CLOCQ: A Toolkit for Fast and Easy Access to Knowledge Bases

Philipp Christmann[1] Rishiraj Saha Roy[2] Gerhard Weikum[3]

Abstract: Curated knowledge bases (KBs) store vast amounts of factual world knowledge, and are therefore ubiquitous in many information retrieval (IR) and natural language processing (NLP) applications like question answering, named entity disambiguation, or knowledge exploration. Despite that, accessing information from complete knowledge bases is often a daunting task. Researchers and practitioners typically have crisp use cases in mind, for which standard querying interfaces can be overly complex and inefficient. We aim to bridge this gap, and release a public toolkit that provides functionalities for common KB access use cases, and make it available via a public API. Experiments show efficiency improvements over existing KB interfaces for various important functionalities.

Keywords: Knowledge Base; Knowledge Graph; RDF; Efficiency

1 Introduction

Large curated knowledge bases (KBs), also known as knowledge graphs (KGs), like Wikidata [VK14], DBpedia [Au07], YAGO [SKW07], Freebase [Bo08], and industrial counterparts (e.g. at Amazon, Apple, Google, or Microsoft), store *factual* world knowledge in compact RDF (Resource Description Framework) triples.

Such knowledge bases empower question answering (QA) systems [BH15, Be13, Ch19, SRA22], that offer natural interfaces for accessing information. Most digital personal assistants like Alexa, Siri, Google Assistant, or Cortana are essentially QA systems accessing a variety of information at their backends, including curated KBs. There is a wide range of research on QA, ranging from methods that target simple single-shot questions [Ab18, Be13], spanning methods dedicated for complex multi-hop questions [Su18], to algorithms that keep track of an ongoing conversational context [CSRW22b, KSRW21, LJ21, Sa18].

For implementing QA systems, one can identify a common set of basic KB functionalities that are very often necessary. For example, retrieving *all KB facts with a specific entity*, or computing the shortest KG path between two entities, are two such frequent needs.

Apart from QA, there is a range of other tasks/applications that benefit from quick and easy access to large KBs. Named entity recognition and detection (NERD) systems can be

[1] Max-Planck-Institut für Informatik & Saarland University, Saarbrücken, Deutschland, pchristm@mpi-inf.mpg.de
[2] Max-Planck-Institut für Informatik & Saarland University, Saarbrücken, Deutschland, rishiraj@mpi-inf.mpg.de
[3] Max-Planck-Institut für Informatik & Saarland University, Saarbrücken, Deutschland, weikum@mpi-inf.mpg.de

doi:10.18420/BTW2023-28

used for mapping entities in text to canonicalized objects, which can give insights about a text at hand [FS10, Ho11, Li20]. Another example use case is entity ranking [CD22], which features in search engines to show entity-centric information for queries like *"Angela Merkel"*. Further, information or statistics extracted from the KB can be leveraged for improving performance on downstream tasks: the distance between entities can serve as a proxy for their semantic similarity [ZI16], the frequency of an entity in the KB can be used for understanding how popular an entity is [Ch19], and the KB ontology can help in relation extraction [Ko14] or answer verification [BH15].

For implementing such methods, several shared KB functionalities are required: identifying the distance between two entities, retrieving the frequency of a KB item, retrieving the type of an entity, or computing the shortest path between two KB items.

Available interfaces to large KBs, with multiple terabytes of data, are often based on query languages like SPARQL. Such interfaces allow for a very general access to the KB with arbitrary complexity, and are heavily optimized for different query patterns and workloads [Fe13, Ur16]. However, implementing some of the basic operations mentioned above can lead to a high degree of query complexity, manual effort and efficiency overhead.

One key problem is that using existing interfaces for accessing KBs *requires deep knowledge and understanding of the respective KB schema*, which is different for every KB. Another problem that we identify is that the whole *KB storage and the corresponding query languages are optimized for the native RDF triple structure*. However, modern KBs store *n*-ary facts, using *reification* (e.g. via qualifier statements in Wikidata, or Compound Value Types (CVTs) in Freebase), going beyond the self-contained triples [HHK15]. In fact, one fourth of the facts in Wikidata provide additional information via such qualifier statements. Consider the real-life fact that Angela Merkel was chancellor of Germany, in triple structure:

⟨Angela Merkel, position held, Federal Chancellor of Germany⟩

From this fact, important contextual information is missing: she was chancellor in the past, from 2005 to 2021, she was the 8th German chancellor, she replaced Gerhard Schröder, etc. In triple structure, this additional context is represented by reification of the basic fact, and adding more triples that refer to the basic fact's identifier and express the contextual information. The single *n*-ary (compound) fact discussed above would be represented as:

⟨Angela Merkel, position held, fact-id⟩
⟨fact-id, position held, Federal Chancellor of Germany⟩
⟨fact-id, start time, 2005⟩
⟨fact-id, end time, 2021⟩
⟨fact-id, series ordinal, 8⟩
⟨fact-id, replaces, Gerhard Schröder⟩

Using traditional query interfaces, retrieving *all facts* with `Angela Merkel` from the knowledge base can be cumbersome and inefficient: queries with the entity as subject, object, and

qualifier-object of the fact are required. An example SPARQL query for detecting all facts with `Angela Merkel` as subject from Wikidata is shown below:

```
SELECT DISTINCT ?fact_id ?subject ?predicate ?object ?qual_pred ?qual_obj {
    VALUES (?subject) {(Angela Merkel)}
    ?subject ?p ?fact_id .
    ?fact_id ?ps ?object .
    ?predicate wikibase:claim ?p .
    ?predicate wikibase:statementProperty ?ps .
    OPTIONAL{
        ?fact_id ?pq ?qual_obj .
        ?qual_pred wikibase:qualifier ?pq
    }
}
```

Similar queries need to be run binding `Angela Merkel` to the object and qualifier-object position. Afterwards, the results need to be post-processed, joining all constituents for one fact-id. Overall, this causes quite a few KB interactions and processing overhead.

Further, certain KB concepts are not well-defined for such scenarios. How should this fact be represented in graphical form, where graphs of all facts would be overlaid to obtain a knowledge graph? Would `2009` belong to the 1-hop neighborhood of `Barack Obama`? What would be the KB distance between `Barack Obama` and `George W. Bush`? Should the shortest path include non-content like fact-ids?

We present a fact-centric definition of the KB, which represents KB facts as arbitrary-length lists, considering context information in qualifiers to be of comparable importance as that in the main triple[4]. This allows us to establish intuitive definitions for KB neighborhood, KB distance, and shortest paths between entities. Based on these definitions, we implement a fact-centric KB index, CLOCQKB, that allows efficient KB access to perform the functionalities described above. The resulting KB interface is made available to the community for accessing KBs more conveniently, both as a public API and open-source code repository.

Contributions. Our contributions in this work are as follows:

- Identifying and outline key problems with existing interfaces for common KB access needs in many NLP and IR use cases;
- Proposing an efficient solution to the problems with existing triple-centric interfaces, based on a fact-centric view of the KB;
- Proposing concise and unambiguous definitions for basic KB concepts;
- Making our code available, and release a public API that allows people to conveniently access Wikidata, without having to deal with multiple terabytes of data[5].

[4] This implements and extends our work in WSDM 2022 [CSRW22a].

[5] `https://clocq.mpi-inf.mpg.de`

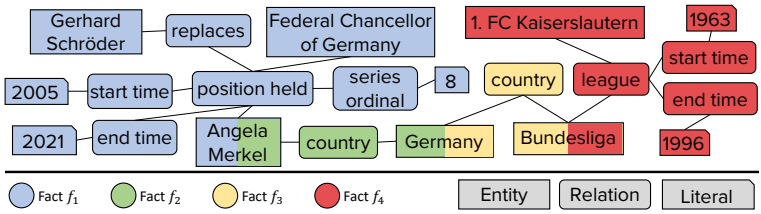

Fig. 1: Graph representation of a KB fact.

2 KB Index

Our primary design choice is to take a *fact-centric* view of the KB. Therefore, we treat the KB as a set of *facts*, instead of a set of triples. As a result, we also directly store and index the facts as a whole. With that, we avoid any complex query structures or post-hoc joining of separate fact constituents at runtime. This helps us to significantly improve the runtime efficiency of several important KB functionalities, like retrieving all KB facts with a specific KB item, or computing the distance between two KB items.

2.1 Concepts and definitions

Based on a fact-centric view of the KB, we establish the following definitions, answering the questions posed in Sec. 1.

Knowledge base. A *knowledge base K* is a set of KB facts.

KB fact. A *KB fact f* is a list of KB items, and expresses objective information or knowledge. It consists of the main triple, which has a subject, predicate, and object, and an optional set of qualifiers represented as ⟨qualifier-predicate, qualifier-object⟩ pairs. The subject is an entity, the predicate and qualifier-predicate are relations. The object is an entity, type or literal (date or string), and the qualifier-object can be an entity or literal. An example fact is ⟨`Angela Merkel, position held, Federal Chancellor of Germany; start time, 2005; end time, 2021; series ordinal, 8; replaces, Gerhard Schröder`⟩, which represents the same information as the compound fact in Sec. 1. A set of facts can be represented in a graph as illustrated in Fig. 1. In this representation, the qualifiers are connected to the predicate of a fact, describing the relation between the subject and object in more detail. Note also, that each instance of a relation becomes an individual node. If, for example, the two `country` nodes were merged, information would be lost: ⟨`Bundesliga, country, Angela Merkel`⟩ could be inferred as a fact then.

KB item. A *KB item x* is either an entity (`Angela Merkel`), relation (`position held`), type (`human`), or literal (`2021`).

Neighborhood. The 1-hop neighborhood N_f of an item x is the set of *all facts* in K with x: $N_f(x) = \{f | x \in f \wedge f \in K\}$. To generalize, the h-hop neighborhood $N_f^h(x)$ is then the union of all facts with any of the items in the (h-1)-hop neighborhood.

Neighbors. The 1-hop neighbors N_i of x are all KB items in the 1-hop neighborhood of x: $N_i(x) = \{x' | x' \in f \wedge f \in N_f(x)\}$. Analogously, the set of h-hop neighbors $N_i^h(x)$ is given by all KB items in the h-hop neighborhood.

Frequency. The frequency of a KB item is given by the size of its 1-hop neighborhood $|N_f(x)|$, i.e. the number of facts x appears in. Without loss of generality, the frequency can be measured w.r.t. a specific position in the fact (e.g. frequency in subject-position).

KB distance. The KB distance between two KB items x and y is h, if the items are h hops away from each other, i.e. $h = \min_{h'} x \in N_i^{h'}(y)$. For example, the distance of `Angela Merkel` and `Gerhard Schröder` is 1, since they appear in the same fact. The distance between `Angela Merkel` and `Bundesliga` is 2.

Shortest path. The shortest path between two KB items is given by the (set of) fact(s) in between. This ensures that important contextual information is not skipped, considering the facts as a whole. For example, if we used only the direct connection between `Angela Merkel` and `Federal Chancellor of Germany` in the graph, incorrect inferences could be made. Instead, the whole fact f_1 is the shortest path. The shortest path between `Angela Merkel` and `Bundesliga` would be $f_2 \circ f_3$, where $\circ$ denotes concatenation.

2.2 Implementation

For improving space efficiency, each KB item x is first integer-encoded as $int(x)$ [Fe13, Ur16], and we create mappings from $x \rightarrow int(x)$ and $int(x) \rightarrow x$. Each fact $f = \langle x_0, x_1, \ldots \rangle$ is then stored as $int(f) = \langle int(x_0), int(x_1), \ldots \rangle$. For facilitating our computations, we index the following information for each item in the KB: the 1-hop neighborhood $N_f(x)$ and the set of 1-hop neighbors $N_i(x)$, both in an integer-encoded manner. This allows us to compute the most important functionalities via simple lookups or set operations at runtime, improving efficiency. Details on the specific implementations of individual functionalities can be found in Sec. 3.

The neighborhoods for each item x are stored in two lists, holding pointers to the individual facts: one list for facts with x in subject-position, and one for all other facts with x. This will be useful later, for retrieving only salient facts with a KB item. The neighborhood lists of x are then stored in position $int(x)$ in the index, which is implemented by another list.

The neighbors of an item x are stored as a set, which is again stored in position $int(x)$ in the corresponding index list. Another possibility would be to extract the set of neighbors from the 1-hop neighborhood at runtime. This reduces the memory footprint, but increases the costs for computing KB distances.

3 Functionalities

In this section, we will describe the functionalities provided with our KB interface in more detail, elaborating some of the implementation details. All functions are implemented for Wikidata, the largest public KB that is actively maintained. However, the principles would still hold for other large curated KBs like DBpedia as well.

3.1 Direct lookups

Label. To avoid collisions due to duplicate labels, KBs typically use identifiers for representing KB items. For example, `Angela Merkel` is stored as `Q567` in Wikidata, and `position held` as `P39` (for simplicity, we use labels to refer to items in this paper). We provide a simple function to look up the (English) label for a provided Wikidata ID.

Aliases. Another useful information on a KB item are aliases. These are alternative labels that one can use for the same KB item. For example, *"CR7"* is an alias of `Cristiano Ronaldo`, and *"office held"* is an alias of `position held`. Such aliases can be used for improving relation extraction [Ba21, Va18] or NERD systems [BOM15]. Relation aliases can also be used for training crisp paraphrase models.

Description. Further, Wikidata stores a crisp description for each KB item, which can e.g. be helpful for deriving latent representations of an item [GSR17].

Types. Another important information on an entity are the KB types. Typical use cases are entity linking [GSR17], answer verification [BH15] in QA systems, or enhanced efficiency [Zi17] of QA systems. For example, `Angela Merkel` is a `human`, and Germany is associated with the types `sovereign state`, `republic`, and `country`. To enhance the expressiveness of the types, we add the occupations of a human, which are not stored as types in Wikidata. E.g. `Angela Merkel` would also be associated with `politician` or `physicist`.

Most frequent type. In some use cases, having exactly one type for an entity is desirable [CSRW22b]. Unlike the deprecated KB Freebase, Wikidata does not indicate the most notable type of an entity. As a proxy, we use the most frequent type of an entity in the KB.

3.2 More complex functionalities

1**-hop neighborhood**. This function is used for retrieving the 1-hop neighborhood of an item, which is a frequent use case in QA [Ch19, SRA22, Su18], but can also be useful for other applications like KB completion [Ba19] or entity alignment [Su20]. For implementing this function, we can simply look up the neighborhood in our fact-centric KB index. We further implement a mechanism to retrieve the more salient facts for a specific KB item: frequent KB items like `Germany` can easily have millions of facts in their neighborhood. However, retrieving all these facts is often not desired in IR or NLP applications [SRA22].

A parameter p is used to control the amount of facts returned as follows: if there are more than p facts with x in the object or qualifier-object position, then these facts are dropped, i.e. only facts with x in the subject-position are kept. This can also help improve the efficiency of downstream applications: the I/O time can be drastically reduced, and only a subset of more salient facts needs to be processed. The output of this function is a set of KB facts.

Frequency. For computing the frequency, we count the number of facts with x as subject, and the number of facts with x not in subject position, using the neighborhood index. The output of this function are the two resulting counts.

Connectivity. We can also compute a connectivity score for two KB items. Note that when retrieving the 3 or 4-hop neighborhood of an item, this will give almost the entire KB. Therefore, we define a connectivity score of two KB items, which we found more useful than a standard distance function in practice [CSRW22a]. The connectivity is 1 if the items have a KB distance of 1, 0.5 if they have a distance of 2, and 0 if they are not connected within 2 hops. The function can be efficiently implemented using basic set operations:

$$connectivity(x, y) = \begin{cases} 1 & \text{if } x \in N_i(y) \vee y \in N_i(x) \\ 0.5 & \text{if } N_i(x) \cap N_i(y) \neq \emptyset \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

Note that the 1-hop neighbors $N_i(x)$ and $N_i(y)$ are simply looked up in the index.

Shortest path. Another often-needed KB functionality is obtaining the shortest path between two items, which can be used to analyze their semantic relationship [JJ19], obtaining training data for QA [Su18], or for connecting multiple subgraphs [Pr21]. The implementation makes use of the connectivity function outlined above: if two items x, y have a connectivity of 1, we search the smaller of the two 1-hop neighborhoods for a fact with both items. For a connectivity of 0.5, we identify the *"items in the middle"* $\{m_i\}$, i.e. the set of items that are connected with x in 1 hop and with y in 1 hop. For these items $\{m_i\}$, we search for the joint facts with x and y, respectively. The output of this function is a set of facts (connectivity = 0.5), or a set of 2-hop paths, one for each m_i, where each such path is given by a set of facts connecting x and m_i, and a set of facts connecting m_i and y (connectivity = 1).

3.3 Search Space Reduction

Finally, we briefly introduce the key algorithm introduced in our recent work [CSRW22a], which retrieves a set of relevant KB facts for a given question. This is typically the first step in QA systems [CSRW22b, SRA22, Su18] for reducing the search space. Consider an input like *"Who scored the final goal for Germany in 2014?"*. For a human interested in football it is obvious that *"Germany"* refers to the `German national football team`, and that the question is on the `2014 FIFA World Cup`. However, this only becomes clear when considering the question as a whole, taking the respective context into account. Therefore, mapping the question words to KB items is a non-trivial task.

We propose a method that first retrieves a set of candidate disambiguations for each question word. These candidate disambiguations are then scored based on four different signals. We use the word-level scores for term matching and question relatedness. Further, to understand the text as a whole, we also consider the connectivity and coherence among disambiguations. Note that certain question words may be more ambiguous than others, and providing additional candidate disambiguations for such question words could alleviate potential errors. Therefore, we dynamically adjust the number of disambiguations k for each individual question word. The algorithm provides the disambiguations, and facts with the disambiguated items, as the output.

Entity and relation linking. Based on this algorithm, we also make functionalities for entity linking and relation linking available. The idea is to prune less relevant entities or relations from the disambiguations provided by the search space reduction method outlined above. These functions have been developed using data from the SMART 2022 Task[6].

4 Experiments

In this section, we conduct experiments to compare the efficiency of our fact-centric KB interface with existing triple-centric ones on a set of important KB functionalities.

4.1 Experimental Setup

The ClocqKB is initiated with a Wikidata dump from 31 January 2022, that has been cleaned for removing information irrelevant for most common tasks[7]. We show runtimes of a locally running version of ClocqKB, a version without the neighbors' index (ClocqKB w/o N_i), and the publicly available ClocqKB-API[8].

We compare our fact-centric KB inferface with two triple-centric ones: Hdt [Fe13] which implements fast triple lookups based on efficient bitmap encodings, and the publicly available QueryService[9] provided by Wikidata. For Hdt, we used the latest dump made available (3 March 2021)[10], and run it on our local machine. Note that it is highly non-trivial to perform the cleaning steps (mentioned above) for the HDT dump, since it comes in a highly compressed form. While function results may therefore not be exactly the same, the runtimes are still comparable. All experiments with the QueryService are conducted in September 2022. Note that we measure all network latencies (if any), and subtract these from the measured timings for fair comparisons. The local experiments are run on an AMD EPYC 7702 Processor, and the public API is running on an AMD EPYC 7302P Processor.

[6] https://smart-task.github.io/2022/
[7] https://github.com/PhilippChr/wikidata-core-for-QA
[8] https://clocq.mpi-inf.mpg.de
[9] https://query.wikidata.org
[10] https://www.rdfhdt.org/datasets/

KB interface	HDT [Fe13]	CLOCQKB	CLOCQKB w/o N_i	QUERYSERVICE	CLOCQKB-API
RAM consumed	150 GB	470 GB	360 GB	–	–
Neighborhood	1.21 s	5.99×10^{-5} s	6.01×10^{-5} s	0.561 s	4.36×10^{-3} s
Frequency	3.12×10^{-2} s	1.02×10^{-5} s	1.15×10^{-5} s	0.122 s	4.24×10^{-3} s
Connectivity	0.802 s	1.83×10^{-5} s	5.02×10^{-5} s	1.11 s	4.22×10^{-3} s
Shortest path	3,046 s	0.553 s	15.8 s	1.18 s	0.591 s

Tab. 1: Large-scale runtime efficiency analysis of KB interfaces.

4.2 Large-scale efficiency analysis

We first compare the efficiency of the KB interfaces for the following four functionalities: i) retrieving the 1-hop **neighborhood**, ii) computing the **frequency**, iii) computing the **connectivity**, and iv) identifying the **shortest path**. We restrict our experiments on the more complex functionalities (Sec. 3.2) here. However, even for accessing the simpler functionalities (Sec. 3.1), deep knowledge of the KB schema is required for the baselines.

As input, we use 10,000 random item (pairs when applicable) for each functionality, using the same random seed for all interfaces. We only used 100 random pairs for shortest path, due to substantially higher runtimes. In case an error is thrown for a function call, which might sometimes occur for the QUERYSERVICE, the corresponding instance is dropped from the analysis for all methods. The results are shown in Table 1.

The key observations are as follows. Due to its smart KB indices, CLOCQKB can achieve extremely low runtimes for retrieving 1-hop neighborhoods, frequencies, or performing connectivity checks. Removing the neighbors' index only has a minor effect on the connectivity check, while the effect on the shortest path functionality is notable. The runtime of the triple-centric baselines is higher by a factor of 10^3 to 10^5 than that of CLOCQKB in most scenarios, indicating the runtime benefits of a fact-centric approach for such use cases. The public CLOCQKB-API has higher runtimes than the version running locally, due to a different processor and some I/O overhead, but is still substantially faster than the baselines.

Note that CLOCQKB consumes quite some RAM. One could store the indexes in a dedicated database to save memory. However, in an industrial use case, ~500 GB RAM is not a major concern, and for scientific use cases the CLOCQKB-API can be used.

4.3 Anecdotal examples

Results in Table 1 are for a random sample, and can show trends among the different KB interfaces. We also conducted experiments on a set of popular items that may often appear in a real application. We chose the following KB items: `Angela Merkel, Germany, Bundesliga`, and run the same functionalities on these items. Results can be seen in Table 2. Time-outs are indicated by "n/a".

KB interface	HDT [Fe13]	CLOCQKB	QUERYSERVICE	CLOCQKB-API
Neighborhood(`Angela Merkel`)	20.8 s	2.55×10^{-3} s	2.12 s	1.07×10^{-2} s
Neighborhood(`Germany`)	2,990 s	2.73 s	"n/a"	15.6 s
Neighborhood(`Bundesliga`)	15.2 s	1.10×10^{-2} s	"n/a"	3.56×10^{-2} s
Frequency(`Angela Merkel`)	2.85×10^{-2} s	2.55×10^{-5} s	0.186 s	5.34×10^{-3} s
Frequency(`Germany`)	5.20×10^{-5} s	2.56×10^{-5} s	0.280 s	5.39×10^{-3} s
Frequency(`Bundesliga`)	5.20×10^{-5} s	2.47×10^{-5} s	8.33×10^{-2} s	5.44×10^{-3} s
Connectivity(`Angela Merkel`, `Germany`)	61.3 s	3.48×10^{-5} s	"n/a"	5.37×10^{-3} s
Connectivity(`Germany`, `Bundesliga`)	60.3 s	3.27×10^{-5} s	"n/a"	5.21×10^{-3} s
Connectivity(`Angela Merkel`, `Bundesliga`)	0.328 s	8.28×10^{-4} s	"n/a"	5.10×10^{-3} s
Shortest path(`Angela Merkel`, `Germany`)	118 s	7.80×10^{-2} s	"n/a"	8.42×10^{-2} s
Shortest path(`Germany`, `Bundesliga`)	120 s	8.10×10^{-2} s	"n/a"	8.89×10^{-2} s
Shortest path(`Angela Merkel`, `Bundesliga`)	5,260 s	0.156 s	"n/a"	0.178 s

Tab. 2: Runtime efficiency analysis on manually chosen function calls for popular entities.

The results reveal that the public QUERYSERVICE cannot cope with use cases in which larger intermediate results are obtained (like for `Germany` or even `Bundesliga`). Most such runs faced server-side time-outs. HDT obtains results for all experiments, but can take quite some time to do so: runtimes > 1 s are rarely acceptable in IR or NLP applications. While runtimes for CLOCQKB can get higher in extreme cases, like for retrieving the 1-hop neighborhood of Germany with more than 1.4 million facts, the runtimes remain tractable, indicating runtime benefits of 10^3 to 10^5 of our fact-centric approach in most cases.

5 Related Work

There has been substantial work on optimizing query performance on KBs [EM10, NW08, Ur16]. Jacobs [UJ20] proposed TRIDENT, for enabling different kinds of workloads (e.g. SPARQL, graph analytics) on large KBs. HDT [Fe13] is an efficient representation of RDF data, that is both space and runtime efficient. The individual triples of the KB are encoded using bitmaps. Two integer-streams holding all predicates and objects for a fixed subject are established. The relations between these predicates and objects are encoded using bit-streams. Using several indexes, HDT can search for triple pattern very efficiently. However, there is no dedicated mechanism for querying *n*-ary facts.

While the proposed approaches show high performance on traditional SPARQL(-like) queries, we showed through comparison that important KB functionalities can be more efficiently implemented taking a fact-centric view of the KB.

6 Conclusion

We present a fact-centric view of the KB, and based on this, provide simple and efficient implementations of several important KB functionalities. These are either not available in existing implementations, or would be quite complex to use, especially for new users. Experiments show that we outperform existing KB interfaces w.r.t. runtime efficiency. We make our code available, and provide a public API to enhance KB applications and research.

Bibliography

[Ab18] Abujabal, Abdalghani; Saha Roy, Rishiraj; Yahya, Mohamed; Weikum, Gerhard: Never-ending learning for open-domain question answering over knowledge bases. In: WWW. pp. 1053–1062, 2018.

[Au07] Auer, Sören; Bizer, Christian; Kobilarov, Georgi; Lehmann, Jens; Cyganiak, Richard; Ives, Zachary: DBpedia: A nucleus for a Web of open data. In: The Semantic Web. pp. 722–735, 2007.

[Ba19] Bansal, Trapit; Juan, Da-Cheng; Ravi, Sujith; McCallum, Andrew: A2N: Attending to neighbors for knowledge graph inference. In: ACL. pp. 4387–4392, 2019.

[Ba21] Bastos, Anson; Nadgeri, Abhishek; Singh, Kuldeep; Mulang, Isaiah Onando; Shekarpour, Saeedeh; Hoffart, Johannes; Kaul, Manohar: RECON: relation extraction using knowledge graph context in a graph neural network. In: WWW. pp. 1673–1685, 2021.

[Be13] Berant, Jonathan; Chou, Andrew; Frostig, Roy; Liang, Percy: Semantic parsing on freebase from question-answer pairs. In: EMNLP. pp. 1533–1544, 2013.

[BH15] Bast, Hannah; Haussmann, Elmar: More accurate question answering on freebase. In: CIKM. pp. 1431–1440, 2015.

[Bo08] Bollacker, Kurt; Evans, Colin; Paritosh, Praveen; Sturge, Tim; Taylor, Jamie: Freebase: A collaboratively created graph database for structuring human knowledge. In: SIGMOD. pp. 1247–1250, 2008.

[BOM15] Blanco, Roi; Ottaviano, Giuseppe; Meij, Edgar: Fast and space-efficient entity linking for queries. In: WSDM. pp. 179–188, 2015.

[CD22] Chatterjee, Shubham; Dietz, Laura: BERT-ER: Query-specific BERT Entity Representations for Entity Ranking. In: SIGIR. pp. 1466–1477, 2022.

[Ch19] Christmann, Philipp; Saha Roy, Rishiraj; Abujabal, Abdalghani; Singh, Jyotsna; Weikum, Gerhard: Look before you hop: Conversational question answering over knowledge graphs using judicious context expansion. In: CIKM. pp. 729–738, 2019.

[CSRW22a] Christmann, Philipp; Saha Roy, Rishiraj; Weikum, Gerhard: Beyond NED: Fast and Effective Search Space Reduction for Complex Question Answering over Knowledge Bases. In: WSDM. pp. 172–180, 2022.

[CSRW22b] Christmann, Philipp; Saha Roy, Rishiraj; Weikum, Gerhard: Conversational Question Answering on Heterogeneous Sources. In: SIGIR. pp. 144–154, 2022.

[EM10] Erling, Orri; Mikhailov, Ivan: Virtuoso: RDF support in a native RDBMS. In: Semantic Web Information Management. pp. 501–519, 2010.

[Fe13] Fernández, Javier D; Martínez-Prieto, Miguel A; Gutiérrez, Claudio; Polleres, Axel; Arias, Mario: Binary RDF representation for publication and exchange (HDT). In: Journal of Web Semantics. pp. 22–41, 2013.

[FS10] Ferragina, Paolo; Scaiella, Ugo: TAGME: On-the-fly annotation of short text fragments (by Wikipedia entities). In: CIKM. pp. 1625–1628, 2010.

[GSR17] Gupta, Nitish; Singh, Sameer; Roth, Dan: Entity linking via joint encoding of types, descriptions, and context. In: EMNLP. pp. 2681–2690, 2017.

[HHK15] Hernández, Daniel; Hogan, Aidan; Krötzsch, Markus: Reifying RDF: What Works Well With Wikidata? In: SSWS. p. 32, 2015.

[Ho11] Hoffart, Johannes; Yosef, Mohamed Amir; Bordino, Ilaria; Fürstenau, Hagen; Pinkal, Manfred; Spaniol, Marc; Taneva, Bilyana; Thater, Stefan; Weikum, Gerhard: Robust Disambiguation of Named Entities in Text. In: EMNLP. pp. 782–792, 2011.

[JJ19] Joseph, Kevin; Jiang, Hui: Content based news recommendation via shortest entity distance over knowledge graphs. In: WWW. pp. 690–699, 2019.

[Ko14] Koch, Mitchell; Gilmer, John; Soderland, Stephen; Weld, Daniel S: Type-aware distantly supervised relation extraction with linked arguments. In: EMNLP. pp. 1891–1901, 2014.

[KSRW21] Kaiser, Magdalena; Saha Roy, Rishiraj; Weikum, Gerhard: Reinforcement Learning from Reformulations in Conversational Question Answering over Knowledge Graphs. In: SIGIR. pp. 459–469, 2021.

[Li20] Li, Belinda Z.; Min, Sewon; Iyer, Srinivasan; Mehdad, Yashar; Yih, Wen-tau: Efficient One-Pass End-to-End Entity Linking for Questions. In: EMNLP. pp. 6433–6441, 2020.

[LJ21] Lan, Yunshi; Jiang, Jing: Modeling transitions of focal entities for conversational knowledge base question answering. In: ACL-IJCNLP. pp. 3288–3297, 2021.

[NW08] Neumann, Thomas; Weikum, Gerhard: RDF-3X: a RISC-style engine for RDF. Proceedings of the VLDB Endowment, pp. 647–659, 2008.

[Pr21] Pramanik, Soumajit; Alabi, Jesujoba; Saha Roy, Rishiraj; Weikum, Gerhard: UNIQORN: unified question answering over RDF knowledge graphs and natural language text. In: arXiv. 2021.

[Sa18] Saha, Amrita; Pahuja, Vardaan; Khapra, Mitesh; Sankaranarayanan, Karthik; Chandar, Sarath: Complex sequential question answering: Towards learning to converse over linked question answer pairs with a knowledge graph. In: AAAI. pp. 705–713, 2018.

[SKW07] Suchanek, Fabian M; Kasneci, Gjergji; Weikum, Gerhard: YAGO: A core of semantic knowledge. In: WWW. pp. 697–706, 2007.

[SRA22] Saha Roy, Rishiraj; Anand, Avishek: Question Answering for the Curated Web: Tasks and Methods in QA over Knowledge Bases and Text Collections. Springer, 2022.

[Su18] Sun, Haitian; Dhingra, Bhuwan; Zaheer, Manzil; Mazaitis, Kathryn; Salakhutdinov, Ruslan; Cohen, William: Open Domain Question Answering Using Early Fusion of Knowledge Bases and Text. In: EMNLP. pp. 4231–4242, 2018.

[Su20] Sun, Zequn; Wang, Chengming; Hu, Wei; Chen, Muhao; Dai, Jian; Zhang, Wei; Qu, Yuzhong: Knowledge graph alignment network with gated multi-hop neighborhood aggregation. In: AAAI. pp. 222–229, 2020.

[UJ20] Urbani, Jacopo; Jacobs, Ceriel: Adaptive Low-level Storage of Very Large Knowledge Graphs. In: WWW. pp. 1761–1772, 2020.

[Ur16] Urbani, Jacopo; Dutta, Sourav; Gurajada, Sairam; Weikum, Gerhard: KOGNAC: Efficient encoding of large knowledge graphs. In: IJCAI. pp. 3896–3902, 2016.

[Va18] Vashishth, Shikhar; Joshi, Rishabh; Prayaga, Sai Suman; Bhattacharyya, Chiranjib; Talukdar, Partha: Reside: Improving distantly-supervised neural relation extraction using side information. In: EMNLP. pp. 1257–1266, 2018.

[VK14] Vrandečić, Denny; Krötzsch, Markus: Wikidata: A free collaborative knowledgebase. In: CACM. pp. 78–85, 2014.

[ZI16] Zhu, Ganggao; Iglesias, Carlos A: Computing semantic similarity of concepts in knowledge graphs. In: IEEE TKDE. pp. 72–85, 2016.

[Zi17] Ziegler, David; Abujabal, Abdalghani; Roy, Rishiraj Saha; Weikum, Gerhard: Efficiency-aware Answering of Compositional Questions using Answer Type Prediction. In: IJCNLP. pp. 222–227, 2017.