Beyond NED: Fast and Effective Search Space Reduction for Complex Question Answering over Knowledge Bases

Philipp Christmann
Max Planck Institute for Informatics
Saarland Informatics Campus
Saarbruecken, Germany
pchristm@mpi-inf.mpg.de

Rishiraj Saha Roy
Max Planck Institute for Informatics
Saarland Informatics Campus
Saarbruecken, Germany
rishiraj@mpi-inf.mpg.de

Gerhard Weikum
Max Planck Institute for Informatics
Saarland Informatics Campus
Saarbruecken, Germany
weikum@mpi-inf.mpg.de

ABSTRACT

Answering complex questions over knowledge bases (KB-QA) faces huge input data with billions of facts, involving millions of entities and thousands of predicates. For efficiency, QA systems first reduce the answer search space by identifying a set of facts that is likely to contain all answers and relevant cues. The most common technique for doing this is to apply named entity disambiguation (NED) on the question, and retrieve KB facts for the disambiguated entities. This work presents CLOCQ, an efficient method that prunes irrelevant parts of the search space using KB-aware signals. CLOCQ uses a top-k query processor over score-ordered lists of KB items that combine signals about lexical matching, relevance to the question, coherence among candidate items, and connectivity in the KB graph. Experiments with two recent QA benchmarks for complex questions demonstrate the superiority of CLOCQ over state-of-the-art baselines with respect to answer presence, size of the search space, and runtimes.

CCS CONCEPTS

• **Information systems** → *Question answering*.

KEYWORDS

Question Answering, Knowledge Bases, Entity Linking

1 INTRODUCTION

Motivation. Large knowledge bases (KBs) like Wikidata [54], DBpedia [4], YAGO [47], and Freebase [10] are ideal sources for answering factual questions that have crisp entities as answers [1, 6, 8, 27, 41, 57, 60, 60]. Such KBs are comprised of *facts*, structured as `<subject, predicate, object>` triples, often augmented with

qualifier predicates and objects for context [20, 24, 30, 37]. Answering complex questions with multiple entities and predicates is one of the most actively researched topics in question answering over knowledge bases (KB-QA) today [9, 23, 39, 43, 52], and this is the setting for this work. Systems answering such complex questions either build explicit structured queries [9, 13, 33], or perform approximate graph search [32, 48, 52] to arrive at the answer. To this end, systems learn models for mapping question words to KB items, where the huge size of the KB poses a stiff challenge. Concretely, whole KBs are often more than 2 Terabytes in size: this makes the development of QA systems over them a rather daunting task. Most KB-QA systems thus prune the search space for candidate answers using Named Entity Disambiguation (NED).

Limitations of the state-of-the-art. NED methods [15, 19, 25, 31, 46, 53] map mentions in questions (single words or short phrases) to KB entities, and the QA system subsequently uses only the facts containing these entities as its *search space* for locating the answer(s). However, general-purpose NED tools have major limitations in this context: i) they are not tailored for downstream use by KB-QA systems; ii) they usually disambiguate only named entities and disregard words and phrases that denote general concepts, types or predicates; and iii) they typically output merely the top-1 entity per mention, missing out on further candidates that can serve as relevant cues. Even methods designed for short input texts, like TAGME [19] and ELQ [31], have such limitations.

Approach. To address these concerns, we propose CLOCQ (Contracting answer spaces with Lists and top-k Operators for Complex QA, pronounced *"Clock"*), a time- and memory-efficient method that operates over *all KB items* to produce top-k candidates for entities, types, concepts and predicates. Consider the complex question on the FIFA Football World Cup 2018:

> *Who scored in the 2018 final between France and Croatia?*

Most systems for complex KB-QA tackle the answering process in two phases. First, they disambiguate question tokens to KB entities. These entities *establish a reduced search space* for the QA system, that can either be an *explicit* set of facts containing these KB entities [32, 48, 49, 52], or involve *implicit* grounding to a small zone in the KB via structured queries containing these entities [9, 29, 57, 60]. Second, depending upon the approach in the first phase, KB-QA systems either search for the answer in the retrieved facts, or build a complex query in SPARQL-like syntax that would return the answer when executed [43]. CLOCQ tries to improve the effectiveness and the efficiency of the first phase above. The output of CLOCQ is thus a small set of disambiguated KB items and facts containing these items, and this is fed into the second phase. Answer presence in the

Table 1: Notation for salient concepts in Clocq.

Notation	Concept
K	Knowledge base
x	KB item
$\langle s, p, o, qp_1, qo_1, \ldots qo_r \rangle$	Fact in K
$NF(x)$	1-hop neighborhood of x (set of facts)
$NI(x)$	1-hop neighbors of x (set of items)
$q = \langle q_1 \ldots q_m \rangle$	Question, cue words in question
s_j	Scoring signal j
l_{ij}	Score-ordered KB-item list for q_i and s_j
$\mathcal{S}(q)$	Search space of facts for question q

KB subspace inherently sets an upper bound on the performance of the downstream KB-QA system, making fast and effective search space reduction a vital step in the QA pipeline.

Method. Clocq first builds inverted lists of KB items per question word with *term matching scores* based on TF-IDF. Top-ranked items from these lists, up to a certain depth, are then scored and ranked by a combination of *global signals*, like semantic coherence between items and connectivity in the KB graph, and *local signals* like relatedness to the question and term-matching score. These scoring signals are computed at question time: this is made feasible with Clocq's novel KB representation and storage model, that substantially speeds up lookups with respect to existing solutions. The threshold algorithm (TA) [17] is applied for extracting the top-k candidates for each question term separately. Since it may not always be obvious how to choose k for every term, we provide an entropy-based mechanism for making this choice automatically. The union of the per-term top-k items forms a pool of relevant KB items, and their KB facts is the output of Clocq that would be passed on to the answering phase of a KB-QA system. Experiments with two recent KB-QA benchmarks and a suite of NED-based competitors [15, 19, 25, 31, 53] show the benefits of Clocq: it obtains the highest *answer presence* in the retained subset of the KB, with tractable search space size and sub-second runtimes. We show proof-of-concept of Clocq's impact on KB-QA by feeding the output of Clocq into the popular QA system GRAFT-Net [49], and obtain significant boosts in answering performance.

Contributions. We make the following salient contributions:

- identifying *answer search space reduction* as a critical task in KB-QA pipelines;
- proposing the Clocq method for computing answer-containing KB subsets with scored lists and the threshold algorithm;
- conducting extensive experiments that show the superiority of Clocq over a number of baselines using NED;
- devising a novel KB indexing scheme that is shown to notably improve runtimes for all methods, including baselines;
- releasing the complete Clocq code that any QA system developer can use for quickly exploring algorithms over much smaller KB subsets at **https://clocq.mpi-inf.mpg.de/**.

2 CONCEPTS AND NOTATION

We now introduce concepts necessary for understanding Clocq.
Knowledge base. A knowledge base K is a compilation of facts.
Fact. A fact is a <subject, predicate, object> triple, that is optionally augmented by <qualifier predicate, qualifier object>

Figure 1: Fact-based definition of KB neighborhoods.

pairs which specify context information for the main triple. For example, <2018 FIFA World Cup Final, participating team, France national football team; location, Luzhniki Stadium; point in time, 15 July 2018> is such a fact, where the first three items constitute the main triple, and the last four make up two <qualifier predicate, qualifier object> pairs. Subjects of facts are entities, while objects are entities, types or literals. Predicates denote relationships between the other categories.

KB item. We refer to entities (2018 FIFA World Cup Final), predicates (participating team), types (footballer) and literals (15 July 2018) as *KB items x*.

1-hop neighborhood. This is defined as $NF(x)$ of a KB item x and is given by all facts in which x occurs. The set of KB items $NI(x)$ in the 1-hop neighborhood of x is termed as its *1-hop neighbors*.

Question. A question q is specified by a sequence of keywords $\langle q_1, q_2, \ldots q_m \rangle$, where stopwords are not considered. For our running example question *Who scored in the 2018 final between France and Croatia?*, we would have q = <scored, 2018, final, france, croatia>. Without loss of generality, q_i may also be a phrase (*"2018 final"*).

Answer. An answer $\{a\}$ to q is a small set of KB entities or literals that satisfy the intent in q ({Paul Pogba, Ivan Perisic, ... }).

Score-ordered list. These are lists $\{l\}$ that hold KB items $\{x\}$, sorted in descending order of some relevance score. Depending upon the situation, we can have one list l_i per question term q_i, or one list l_{ij} per score s_j per q_i.

Search space. A search space $\mathcal{S}(q)$ for a question q is a set of facts $\mathcal{S}(q) \subseteq K$, that is expected to contain each $\{a\}$. For example, {⟨ 2018 FIFA World Cup Final, goal scored by, **Paul Pogba**; for team, France football team ⟩, ⟨ 2018 FIFA World Cup Final, goal scored by, **Ivan Perisic**; for team, Croatia football team ⟩, ...} comprise a search space for the running example question, where the answers are shown in **bold**.

3 KB REPRESENTATION AND STORAGE

One of the recurrent requirements in QA and specifically in answer search space reduction is to retrieve the facts of a given KB item (like entities returned by a NED system). Existing KBs are stored as collections of RDF triples. One can query these triple stores in SPARQL-like languages: however, the functionality of fact-retrieval is not built-in, and getting all facts of a single item may often entail issuing a substantial volume of queries (explained later). The
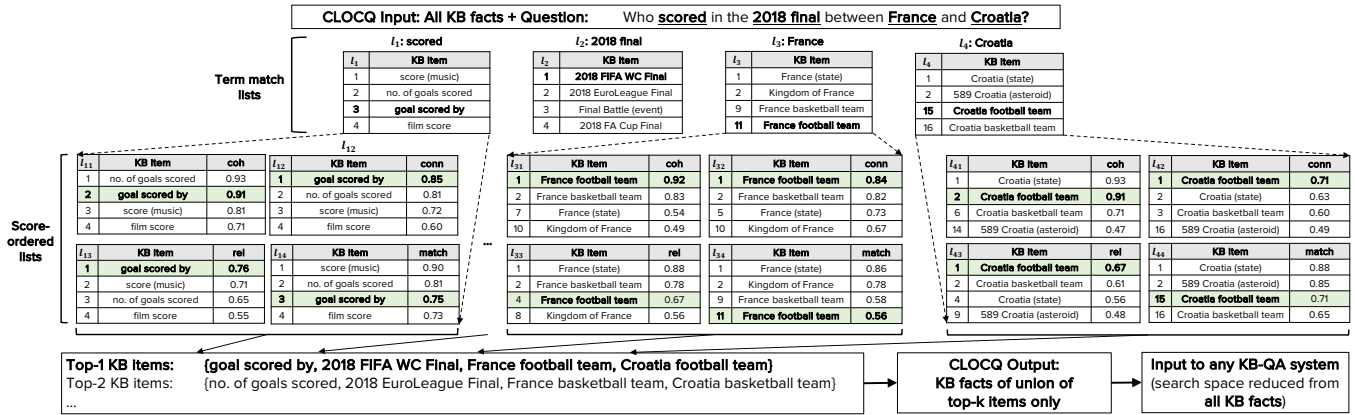
Figure 2: Illustrating the workflow of CLOCQ for our running example. Input of CLOCQ: Question + All KB facts; Output of CLOCQ: Disambiguated question-relevant KB items + Only KB facts with these items.

consequence is that the total time taken for this step can often be too high, and this is detrimental to any system that relies on these retrieval results. As a result, we devise our own KB representation and storage, that are detailed in this section.

Concerns with a triple-based KB view. In standard triple stores, facts containing qualifiers are stored in a *reified* form. Qualifiers are conceptually modeled as <qualifier predicate, qualifier object> pairs that are appended to the main triple. However, this is not amenable to store in a uniform triple store. Reification is a technical trick that stitches the main triple with its qualifiers using fact-specific identifiers, and at the same time achieves a "triplified" format for all nuggets of information. For example, the single fact <2018 FIFA World Cup Final, participating team, France football team; location, Luzhniki Stadium; point in time, 15 July 2018> in reified form would be represented as a set of four triples: <2018 FIFA World Cup Final, participating team, fact-id>, <fact-id, participating team, France football team>, <fact-id, location, Luzhniki Stadium>, <fact-id, point in time, 15 July 2018>.

Joining reified triples into their original facts is more amenable to downstream use. However, such an aggregation requires the execution of thousands of structured queries over the KB (equivalently, matching a large number of triple patterns). For example, querying for the triples of France football team with this item in the object position will only match the second reified triple above; the whole fact needs to be reconstituted using sequential lookups. Moreover, this needs to be done for every reified fact that the KB item is a part of, which are often several thousands, and additional lookups are also necessary to get facts with the item as a subject.

A fact-based view as a solution. This motivates us to adopt a fact-based view of the KB, that we instantiate as follows. We start with a standard RDF triple dump. We aggregate all reified triples by their fact-id upfront, remove the respective fact-ids, and postprocess them to the form shown in Table 1 (Row 3). Two different indexes are then established: one stores the 1-hop neighborhood of every item ($x \mapsto NF(x)$), and the other stores the set of 1-hop neighbors of each KB item ($x \mapsto NI(x)$). Instead of using long alphanumeric strings that are typical in most raw dumps, KB items are integer-encoded [18, 50]. To reduce memory footprint, both indexes use appropriate pointers inside their representations. The final set of facts thus obtained is referred to as our KB K.

With a fact-based indexing, at runtime, the 1-hop neighborhood of an item can simply be *looked up*, eliminating the need for expensive querying or joining. Further, the index of 1-hop neighbors allows for very fast computation of KB distances: two KB items x_1, x_2 are within 1-hop distance if $x_1 \in NI(x_2)$, and in 2-hop distance if $NI(x_1) \bigcup NI(x_2) \neq \emptyset$ (verified via set-overlap tests). This proves decisive for connectivity checks later on (Sec. 4.2).

Additional benefits of a fact-based view. When a postprocessed fact is directly modeled as a graph (Fig. 1 top), traditional distance conventions in graphs would imply that even KB items that are part of the same fact may be at a high distance of three (France football team and 2018 FIFA World Cup Final). Distances to KB items in connected facts may be even higher, like five (France football team and Moscow). 1-hop and 2-hop neighborhoods are vital intuitions of close proximity in KB-QA and such arbitrary distance conventions are far from ideal. In a fact-centric view, France football team and 2018 FIFA World Cup Final are now at a distance of 1, while France football team and Moscow are 2 hops apart (Fig. 1 bottom): this is more practical in terms of several KB-related applications. Our approach lifts qualifiers to first-class citizens, this way enhancing the expressiveness of the QA method within limited neighborhoods.

The concept of a KB neighborhood in the literature is primarily entity-centric. An ideal representation should enable definitions that uniformly apply to entities, predicates, types and literals. Predicates are often modeled as edge labels, and this precludes a seamless notion of neighborhood. A fact-based neighborhood can easily be envisioned for all types of KB items.

4 THE CLOCQ METHOD

We now explain the complete CLOCQ workflow (illustrated in Fig. 2).

4.1 Retrieving candidate KB items per term

Creating term match lists. Consider our running example question: *Who scored in the 2018 final between France and Croatia?* As our goal is to disambiguate keywords or phrases in the question (*"scored"*, *"2018 final"*, *"France"*, *"Croatia"*) to items in a KB, we first collect candidates from the KB using a standard lexical matching score (like TF-IDF or BM25) for each question term $q_1 \dots q_m$ ($m = 4$ in our example, stopwords are dropped). Here

q_i is analogous to a search query, while each item x in the KB resembles a document in a corpus. This "document" is created by concatenating the item label with textual aliases and descriptions available in most KBs [10, 54]. This results in m ranked lists $\langle l_1 = \{x_{11}, x_{12}, \ldots\}; l_2 = \{x_{21}, x_{22}, \ldots\}; \ldots l_m = \{x_{m1}, x_{m2}, \ldots\}\rangle$ of KB items x_{ij}, one list l_i for each q_i, scored by degree of match between question tokens and KB items. A ranked lexical match list (ideal disambiguation in **bold**) for *"scored"* could look like: $l_1 = \langle$ 1: score (music), 2: no. of goals scored, **3: goal scored by**, 4: film score, ...$\rangle$, while that for *"Croatia"* could be: $l_4 = \langle$ 1: Croatia (state), 2: 589 Croatia (asteroid), ..., 15: **Croatia football team**, ..., 19: Croatia basketball team, ...$\rangle$. Note that the best matching KB item x_i^* for q_i can sometimes be *very deep in individual lists* l_i (Croatia football team is at rank 15 in l_4).

Next, each list l_i is traversed up to a depth d to fetch the top-d items (computational cost $O(m \cdot d)$), that are per-term question-relevant KB candidates for the next phase of Clocq. The goal is to find *combinations* of KB items $\langle x_i \rangle_{i=1}^m$ that best match the question, since these items have a high likelihood of having the *answer within their facts* $\bigcup_{i=1}^m NF(x_i)$. For instance, an ideal combination for us would be: {goal scored by, 2018 FIFA WC final, France football team, Croatia football team}. These combinations come from the Cartesian product of items in the m lists, and would have d^m possibilities if each combination is explicitly enumerated and scored. This is cost-prohibitive: since we are only interested in some top-k combinations, as opposed to a full or even extended partial ordering, a more efficient way of doing this would be to apply top-k algorithms [3, 28, 34]. These prevent complete scans and return the top-k best combinations efficiently.

4.2 Computing relevance signals for each item

To go beyond shallow lexical matching, our proposal is to construct *multiple lists per question token, each reflecting a different relevance signal*, and to apply top-k algorithms *on these lists* to obtain the disambiguation of each question token individually. Unlike prior works on NED that are restricted to individual named entities [19, 25, 31], Clocq includes mentions of types, predicates and general concepts in the input question and maps them to KB items. A candidate KB item combination that fits well with the intent in the question is expected to have high semantic coherence and high graph connectivity (these can be viewed as proximity in latent and symbolic spaces) within its constituents, as well as match the question well at global and term-levels. These motivate our four indicators of relevance for each item x_{ij} in list l_i below. The cost of this scoring is $O(m^2 \cdot d^2)$: while this looks expensive, it is still fast with a parallelized implementation.

Coherence. Clocq targets a joint disambiguation of question-relevant KB items. It thus considers semantic coherence and graph connectivity, which are inherently defined for KB item pairs, instead of single items. Therefore, we need a technique to convert these signals into item-level scores. The first signal, semantic coherence, is transformed into an item-level score using the max operator. More precisely, the coherence score of an item x_{ij} is defined in Eq. 1 as the maximum item-item similarity (averaged over pairs of lists) this item can contribute to the combination, where pairwise similarity is obtained by the cosine value between the embedding vectors of

two KB items (min-max normalized from $[-1, +1]$ to $[0, 1]$):

$$coh(x_{ij}) = \frac{1}{m-1} \sum_{k \neq i} \max_l cosine(\vec{x_{ij}}, \vec{x_{kl}}) \quad (1)$$

Connectivity. This is the second context-level signal in Clocq, and captures a very different form of proximity. We assign items in 1-hop of each other to have a distance of 1 (recall KB-distance computations from Sec. 3), those in 2-hops to have a distance of 2, and ∞ otherwise (most KB items are in 3 hops of each other, and thus distance > 2 hops ceases to be a discriminating factor). We define connectivity scores as the inverse of this KB distance, thereby obtaining $1, 0.5$, and 0, respectively for 1-, 2-, and > 2-hop neighbors. Connectivity as a context-level signal is converted to an item-level score analogously using the max aggregator over pairs. We thus define connectivity ($\in [0, 1]$) of x_{ij} in Eq. 2:

$$conn(x_{ij}) = \frac{1}{m-1} \sum_{k \neq i} \max_l conn(x_{ij}, x_{kl}) \quad (2)$$

Question relatedness. We estimate semantic relatedness of the KB item x_{ij} to the overall input question q by averaging pairwise cosine similarities (same min-max normalization as for coherence) between the embeddings of the item and each term q_i in Eq. 3. To avoid confounding this estimate with the question term for which x_{ij} was retrieved, we exclude this from the average to define semantic relatedness as:

$$rel(x_{ij}) = \operatorname{avg}_{q_i \neq q_k} cosine(\vec{x_{ij}}, \vec{q_k}) \quad (3)$$

Term match. This score is intended to take into account the original degree of lexical term match (via TF-IDF, BM25, or similar) for which x_{ij} was admitted into l_i. However, such TF-IDF-like weights are often unbounded and may have a disproportionate influence when aggregated with the other signals, that all lie in the interval $\in [0, 1]$. Thus, we simply take the reciprocal rank of x_{ij} in l_i as the match score (Eq. 4) to have it in the same $[0, 1]$ interval:

$$match(x_{ij}) = 1/rank(x_{ij}, l_i) \quad (4)$$

4.3 Finding top-k across sorted lists

We now sort each of these $4 \cdot m$ lists in descending score-order. Note that for each q_i, all lists l_{ij} hold the same items (those in the original l_i). Fig. 2 shows lists l_{ij} in the center. Top-k algorithms operating over such multiple score-ordered lists, where each list holds the same set of items, require a monotonic aggregation function over the item scores in each list [3, 7, 11, 17]. Here, we use a linear combination of the four relevance scores as this aggregate: $aggScore(x_{ij}) = h_{coh} \cdot coh(x_{ij}) + h_{conn} \cdot conn(x_{ij}) + h_{rel} \cdot rel(x_{ij}) + h_{match} \cdot match(x_{ij})$, where hyperparameters are tuned on a dev set, and $h_{coh} + h_{conn} + h_{rel} + h_{match} = 1$. Since each score lies in $[0, 1]$, we also have $aggScore(\cdot) \in [0, 1]$.

We use the threshold algorithm (TA) over these score-ordered lists with early pruning [17]. TA is run over each set of 4 sorted lists $\langle l_{i1}, l_{i2}, l_{i3}, l_{i4} \rangle$, corresponding to one question term q_i, to obtain the top-k best KB items $\{x_i^*\}_k$ per q_i. Once we have these $\{x_i^*\}_k$, we take the union of these items $\bigcup_{i=1\ldots m} \{x_i^*\}_k$ to create our final combination of KB items. *KB facts* of items in this final list comprise $\bigcup_{i=1\ldots m} \{NF(x_i^*)\}_k$, which is the output search space S of Clocq and would be passed on to the downstream QA system.

4.4 Pruning facts of highly frequent KB items

To avoid including all facts of extremely frequent KB items into our search space $\mathcal{S}$ (U.K. brings in millions of entities), we use a pruning threshold p as follows. An entity x can appear in a fact as the subject, object or qualifier object, where usually the first role is the most salient. Whenever the last two total more than p, we take only the subject facts of x, and all facts otherwise. This is a proxy for keeping only salient facts in $\mathcal{S}$. For disambiguated predicates x, p directly acts as a frequency threshold. Thus, parameter p essentially controls the amount of potentially noisy facts that goes into $\mathcal{S}$.

4.5 Automatically setting k and p

The choice of k and p might not always be obvious, and in the methodology described above, all question words have the same values for k and p ($k = 5$ and $p = 1000$, say). Therefore, we propose a simple but effective mechanism to automatically choose k and p, dynamically depending upon the specific question word.

Choosing k. Intuitively, one would like to increase k for *ambiguous* question words. For example, *"France"* can refer to many KB items. By increasing k one can account for potential disambiguation errors. On the other hand, *"Paul Pogba"* is not as ambiguous, and hence k=1 should suffice. The ambiguity of a question word is closely connected to that of uncertainty or randomness. The more uncertainty there is in predicting what a word refers to, the more ambiguous it is. This makes *entropy* a suitable measure of ambiguity. More specifically, each question word is linked to d KB items. These items form the sample space of size d for the probability distribution. The numbers of KB facts of these items form a frequency distribution that can be normalized to obtain the required probability distribution. We compute the entropy of this probability distribution as the ambiguity score of a question word, and denote it as $ent(q_i)$. Incidentally, by definition, $0 \le ent(q_i) \le \log_2 d$. Practical choices of k and d do not exceed 5 and 50 respectively, and hence k and $\log_2 d$ are in the same ballpark ($\log_2 50$=5.6). This motivates us to make the simple choice of directly setting k as $ent(q_i)$. Specifically, we use $k = \lfloor ent(q_i) \rfloor + 1$ to have integral k, and avoid k=0.

Choosing p. We identify a logical connection between k and p: the less uncertainty there is in the disambiguation of a question word (i.e. the lower the k), the more facts one wants to include in $\mathcal{S}$ for this word. On the contrary, for highly ambiguous question words, less facts should be admitted for avoiding a higher amount of noise. Therefore, we set p automatically, by having $p=f(k)$. For example, we could set $p=10^{(5-k)}$, such that p is set to a high value ($p=10^4$) for k=1, but for a highly ambiguous word for which k=5, only subject facts are considered (p=1). We experiment with different variations of the function f that meet the desiderata above.

5 EXPERIMENTAL SETUP

Benchmarks. We use two recent QA benchmarks: LC-QuAD 2.0 [14] and ConvQuestions [12]. To make our case, we sampled 10k of the more complex questions from LC-QuAD 2.0 (LC-QuAD2.0-CQ in Table 2 with 2k dev, 8k test; no training required in Clocq). Complexity is loosely identified by the presence of multiple entities, as detected by Tagme [19], and/or predicates where main verbs were used as a proxy, detected with Stanza [38]. ConvQuestions was built for incomplete utterances in conversational QA, but also

has well-formed complete questions that exhibit several complex phenomena. For ConvQuestions, we considered full questions from the benchmark (ConvQuestions-FQ in Table 2; 338 dev, 1231 test).

Metrics. We use three metrics: i) *answer presence*, the percentage of times the correct answer is found in the reduced search space; ii) *size of the search space* $|\mathcal{S}|$, measured by the number of entities and literals, that would be answer candidates to be considered by the downstream QA engine; and iii) *runtime*, summed over all steps that happen at answering time and measured in seconds.

Baselines. We compare Clocq with a variety of NED baselines [15, 19, 25, 31, 53]. To provide baselines with competitive advantage w.r.t. efficient retrieval, we use the state-of-the-art HDT RDF [18] for KB storage and indexing. An example baseline would be Tagme+Hdt. For convenience, we omit the Hdt when referring to baselines in text. NED systems that link to Wikipedia are mapped to Wikidata using Wikipedia URLs that are also present in Wikidata. Baselines are either run on our data with original code when available, or through APIs. Internal confidence thresholds were set to zero (no cut-off) in configurable baselines like Tagme and Aida to allow for as many disambiguations (linkings) as possible, to help boost answer presence. Otherwise, default configurations were retained.

Initialization. We perform all experiments over Wikidata: we use the 2 TB uncompressed NTriples dump from 26 April 2020 with about 12B triples, on which Clocq is applied. Note that we applied the same pruning strategy and underlying Wikidata dump when using Hdt for retrieval, i.e. $NF(x)$ is the same for the Clocq KB interface and Hdt. For baselines, we uniformly set p=10k to boost their answer presence. To build term matching lists of question terms against KB items, we used Elasticsearch [21]. We use Wikipedia2Vec [58] to compute embeddings for terms and KB items. Questions were segmented into phrases like *"Harry Potter"* and *"theme music"* using named entity recognition [19]. The depth of the term-matching lists was set to d=20, and hyperparameters were tuned via dev sets to h_{coh}=0.1, h_{conn}=0.3, h_{rel}=0.2, h_{match}=0.4 for both benchmarks. The default setting for Clocq is an automatically chosen k and p=1k (Sec. 4.5). Since d=20, we have $k \in [1, 5]$. This configuration is implied when writing just "Clocq".

6 RESULTS AND INSIGHTS

6.1 Key findings

Our main results on search space reduction are in Table 2. As a reference point, the a-priori answer search space consists of all entities and literals in the whole KB K, a total of about 152M items. **Clocq keeps more answers in its search space.** Clocq outperforms the best baseline on answer presence for each benchmark by 5.8% (LC-QuAD) and 7.2% (ConvQuestions), pushing the upper bound for performance of QA systems. Clocq is able to keep 82.6% (LC-QuAD) and 84.7% (ConvQuestions) answers in its search space, which is statistically significant for all pairwise comparisons with Elq and Tagme, the strongest baselines for this task. Importantly, Clocq achieves this in sub-second runtimes, slightly faster than Elq, the fastest baseline. Representative examples are in Table 3. While Clocq (default) performs best, we note that Clocq (k=1) achieves an answer presence that is substantially better than that of all baselines as well, showing the effectiveness of KB-aware signals for this task.

Table 2: Performance of Clocq w.r.t. baselines. Statistical significance of Clocq's answer presence over Tagme and Elq, the strongest baselines, is marked with † and * respectively (McNemar's test as answer presence is a binary variable, with $p < 0.05$).

| Benchmark	LC-QuAD2.0-CQ [14]			ConvQuestions-FQ [12]		
Metric → Method ↓	Answer presence (Percentage)	Search space size (No. of KB items)	Runtime (Seconds)	Answer presence (Percentage)	Search space size (No. of KB items)	Runtime (Seconds)
Tagme [19]+Hdt [18]	76.8	2.9k	1.14	69.1	1.8k	1.43
Aida [25]+Hdt [18]	60.5	2.2k	0.75	44.4	2.2k	1.19
Earl [15]+Hdt [18] (k=1)	53.8	1.1k	2.50	44.6	1.1k	2.49
Earl [15]+Hdt [18] (k=5)	65.9	2.2k	2.50	53.4	2.0k	2.49
Rel [53]+Hdt [18]	55.8	0.7k	0.72	45.6	0.4k	0.61
Elq [31]+Hdt [18]	76.7	1.1k	0.62	77.5	0.6k	0.47
Clocq (Default: k=Auto, p=1k)	$82.6^{\dagger *}$	1.5k	0.50	$84.7^{\dagger *}$	1.3k	0.42
Clocq (k=1, p=10k)	$80.0^{\dagger *}$	3.9k	0.48	$78.4^{\dagger}$	2.3k	0.39
Clocq (k=5, p=100)	$80.9^{\dagger *}$	0.6k	0.49	$84.2^{\dagger *}$	0.6k	0.40

(a) Answer presence. (b) Search space size. (c) Runtimes.
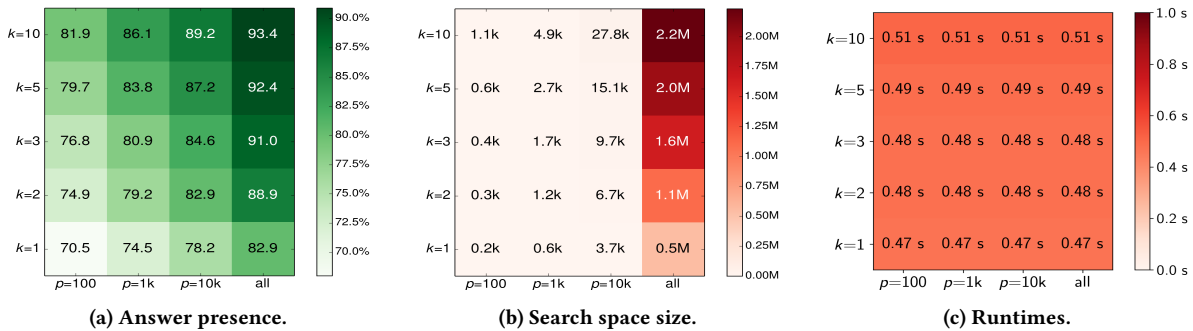
Figure 3: Varying Clocq parameters on the LC-QuAD dev set. Trends on the ConvQuestions dev set are almost the same.

Top-k results add value over top-1. The true power of Clocq comes from the flexibility of top-k outputs, coupled with the pruning threshold p. Fig. 3 shows variation in answer presence, search space size, and runtime with k and p on the LC-QuAD dev set. We see that by increasing k from 1 to 10, Clocq achieves very good answer presence (going above 80%, Fig. 3a), while keeping a tight threshold on items admitted into the search space (columns 1 and 2, p=100 or 1k). Here, the search space stays fairly small, in the order of a few thousand KB items (Fig. 3b). If, on the other hand, a QA system requires very high recall, Clocq can achieve this by increasing p (columns 3 and 4 in Fig. 3b): answer presence is well above 90%. The price is a much larger search space. Another observation is that due to the use of our efficient top-k architecture and novel KB index, the timings are fairly stable when increasing k and p. For change in p, we did not observe any increase in runtimes, and for k, the increase is ≤ 0.04 seconds. These trends are almost the same for ConvQuestions. We added one top-k variant with a good trade-off on the dev-set (k=5, p=100) to Table 2 (last row). This significantly outperforms all baselines w.r.t. answer presence and runtime, with a very small search space size of only about 600 items. Among our baselines, Earl [15] can produce top-k disambiguations: using k=5 for Earl (fourth row) also increases its answer presence, but this is far below that of Clocq.

We identify a *trade-off* between answer presence and search space size as a major consideration for QA. The best setting for k and p highly depends on the QA system operating on the contracted search space. In general, for improving the answer presence,

Table 3: Anecdotal questions from test sets for which only Clocq had an answer in the search space.

Who is the composer of All We Know? (LC-QuAD)
Who is the son of the brother of Queenie Padilla? (LC-QuAD)
what was the name of Dorethea Lange's spouse in 1920? (LC-QuAD)

Who was the screenwriter for Crazy Rich Asians? (ConvQuestions)
What is the name of the first book in Bill Hodges trilogy? (ConvQuestions)
What actor played the role of Jason Bateman's older brother in the sitcom Arrested Development? (ConvQuestions)

we recommend increasing k rather than p. Even though increasing k and p cannot decrease the answer presence, the additional facts admitted into S could still distract the QA system and lead to longer runtimes. Therefore, the choice of k and p depends on the maximum search space size and potential disambiguations per mention (manifested as k) a specific QA system can handle.

Impact on KB-QA. While answer presence is an important measure creating an upper bound for the QA system, the key goal of this work is to enhance the performance on the downstream QA task. To study these effects, we feed the outputs of Clocq and the two best baselines into the popular KB-QA system GRAFT-Net [49] on the two benchmarks. We report the standard QA metrics *precision at 1* (P@1), *mean reciprocal rank* (MRR) and *hit at 5* (Hit@5) [43]. Results are in Table 4. For LC-QuAD, the configuration with Clocq significantly outperforms the two strongest baselines on all metrics. For ConvQuestions, Clocq has the best performance on MRR and Hit@5, and is only slightly behind Elq on P@1. These results show the benefits of Clocq for downstream QA. Clocq generates the

Table 4: Impact of Clocq on KB-QA.

Benchmark	LC-QuAD2.0-CQ			ConvQuestions-FQ		
QA system →	GRAFT-Net [49]			GRAFT-Net [49]		
Method ↓	P@1	MRR	Hit@5	P@1	MRR	Hit@5
Clocq	**0.197***	**0.268***	**0.350***	0.207	**0.264**	**0.337**
Elq+Hdt	0.168	0.224	0.288	**0.213**	0.256	0.313
Tagme+Hdt	0.171	0.225	0.291	0.167	0.204	0.237

Table 5: Ablation study of configurations in Clocq.

Benchmark	LC-QuAD2.0-CQ			ConvQuestions-FQ		
Method ↓	Ans. pres.	$|\mathcal{S}|$	Time	Ans. pres.	$|\mathcal{S}|$	Time
Clocq	0.803	1.5k	0.47 s	0.760	1.1k	0.34 s
w/o *match*	0.726*	1.3k	0.46 s	0.607*	0.9k	0.30 s
w/o *rel*	0.806	1.5k	0.47 s	0.746*	1.2k	0.32 s
w/o *conn*	0.790*	1.5k	0.41 s	0.746*	1.2k	0.26 s
w/o *coh*	0.802	1.5k	0.40 s	0.750	1.1k	0.24 s
w/o *match* + *rel*	0.733*	1.3k	0.47 s	0.618*	0.9k	0.30 s
w/o *coh* + *conn*	0.791*	1.5k	0.34 s	0.743*	1.2k	0.22 s

search space faster: the average runtimes per question are 0.49 s for Clocq, 0.60 s for Elq+Hdt and 1.18 s for Tagme+Hdt.

6.2 In-depth analysis

Clocq identifies relevant concepts and types. For many questions, Clocq identifies not just additional entities but also concepts and types that are missed by baselines. Since $k>1$ trivially adds more KB items, we set $k=1$ for fair comparison in this analysis. For example, in *What was the name of the theme music for the television series Mash?*, Elq disambiguates only *"Mash"* (incorrectly), to the 1970 film. Clocq, on the other hand, finds: *"name"* ↦ personal name, *"theme music"* ↦ theme music, *"television series"* ↦ television series, and *"Mash"* ↦ M∗A∗S∗H (the TV series, correct). On average, Clocq finds 4.68 KB items per question (LC-QuAD), while Elq, Aida and Tagme find 1.82, 2.65 and 3.75, respectively. We verified that these additionally disambiguated types and concepts help: when removed from Clocq's output, answer presence drops from 80.3% to 65.5% (LC-QuAD dev). Note that standalone NED evaluation is out of scope here, because QA benchmarks have no ground-truth for KB item disambiguation.

Ablation studies. Clocq includes four signals in its architecture, so this naturally calls for ablations (Table 5, dev sets). Answer presence on ConvQuestions dropped for each single signal that is removed, showing that all four matter (* = significant drop from full configuration). On LC-QuAD, trends are similar, just that removing relevance led to a slightly increased answer presence. While removing a single component has only small influence, dropping the pair of local and global signals (like *match* + *rel*, or *coh* + *conn*) often results in noticeable loss. However, such choices may need to be made when runtime is of utmost importance, since computing *coh* and *conn* are the most time-consuming steps in Clocq.

Error analysis. Clocq misses the answer in $\mathcal{S}$ just about 20% of the time (both benchmarks), arising from two error cases: i) the answer is missing in the computed set of facts, as the depth-d term matching does not retrieve the relevant items (LC-QuAD 44.8%, ConvQuestions 46.7%); ii) the answer is in the candidate space, but

Table 6: Effect of choosing k and p dynamically per term.

Benchmark	LC-QuAD2.0-CQ		ConvQuestions-FQ	
Clocq ↓	Ans. pres.	$|\mathcal{S}|$	Ans. pres.	$|\mathcal{S}|$
k=Auto, p=1k (default)	0.803	1.5k	0.760	1.0k
k=Auto, p=10k	0.837	7.8k	0.778	4.6k
k=Auto, p=10^{5-k}	0.776	1.5k	0.728	0.9k
k=Auto, p=$10^{5-0.5 \cdot k}$	0.835	6.3k	0.778	4.3k
k=Auto, p=$10^{4-0.5 \cdot k}$	0.798	1.0k	0.734	0.8k

Table 7: Effect of BM25 on verbalized KB facts.

Benchmark	LC-QuAD2.0-CQ		ConvQuestions-FQ	
Method ↓	Ans. pres.	$|\mathcal{S}|$	Ans. pres.	$|\mathcal{S}|$
Clocq (k=1, p=10k)	0.782	3.7k	0.704	1.9k
+ BM25 (top-100)	0.625	0.1k	0.509	0.1k
+ BM25 (top-1000)	0.726	0.8k	0.630	0.7k
Clocq (k=3, p=10k)	0.846	9.7k	0.775	5.9k
+ BM25 (top-100)	0.614	0.1k	0.462	0.1k
+ BM25 (top-1000)	0.742	1.0k	0.648	0.9k
Clocq (k=5, p=10k)	0.872	15.1k	0.796	10.3k
+ BM25 (top-100)	0.605	0.1k	0.456	0.1k
+ BM25 (top-1000)	0.747	1.0k	0.648	0.9k

the top-k algorithm fails to return one or more relevant items (LC-QuAD 55.2%, ConvQuestions 53.3%). Both cases could be countered by increasing d or the range of k, at the cost of increased runtimes.
Automatic choices for k and p. Table 6 shows results of various choices. As discussed in Sec. 4.5, p can be set as $f(k)$. We tried p=10^{5-k} first, and found that p is reduced too drastically. Therefore, we compared with smoother versions p=$10^{5-0.5k}$ and p=$10^{4-0.5k}$. Again, there is a trade-off between answer presence and search space size: having p=$10^{5-0.5k}$ gives the best answer presence, but p=$10^{4-0.5k}$ has a much smaller $|\mathcal{S}|$. The runtime was almost same across all variants. Overall, we found a static setting of p to perform slightly better w.r.t. the trade-off.
IR-based extension. An intuitive extension or alternative is to fetch a larger subset of the KB, verbalize these facts [2, 36] and use a standard IR pipeline to retrieve the most relevant facts for use by the QA system. We implemented such a variant, treating the question as the query and the collection of verbalized facts as the corpus. BM25 [42] is used for scoring fact-relevance, and returns the top-100 or top-1000 facts. Results on dev sets are shown in Table 7. Different variants of Clocq are used for retrieving the KB subset, where the focus is on larger initial $\mathcal{S}$ to measure the impact of BM25 (therefore the choice of a large p of 10k). Answer presence for top-1000 facts is comparable to the initial answer presence; but a significant drop was observed when taking only the top-100 facts. This indicates that basic bag-of-words in BM25 matching falls short for complex questions. However, an IR-based filter is a viable choice when the number of facts that can be consumed is budgeted.

6.3 Effect of fact-based KB indexing

Fact-centric KB storage is a foundation for Clocq: we now analyze its effect on runtimes for search space reduction. Our comparison points are the available Wikidata SPARQL endpoint[1] (QueryService)

---

[1] https://query.wikidata.org/bigdata/namespace/wdq/sparql?format=json

Table 8: Comparison of KB interfaces w.r.t. functionalities.

KB interface	QUERYSERVICE	HDT [18]	CLOCQ
RAM consumed	–	$220GB$	$340GB$
Neighborhood	$1.48 \times 10^{-2}s$	$6.73 \times 10^{-4}s$	$4.98 \times 10^{-5}s$
KB-distance	$2.46 \times 10^{-2}s$	$5.43 \times 10^{-3}s$	$3.23 \times 10^{-6}s$

Table 9: Timing KB interfaces for search space reduction.

KB interface	QUERYSERVICE	HDT [18]	CLOCQ
CLOCQ	–	971 s	0.54 s
ELQ [31]	0.89 s	0.62 s	0.12 s
TAGME [19]	19 s	1.25 s	0.52 s

and triple pattern queries issued to the HDT [18] KB interface. We subtracted network latencies when measuring runtimes.

Basic functionalities. Our first experiment was on the two basic functionalities required for KB-QA: retrieving all facts of a given KB item (neighborhood), and measuring the distance between two given KB items (KB-distance). For baselines, we optimized the number of required queries and implemented distance checks as for CLOCQ (Sec. 3). We took 1 million random KB items for the neighborhood lookups, and 1 million random KB item pairs for the connectivity checks. Average runtimes (per KB item / KB item pair) are shown in Table 8. We found that HDT has a better performance than the Wikidata QUERYSERVICE, making use of its efficient implementation via bitstreams. However, CLOCQ can improve neighborhood lookups by a factor of 10 and 10^3 over HDT and QUERYSERVICE, respectively. When measuring KB-distances, the effect becomes even larger: CLOCQ is 10^3 and 10^4 times faster than HDT and the QUERYSERVICE. The memory consumption for the CLOCQ KB index is slightly higher than that of HDT, but this is still much lower than what loading the raw KB dump into memory would consume.

Effect on search space reduction. We now compare runtimes with these KB interfaces for search space reduction on the LC-QuAD dev set. While CLOCQ makes use of the neighborhood and KB-distance functions, only the neighborhood function is necessary in ELQ and TAGME. We observe similar trends as before: runtimes of CLOCQ are much better when using the CLOCQ KB index. The QUERYSERVICE script did not terminate within a reasonable amount of time. Interestingly, these trends also hold for ELQ and TAGME: when using the CLOCQ KB index for search space reduction, the runtime is significantly reduced. This shows that our fact-based KB index is valuable beyond its specific use in CLOCQ.

Gains in runtime are due to the fact-centric KB index, which is specifically designed for providing efficient KB access for QA functionalities. KB interface baselines may provide very fast KB access for general-purpose querying, but fall short for the more specific requirements of QA.

Summary. Disambiguating not only entities, but also general concepts, types, or predicates when establishing the search space, is generally beneficial for QA systems. This is something that is done by CLOCQ but is *beyond NED* systems. The detected trade-off between answer presence and search space size is an important factor: increasing $|\mathcal{S}|$ improves answer presence but also injects noise, whereas a smaller search space could potentially be cleaner and easier to explore by the QA system. This trade-off is closely connected to the choice of k and the amount of facts for a specific KB item that is admitted into $\mathcal{S}$, that is controlled by our other parameter p.

7 RELATED WORK

KB interfaces. Optimizing KBs for executing SPARQL queries is a well-studied problem [16, 22, 35, 50, 55]. TRIDENT [51] enables workloads like query and graph analytics on large KBs. HDT [18] encodes triples using bitmaps. It constructs two individual integer-streams holding predicates and objects, adjacent to some given subject, and two additional bitstreams for encoding connections between these predicates and objects. Due to multiple indexes, triple pattern queries can be answered very efficiently using HDT. These works focus on optimizing queries on triple stores. However, the problem of retrieving the complete facts of a KB item including qualifier information is a typical task in KB-QA, but is not targeted.

Named entity disambiguation. NED tries to map entity mentions to corresponding real-life concepts: pages in Wikipedia or entries in curated KBs like Wikidata. TAGME [19] leverages Wikipedia anchors to detect entity mentions, looks up possible mappings, and scores these with regard to a collective agreement implemented by a voting scheme. In AIDA [25], a mention-entity graph is established, and the mentions are disambiguated jointly by approximating the densest subgraph. More recently, van Hulst et al. [53] proposed the framework REL for end-to-end entity linking, building on state-of-the-art neural components. ELQ [31] jointly performs mention detection and disambiguation leveraging a BERT-based bi-encoder. These methods are optimized for computing the top-1 entity per mention, and mostly return only the top-ranked entity in the disambiguation. Top-1 NED is prone to errors that can propagate through the answering pipeline [45, 60]. Early work in S-MART [59] applied statistical models of regression trees on a set of (mention, entity)-pairs and corresponding features. Unlike most other works, S-MART returned top-k disambiguations per mention. However, owing to proprietary code, a comparison was not possible.

Search space reduction. Methods in complex KB-QA mostly follow one of two approaches [43]: i) disambiguating entities, predicates and types over the whole KB [13, 26, 45, 52] by leveraging question-word specific index lists [45, 52] for subsequent semantic parsing; and, ii) applying NED as an initial step to focus the remaining computation on a restricted search space [5, 6, 9, 33, 40, 44, 48, 56, 60]. In this work, the focus is on improving the second line of work: instead of performing top-1 or top-k NED, we disambiguate *all* question cue words and compute the top-k results per question token. This leads to a search space that is more likely to contain the relevant KB items *and* the answer. EARL [15] takes an approach of disambiguating both entity and predicate mentions. We generalize this direction by disambiguating all keywords in the question.

8 CONCLUSIONS AND FUTURE WORK

We introduced answer search space reduction as a vital task in KB-QA. We showed that our proposal CLOCQ, based on the threshold algorithm over score-ordered lists containing KB items with different relevance signals, is more successful in retaining answers in its reduced search space than a wide variety of general-purpose methods for named entity disambiguation.

REFERENCES

[1] Abdalghani Abujabal, Rishiraj Saha Roy, Mohamed Yahya, and Gerhard Weikum. 2018. Never-ending learning for open-domain question answering over knowledge bases. In *WWW*.

[2] Oshin Agarwal, Heming Ge, Siamak Shakeri, and Rami Al-Rfou. 2010. Knowledge graph based synthetic corpus generation for knowledge-enhanced language model pre-training. *NAACL* (2010).

[3] Vo Ngoc Anh and Alistair Moffat. 2006. Pruned query evaluation using precomputed impacts. In *SIGIR*.

[4] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. 2007. DBpedia: A nucleus for a Web of open data. (2007).

[5] Junwei Bao, Nan Duan, Zhao Yan, Ming Zhou, and Tiejun Zhao. 2016. Constraint-based question answering with knowledge graph. In *COLING*.

[6] Hannah Bast and Elmar Haussmann. 2015. More accurate question answering on freebase. In *CIKM*.

[7] Hannah Bast, Debapriyo Majumdar, Ralf Schenkel, Martin Theobald, and Gerhard Weikum. 2006. IO-Top-k: Index-access Optimized Top-k Query Processing. In *VLDB Conference*.

[8] Jonathan Berant, Andrew Chou, Roy Frostig, and Percy Liang. 2013. Semantic parsing on freebase from question-answer pairs. In *EMNLP*.

[9] Nikita Bhutani, Xinyi Zheng, and HV Jagadish. 2019. Learning to answer complex questions over knowledge bases with query composition. In *CIKM*.

[10] Kurt Bollacker, Colin Evans, Praveen Paritosh, Tim Sturge, and Jamie Taylor. 2008. Freebase: A collaboratively created graph database for structuring human knowledge. In *SIGMOD*.

[11] Chris Buckley and Alan F Lewit. 1985. Optimization of inverted vector searches. In *SIGIR*.

[12] Philipp Christmann, Rishiraj Saha Roy, Abdalghani Abujabal, Jyotsna Singh, and Gerhard Weikum. 2019. Look before you hop: Conversational question answering over knowledge graphs using judicious context expansion. In *CIKM*.

[13] Jiwei Ding, Wei Hu, Qixin Xu, and Yuzhong Qu. 2019. Leveraging Frequent Query Substructures to Generate Formal Queries for Complex Question Answering. In *EMNLP-IJCNLP*.

[14] Mohnish Dubey, Debayan Banerjee, Abdelrahman Abdelkawi, and Jens Lehmann. 2019. Lc-quad 2.0: A large dataset for complex question answering over wikidata and dbpedia. In *ISWC*.

[15] Mohnish Dubey, Debayan Banerjee, Debanjan Chaudhuri, and Jens Lehmann. 2018. EARL: joint entity and relation linking for question answering over knowledge graphs. In *ISWC*.

[16] Orri Erling and Ivan Mikhailov. 2010. Virtuoso: RDF support in a native RDBMS. In *Semantic Web Information Management*.

[17] Ronald Fagin, Amnon Lotem, and Moni Naor. 2003. Optimal aggregation algorithms for middleware. *Journal of computer and system sciences* 66, 4 (2003).

[18] Javier D Fernández, Miguel A Martínez-Prieto, Claudio Gutiérrez, Axel Polleres, and Mario Arias. 2013. Binary RDF representation for publication and exchange (HDT). In *Journal of Web Semantics*.

[19] Paolo Ferragina and Ugo Scaiella. 2010. TAGME: On-the-fly annotation of short text fragments (by Wikipedia entities). In *CIKM*.

[20] Mikhail Galkin, Priyansh Trivedi, Gaurav Maheshwari, Ricardo Usbeck, and Jens Lehmann. 2020. Message Passing for Hyper-Relational Knowledge Graphs. In *EMNLP*.

[21] Clinton Gormley and Zachary Tong. 2015. *Elasticsearch: the definitive guide: a distributed real-time search and analytics engine*. O'Reilly Media, Inc.

[22] Sairam Gurajada, Stephan Seufert, Iris Miliaraki, and Martin Theobald. 2014. TriAD: a distributed shared-nothing RDF engine based on asynchronous message passing. In *SIGMOD*.

[23] Gaole He, Yunshi Lan, Jing Jiang, Wayne Xin Zhao, and Ji-Rong Wen. 2021. Improving multi-hop knowledge base question answering by learning intermediate supervision signals. In *WSDM*.

[24] Daniel Hernández, Aidan Hogan, and Markus Krötzsch. 2015. Reifying RDF: What works well with Wikidata? *SSWS@ ISWC* (2015).

[25] Johannes Hoffart, Mohamed Amir Yosef, Ilaria Bordino, Hagen Fürstenau, Manfred Pinkal, Marc Spaniol, Bilyana Taneva, Stefan Thater, and Gerhard Weikum. 2011. Robust Disambiguation of Named Entities in Text. In *EMNLP*.

[26] Sen Hu, Lei Zou, and Xinbo Zhang. 2018. A state-transition framework to answer complex questions over knowledge base. In *EMNLP*.

[27] Xiao Huang, Jingyuan Zhang, Dingcheng Li, and Ping Li. 2019. Knowledge graph embedding based question answering. In *WSDM*.

[28] Ihab F Ilyas, George Beskales, and Mohamed A Soliman. [n.d.]. A survey of top-k query processing techniques in relational database systems. *CSUR*.

[29] Yunshi Lan and Jing Jiang. 2020. Query graph generation for answering multi-hop complex questions from knowledge bases. In *ACL*.

[30] Jyoti Leeka, Srikanta Bedathur, Debajyoti Bera, and Medha Atre. 2016. Quark-X: An efficient top-k processing framework for RDF quad stores. In *CIKM*.

[31] Belinda Z. Li, Sewon Min, Srinivasan Iyer, Yashar Mehdad, and Wen-tau Yih. 2020. Efficient One-Pass End-to-End Entity Linking for Questions. In *EMNLP*.

[32] Xiaolu Lu, Soumajit Pramanik, Rishiraj Saha Roy, Abdalghani Abujabal, Yafang Wang, and Gerhard Weikum. 2019. Answering Complex Questions by Joining Multi-Document Evidence with Quasi Knowledge Graphs. In *SIGIR*.

[33] Kangqi Luo, Fengli Lin, Xusheng Luo, and Kenny Zhu. 2018. Knowledge base question answering via encoding of complex query graphs. In *EMNLP*.

[34] Joel Mackenzie and Alistair Moffat. 2020. Examining the Additivity of Top-k Query Processing Innovations. In *CIKM*.

[35] Thomas Neumann and Gerhard Weikum. 2008. RDF-3X: a RISC-style engine for RDF. *Proceedings of the VLDB Endowment*.

[36] Barlas Oguz, Xilun Chen, Vladimir Karpukhin, Stan Peshterliev, Dmytro Okhonko, Michael Schlichtkrull, Sonal Gupta, Yashar Mehdad, and Scott Yih. 2020. Unified Open-Domain Question Answering with Structured and Unstructured Knowledge. *arXiv*.

[37] Thomas Pellissier Tanon, Denny Vrandečić, Sebastian Schaffert, Thomas Steiner, and Lydia Pintscher. 2016. From Freebase to Wikidata: The great migration. In *WWW*.

[38] Peng Qi, Yuhao Zhang, Yuhui Zhang, Jason Bolton, and Christopher D Manning. 2020. Stanza: A python natural language processing toolkit for many human languages. *arXiv*.

[39] Yunqi Qiu, Yuanzhuo Wang, Xiaolong Jin, and Kun Zhang. 2020. Stepwise reasoning for multi-relation question answering over knowledge graph with weak supervision. In *WSDM*.

[40] Yunqi Qiu, Kun Zhang, Yuanzhuo Wang, Xiaolong Jin, Long Bai, Saiping Guan, and Xueqi Cheng. 2020. Hierarchical Query Graph Generation for Complex Question Answering over Knowledge Graph. In *CIKM*.

[41] Ridho Reinanda, Edgar Meij, and Maarten de Rijke. 2020. Knowledge Graphs: An Information Retrieval Perspective. *Found. Trends Inf. Retr.* (2020).

[42] Stephen Robertson and Hugo Zaragoza. 2009. The Probabilistic Relevance Framework: BM25 and Beyond. *Foundations and Trends in Information Retrieval* (2009).

[43] Rishiraj Saha Roy and Avishek Anand. 2021. Question Answering for the Curated Web: Tasks and Methods in QA over Knowledge Bases and Text Collections. *Synthesis Lectures on Information Concepts, Retrieval, and Services* 13, 4 (2021), 1–194.

[44] Uma Sawant, Saurabh Garg, Soumen Chakrabarti, and Ganesh Ramakrishnan. 2019. Neural architecture for question answering using a knowledge graph and web corpus. In *Information Retrieval Journal*.

[45] Tao Shen, Xiubo Geng, QIN Tao, Daya Guo, Duyu Tang, Nan Duan, Guodong Long, and Daxin Jiang. 2019. Multi-Task Learning for Conversational Question Answering over a Large-Scale Knowledge Base. In *EMNLP-IJCNLP*.

[46] Wei Shen, Jianyong Wang, and Jiawei Han. 2015. Entity Linking with a Knowledge Base: Issues, Techniques, and Solutions. *IEEE Transactions on Knowledge and Data Engineering* (2015).

[47] Fabian M Suchanek, Gjergji Kasneci, and Gerhard Weikum. 2007. YAGO: A core of semantic knowledge. In *WWW*.

[48] Haitian Sun, Tania Bedrax-Weiss, and William Cohen. 2019. PullNet: Open Domain Question Answering with Iterative Retrieval on Knowledge Bases and Text. In *EMNLP-IJCNLP*.

[49] Haitian Sun, Bhuwan Dhingra, Manzil Zaheer, Kathryn Mazaitis, Ruslan Salakhutdinov, and William Cohen. 2018. Open Domain Question Answering Using Early Fusion of Knowledge Bases and Text. In *EMNLP*.

[50] Jacopo Urbani, Sourav Dutta, Sairam Gurajada, and Gerhard Weikum. 2016. KOGNAC: Efficient encoding of large knowledge graphs. In *IJCAI*.

[51] Jacopo Urbani and Ceriel Jacobs. 2020. Adaptive Low-level Storage of Very Large Knowledge Graphs. In *WWW*.

[52] Svitlana Vakulenko, Javier David Fernandez Garcia, Axel Polleres, Maarten de Rijke, and Michael Cochez. 2019. Message passing for complex question answering over knowledge graphs. In *CIKM*.

[53] Johannes M van Hulst, Faegheh Hasibi, Koen Dercksen, Krisztian Balog, and Arjen P de Vries. 2020. REL: An Entity Linker Standing on the Shoulders of Giants. In *SIGIR*.

[54] Denny Vrandečić and Markus Krötzsch. 2014. Wikidata: A free collaborative knowledgebase. *CACM*.

[55] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. 2008. Hexastore: sextuple indexing for semantic web data management. *Proceedings of the VLDB Endowment*.

[56] Kun Xu, Siva Reddy, Yansong Feng, Songfang Huang, and Dongyan Zhao. 2016. Question Answering on Freebase via Relation Extraction and Textual Evidence. In *ACL*.

[57] Mohamed Yahya, Klaus Berberich, Shady Elbassuoni, Maya Ramanath, Volker Tresp, and Gerhard Weikum. 2012. Natural language questions for the web of data. In *EMNLP*.

[58] Ikuya Yamada, Akari Asai, Jin Sakuma, Hiroyuki Shindo, Hideaki Takeda, Yoshiyasu Takefuji, and Yuji Matsumoto. 2020. Wikipedia2Vec: An Efficient Toolkit for Learning and Visualizing the Embeddings of Words and Entities from Wikipedia. In *EMNLP*.

[59] Yi Yang and Ming-Wei Chang. 2015. S-MART: Novel Tree-based Structured Learning Algorithms Applied to Tweet Entity Linking. In *ACL-IJCNLP*.

[60] Wen-tau Yih, Ming-Wei Chang, Xiaodong He, and Jianfeng Gao. 2015. Semantic Parsing via Staged Query Graph Generation: Question Answering with Knowledge Base. In *ACL-IJCNLP*.