, it has drawbacks. Imagine a query about a place and a person. Now let's say that there is a document containing three different people and three places and the intervals between them are overlapping. MG4J would return only one interval connecting one person with one place. In Enticing, we prefer to return all three combinations that occur, even though they overlap. MG4J does not support that, but it will at least return the document for us and we can then perform the post-processing generating all possible combinations. The con of our solution is that the number of results can grow exponentially. To cope with that, we decided to limit the number of snippets per document.

7. Testing and evaluation

This section covers testing and evaluation of the platform. Since the project is very big, multiple different types of tests were used together to ensure its quality. Each module has its own *unit tests* verifying its functionality. Different modules are combined together

into components and *integration tests* verify the communication between them.

A significant part of testing is centered around EQL and its searching functionality. They form the very core of the search engine and therefore their correctness is of the highest importance. EQL compiler has a suite of tests verifying that the translation from EQL to MG4J works as expected. Another suite of EQL tests was created to ensure that semantic errors are discovered before the query is executed. More *functional tests* were added to verify that given a query and a group of test documents, correct results are provided. These tests are done on multiple layers, from the core EQL algorithms through IndexServers all the way to the Webserver, to ensure that the results are not malformed as they are post-processed.

Performance

Apart from the correctness itself, performance is very important. Users expect search results to be delivered fast and therefore the whole system has to be optimized to satisfy that. In order to perform any optimizations, measurements have to be taken first, to make sure that the optimizations work as expected.

The current measurements at the time of writing this paper are presented in the table 2. Please note that the system is still being developed, so the results presented here might not be aligned with the performance you are experiencing when using Enticing. We are currently working on optimizing the search performance, so hopefully the system will be even faster by the time you experiment with it. We measured the duration of querying one IndexServer, as it is the most important use case to optimize. Wanted amount of snippets was 20. Tested IndexServer instance was deployed on KNOT server *knot01.fit.vutbr.cz*. This server has Intel Xeon E5-2630 2,3 GHz processor with 15MB cache and 6 cores. Total ram size is 65536 MB. IndexServer instance was handling 10 mg4j files, which together had 2.9 GB and contained 24371 documents. We created a list of queries and then submitted them 100 times, measuring the execution time of each query. We then computed the average value, the deviation, min and max of the execution time for each query. We started with simple single word queries, then combined them together, also adding restrictions later on. Contrary to what we expected, more complex queries are not always significantly slower. The number of occurrences of searched terms and their locations play a significant role as well. For example, there are only 1285 matches of the word water, but there are more than 10000 entities of type person. Therefore less documents have to be iterated when providing 20

results. You can also notice a significant slowdown in the query `water nertag:person nertag:location ctx:sent`. This happens because the context restrictions operators are currently not very optimized. They will be our first target in future optimizations.

8. Conclusions

The main topic of this paper was a search engine for querying semantically enhanced documents. The Knowledge Technology Research Group (KNOT)⁹ at FIT BUT has a Natural language processing (NLP) pipeline which can analyse documents written in natural languages and add additional meta information to them. Such information can be *syntactic*, such as lemma of the word or their position within sentences and paragraphs, or *semantic*, such as entities like people and places. The search engine should query the documents efficiently while allowing to use all the meta information in the queries.

The first section focused on the problem of indexing and searching inside search engines. It also described MG4J¹⁰, a search engine that is used internally in the resulting infrastructure. The following section focused on semantic enhancement of natural languages. The corpora processing pipeline¹¹ used within KNOT to create semantically enhanced documents was described. Afterwards the design of the new search engine called Enticing was presented. It is a distributed system, with each of its components consisting of several modules with well-defined interfaces. Each component is designed to be deployed as a separate process on its own server. Therefore the system should be able handle most of exceptions in its components without shutting down totally and at least partial results should be presented to the user whenever possible.

In order to query the semantic metadata, new query language called EQL (Enticing Query Language) was designed. This language is powerful enough to query all the entities inside semantically enhanced documents but also simple to understand, so that users from other domains can use it as well.

The system was built incrementally. First, basic prototypes of all components were developed to better understand the domain and gather more precise requirements. Afterwards, the architecture was carefully designed and all components were extended accordingly. Some parts were rewritten multiple times until their quality was sufficient. After the implementa-

tion, the system was tested and deployed on KNOT servers. The core was then further extended by adding a monitoring infrastructure and maintenance system. A follow-up work can extend the platform by adding new types of IndexServers, for example backed by a neural network, or by adding a native mobile client.

Acknowledgements

I'd like to thank my supervisor, Ing. Jaroslav Dytrych, Ph.D., and Doc. RNDr. Pavel Smrž, Ph.D for their professional help and guidance. I also want to thank Ing. Jan Doležal for his guidance and support in the design process. Last but not least, I'd like to thank my parents for their neverending support.

References

- [1] Panov Sergey. Indexing and searching semantically annotated texts. Bachelor thesis, Brno University of Technology, Faculty of Information Technology, Brno, 2017.
- [2] Grešová Katarína. Searching semantically annotated texts. Bachelor thesis, Brno University of Technology, Faculty of Information Technology, Brno, 2018.
- [3] Sebastiano Vigna. Quasi-succinct indices. In *Proceedings of the 6th ACM International Conference on Web Search and Data Mining*, pages 83–92, Rome, Italy, 2013. ACM.
- [4] Paolo Boldi and Sebastiano Vigna. Efficient optimally lazy algorithms for minimal-interval semantics. In *Theoretical Computer Science*, pages 8–25, Dipartimento di Informatica, Università degli Studi di Milano, Italy, 2016.
- [5] Paolo Boldi and Sebastiano Vigna. Mg4j: high-performance text indexing for java. online, 2005.
- [6] Paolo Boldi and Sebastiano Vigna. *Mg4j (big) The Manual*. Available at <http://mg4j.di.unimi.it/man-big/manual.pdf>.
- [7] V. Tablan, K. Bontcheva, I. Roberts, and H. Cunningham. Mimir: an open-source semantic search framework for interactive information seeking and discovery. *Journal of Web Semantics*, 2014.
- [8] Adam Kilgarriff, Pavel Rychlý, Miloš Jakubíček, et al. Sketchegine. online.

⁹<https://www.fit.vut.cz/research/group/knot/>

¹⁰<http://mg4j.di.unimi.it/>

¹¹http://knot.fit.vutbr.cz/corpproc/corpproc_en.html

Table 2. Results of performance tests

EQL Query	Average[ms]	Deviation[ms]	min[ms]	max[ms]
water	711.4	65843.12	337	1823
nertag:person	155.4	2263.76	104	421
nertag:location	177.73	24321.48	93	835
water nertag:person	503.0	35445.24	355	1259
water nertag:person ctx:sent	423.15	8891.11	321	771
nertag:person nertag:location	130.24	1919.32	96	433
nertag:person nertag:location ctx:sent	287.09	2490.72	220	463
water nertag:person nertag:location	453.82	14603.47	331	1056
water nertag:person nertag:location ctx:sent	4317.23	16767.16	4071	4828
nertag:person nertag:person	119.28	1071.40	96	403
nertag:person nertag:person ctx:sent	124.48	828.09	100	330
a:=nertag:person b:=nertag:person	117.31	699.074	97	287
a:=nertag:person b:=nertag:person ctx:sent	130.61	667.64	104	247
a:=nertag:person b:=nertag:person && a.url != b.url	126.33	801.48	98	263
a:=nertag:person b:=nertag:person ctx:sent && a.url != b.url	353.91	10034.46	250	801