GeeseDB: A Python Graph Engine for Exploration and Search

Chris Kamphuis[1], Arjen P. de Vries[1]

[1]Radboud University, Toernooiveld 212, Nijmegen, The Netherlands

Abstract

GeeseDB is a Python toolkit for solving information retrieval research problems that leverage graphs as data structures. It aims to simplify information retrieval research by allowing researchers to easily formulate graph queries through a graph query language. GeeseDB is built on top of DuckDB, an embedded column-store relational database designed for analytical workloads. GeeseDB is available as an easy to install Python package. In only a few lines of code users can create a first stage retrieval ranking using BM25. Queries read and write Numpy arrays and Pandas dataframes, at zero or negligible data transformation cost (dependent on base datatype). Therefore, results of a first-stage ranker expressed in GeeseDB can be used in various stages in the ranking process, enabling all the power of Python machine learning libraries with minimal overhead. Also, because data representation and processing are strictly separated, GeeseDB forms an ideal basis for reproducible IR research.

Keywords

Open-Source Search Engine, Information Retrieval, Graph Databases

1. Introduction

In recent years there has been a lot of exciting new information retrieval research that makes use of non-text data to improve the effectiveness of search systems. Consider for example dense representations for retrieval [1, 2, 3], knowledge graphs to leverage entity information [4, 5, 6], and non-textual learning-to-rank features [7, 8]. All these research directions have improved the effectiveness of search systems by making use of more diverse data. Despite the fact that search systems consider more diverse sources of data, the usage of this data is often implemented through the use of a coupled architecture. In particular, first-stage retrieval is often carried out with different software compared to later retrieval stages where these novel reranking techniques tend to be used. In our view, researchers could benefit from a system where retrieval stages are more tightly integrated, that facilitates the exploration on how to use non-content data for ranking, and serves the data in a format suitable for reranking with e.g. transformers or tree based methods.

In order to fulfill these needs we propose GeeseDB[1], a prototype Python toolkit for information retrieval that leverages graphs as data structures, allowing metadata and graphs to be easily included in the ranking pipeline. The toolkit is designed to quickly set up first stage retrieval, and make it easy for researchers to explore new ranking models quickly. In short, GeeseDB aims to provide the following functionalities:

- GeeseDB is an easy-to-install, self-contained Python package available through `pip install` with as few as possible dependencies. It contains topics and relevance judgements for several standard IR collections out-of-the-box, allowing researchers to quickly start developing new ranking models.
- First stage (sparse) retrieval is directly supported. In only a few lines of code it is possible to load documents and create a first stage ranking.
- Data is served in a usable format for later retrieval stages. GeeseDB allows to directly run queries on Pandas data frames for efficient data transfer to sequential reranking algorithms.
- Data exploration is supported through querying data with SQL, but more interestingly, also using a graph query language, making the exploration of new research avenues easier. This prototype supports a subset of the graph query language Cypher [9], a graph query language originally proposed for Neo4j, similar to the property graph database model query language as described by Angles [10].

GeeseDB began as a project after identifying the opportunities for graph queries to improve reproducible IR [11] at the Open-Source IR Replicability Challenge SIGIR workshop [12]. Prior work observed many BM25 implementations [13, 14], that resulted in wildly varying effectiveness scores, and the variety of systems participating in this workshop also found varying BM25 effectiveness scores between them. Is this really a problem? Several valid

DESIRES 2021 – 2nd International Conference on Design of Experimental Search & Information REtrieval Systems, September 15–18, 2021, Padua, Italy

chris@cs.ru.nl (C. Kamphuis); arjen@cs.ru.nl (A. P. de Vries)

© 2021 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings (CEUR-WS.org)

[1] https://github.com/informagi/geesedb

reasons could explain these differences in effectiveness; document pre-processing, parameter tuning, or even interpretation of the theory to arrive at the exact ranking formula to be used. When using these scores as a baseline however, the effectiveness gain of novel methods could be exaggerated due to the (coincidental) choice for an implementation of the baseline that gives low effectiveness. Indeed, Yang et al. [15] showed empirically that the comparison against weak baselines *is* a real problem, that can obfuscate the real gain in effectiveness.

A method introduced into the community to help simplify the comparison between open source search systems has been the introduction of the Common Index File Format (CIFF) [16]. CIFF is a binary *data exchange format* that can be used by search systems to share their index structures. This way, researchers ensure that the exact same pre-processing has been applied when comparing different systems to each other. Experiments in [16] show how differences in (BM25) effectiveness scores between different implementations do decrease when their indexes are exchanged using CIFF. GeeseDB therefore adopts the CIFF index format to exchange data between systems.

A second approach to improve the reproducibility of IR research results has been adopted less widely. By making use of a database system, the way how data is stored and the plans on how this data is processed are explicitly separated. This enables easier inspection on differences between ranking formulas. In that perspective, it may be not so surprising that the only two systems that produced the exact same effectiveness scores for their BM25 rankings in the studies mentioned above, were the two relational database systems used to rank documents; even though their execution engines were completely different and implemented by different teams. Also, in the work by Kamphuis et al. [17], using a shared database back-end for a series of retrieval experiments, testing a number of previously proposed 'improvements' of BM25, demonstrated that these differences between variants turn out insignificant once everything but the ranking formula is fixed.

Given these findings, we fully subscribe to the position that the declarative specification of ranking in a database query language offers the potential to improve reproducibility in IR research. SQL queries that express more complex ranking functions than the default combination of term frequency and document frequency, can however easily become overly tedious to write, elaborate and error-prone. As the way forward, GeeseDB therefore introduces the property graph data model with a graph query language to express IR retrieval models in a more compact manner. We show in this work that this is especially useful when introducing representations of documents and queries that include information beyond just text.

2. Design

At the core of GeeseDB lies the full text search design presented by Mühleisen et al. [14]. In this work, a column-store database for IR prototyping is proposed, which uses the database schema described in Figure 1, consisting of three database tables. (One for all term information,

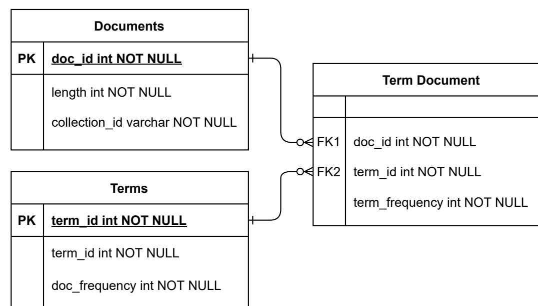

Figure 1: Database schema by Mühleisen et al. [14] for full text search in relational databases

one for all document information, and one that contains the information on how terms relate to documents; the information that is found in a posting list of an inverted index). Using these three tables they show that BM25 can be easily expressed as a SQL query, with latencies that are on par with custom-build IR engines. In GeeseDB we use the exact same relational schema for full text search. Instead of seeing the document data and term data as tables that relate to each other through a many-to-many join table, it is also possible to consider this schema as a bipartite graph. In this graph both documents and terms are considered as nodes, connected to each other through edges. If a term occurs in a document there exists an edge between that term and document. GeeseDB uses the data model of property graphs; labeled multigraphs where both edges and nodes can have property-value pairs. The database schema as described in Figure 1 would then translate to the property graph schema shown in Figure 2. A small example of a graph represented by this

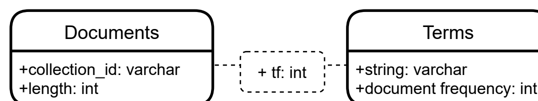

Figure 2: Graph schema representing bipartite document-term graph

schema is shown in Figure 3, document nodes contain document specific information (i.e. document length and the collection identifier), term nodes contain information relevant to the term (i.e. the term string and the

term's document frequency), and the the edges between document and terms nodes contain term frequency information (i.e. how often is the term mentioned in the document represented the respective nodes it connects). If one wants to also store position data, this graph can

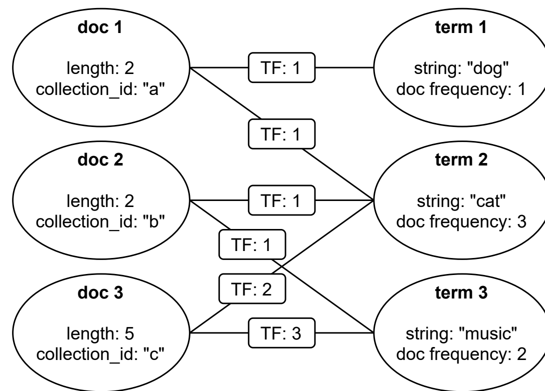

Figure 3: Example term-document graph that maps to relational database schema

easily be changed to a graph where the edges store the position of a term. If a term would appear multiple times in a document, the property graph model would allow for multiple edges to exist between two nodes. The graph schema that we described by Figure 2 maps one-to-one to the relational database schema described by Figure 1, so nodes are represented by normal relational tables that represent specific data units (terms, documents), while edges are represented by many-to-many join tables. So, even though we think of the data as graphs, in the back-end they are represented as relational tables. When using GeeseDB for search we expect at least the document-term graph to be present, of course new node types can be introduced in order to explore new search strategies.

2.1. Backend

GeeseDB is built on top of DuckDB [18], an in-process SQL OLAP (analytics optimized) database management system. DuckDB is designed to support analytical query workloads, meaning that it specifically aims to process complex long-running queries where a significant portion of the data is accessed, conditions matching the case of IR research. DuckDB has a client Python API which can be installed using `pip`, afterwards it can be used directly. DuckDB has a separated API built around both NumPy and Pandas, providing NumPy/Pandas views over the same underlying data representation, without incurring data transfer (usually referred to as "zero-copy" reading). Pandas DataFrames can be registered as virtual tables, allowing to directly query the data present in Pandas DataFrames. GeeseDB inherits all these functionalities from DuckDB.

As DuckDB is a SQL database management system, we can execute analytical SQL queries on the tables that contain our data, including the BM25 rankings described by Mühleisen et al. [14]. By default, the BM25 implementation provided with GeeseDB implements the disjunctive variant of BM25, instead of the conjunctive variant they used. Although the conjunctive variant of BM25 can be calculated more quickly, we chose to use the disjunctive variant as it is more commonly used by IR researchers and the differences between effectiveness scores are noticeable on smaller collections. For now we only support the original formulation of BM25 by Robertson et al. [19], however support of or adding other versions of BM25 [17] is trivial.

2.2. Graph Query Languange

What distinguishes GeeseDB from alternatives, database-backed (olddog) [20] or native systems (Anserini [21], Terrier [22]) is the graph query language, based on Cypher [9]. Systems like Elasic[2] and Solr[3] do support querying graphs, but not declaratively. For now, GeeseDB implements Cypher's basic graph pattern matching queries for retrieving data. An example of a graph query supported by GeeseDB is presented in Figure 4. This

```
MATCH (d:docs)-[]-(:authors)-[]-(d2:docs)
WHERE d.collection_id = "96ab542e"
RETURN DISTINCT d2.collection_id
```

Figure 4: An example cypher query that finds all documents that were written by the same author that wrote the document with the `collecion_id` "96ab542e"

query finds all documents written by the same authors as those who wrote document "96ab542e". For comparison, Figure 5 illustrates the same query represented in SQL; much more complex than the Cypher version, due to the join conditions that have to be made explicit. In order to connect the "docs" table with the "authors" table 2 joins are needed, reconnecting the "docs" table again introduces two more joins.

At the moment of writing, GeeseDB supports the following Cypher keywords: MATCH, RETURN, WHERE, AND, DISTINCT, ORDER BY, SKIP, and LIMIT. Instead of using WHERE to filter data, it is also possible to use graph matching using the keyword MATCH, as shown in Figure 6; the query returns the length of document "96ab542e". We

[2] https://www.elastic.co/what-is/elasticsearch-graph (accessed 19-08-2021)

[3] https://solr.apache.org/guide/6_6/graph-traversal.html (accessed 19-08-2021)

```sql
SELECT DISTINCT d2.collection_id
FROM docs AS d2
JOIN doc_author AS da2
    ON (d2.collection_id = da2.doc)
JOIN authors AS a2
    ON (da2.author = a2.author)
JOIN doc_author AS da3
    ON (a2.author = da3.author)
JOIN docs AS d
    ON (d.collection_id = da3.doc)
WHERE d.collection_id = '96ab542e'
```

Figure 5: SQL query that corresponds to the graph query described in Figure 4.

```
MATCH (d:docs {d.collection_id: "96ab542e"})
RETURN d.len
```

Figure 6: Graph query where the length of document with `collection_id` is returned.

plan to support the other keywords of Cypher in the future, as well as directed edges. Everything that is not yet directly supported yet by our implementation can of course still be expressed in SQL, which is fully supported[4]. In order to know how to join nodes to each other if no edge information has been provided, GeeseDB stores information on the schema. This way GeeseDB knows how nodes relate to each other through which edges. GeeseDB has a module for updating the graph schema, allowing researchers to easily set up the graph they want represented in the database.

3. Usage

GeeseDB comes as an easy-to-install Python package that can be installed using pip, the standard package installer for Python:

```
$ pip install geesedb==0.0.1
```

We can start using GeeseDB after installing it. All examples we show in this paper were run on version v0.0.1 of GeeseDB. However, as GeeseDB is actively being developed, we advise readers to use the latest version of GeeseDB, which can be installed when not specifying a package version. It is also possible to install the latest commit by installing the latest version directly from GitHub[5].

As an example, we will show how to use GeeseDB for the background linking task of the TREC News Track [23]. The goal of this task is: *Given a news story, find other news articles that can provide important context or background information.* These articles can then be recommended to the reader to help them understand the context in which these news articles take place. The collection used for this task is the Washington Post V3 collection[6] released for the 2020 edition of TREC. It contains 671.945 news articles published by the Washington Post published between 2012 and 2020, and 50 topics with relevance assessments (topics correspond to collection identifiers of documents for which relevant data has to be found). The articles in this collection contain useful metadata; in particular, we will use authorship information. We extracted 25.703 unique article authors, where it is possible that multiple authors co-wrote a news article. We also annotate documents with entity information which was obtained by using the Radboud Entity Linker [24]. In total 31.622.419 references to 541.729 unique entities were found. An edge between entity and document nodes contains mention and location information, as well as the `ner_tag` found by the linker's entity recognition module (the entity linker can assign different tags to the same entity).[7] Figure 7 illustrates the data schema that we use for the background linking task.

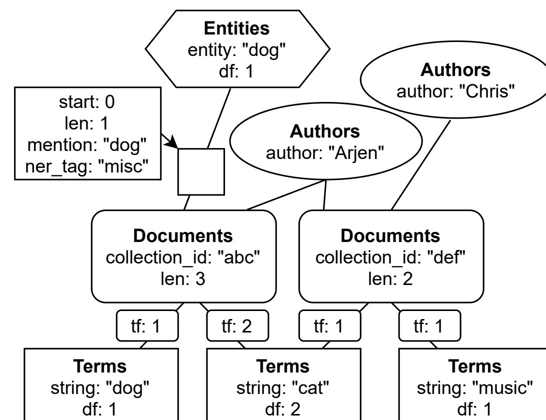

Figure 7: Example property graph for the TREC News Track's background linking task. The node types are authors, entities, terms and documents. Edges connect document nodes to other types of nodes. Both edges and nodes can have properties (following the property graph model). Multiple edges may exist between one entity node and one document node, as one entity can be linked multiple times to one document.

[4] GeeseDB supports the graph queries by translating them to their corresponding SQL queries, both nodes and edges are after all just tables in the backend.

[5] https://github.com/informagi/GeeseDB#package-installation

[6] https://trec.nist.gov/data/wapost/

[7] The annotated data will be made publicly available.

3.1. Indexing and Search

In order to start, a database containing at least the document and term information needs to be created. Figure 8 shows how the data can be easily loaded using CSV files.

```
from geesedb.index import FullTextFromCSV

index = FullTextFromCSV(
    database='/path/to/database',
    docs_file='/path/to/docs.csv',
    term_dict_file='/path/to/term_dict.csv',
    term_doc_file='/path/to/term_doc.csv'
)
index.load_data()
```

Figure 8: Load text data from the WashingtonPost collection formatted as csv files in the format as described by Mühleisen et al. [14]

Instead of loading the data from CSV files it is also possible to load the text data directly using the CIFF format for data exchange [16]. GeeseDB also has functionalities to create the CSV files used here from the CIFF format. Authorship information and entity links can be loaded similarly. Processing Cypher queries depends on the schema information that needs to be loaded as well. We have a supporting class (called metadata) for this, and the schema data used in this paper will be available via GitHub. After loading the data we can quickly create a BM25 ranking for ad hoc search in the Washington Post collection as shown in Figure 9.

```
from geesedb.search import Searcher

searcher = Searcher(
    database='/path/to/database',
    n=10
)
topic = 'obama and trump'
hits = searcher.search_topic(topic)
```

Figure 9: Example on how to create a BM25 ranking for the query "obama and trump" that returns the top 10 documents.

For the background linking task however, we do not have regular topics; we only have the collection identifiers of the documents we need to find relevant background info for. In order to search for relevant background reading, queries that represent our information need to be constructed. A common approach is to use the top-k TF-IDF terms of the source article. These can easily be found using the Cypher statement shown in Figure 10. Instead of using Cypher it would also be possible to use SQL, as shown in Figure 11; however this example shows again the Cypher query is more elegant.

```
MATCH (d:docs {collection_id:
↪   ?})-[]-(t:term_dict)
RETURN string
ORDER BY tf*log(671945/df)
DESC
LIMIT 5
```

Figure 10: Prepared Cypher statement that finds the top-5 TF-IDF terms in a document.

```
SELECT term_dict.string
FROM term_dict
JOIN term_doc ON
    (term_dict.term_id = term_doc.term_id)
JOIN docs ON
    (docs.doc_id = term_doc.doc_id)
WHERE docs.collection_id = ?
ORDER BY term_doc.tf *
↪   log(671945/term_dict.df
DESC
LIMIT 5;
```

Figure 11: Prepared SQL statement that finds the top-5 TF-IDF terms in a document.

Using the terms found with Cypher, we can construct queries that we can pass to the searcher, and create a BM25 ranking. The code that generates the rankings for all topics is presented in Figure 12. As you can see, with only a limited number of lines of Python code it is quite easy to create rankings. Note that the collection size is hardcoded as version v0.0.1 does not support aggregation yet. From this point it is quite trivial to write the content of `hits` to a runfile, and evaluate using `trec_eval`.
Instead of "just" ranking with BM25, using e.g. the metadata in order to adapt the ranking is straightforward. In the case of background linking, it makes sense to consider authorship information when recommending articles that might be suitable as background reading. As journalists are often specialized in certain news topics (e.g. politics, foreign affairs, tech), the stories they write often share context. Also, when journalists collaborate on stories they write together on topics they specialize in as well. As authorship information is available to us, we can decide to use the information whether an article is written by the authors of the topic article, or by someone they have collaborated with in the past. Finding the articles that are written by this group of people can easily be done using a graph query, the query that finds these articles is shown in Figure 13.
Depending on the number of documents found by this query, different rescoring strategies can be decided upon. If the set of documents written by the authors or their co-authors is large, perhaps it is possible to only consider these documents, but if the set is small, a score boost

```python
from geesedb.search import Searcher
from geesedb.connection import get_connection
from geesedb.resources import
↪   get_topics_backgroundlinking
from geesedb.interpreter import Translator

db_path = '/path/to/database'
searcher = Searcher(
        database=db_path,
        n=1000
)

translator = Translator(db_path)
c_query = """cypher TFIDF query"""

query = translator.translate(c_query)
cursor = get_connection(db_path).cursor
topics = get_topics_backgroundlinking(
    '/path/to/topics'
)
for topic_no, collection_id in topics:
    cursor.execute(query, [collection_id])
    topic = ' '.join(cursor.fetchall()[0])
    hits = searcher.search_topic(topic)
```

Figure 12: Create a BM25 ranking for all background linking topics using the top-5 TFIDF terms. Note that in this case a processed topic file was used that only contains the topic identifier and the topic article id. The topic file in this format is provided on our GitHub.

```
MATCH (d:docs)-[]-(:authors)-[]-(:docs)-[]-
↪   (:authors)-[]-(d2:docs {collection_id:
↪   ?})
RETURN DISTINCT d.collection_id
```

Figure 13: Cypher query to find documents written by co-authors of the authors of the topic article.

```python
# import and first lines the same as example

author_c_query = """cypher authorship
↪   query"""
author_query = t.translate(author_c_query)

cursor = get_connection(db_path).cursor
topics = get_topics_backgroundlinking(
    '/path/to/topics'
)
for topic_no, collection_id in topics:
    cursor.execute(query, [collection_id])
    topic = ' '.join(cursor.fetchall()[0])
    hits = searcher.search_topic(topic)

    cursor.execute(author_query,
    ↪   [collection_id])
    docs_authors = {
        e[0] for e in cursor.fetchall()
    }
    if len(docs_authors) > 2000:
        hits = hits[hits.collection_id.isin(
            docs_authors)]
```

Figure 14: Find documents written by all authors that collaborated with the authors of the topic article, if there are more than 2000 documents found only consider these documents as background reading candidates.

```
    MATCH (d:docs {collection_id:
↪   ?})-[]-(e:entities)
    RETURN mention
    ORDER BY start
    LIMIT 5
```

Figure 15: Retrieve the first five entities mentioned in the topic article; and return the terms used to mention the entity.

might be more appropriate. Figure 14 shows an example on how to only consider documents found with the query in Figure 13, in this particular case we ensure that at least 2000 documents are found before filtering.

To give another example; the graph query language is also useful when considering entities. When journalists write news articles, the articles relate to events concerning e.g. people, organisations, or countries. In other words, the basis of news articles lay the entities as they are often the subject of news. So, instead of using the most informative terms in a news article, it could be useful to consider the entities identified in the article instead. Important entities tend to be mentioned in the beginning of a news article [25]; Figure 15 shows the Cypher query to retrieve the text mentions of the first five mentioned entities.

Before it is possible to search using the text describing the first five entity mentions, the text needs to be processed. The term data loaded in GeeseDB was already processed, as it was data loaded from CSV files built from a CIFF file created from an Anserini [21] (Lucene) index. Anserini has an easy to use Python extension, Pyserini [26], that can be used to tokenize the text in the same way as the documents were tokenized. Figure 16 shows the Python code where we extract the mentions, process them such that they become a usable query for GeeseDB, and then BM25 ranking is created with this query.

In summary, GeeseDB allows researchers to index and search data with only a few lines of Python code. It can be used to explore new IR research ideas through both SQL and the Cypher graph query language. As GeeseDB can query directly on top of Panda's DataFrames, no data transfer has to be done, making this framework ideal to set up the data for other Python reranking pipelines (i.e. it is trivial to store learning-to-rank features in the

```python
from geesedb.search import Searcher
from geesedb.connection import get_connection
from geesedb.resources import
↪ get_topics_backgroundlinking
from geesedb.interpreter import Translator
from pyserini.analysis import Analyzer,
↪ get_lucene_analyzer

db_path = '/path/to/database'
searcher = Searcher(
    database=db_path,
    n=1000
)

analyzer = Analyzer(get_lucene_analyzer())

translator = Translator(db_path)
c_query = """cypher entity query"""
query = translator.translate(c_query)

cursor = get_connection(db_path).cursor
topics = get_topics_backgroundlinking(
    '/path/to/topics'
)

for topic_no, collection_id in topics:
    cursor.execute(query, [collection_id])
    topic = ' '.join([e[0] for e in
    ↪ cursor.fetchall()])
    topic = ' '.join(analyzer.analyze(topic))
    hits = searcher.search_topic(topic)
```

Figure 16: Create a BM25 ranking for all background linking topics using the mention text of the first five linked entities in the source article.

database that can then directly be used).

4. Future Work

As the current GeeseDB version is still an early prototype, many future improvements have been envisioned. We have identified four improvements we want to pursue as a priority:

- We have implemented the graph query language Cypher only partially; in the near future, we would like to support this fully. For now, it is only possible to use the graph query language to query data, but ideally it could also be used to load or update data. Of course it is already possible to do this through the SQL backend, but this should only be necessary for extending the backend support for new use-cases.
- As the goal of GeeseDB is to serve as an IR toolkit, we would like to extend GeeseDB with functionalities that make the package easy to use for IR researchers. A few obvious extensions would be IR dataset support, native document processing, and implementations of popular first-stage rankers.
- This version of GeeseDB lacks extensive benchmarking. We plan to release benchmarks on popular IR datasets, including instruction on how to reproduce these benchmarks.
- In recent years, dense graph representations have become popular. We would like to add the functionality to analyse these dense representations for graphs managed in GeeseDB.

Eventually, we would like to extend the query language with proper support to define ranking over graphs. (Now, the ranking function is hidden in the 'searcher' module.)

5. Conclusion

In this work we have described our prototype implementation of GeeseDB, and how we envision graph databases can be used for information retrieval research. GeeseDB is still in active development, and we are open to additional contributions from the community.

Acknowledgments

This work is part of the research program Commit2Data with project number 628.011.001 (SQIREL-GRAPHS), which is (partly) financed by the Netherlands Organisation for Scientific Research (NWO).

References

[1] L. Gao, Z. Dai, T. Chen, Z. Fan, B. Van Durme, J. Callan, Complement lexical retrieval model with semantic residual embeddings, in: Advances in Information Retrieval, ECIR '21, Springer International Publishing, Cham, 2021, pp. 146–160.

[2] Y. Luan, J. Eisenstein, K. Toutanova, M. Collins, Sparse, dense, and attentional representations for text retrieval, Transactions of the Association for Computational Linguistics 9 (2021) 329–345.

[3] S. Lin, J. Yang, J. Lin, Distilling dense representations for ranking using tightly-coupled teachers, CoRR abs/2010.11386 (2020). URL: https://arxiv.org/abs/2010.11386. arXiv:2010.11386.

[4] F. Hasibi, K. Balog, S. E. Bratsberg, Exploiting entity linking in queries for entity retrieval, in: Proceedings of the 2016 ACM International Conference on the Theory of Information Retrieval, ICTIR '16, Association for Computing Machinery, New York, NY, USA, 2016, p. 209–218. URL: https://doi.org/10.1145/2970398.2970406. doi:10.1145/2970398.2970406.

[5] K. Balog, Entity-oriented search, Springer Nature, Gewerbestrasse 11, 6330 Cham, Switzerland, 2018.

[6] J. Dalton, L. Dietz, J. Allan, Entity query feature expansion using knowledge base links, in: Proceedings of the 37th International ACM SIGIR Conference on Research; Development in Information Retrieval, SIGIR '14, Association for Computing Machinery, New York, NY, USA, 2014, p. 365–374. URL: https://doi.org/10.1145/2600428.2609628. doi:10.1145/2600428.2609628.

[7] R. Deveaud, M.-D. Albakour, C. Macdonald, I. Ounis, On the importance of venue-dependent features for learning to rank contextual suggestions, in: Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management, CIKM '14, Association for Computing Machinery, New York, NY, USA, 2014, p. 1827–1830. URL: https://doi.org/10.1145/2661829.2661956. doi:10.1145/2661829.2661956.

[8] C. Macdonald, R. L. Santos, I. Ounis, On the usefulness of query features for learning to rank, in: Proceedings of the 21st ACM International Conference on Information and Knowledge Management, CIKM '12, Association for Computing Machinery, New York, NY, USA, 2012, p. 2559–2562. URL: https://doi.org/10.1145/2396761.2398691. doi:10.1145/2396761.2398691.

[9] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, A. Taylor, Cypher: An evolving query language for property graphs, in: Proceedings of the 2018 International Conference on Management of Data, SIGMOD '18, Association for Computing Machinery, New York, NY, USA, 2018, p. 1433–1445. URL: https://doi.org/10.1145/3183713.3190657. doi:10.1145/3183713.3190657.

[10] R. Angles, The property graph database model., in: Proceedings of the 12th Alberto Mendelzon International Workshop on Foundations of Data Management, AMW '18, CEUR-WS.org, Aachen, 2018.

[11] C. Kamphuis, A. P. de Vries, Reproducible IR needs an (IR) (graph) query language, in: Proceedings of the Open-Source IR Replicability Challenge co-located with 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval, OSIRRC@SIGIR 2019, Paris, France, July 25, 2019, CEUR-WS.org, Aachen, 2019, pp. 17–20. URL: http://ceur-ws.org/Vol-2409/position03.pdf.

[12] R. Clancy, N. Ferro, C. Hauff, J. Lin, T. Sakai, Z. Z. Wu, The sigir 2019 open-source ir replicability challenge (osirrc 2019), in: Proceedings of the 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR'19, Association for Computing Machinery, New York, NY, USA, 2019, p. 1432–1434. URL: https://doi.org/10.1145/3331184.3331647. doi:10.1145/3331184.3331647.

[13] J. Arguello, F. Diaz, J. Lin, A. Trotman, Sigir 2015 workshop on reproducibility, inexplicability, and generalizability of results (rigor), in: Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR '15, Association for Computing Machinery, New York, NY, USA, 2015, p. 1147–1148. URL: https://doi.org/10.1145/2766462.2767858. doi:10.1145/2766462.2767858.

[14] H. Mühleisen, T. Samar, J. Lin, A. de Vries, Old dogs are great at new tricks: Column stores for ir prototyping, in: Proceedings of the 37th International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR '14, Association for Computing Machinery, New York, NY, USA, 2014, p. 863–866. URL: https://doi.org/10.1145/2600428.2609460. doi:10.1145/2600428.2609460.

[15] W. Yang, K. Lu, P. Yang, J. Lin, Critically examining the "neural hype": Weak baselines and the additivity of effectiveness gains from neural ranking models, in: Proceedings of the 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR'19, Association for Computing Machinery, New York, NY, USA, 2019, p. 1129–1132. URL: https://doi.org/10.1145/3331184.3331340. doi:10.1145/3331184.3331340.

[16] J. Lin, J. Mackenzie, C. Kamphuis, C. Macdonald, A. Mallia, M. Siedlaczek, A. Trotman, A. de Vries, Supporting interoperability between open-source search engines with the common index file format, in: Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR '20, Association for Computing Machinery, New York, NY, USA, 2020, p. 2149–2152. URL: https://doi.org/10.1145/3397271.3401404. doi:10.1145/3397271.3401404.

[17] C. Kamphuis, A. P. de Vries, L. Boytsov, J. Lin, Which bm25 do you mean? a large-scale reproducibility study of scoring variants, in: Advances in Information Retrieval, ECIR '20, Springer International Publishing, Cham, 2020, pp. 28–34.

[18] M. Raasveldt, H. Mühleisen, Duckdb: An embeddable analytical database, in: Proceedings of the 2019 International Conference on Management of Data, SIGMOD '19, Association for Computing Machinery, New York, NY, USA, 2019, p. 1981–1984. URL: https://doi.org/10.1145/3299869.3320212. doi:10.1145/3299869.3320212.

[19] S. E. Robertson, S. Walker, S. Jones, M. M. Hancock-Beaulieu, M. Gatford, et al., Okapi at trec-3, Nist Special Publication Sp 109 (1995) 109.

[20] C. Kamphuis, A. P. de Vries, The olddog docker

image for OSIRRC at SIGIR 2019, in: Proceedings of the Open-Source IR Replicability Challenge co-located with 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval, OSIRRC@SIGIR 2019, Paris, France, July 25, 2019, CEUR-WS.org, Aachen, 2019, pp. 47–49. URL: http://ceur-ws.org/Vol-2409/docker07.pdf.

[21] P. Yang, H. Fang, J. Lin, Anserini: Enabling the use of lucene for information retrieval research, in: Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR '17, Association for Computing Machinery, New York, NY, USA, 2017, p. 1253–1256. URL: https://doi.org/10.1145/3077136.3080721. doi:10.1145/3077136.3080721.

[22] I. Ounis, G. Amati, V. Plachouras, B. He, C. Macdonald, D. Johnson, Terrier information retrieval platform, in: D. E. Losada, J. M. Fernández-Luna (Eds.), Advances in Information Retrieval, Springer Berlin Heidelberg, Berlin, Heidelberg, 2005, pp. 517–519.

[23] I. Soboroff, S. Huang, D. Harman, Trec 2018 news track overview., in: Proceedings of The Twenty-Seventh Text REtrieval Conference, TREC '18, National Institute for Standards and Technology (NIST), Gaithersburg, Maryland, USA, 2018.

[24] J. M. van Hulst, F. Hasibi, K. Dercksen, K. Balog, A. P. de Vries, Rel: An entity linker standing on the shoulders of giants, in: Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR '20, Association for Computing Machinery, New York, NY, USA, 2020, p. 2197–2200. URL: https://doi.org/10.1145/3397271.3401416. doi:10.1145/3397271.3401416.

[25] C. Kamphuis, F. Hasibi, A. P. de Vries, T. Crijns, Radboud university at trec 2019., in: Proceedings of The Twenty-Eight Text REtrieval Conference, TREC '19, National Institute for Standards and Technology (NIST), Gaithersburg, Maryland, USA, 2019.

[26] J. Lin, X. Ma, S.-C. Lin, J.-H. Yang, R. Pradeep, R. Nogueira, Pyserini: An easy-to-use python toolkit to support replicable ir research with sparse and dense representations, arXiv preprint arXiv:2102.10073 (2021).