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

Philip 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

ACM Reference Format:

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, 66]. 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
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/](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 \(<\text{subject}, \text{predicate}, \text{object}>\) triple, that is optionally augmented by \(<\text{qualifier predicate, qualifier object}>\) pairs which specify context information for the main triple. For example, \(<\text{2018 FIFA World Cup Final, goal scored by, Paul Pogba}>\), \(<\text{France national football team, location, Luzhniki Stadium}>\); 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 \(<\text{qualifier predicate, qualifier object}>\) 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 \( q_1, q_2, \ldots, q_m \), where stopwords are not considered. For our running example question Who scored in the 2018 final between France and Croatia?, we would have \( q = <\text{scored}, \text{2018, final, france, croatia}>. \) Without loss of generality, \( q \) 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 \( S(q) \) for a question \( q \) is a set of facts \( S(q) \subseteq K \), that is expected to contain each \( a \). For example, \(<\text{2018 FIFA World Cup Final, goal scored by, Paul Pogba}>\) for team, France football team, \(<\text{2018 FIFA World Cup Final, goal scored by, Ivan Perisic}>\) for team, Croatia football team, ...}

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
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, 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 → NF(x)`), and the other stores the set of 1-hop neighbors of each KB item (`x → 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 `x1, x2` are within 1-hop distance if `x1 ∈ NI(x2)`, and in 2-hop distance if `NI(x1) ∪ NI(x2) ≠ ∅` (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, and 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 `q1 ··· qm (m = 4 in our example, stopwords are dropped). Here
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 \([0, 1]\)):

\[
coh(x_{ij}) = \frac{1}{m-1} \sum_{k \neq i} \text{max cosine}(x_{ij}^k, x_{kl}^k) \tag{1}
\]

**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 \(\infty\) 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:

\[
\text{conn}(x_{ij}) = \frac{1}{m-1} \sum_{k \neq i} \text{max conn}(x_{ij}, x_{kl}) \tag{2}
\]

**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:

\[
\text{rel}(x_{ij}) = \frac{1}{q_i \neq q_j} \text{cosine}(x_{ij}^k, q_k) \tag{3}
\]

**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 \([0, 1]\). 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:

\[
\text{match}(x_{ij}) = 1/\text{rank}(x_{ij}, l_i) \tag{4}
\]

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:

\[
\text{aggScore}(x_{ij}) = \text{coh} \cdot \text{coh}(x_{ij}) + \text{conn} \cdot \text{conn}(x_{ij}) + \text{rel} \cdot \text{rel}(x_{ij}) + \text{match} \cdot \text{match}(x_{ij}) \tag{5}
\]

where hyperparameters are tuned on a dev set, and \(\text{coh} + \text{conn} + \text{rel} + \text{match} = 1\). Since each score lies in \([0, 1]\), we also have \(\text{aggScore}(\cdot) \in [0, 1]\).

We use the threshold algorithm (TA) over these score-ordered lists with early pruning \([17]\). TA is run over each of \(4\) sorted lists \((l_{ij}, l_{i2}, l_{i3}, l_{i4})\), corresponding to one question term \(q_i\), to obtain the top-k best KB items \(\{x_{ij}^k\}_{k}\) per \(q_i\). Once we have these \(\{x_{ij}^k\}\), we take the union of these items \(\bigcup_{i=1..m} \{x_{ij}^k\}\) to create our final combination of KB items. KB facts of items in this final list comprise \(\bigcup_{i=1..m} (NF(x_{ij}))\), which is the output search space \(\mathcal{S}\) 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 $S$ (e.g., 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 $S$. 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 $S$.

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, $0 \leq ent(q_i) \leq \log d$. Practical choices of $k$ and $d$ do not exceed 5 and 50 respectively, and hence $k$ and $\log d$ are in the same ballpark ($\log 50 \approx 5.6$). 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 $S$ 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 ($p=10^5$) 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 $|S|$, 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 \( p < 0.05 \)).

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>LC-QuAD2.0-CQ ([14])</th>
<th>ConvQuestions-FQ ([12])</th>
</tr>
</thead>
<tbody>
<tr>
<td>Metric ( \rightarrow ) Method ( \downarrow )</td>
<td>Answer presence (Percentage)</td>
<td>Search space size (No. of KB items)</td>
</tr>
<tr>
<td>Tagme [19]+Hdt [18]</td>
<td>76.8</td>
<td>2.9k</td>
</tr>
<tr>
<td>Aida [25]+Hdt [18]</td>
<td>60.5</td>
<td>2.2k</td>
</tr>
<tr>
<td>Earl [15]+Hdt [18] ((k=1))</td>
<td>53.8</td>
<td>1.1k</td>
</tr>
<tr>
<td>Earl [15]+Hdt [18] ((k=5))</td>
<td>65.9</td>
<td>2.2k</td>
</tr>
<tr>
<td>Rel [53]+Hdt [18]</td>
<td>55.8</td>
<td>0.7k</td>
</tr>
<tr>
<td>Elq [31]+Hdt [18]</td>
<td>76.7</td>
<td>1.1k</td>
</tr>
</tbody>
</table>

Clocq (Default: \( k=\text{Auto}, p=1k \))
- \( 82.6^\dagger \) | 1.5k | 0.50 | 84.7^\dagger | 1.3k | 0.42 |

Clocq \((k=1, p=10k)\)
- \( 80.0^{*\dagger} \) | 3.9k | 0.48 |

Clocq \((k=5, p=100)\)
- \( 80.9^{*\dagger} \) | 0.6k | 0.49 |

Figure 3: Varying Clocq parameters on the LC-QuAD dev set. Trends on the ConvQuestions dev set are almost the same.

(a) Answer presence. (b) Search space size. (c) Runtimes.

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 \( \leq 0.04 \) 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, 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 3: Anecdotal questions from test sets for which only Clocq had an answer in the search space.

<table>
<thead>
<tr>
<th>Question</th>
<th>LC-QuAD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Who is the composer of All We Know? (LC-QuAD)</td>
<td></td>
</tr>
<tr>
<td>Who is the son of the brother of Queenie Padilla? (LC-QuAD)</td>
<td></td>
</tr>
<tr>
<td>what was the name of Dorethea Lange’s spouse in 1920? (LC-QuAD)</td>
<td></td>
</tr>
<tr>
<td>Who was the screenwriter for Crazy Rich Asians? (ConvQuestions)</td>
<td></td>
</tr>
<tr>
<td>What is the name of the first book in Bill Hodges trilogy? (ConvQuestions)</td>
<td></td>
</tr>
<tr>
<td>What actor played the role of Jason Bateman’s older brother in the sitcom Arrested Development? (ConvQuestions)</td>
<td></td>
</tr>
</tbody>
</table>

Note: The presence of a question is marked with * (McNemar’s test as answer presence is a binary variable, with \( p < 0.05 \)).
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" $\rightarrow$ personal name, "theme music" $\rightarrow$ theme music, "television series" $\rightarrow$ television series, and "Mash" $\rightarrow$ *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 relevances 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 $S$ 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 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 $p=10^{\beta-k}$ first, and found that $p$ is reduced too drastically. Therefore, we compared with smoother versions $p=10^{\beta-0.5k}$ and $p=10^{\beta-0.5k}$. Again, there is a trade-off between answer presence and search space size: having $p=10^{\beta-0.5k}$ gives the best answer presence, but $p=10^{\beta-0.5k}$ has a much smaller $|S|$. 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 $S$ 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.

### Table 4: Impact of Clocq on KB-QA.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>LC-QuAD2.0-CQ</th>
<th>ConvQuestions-FQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>QA system \rightarrow Method ↓</td>
<td>GRAFT-Net [49]</td>
<td>GRAFT-Net [49]</td>
</tr>
<tr>
<td>Clocq</td>
<td>0.197*</td>
<td>0.268*</td>
</tr>
<tr>
<td>ELq+Hdt</td>
<td>0.168</td>
<td>0.224</td>
</tr>
<tr>
<td>Tagme+Hdt</td>
<td>0.171</td>
<td>0.225</td>
</tr>
<tr>
<td>ConvQuestions-FQ</td>
<td>0.207</td>
<td>0.264</td>
</tr>
<tr>
<td>MRR</td>
<td>30.50*</td>
<td>28.88</td>
</tr>
<tr>
<td>Hit@5</td>
<td>0.213</td>
<td>0.256</td>
</tr>
<tr>
<td>Hit@1</td>
<td>0.167</td>
<td>0.204</td>
</tr>
</tbody>
</table>

### Table 5: Ablation study of configurations in Clocq.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>LC-QuAD2.0-CQ</th>
<th>ConvQuestions-FQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method ↓</td>
<td>Ans. pres.</td>
<td>Time</td>
</tr>
<tr>
<td>Clocq</td>
<td>0.803 1.5k 0.47 s</td>
<td>0.760 1.1k 0.34 s</td>
</tr>
<tr>
<td>w/o match</td>
<td>0.726* 1.3k 0.46 s</td>
<td>0.607* 0.9k 0.30 s</td>
</tr>
<tr>
<td>w/o rel</td>
<td>0.806 1.5k 0.47 s</td>
<td>0.746* 1.2k 0.32 s</td>
</tr>
<tr>
<td>w/o conn</td>
<td>0.790* 1.5k 0.41 s</td>
<td>0.746* 1.2k 0.26 s</td>
</tr>
<tr>
<td>w/o coh</td>
<td>0.802 1.5k 0.40 s</td>
<td>0.750 1.1k 0.24 s</td>
</tr>
<tr>
<td>w/o match + rel</td>
<td>0.733* 1.3k 0.47 s</td>
<td>0.618* 0.9k 0.30 s</td>
</tr>
<tr>
<td>w/o match + conn</td>
<td>0.791* 1.5k 0.34 s</td>
<td>0.743* 1.2k 0.22 s</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>LC-QuAD2.0-CQ</th>
<th>ConvQuestions-FQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method ↓</td>
<td>Ans. pres.</td>
<td></td>
</tr>
<tr>
<td>Clocq</td>
<td>Ans. pres.</td>
<td></td>
</tr>
<tr>
<td>$k=Auto$, $p=1k$ (default)</td>
<td>0.803</td>
<td>1.5k</td>
</tr>
<tr>
<td>$k=Auto$, $p=10k$</td>
<td>0.837</td>
<td>7.8k</td>
</tr>
<tr>
<td>$k=Auto$, $p=10^{\beta-0.5k}$</td>
<td>0.776</td>
<td>1.5k</td>
</tr>
<tr>
<td>$k=Auto$, $p=10^{\beta-0.5k}$</td>
<td>0.835</td>
<td>6.3k</td>
</tr>
<tr>
<td>$k=Auto$, $p=10^{\beta-0.5k}$</td>
<td>0.798</td>
<td>1.0k</td>
</tr>
</tbody>
</table>

### Table 7: Effect of BM25 on verbalized KB facts.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>LC-QuAD2.0-CQ</th>
<th>ConvQuestions-FQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method ↓</td>
<td>Ans. pres.</td>
<td></td>
</tr>
<tr>
<td>Clocq</td>
<td>Ans. pres.</td>
<td></td>
</tr>
<tr>
<td>$k=1$, $p=0.1k$</td>
<td>0.782</td>
<td>3.7k</td>
</tr>
<tr>
<td>+ BM25 (top-100)</td>
<td>0.625</td>
<td>0.1k</td>
</tr>
<tr>
<td>+ BM25 (top-1000)</td>
<td>0.726</td>
<td>0.8k</td>
</tr>
<tr>
<td>$k=3$, $p=0.1k$</td>
<td>0.846</td>
<td>9.7k</td>
</tr>
<tr>
<td>+ BM25 (top-100)</td>
<td>0.614</td>
<td>0.1k</td>
</tr>
<tr>
<td>+ BM25 (top-1000)</td>
<td>0.742</td>
<td>1.0k</td>
</tr>
<tr>
<td>$k=5$, $p=0.1k$</td>
<td>0.872</td>
<td>15.1k</td>
</tr>
<tr>
<td>+ BM25 (top-100)</td>
<td>0.605</td>
<td>0.1k</td>
</tr>
<tr>
<td>+ BM25 (top-1000)</td>
<td>0.747</td>
<td>1.0k</td>
</tr>
</tbody>
</table>

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

---

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)\(^2\).
Table 8: Comparison of KB interfaces w.r.t. functionalities.

<table>
<thead>
<tr>
<th>KB interface</th>
<th>QueryService</th>
<th>Hdt [18]</th>
<th>Clocq</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAM consumed</td>
<td>–</td>
<td>220GB</td>
<td>340GB</td>
</tr>
<tr>
<td>Neighborhood</td>
<td>1.48 × 10⁻²s</td>
<td>6.73 × 10⁻³s</td>
<td>4.98 × 10⁻⁵s</td>
</tr>
<tr>
<td>KB-distance</td>
<td>2.46 × 10⁻²s</td>
<td>5.43 × 10⁻³s</td>
<td>3.23 × 10⁻⁵s</td>
</tr>
</tbody>
</table>

Table 9: Timing KB interfaces for search space reduction.

<table>
<thead>
<tr>
<th>KB interface</th>
<th>QueryService</th>
<th>Hdt [18]</th>
<th>Clocq</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clocq</td>
<td>–</td>
<td>971 s</td>
<td>0.54 s</td>
</tr>
<tr>
<td>Elq [31]</td>
<td>0.89 s</td>
<td>0.62 s</td>
<td>0.12 s</td>
</tr>
<tr>
<td>TagMe [19]</td>
<td>19 s</td>
<td>1.25 s</td>
<td>0.52 s</td>
</tr>
</tbody>
</table>

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 10⁴ over Hdt and QueryService, respectively. When measuring KB-distances, the effect becomes even larger: Clocq is 10³ and 10⁴ 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 |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 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.

**Acknowledgments.** We thank Jesujoa Alabi, Zhen Jia, and Magdalena Kaiser for their useful inputs at various stages of this work. Part of this work was supported by the ERC Synergy Grant “imPACT” (No. 610150).
REFERENCES


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


[34] Nikita Buitani, Xinyi Zhang, and HV Jagadish. 2019. Learning to answer complex questions over knowledge bases with query composition. In CIKM.

