DBee: A Database for Creating and Managing Knowledge Graphs and Embeddings

Viktor Schlegel, André Freitas
Department of Computer Science
University of Manchester
{viktor.schlegel,andre.freitas}@manchester.ac.uk

Abstract

This paper describes DBee, a database to support the construction of data-intensive AI applications. DBee provides a unique data model which operates jointly over large-scale knowledge graphs (KGs) and embedding vector spaces (VSs). This model supports queries which exploit the semantic properties of both types of representations (KGs and VSs). Additionally, DBee aims to facilitate the construction of KGs and VSs, by providing a library of generators, which can be used to create, integrate and transform data into KGs and VSs.

1 Introduction

Many AI tasks can be summarised into the cycle of collecting data, overlaying a representation (schema) on the top of the data and performing learning and inference algorithms, which will eventually produce new data or extend the representation. While in many cases learning and inference are put at the centre of the stage, managing the data and the supporting representations are fundamental parts of the design and delivery of an AI system.

Currently, the prevalence of workflow architectures for many types of AI systems reflects the emphasis on learning and inference, where data management becomes a secondary concern. However, complex AI tasks such as Question Answering (QA) (Kumar et al., 2016), Text Entailment (Hashimoto et al., 2016) or Natural Language Inference are either directly dependent on or can benefit from the construction of supporting Knowledge Bases.

Recently, latent and explicit semantic representations are emerging as fundamental elements for supporting those tasks, due to their dependency on commonsense and domain specific knowledge. Moreover, the recent rise of successful approaches operating at the neuro-symbolic representation level (Parisotto et al., 2016; Liang et al., 2016), demands for a closer dialogue between explicit and latent models. Word embeddings (Mikolov et al., 2013) and Knowledge Graphs (lexico-semantic graphs) (Bollacker et al., 2008) are becoming the de-facto representation models within different AI tasks. Moreover, they have complementary properties, where word embeddings provide more coarse-grained semantics which are complemented by the fine-grained semantics of KGs being commonly used in coordination (Silva et al., 2018; Xie et al., 2017).

This paper describes DBee, a database for creating, querying and consuming embeddings and knowledge graphs. DBee aims to be a database designed for satisfying recurring demands from AI applications. DBee provides a seamless layer to jointly query knowledge graphs and embeddings, simultaneously exploiting the semantic properties of both resources, taking into account performance and scalability aspects. At the centre of the proposed database is the goal of bridging the gap between data, representation, learning and inference algorithms, where classifiers and extractors directly interface with the schema. By design, DBee provides a declarative layer for data and representation management in AI systems. Finally, DBee also supports the combination of different models and representations (cross-model and cross-representation queries) and their customisation.

In the following sections of the paper we motivate our approach with an initial scenario, discuss background and and related work, describe the proposed framework, present the implemented system by instantiating it for archetypal use cases and conclude with a discussion outlining the expected performance, hardware requirements and current limitations of the system.

2 Motivational Scenario

An AI application engineer wants to build a QA system to support investors in NASDAQ companies. Most of the data relevant for this task such as finan-

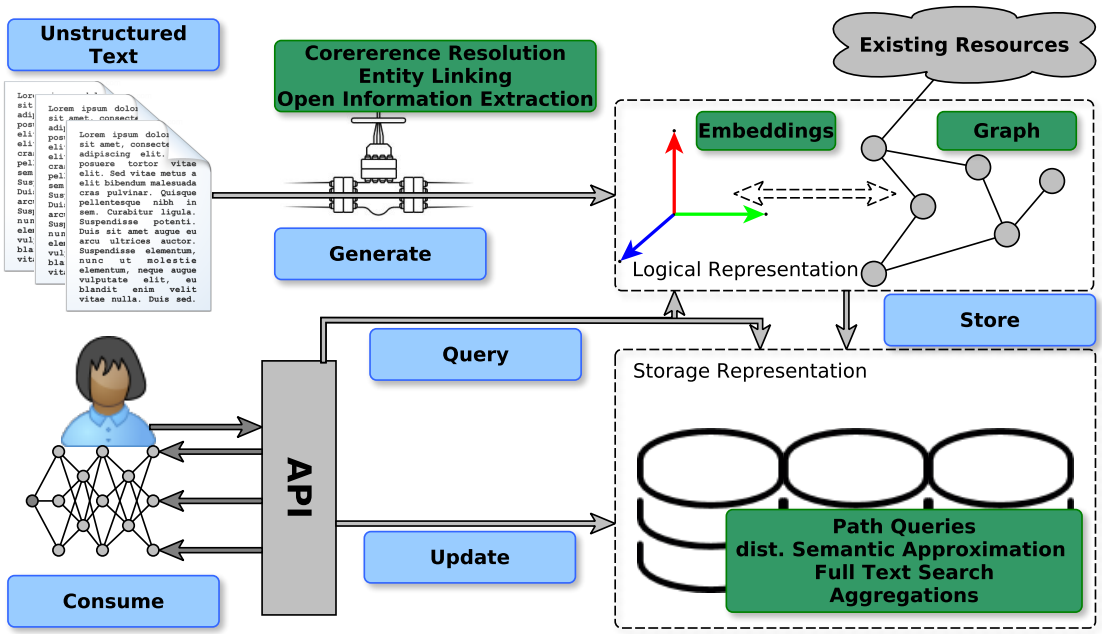

Figure 1: Architecture and Workflow overview: the overall architecture (blue) supports the implementation of the motivational scenario (green).

cial reports, blog articles and recent news only exist in the form of unstructured text. The engineer also anticipates the benefits of integrating structured Knowledge Graphs such as DBpedia (Auer et al., 2007), integrating KGs to the textual data sources. Realizing the importance of his application to be able to generate traceable and explainable answers, he decides to use an explicit internal representation, such as the graph-based RDF-NL (Cetto et al., 2018). With the associated chain of classifiers and extractors available at DBee, he performs open-information extraction (OIE), Coreference Resolution (CR), Entity Linking (EL) and Rhetorical Structure Classification (RSC) to obtain the graph from a chosen set of documents. After the extraction, the graph is indexed to ensure efficiency for different types of queries over the graph representation. In order to support semantic approximation during the queries, he associates two pre-trained word embedding models to the KG using the DBee API and uses the provided set of query primitives to query the knowledge graph. Deciding to use triples from the KG as features for a neural stock predictor model, he uses the DBee API to create input and answer sets (for a set of pre-defined queries) ready to be consumed by the automatic differentiation framework of his choice.

3 Background & Related Work

Current machine learning systems such as Keras (Chollet et al., 2015), and PyTorch (Paszke et al., 2017) focus mostly on exposing their user to the definition of neural architectures, abstracting away the computation details of automatic differentiation - or trying to learn even those (Jin et al., 2018) - with TensorFlow (Abadi et al., 2016) being the most complete suite providing assistance from data streaming, over training to model serving. Our approach can be seen as complementary to these efforts since we aim to provide the infrastructure to extract, represent and query structured and unstructured data (with an emphasis on KGs from text and associated embeddings).

Early efforts in a similar direction include (Sales et al., 2018), that present a uniform service-based API for storing, querying and comparing word embeddings, pre-computed with varying models and on different datasets. Another information management tool for unstructured data is Apache UIMA [1].

Contemporary machine comprehension systems based on neural architectures have targeted evaluation settings which have limited document scale (e.g. SQUAD (Rajpurkar et al., 2016)).

Different works explored the connection be-

[1] http://uima.apache.org

tween distributional semantics and structured Knowledge Graph representations in the context of semantic parsing over large-scale RDF graphs (Freitas and Curry, 2014; Freitas, 2015; Sales et al., 2016) and approximative abductive reasoning over commonsense KBs (Freitas et al., 2014, 2013). Comparatively, DBee focuses on explicit semantic representation models (Knowledge Graphs) extracted from text.

4 Proposed Framework

To satisfy the emerging need to work with unstructured text representations, as depicted in the introductory part of this work, we propose a framework that supports the extraction and management of both explicit and latent text representation models and facilitates the integration with downstream machine learning based models. DBee was designed to deliver the following features:

1. **Bridging the gap between unstructured data and semantic representations:** Conforming data into latent and explicit text representations is a primary requirement for many AI applications. DBee allows users to create, reuse and compose a library of text extractors and classifiers which will be used to structure and integrate existing unstructured data. The library includes standard *representation generators* such as syntactic and lexical parsers, open information extractors, named entity recognisers and linkers and discourse-level extractors.

2. **Multi-representation model:** DBee supports users in experimenting with different types of explicit and latent semantic representations and models. Different tasks will require different types of representation. Users should be able to query across multiple representations.

3. **Expressive structured queries and ML integration:** To give its users fine-grained control over the data and to overlay their own machine learning algorithms, DBee features an intuitive query language and seamless integration with existing machine learning algorithms.

4. **Extensibility:** Representation schemas and their supporting generators are extensible and customisable.

5. **Scalability:** Operating over large-scale data sources, large knowledge graphs and embeddings require principled query processing strategies. DBee inherits indexing strategies from databases and kNN embedding queries in order to support scaling to large datasets, memory footprints and storage space requirements.

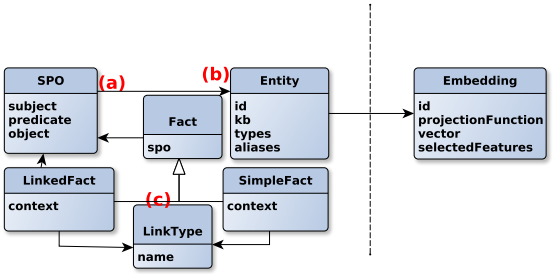

Figure 2: Initial data model

5 The DBee Model

5.1 Data Model

DBee operates over two types of representation: knowledge graphs and word embeddings.

The underlying knowledge graph data model uses RDF-NL, an extension of the RDF (Lassila and Swick, 1999) data model suitable to represent text as a lexico-semantic Knowledge Graph. RDF-NL is built upon a sentence representation model proposed by (Niklaus et al., 2017, 2019, 2018) which splits complex sentences into simpler linked clausal and phrasal elements, later splitting these elements into predicate-argument structures.

The graph data model (Figure 2 (a)) is defined by a subject-predicate-object (SPO) triple which can have contextual relations (C) as reifications or can be linked to other SPO triples. Contextual links can be named. This data model supports the creation of versatile sparse graph representations. For example, the data model smoothly captures linguistic predicate-argument structures and phrasal (e.g. appositive), clausal (coordination and subordination), rhetorical and argumentation relations. Figure 3 shows an example of a concrete knowledge graph extracted from a sentence.

All SPO nodes are defined by their lexical realisation (typically a text chunk) and can be linked to a canonical identifier in the entity component of the data model (Figure 2 (b)), which allows an

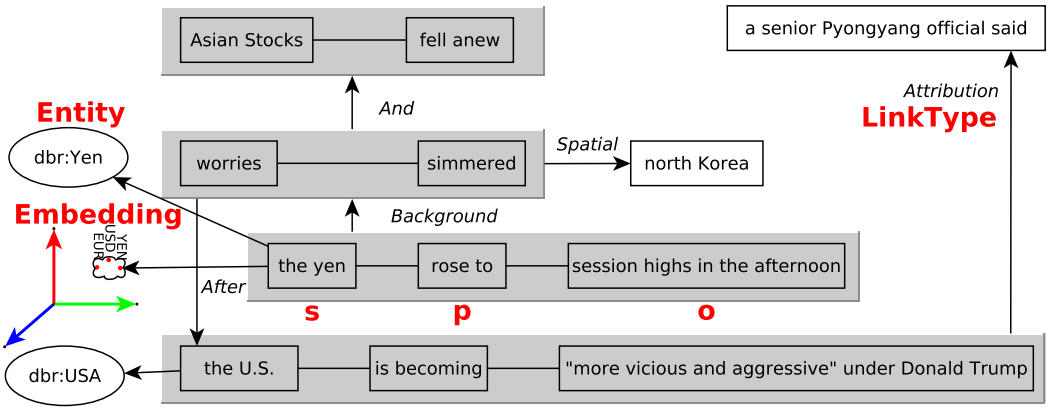

Figure 3: Example generator chain output of the sentence *"Asian stocks fell anew and the yen rose to session highs in the afternoon as worries about North Korea simmered, after a senior Pyongyang official said the U.S. is becoming "more vicious and more aggressive" under President Donald Trump ."*

entity-centric data integration, such as it is performed by co-reference resolution, entity linking or word-vector clustering.

The data model is materialised into different types of supporting indexes in order to enable efficient and scalable query processing. There are two main types of indexes associated with the data model:

- `Embedding Indexing` (${}_{E}I$): Supports the efficient querying of embedding spaces (k-NN similarity queries). By default it uses the random projections of the locality-sensitive hashing method proposed by (Charikar, 2002).

- `Knowledge Graph Lexical Indexing` (${}_{T}I$): Supports information-retrieval style keyword search queries over the KG structure using inverted indexes and associated weighting schemes (by default, TF-IDF is used).

5.2 Operations

At the centre of the DBee data model is the ability to build, transform and combine KGs and Vector Spaces/Embeddings (VSs).

Different KGs and VSs can be combined using a *view* allowing support of querying specific compositions. *Projections* ($\vec{\pi}$) are operators which build VSs (embeddings) from KGs and unstructured datasets.

On the top of the *views*, *query* operators are defined. View, projections and queries can be chained together. If called in the middle of a function chain, these functions serve the purpose of a join in a sense similar to relational databases. This means views and projections later in the operation chain will only operate on the subset of results satisfying the query defined in the chain so far. This behaviour is visualized in Figure 5.

The domain-specific query language (DSL) associated with DBee includes the following functions:

- `query(term, n)`: This operation retrieves up to n best matching candidates from the view/projection it is being executed upon with respect to its type. For a projection, for example, it retrieves n nearest neighbours regarding their embeddings, after embedding the query term using the projection space's corresponding embedding function.

- `filter(attribute=value | bgp | conditional statement)`: This operation filters are selection operators (σ) for predicates defined as the function's parameters. They can be defined as an attribute=value form or with the help of basic graph patterns).

- `rank(wrt)`: This function can be used to rank a set of results with respect to a given term by their distance to it in the corresponding vector space.

- `top(n)`, `count()`: Aggregation operators will retrieve the top results in up to a given limit or count them up, respectively. Provided a name, the result set will consist of attributes of this name.

- `create_view(name)`: Creates a new view with a given name from the current result set.

- `create_projection(name, using, features)`: Similarly, creates a new projection using a given embedding function by extracting the features from every result in the set. Features might be defined simply by providing a list of attribute names to use or any callable operation to extract custom features.

5.3 Representation Generators (g) & Chains (c)

Representation elements have associated *generators* g, which are classifiers, extractors and linkers which operate over *Data*, *KGs* or *VSs*.

The generators are stored into *libraries*, typed according to their *representation function* and tagged with the model metadata (such as training corpus and evaluation score, architecture and hyperparameter configuration). Generators can be composed using generator *chains*. For example, generating a KG from textual data would typically employ the chain: $g^{CR} \circ g^{EL} \circ g^{OIE}_{RDF-NL}$. As with the generators, chains can be named and persisted into libraries.

Generators can also be associated with vector representations, e.g. g^{VS}_{W2V}. The set of pre-defined generators currently present at DBee are described in Table 1.

Figure 1 summarises the main primitives of the system depicting a schematic high-level components diagram of DBee.

Concretely, we propose a pipeline with the following steps: First, using contextualised open information extraction (Cetto et al., 2018), structured information is extracted from the unstructured text, in the form of a set of inter-linked subject-predicate-object triples, thus yielding a graph. With coreference resolution, the graph is further enriched semantically, linking nodes that refer to the same entity in the text. In a final entity linking step, recognised entities are connected to existing resources, contextualising them further regarding existing background knowledge.

Table 1: Library of pre-defined Generators

Symbol	Description
g^{CR}	Coreference Resolution generator
g^{EL}_{X}	Entity Linker to the resource X
g^{NER}	Named Entity Recognizer
g^{OIE}	Open Information Extractor
g^{π}	provider of an embedding function pi

The extracted knowledge graph is then serialised and indexed, while still retaining its logical graph representation. In particular, we use full-text search capable databases and nearest neighbour indices to enable querying and approximation of stored data using string-based as well as embedding-based methods.

The API layer features a chainable IDSL to allow intuitive interaction with the data. Concretely it is designed to support expressive recurring query patterns while reducing impedance mismatch.

6 Usage

6.1 Implementation

DBee was conceptualised and implemented as an extensible Python library. We use the HOCON[2] format to enable for easy generator chain definition and persistence. We provide a pre-defined chain featuring the contextualised open information extraction tool Graphene (Cetto et al., 2018), the Stanford CoreNLP coreference resolution system (Manning et al., 2014) and the entity linker Spotlight (Mendes et al., 2011) that links recognised entities to DBpedia resources. Furthermore, we use ElasticSearch[3] as the full-text search engine and Annoy[4] to index the embeddings for the kNN queries.

Note, that following the design goals of extensibility and scalability, the software is not constrained to use those specific tools. Even the choice of generators and storage types is not fixed, as it requires low effort to add a new generator or storage type, such as a relational database to perform joins more efficiently, for instance.

6.2 DBee in Code

Listed below are example instantiations illustrating the usage of the framework exercised on four exemplar use cases.

6.2.1 Extraction

Code 1 shows the boilerplate code required to instantiate DBee. From a list of Wikipedia article titles one can query the Wikipedia API and apply

[2] https://github.com/chimpler/pyhocon
[3] https://www.elastic.co
[4] https://github.com/spotify/annoy

```
[0]─(Trump)─(withdrew)─(his sponsorship)
    ├[   Temporal   ]─(in 1990 .)
    ├[TemporalBefore]─(after the second Tour de Trump .)
    └[    Cause     ]─(his business ventures)─(were experiencing)─(financial woes)
[1]─(Trumps business ventures)─(were experiencing)─(financial woes)
```

Figure 4: Example output of a DBee fact query.

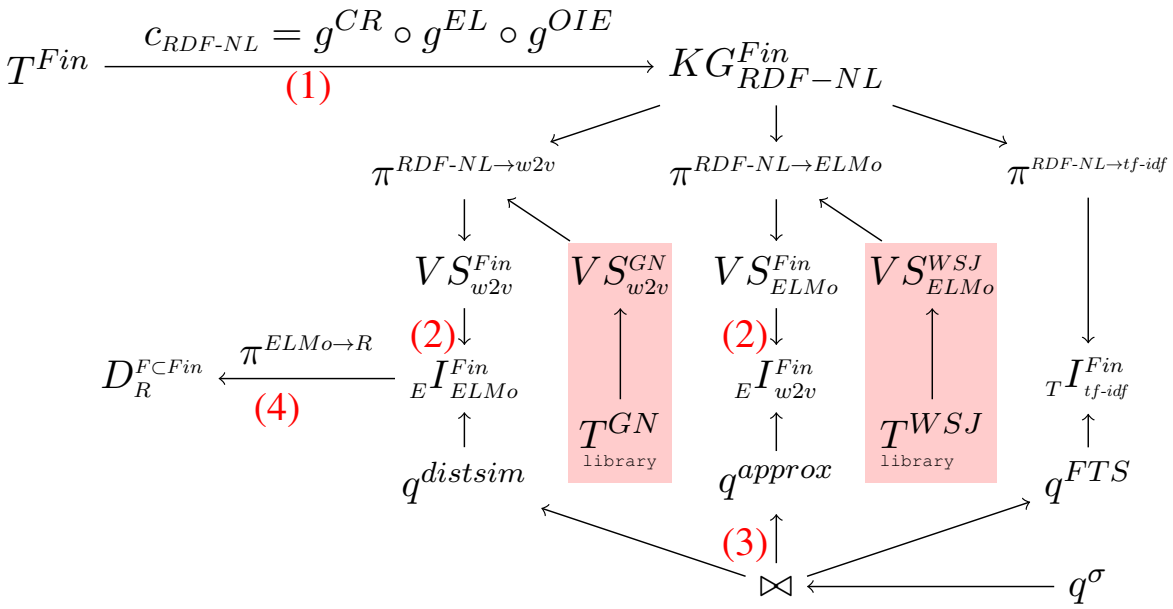

Figure 5: Conceptual overview of the code snippets. From unstructured text, using a chain of generators (1) a knowledge graph is extracted and stored (2) in different indices, which are later queried jointly by the corresponding query types they offer (3) or used to create a data set (4) to train a neural network.

the selected generator chain. The snippet further

```
main = DBee()
docs = main.get_pipeline("extended").
 assemble().
 load("wikipedia-nasdaq-100.txt")
```

```
{
 '()': semantic.OIEGenerator
 coref_provider: {
  class: graphene.FromRESTProvider
  server_address = localhost
 }
}
```

Code Snippet 1: Extraction Example

highlights the definition of a generator as one step of the chain, using HOCON syntax, with semantics similar python's logging module configuration.

6.2.2 Index Creation

From the KG extracted in the previous step, the storage indices can be populated. The DSL-level user does not necessarily need to know

```
kb('nasdaq-100').
 using(docs.get_iterator('fact')).
 create_view('fact')
```

```
kb('nasdaq-100').
 view('spo').
 create_projection('po',
  pi=IndraEmbedder,
  features=['predicate', 'object'])
```

Code Snippet 2: Example of data Storage

the actual name of the data view (`'fact'` in this case) but can obtain it by querying the class of the corresponding generator type (i.e. `OIEGenerator.provides`). Similarly, additional indices can be constructed and stored from the representation generated by an already defined chain or indices - as shown in the example, by utilising one of the pre-trained embedding generators provided by DBee.

6.2.3 Querying

Code 3 shows the query equivalent to the natural language query *Which companies have offices in China?*. The query describes the process of filtering the list of initial entities to retain only those of the type "company", switch the data view to facts (performing a join implicitly), further filtering out facts, and finally projecting the remaining entities into the previously created vector space and ranking them by distance to the computed projection of a given term. An implicit join back to the textual view is made to retrieve the subjects of the re-ordered remaining facts.

```
kb('nasdaq-100')
 view('entity').
 filter(type='dbo:Company').
 view('fact').
 filter(label=Spatial, context='China').
 view('spo').projection('po').
 rank(wrt="have offices").
 get('subject')
```

Code Snippet 3: DSL Querying example

Note that the query uses already resolved filter predicates for brevity, one could likewise use the operation `view('types').query('companies')` to query for the concrete type URI using the expression obtained from the text - given the view was constructed beforehand.

6.2.4 ML Integration

```
kb('nasdaq-100')
 view('fact').
 as_classification_dataset(
  features=bow(["spo", "context.spo"])
  labels=onehot("context.label")
 )
```

Code Snippet 4: Dataset Creation Example

Finally, the example in Code 4 shows the creation of a toy dataset for link type prediction between two interlinked facts. The user-defined defined `bow` and `onehot` functions serve as feature extractors.

7 Discussion

7.1 Analysis of T^{Fin}

While this is not meant to be a thorough empirical analysis, the following section gives insights into the performance of the system regarding time and space requirements.

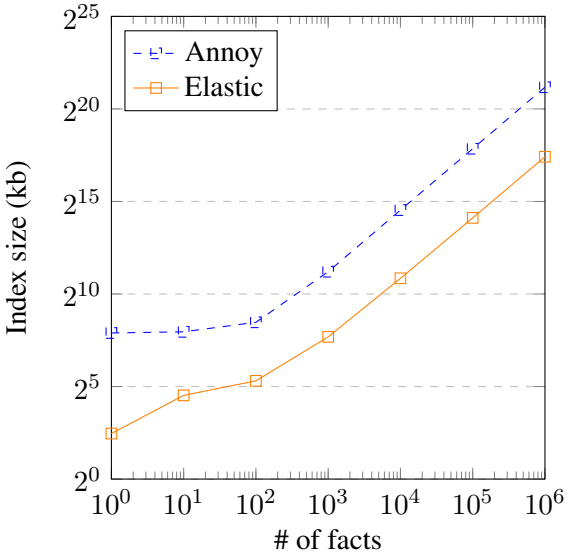

Figure 6: Index sizes for Annoy and ElasticSearch indices for varying number of facts

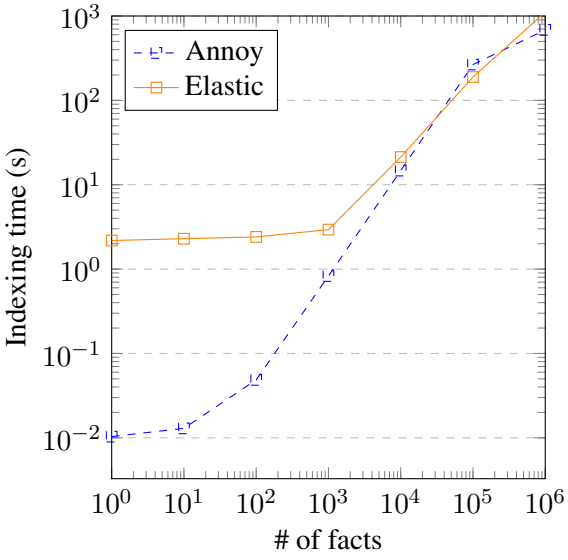

Figure 7: Indexing times for Annoy and ElasticSearch indices

All of the following measurements were carried out on a notebook featuring an SSD, a dual-core i5-6200U CPU performing at 2.3GHz and 16 GB of RAM.

We provide the vector space indexing times and sizes for a varying number of indexed vectors, averaged over ten runs. In particular, we index $10^n, n \in 0..6$ embedding vectors using our vector space storage implementation based on Annoy. The dimension of the embeddings is 300. For full-

text search indices, we performed the same procedure. We populated ElasticSearch indices with $10^n, n \in 0..6$ facts, denormalised according to the data model, using a single local node. Figures 6 and 7 shows the result, revealing that indexing times and index sizes scale linearly with the size of the dataset. Axes within the plots are at logarithmic scale.

The creation of a small dataset from 100 Wikipedia articles yielded 2292 recognised entities, 22633 distinct subject-predicate-object structures and 39456 contextual links. The total required storage space was 8.159 MB for the denormalised textual data stored in ElasticSearch and 148.37 MB for the stored 300 dimensional `word2vec` embeddings. Running all steps in sequence - from obtaining the documents up to storing them in coresponding indices - took approximately 3.5 hours.

It is worth noting that the open information step takes up most of the processing time. However, since by design the extraction process does not require to explore any dependencies between different documents, a speedup factor of up to n can be assumed for n parallel instantiations of the extraction pipeline.

7.2 Current Limitations & Future Work

In its current version, the software does not support data insertion or updates due to an implementation detail of choosing the annoy implementation for nearest neighbour approximation, in favour of its speed. There are, however, recent approaches for nearest neighbour estimation that support dynamic index updates (Li and Malik, 2017).

Furthermore, the current approach requires different tools to store different data representation types such as views and projections. One future direction is to investigate how to build low-level vector space index support into existing DBMS.

Finally, a rigorous analysis regarding scalability, complexity, performance and usability will be carried out in the future.

7.3 Conclusion

In this paper, we formalised an approach to create and manage knowledge graphs and embeddings and to query them jointly and introduced DBee, a system implementing this approach. We hope to provide the community with a tool that facilitates the management, storage and querying of latent and explicit text representation facilitating its integration to downstreal ML/AI applications.

Acknowledgements

The authors would like to express their gratitude towards members of the AI Systems lab at the University of Manchester for many fruitful discussions.

References

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: a system for large-scale machine learning.. In *OSDI*, Vol. 16. 265–283.

Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. 2007. Dbpedia: A nucleus for a web of open data. In *The semantic web*. Springer, 722–735.

Kurt Bollacker, Colin Evans, Praveen Paritosh, Tim Sturge, and Jamie Taylor. 2008. Freebase: a collaboratively created graph database for structuring human knowledge. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. AcM, 1247–1250.

Matthias Cetto, Christina Niklaus, André Freitas, and Siegfried Handschuh. 2018. Graphene: Semantically-Linked Propositions in Open Information Extraction. In *Proceedings of the 27th International Conference on Computational Linguistics*. Association for Computational Linguistics, 2300–2311.

Moses S Charikar. 2002. Similarity estimation techniques from rounding algorithms. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*. ACM, 380–388.

François Chollet et al. 2015. Keras. (2015).

André Freitas. 2015. *Schema-agnositc queries over large-schema databases: a distributional semantics approach.* Ph.D. Dissertation. Digital Enterprise Research Institute (DERI), National University of Ireland, Galway.

André Freitas and Edward Curry. 2014. Natural language queries over heterogeneous linked data graphs: a distributional-compositional semantics approach. In *19th International Conference on Intelligent User Interfaces, IUI 2014, Haifa, Israel, February 24-27, 2014*. 279–288.

André Freitas, João Carlos Pereira da Silva, Edward Curry, and Paul Buitelaar. 2014. A Distributional Semantics Approach for Selective Reasoning on Commonsense Graph Knowledge Bases. In *Natural Language Processing and Information Systems - 19th International Conference on Applications of Natural Language to Information Systems, NLDB 2014, Montpellier, France, June 18-20, 2014. Proceedings*. 21–32.

Andre Freitas, Joao C. P. da Silva, Sean ORiain, and Edward Curry. 2013. Distributional Relational Networks. In *AAAI 2013 Fall Symposium on Semantics for Big Data*.

Kazuma Hashimoto, Caiming Xiong, Yoshimasa Tsuruoka, and Richard Socher. 2016. A joint many-task model: Growing a neural network for multiple NLP tasks. *arXiv preprint arXiv:1611.01587* (2016).

Haifeng Jin, Qingquan Song, and Xia Hu. 2018. Efficient Neural Architecture Search with Network Morphism. *arXiv preprint arXiv:1806.10282* (2018).

Ankit Kumar, Ozan Irsoy, Peter Ondruska, Mohit Iyyer, James Bradbury, Ishaan Gulrajani, Victor Zhong, Romain Paulus, and Richard Socher. 2016. Ask me anything: Dynamic memory networks for natural language processing. In *International Conference on Machine Learning*. 1378–1387.

Ora Lassila and Ralph R Swick. 1999. Resource description framework (RDF) model and syntax specification. (1999).

Ke Li and Jitendra Malik. 2017. Fast k-nearest neighbour search via prioritized dci. *arXiv preprint arXiv:1703.00440* (2017).

Chen Liang, Jonathan Berant, Quoc Le, Kenneth D Forbus, and Ni Lao. 2016. Neural symbolic machines: Learning semantic parsers on freebase with weak supervision. *arXiv preprint arXiv:1611.00020* (2016).

Christopher Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven Bethard, and David McClosky. 2014. The Stanford CoreNLP natural language processing toolkit. In *Proceedings of 52nd annual meeting of the association for computational linguistics: system demonstrations*. 55–60.

Pablo N Mendes, Max Jakob, Andrés García-Silva, and Christian Bizer. 2011. DBpedia spotlight: shedding light on the web of documents. In *Proceedings of the 7th international conference on semantic systems*. ACM, 1–8.

Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*. 3111–3119.

Christina Niklaus, Bernhard Bermeitinger, Siegfried Handschuh, and Andr Freitas. 2017. A Sentence Simplification System for Improving Relation Extraction. (2017). arXiv:cs.CL/1703.09013

Christina Niklaus, Matthias Cetto, André Freitas, and Siegfried Handschuh. 2018. A Survey on Open Information Extraction. In *Proceedings of the 27th International Conference on Computational Linguistics*. Association for Computational Linguistics, Santa Fe, New Mexico, USA, 3866–3878.

Christina Niklaus, Matthias Cetto, André Freitas, and Siegfried Handschuh. 2019. Transforming Complex Sentences into a Semantic Hierarchy. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, Florence, Italy, 3415–3427.

Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. 2016. Neuro-symbolic program synthesis. *arXiv preprint arXiv:1611.01855* (2016).

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in pytorch. (2017).

Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. 2016. Squad: 100,000+ questions for machine comprehension of text. *arXiv preprint arXiv:1606.05250* (2016).

Juliano Efson Sales, Andre Freitas, Brian Davis, and Siegfried Handschuh. 2016. A Compositional-Distributional Semantic Model for Searching Complex Entity Categories. In *Proceedings of the Fifth Joint Conference on Lexical and Computational Semantics*. Association for Computational Linguistics, Berlin, Germany, 199–208.

Juliano Efson Sales, Leonardo Souza, Siamak Barzegar, Brian Davis, André Freitas, and Siegfried Handschuh. 2018. Indra: A Word Embedding and Semantic Relatedness Server. In *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC 2018)*. European Language Resources Association (ELRA), Miyazaki, Japan.

Vivian Dos Santos Silva, Siegfried Handschuh, and André Freitas. 2018. Recognizing and Justifying Text Entailment Through Distributional Navigation on Definition Graphs.. In *AAAI*.

Qizhe Xie, Xuezhe Ma, Zihang Dai, and Eduard Hovy. 2017. An interpretable knowledge transfer model for knowledge base completion. *arXiv preprint arXiv:1704.05908* (2017).