# Efficient and Effective SPARQL Autocompletion on Very Large Knowledge Graphs

Hannah Bast*
University of Freiburg
Freiburg, Germany
bast@cs.uni-freiburg.de

Johannes Kalmbach*
University of Freiburg
Freiburg, Germany
kalmbach@cs.uni-freiburg.de

Theresa Klumpp*
University of Freiburg
Freiburg, Germany
therry.klumpp@gmail.com

Florian Kramer*
University of Freiburg
Freiburg, Germany
kramerfl@cs.uni-freiburg.de

Niklas Schnelle*
University of Freiburg
Freiburg, Germany
niklas.schnelle@gmail.com

## ABSTRACT

We show how to achieve fast autocompletion for SPARQL queries on very large knowledge graphs. At any position in the body of a SPARQL query, the autocompletion suggests matching subjects, predicates, or objects. The suggestions are context-sensitive and ranked by their relevance to the part of the query already typed. The suggestions can be narrowed down by prefix search on the names and aliases of the desired subject, predicate, or object. All suggestions are themselves obtained via SPARQL queries. For existing SPARQL engines, these queries are impractically slow on large knowledge graphs. We present various algorithmic and engineering improvements of an open-source SPARQL engine such that these queries are executed efficiently. We evaluate a variety of suggestion methods on three large knowledge graphs, including the complete Wikidata. We compare our results with two widely used SPARQL engines, Virtuoso and Blazegraph. Our code, benchmarks, and complete reproducibility materials are available on https://ad.cs.uni-freiburg.de/publications .

## CCS CONCEPTS

• **Information systems → Users and interactive retrieval**.

## KEYWORDS

SPARQL, Autocompletion, Efficiency, Knowledge Graphs

**ACM Reference Format:**
Hannah Bast, Johannes Kalmbach, Theresa Klumpp, Florian Kramer, and Niklas Schnelle. 2022. Efficient and Effective SPARQL Autocompletion on Very Large Knowledge Graphs. In *Proceedings of the 31st ACM International Conference on Information and Knowledge Management (CIKM '22), October 17–21, 2022, Atlanta, GA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3511808.3557093

*Author contributions are stated in Section 7.


*CIKM '22, October 17–21, 2022, Atlanta, GA, USA*

ACM ISBN 978-1-4503-9236-5/22/10…$15.00
https://doi.org/10.1145/3511808.3557093

## 1 INTRODUCTION

In the widely used Resource Description Framework (RDF), a knowledge graph can be represented as a set of subject-predicate-object triples. Each subject, predicate, or object is either an *Internationalized Resource Identifier* (IRI), enclosed in angle brackets, or a so-called *literal*, enclosed in quotes. Here is a toy example:

```
<Meryl_Streep>      <is_a>        <Person>
<Meryl_Streep>      <gender>      <Female>
<Meryl_Streep>      <award_won>   <Oscar_Best_Actress>
<Meryl_Streep>      <birth_date>  "1949-06-22"
<Oscar_Best_Actress> <is_a>       <Oscar>
```

The standard query language for RDF is SPARQL. It allows queries with precise and easily understandable semantics. For example, the following query finds all women who won an Oscar.

```
SELECT ?entity ?award WHERE {
  ?entity   <is_a>       <Person>   .
  ?entity   <gender>     <Female>   .
  ?entity   <award_won>  ?award     .
  ?award    <is_a>       <Oscar>    }
```

The result for this query is a table with two columns, where each row contains the name of the person and the name of the award. For the example knowledge graph above, the result is:

```
<Meryl_Streep> <Oscar_Best_Actress>
```

The five example triples above come from *Fbeasy* (362 M triples) [4], an easy-to-use subset of *Freebase* (1.9 B triples) [9]. The largest general-purpose knowledge graph to date is *Wikidata* (9.9 B triples, as of 10-06-2021) [25]. Fbeasy has (human-)understandable IRIs for all entities. In Wikidata, almost all IRIs are abstract, whereas understandable names can be obtained via dedicated predicates; see the example query below. Freebase uses a mix of understandable and abstract IRIs. We consider all three knowledge graphs in this paper; see Section 5.2 for details.

SPARQL is conceptually easy, but it can be hard, even for experts, to formulate queries and to find the right IRIs. It becomes even harder when IRIs are abstract. For example, here is the correct SPARQL query on Wikidata to obtain the list of all Oscars of Meryl Streep and the movies she won them for:

```
SELECT ?award ?film WHERE {
  wd:Q873 p:P166 ?m .
  ?m pq:P1686 ?film_id .
```

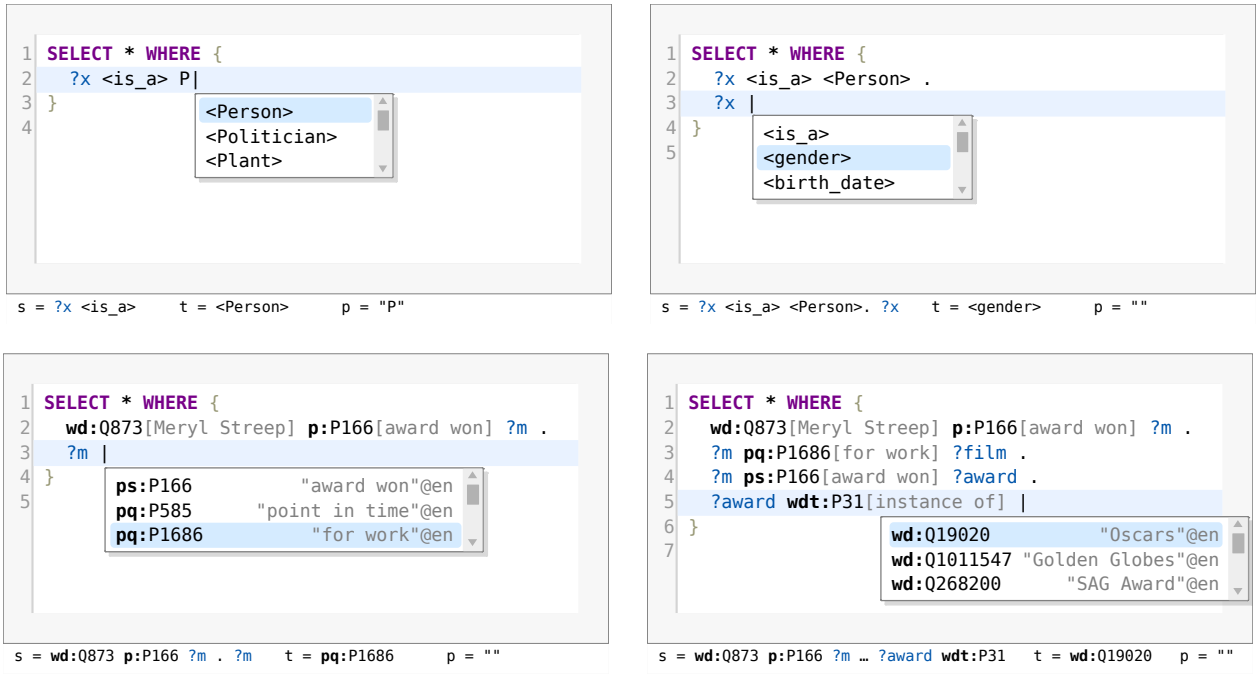


**Figure 1: Four screenshots of our autocompletion in action, with three suggestions each. Top-left and top-right: Examples 1 and 2 from Section 1.1. The assignments below each screenshot show the values of the variables from our problem definition.**

*?m ps:P166 ?award_id .*
*?award_id wdt:P31 wd:Q19020 .*
*?award_id rdfs:label ?award .*
*?film_id rdfs:label ?film }*

The *wd:*, *wdt:*, *p:*, *pq:*, *ps:*, and *rdfs:* are IRI prefixes.[1] The IRI *wd:Q873* stands for *Meryl Streep* and *wd:Q19020* for the *Academy Awards*. The *p:P166* leads to a so-called statement node, representing a particular award. The *ps:P166* leads to the award entity and the *pq:P1686* leads to the awarded film. The *wdt:P31* stands for *instance of*.

## 1.1 Problem Definition and Three Examples

The goal of this paper is to assist the user in typing the body of a SPARQL query by providing suggestions for IRIs and literals at any point in the query.[2] The suggestions should be ranked by relevance to the part of the query already typed. We first provide a formal problem definition and then explain it at length via two examples in the text and four examples depicted in Figure 1.

**Definition.** Consider a valid SPARQL query. Let $s$ be a prefix of the contents of the WHERE clause (the part the user has already typed), just before the next token (subject, predicate, or object) begins. Let $t$ be that next token (which we want the user to help finding). Let $p$ be a prefix of a name or alias of $t$, possibly empty. The *SPARQL Autocompletion via SPARQL* problem then is: Given $s$ and $p$, construct and process a SPARQL query, called *autocompletion query* or *AC query*, with the following properties:

The AC query returns a table with each row corresponding to a suggestion and the following three columns:

*?entity* (an entity from the knowledge graph),
*?name* (a name or alias of that entity, starting with $p$),
*?score* (an estimate of the relevance of this entity suggestion).

The rows are sorted in descending order of *?score*. There are three objectives:

*Relevance*: One of the rows contains $t$ in the *?entity* column and that row should be as high up in the table as possible.

*Sensitivity*: As many of the suggestions as possible should continue the SPARQL query in a meaningful way, namely such that there exists a continuation with a non-empty result. We call such a suggestion *context-sensitive*, or just *sensitive*.

*Efficiency*: The query should be processed as quickly as possible.

**Example 1** Assume that we have typed the body of the first SPARQL query from the introduction until before the first object; see below. This is the $s$ from the definition. The _ symbol marks the cursor position and the prefix $p$ is *"P"*. The token $t$ we are looking for at this position is *<Person>*. The knowledge graph is Fbeasy.

*?x <is_a> P_*

The following AC query computes a table containing each object *?entity* and its name *?name* such that the name starts with *P* and the triple *?x <is_a> ?entity* exists. The table is sorted in descending order of the number of such triples for each *?entity*.

```
1. SELECT ?entity ?name ?score WHERE {
2.   { SELECT ?entity (COUNT(?x) AS ?score) WHERE {
3.       ?x <is_a> ?entity
```

[1] Definitions omitted to save space; see https://en.wikibooks.org/wiki/SPARQL/Prefixes
[2] A typical user interface for SPARQL autocompletion also involves suggestions for variable names or SPARQL constructs like OPTIONAL, FILTER, UNION or GROUP BY at appropriate positions in the query. We omit this aspect here, as such suggestions are not particularly challenging with respect to relevance or efficiency.

```
4.    } GROUP BY ?entity }
5.    BIND(STR(?entity) AS ?name) . FILTER REGEX(?name, "^P")
6.  } ORDER BY DESC (?score)
```

The first three result rows for that query look as follows. Note that for this knowledge graph, the name of an entity is simply the IRI, interpreted as a string (that is what the *STR* function does).

| | | |
|---|---|---|
| <Person> | "Person" | 3970825 |
| <Politician> | "Politician" | 127809 |
| <Plant> | "Plant" | 60459 |

*Relevance:* The desired token $t$ is the first suggestion.

*Sensitivity:* By construction of the AC query, all suggested entities lead to a non-empty result.

*Efficiency:* Virtuoso and Blazegraph (introduced in Section 4) materialize each matching *?name* and check the regular expression for each of them. Our approach avoids this; see Section 4.

**Example 2** Now assume that we have typed the SPARQL query until the following $s$. The desired token $t$ at the cursor position is <*gender*>. The prefix $p$ is empty.

```
?x <is_a> <Person> .
?x _
```

The following AC query gives us a ranked list of *predicates* that lead to a non-empty result. The score for each predicate is the number of distinct persons that have a triple with that predicate. The reason for the DISTINCT is explained more below and in Section 3.

```
1.  SELECT ?entity ?name ?score WHERE {
2.    { SELECT ?entity (COUNT(DISTINCT ?x) AS ?score) WHERE {
3.        ?x <is_a> <Person> . ?x ?entity ?object
4.    } GROUP BY ?entity }
5.    BIND(STR(?entity) AS ?name)
6.  } ORDER BY DESC (?score)
```

The first three result rows for this AC query are as follows:

| | | |
|---|---|---|
| <is_a> | "is_a" | 3970825 |
| <gender> | "gender" | 2276146 |
| <birth_date> | "birth_date" | 1915167 |

*Relevance and Sensitivity:* The row with the desired token $t$ is second in this table and, again by construction, the AC query only returns predicates that lead to a hit. Since <*gender*> is a frequent predicate of entities of type <*Person*>, it occurs high up in the list and the user does not even have to type a single letter to get it on the first page of suggestions. It is important to note that without the first triple in the AC query above (the *Agnostic* approach evaluated in Section 5), <*gender*> would not be among the top suggestions.

*Efficiency:* Virtuoso and Blazegraph first join the two triples from line 3 into a large table (one row for each triple of each person, 37M for the query above), and then group by predicate. This is very expensive and leads to a timeout. For these engines (not for ours), we therefore remove the DISTINCT; see Section 5.3. This gives slightly worse suggestions, but at least some suggestions at all. We can solve such queries very fast (0.1s for the query above), using a general technique described in Section 4.1.

Figure 1 shows two more AC queries for the Wikidata knowledge graph, where IRIs are abstract and we need to filter on the names of these IRIs, obtained via the *rdfs:label* predicate.

It is important to note that the construction of our AC queries (Section 3) is *generic* and works for arbitrary knowledge graphs.

## 1.2 Our contributions

We consider the following as our main contributions:

- We develop the idea of providing SPARQL autocompletion via SPARQL queries. These *AC queries* can be processed by any (standard-conforming) SPARQL engine. The basic idea is already found in previous work, but in less generality, without an efficient solution, and without an extensive quality evaluation. See Sections 2 and 3.
- We extend an existing SPARQL engine such that these AC queries can be processed efficiently. Our extensions are technically challenging, comprise algorithmic ideas and algorithm engineering, and are valuable also beyond autocompletion. See Section 4.
- We show how to to realize AC queries with good (though not optimal) quality also for two existing and widely used SPARQL engines (Virtuoso and Blazegraph), despite their less efficient query processing. Again, see Section 4.
- We provide an extensive evaluation of all three SPARQL engines on three large knowledge graphs, including the complete Wikidata. In particular, we explore the trade-offs between sensitivity, relevance and efficiency. See Section 5.
- We achieve strong results. For example, on Wikidata we can realize sub-second response times with an average relevance (MRR) of 43% per token without typing anything, and over 90% when typing only three characters.
- Our code, benchmark, reproducibility materials, and a web-based tool for the interactive exploration of our results are publicly available on https://ad.cs.uni-freiburg.de/publications .

# 2 RELATED WORK

Campinas et al. [12] present and investigate AC queries for predicates and types (that is, objects of a *type* predicate). Their AC queries operate on a smaller *graph summary*, which helps efficiency, but harms relevance. In contrast, our AC queries operate on the original data and work for objects of any kind.

In a follow-up paper, Campinas [11] presents a system called *Gosparqled*, which uses AC queries similar to ours, but with a LIMIT (of 10, 100 or 500) on the inner subquery before the GROUP BY (line 4 in Section 1.1). They evaluate the effect of the LIMIT by computing the Jaccard similarity of the set of suggestions to that of the respective query without LIMIT. For each suggestion, the desired token is removed from the full query, which helps efficiency a lot because of the restrictive context. In Section 5, we evaluate their method (under the name *Sensitive-Trunc*) more realistically, by computing actual relevance of the suggestions on real queries typed from beginning to end.

Jarrar and Dikaiakos [16] present autocompletion for a variant of SPARQL, called *MashQL*. Their AC queries are only context-sensitive for *linear-shaped* queries. For example, if the user has already typed *?x1 <place of birth> ?x2 . ?x2 <country> ?x3 . ?x3* _, MashQL will consider this context. But if the user has typed *?x*

*<place of birth> <Berlin> . ?x <gender> <Female> . ?x _*, MashQL will suggest predicates without taking the previous context into account. To be able to run the AC queries more efficiently, they use two graph summaries. In one graph summary, entities with the same *outgoing* paths are grouped together and in the other one, entities with the same *incoming* paths are grouped together. These summaries only lead to context-sensitive results for the above mentioned *linear-shaped* queries.

De la Parra and Hogan [13] present a follow-up to an earlier (arXiv) version of our paper [7]. Like [12], they disregard ranking, but instead of computing a subset of the context-sensitive suggestions they compute a superset. Despite the publicly available benchmark from [7] (which they cite), they claim that "there is no existing benchmark for autocompletion" and evaluate on a very narrow class of queries (two triples, one predicate fixed, looking for suggestions for the other predicate).

Bast et al. [3] present a system called *Broccoli*, which provides context-sensitive suggestions for tree-shaped queries and depicts the queries as trees. The underlying query language is equivalent to SPARQL, restricted to trees and basic graph patterns. The focus of the paper is on extending the query language by a text-search component and on providing efficient autocompletion for this component as well.

Ferré [15] presents a system called *SPARKLIS*, which helps the user construct a subclass of SPARQL queries using a faceted-search interface and a user-friendly representation, similar to Broccoli. Suggestions are obtained via AC queries to an arbitrary given SPARQL endpoint. To address efficiency issues, there is a LIMIT as described for Gosparqled above. Prefix search and ranking are supported to a limited extent, synonyms or aliases not at all. There is a user study, but no quality evaluation like the one in Section 5.

There is a wide literature on other approaches to assist the user in creating SPARQL queries (or otherwise getting results from a knowledge graph) by other means than token-based autocompletion. In particular: *SemFacet* [1], *BrowseRDF* [21], *SPARQLets-Finder* [23], *SnipSuggest* [17], *Aqqu* [6], *Question AC* [2], *AutoSPARQL* [19], *SQLSUGG* [14]. They are not directly relevant to our work in this paper, so we omit a detailed discussion due to the space restrictions.

## 3 AC QUERY TEMPLATES

In Section 1.1, we have already seen AC queries for two concrete examples. In this section, we show how these can be generalized to arbitrary knowledge graphs. All AC queries explained in this section are provided in full on https://ad.cs.uni-freiburg.de/publications .

Our approach is completely generic. All we need for a given knowledge graph is a predicate path **%name-path%**, used to obtain names and aliases from IRIs[3] and a predicate path **%ranking-path%**, used to obtain counts for ranking IRIs if no context is given[4]. The next subsection shows how these are used in concrete AC queries. Technically, there is no need to restrict them to predicate paths; this only serves to make our AC queries easier to display.

It is also important to note that the ranking with **%ranking-path%** is only needed for AC queries *without* context, in particular, for our *agnostic* baseline (see Section 5.4). There is a meaningful default setting for any knowledge graph.[5]

**Sensitive AC queries:** We use the part $s$ of the SPARQL query body already typed; see our definition in Section 1.1. We first need to compute the **%context%**, which is the part of $s$ that is actually "connected" to the triple at the current cursor position.

Let $T$ be the partial triple that is currently being typed. Let $\mathcal{S}$ be the set of all finished triples inside $s$. Construct an undirected graph with node set $\mathcal{S} \cup \{T\}$ and an edge between two nodes if they share a variable. Then **%context%** is $s$ without $T$ and without all nodes in $\mathcal{S}$ that are not reachable from $T$.[5] It can be computed with a breadth-first search starting from $T$. Further, let **%subject%** and **%predicate%** be the subject and predicate of the triple at the current cursor position (if they already exist) and let **%prefix%** be the prefix (possible empty) typed by the user.

All our AC queries follow the same template: a subquery computing the suggested entities and their scores and a surrounding part adding the names and filtering by the given **%prefix%**. An entity can have multiple matching names, hence the outer *GROUP BY*.

```
1. SELECT ?entity (SAMPLE(STR(?name)) AS ?name)
2.                (SAMPLE(?score) AS ?score) WHERE {
     %entity-score-subquery%
6.   ?entity %name-path% ?name .
7.   FILTER REGEX(STR(?name), "^%prefix%")
8. } GROUP BY ?entity ORDER BY DESC (?score)
```

Here is **%entity-score-subquery%** for subject suggestions; for these, **%context%** is always empty; see its definition above.

```
3. { SELECT ?entity (COUNT(?r) AS ?score) WHERE {
4.   ?entity %ranking-path% ?r
5. } GROUP BY ?entity }
```

Here is **%entity-score-subquery%** for predicate suggestions. If **%subject%** is a variable, **%x%** is *DISTINCT* **%subject%**, otherwise *?object*. Together with the template above, this and the following subquery generalize Examples 1 and 2 from Section 1.1, which also provide an intuition for the score.

```
3. { SELECT ?entity (COUNT(%x%) AS ?score) WHERE {
4.   %context% . %subject% ?entity ?object
5. } GROUP BY ?entity }
```

Here is **%entity-score-subquery%** for object suggestions.

```
3. { SELECT ?entity (COUNT(?entity) AS ?score) WHERE {
4.   %context% . %subject% %predicate% ?entity
5. } GROUP BY ?entity }
```

**Agnostic and Unranked AC queries:** Agnostic queries trade off relevance for efficiency. For subject and object suggestions, the

[3]Wikidata: *rdfs:label|skos:altLabel* to obtain names for subjects and objects, and *^(<>|!<>)/(rdfs:label|skos:altLabel)* for predicates: the *^(<>|!<>)* follows an arbitrary predicate in reverse direction; Freebase: *fb:type.object.name|fb:common.topic.alias* for all; Fbeasy: the identity predicate *(!(<>|!<>))?* for all, which plays the role of the *BIND(STR ...)* in the examples from Section 1.1.

[4]Wikidata: *^schema:about* (all Wikimedia links of an entity); Freebase: *fb:type.object.type* (all types of an entity); Fbeasy: *<is-a>* (dito). A meaningful default setting would be *<>|!<>*, which matches any predicate.

[5]For the sake of brevity, this simplified description assumes a WHERE clause without FILTER, OPTIONAL, UNION, MINUS, and sub-queries. In our implementation and evaluation, we do consider these properly as well (it is straightforward). The details can be found under https://ad.cs.uni-freiburg.de/publications.

agnostic AC query is identical to the sensitive AC query for subject suggestions above (which never has context). For predicate suggestions, the agnostic AC query is like the respective sensitive query, but with empty **%context%**. In Section 5, we see that we can always process these queries in time below one second.

The unranked AC queries are just like the agnostic AC queries, but without the final ORDER BY.

# 4 EFFICIENT AC QUERIES

This section describes our main techniques to make AC queries efficient. In our evaluation in Section 5, we impose a timeout for each AC query (a user is only willing to wait so long for suggestions). Efficiency is therefore a prerequisite for quality.

We implement the extensions described in the following subsections as extensions of the open-source SPARQL engine QLever [5]. In our evaluation, we compare this to Virtuoso [24] and Blazegraph [8]. Virtuoso is one of the most widely used SPARQL engines and Blazegraph is the SPARQL engine behind the Wikidata Query Service [26]. The general architecture of all three engines is similar: they all map IRIs to internal IDs, and store multiple permutations of the triples (represented via their IDs) in order to speed up the basic query processing operations, most notably joins.

## 4.1 AC Queries for Predicates Using Patterns

Our predicate AC queries involve queries of the following kind, where **%context%** are the completed triples from the part $s$ of the SPARQL query that has already been typed; see our definition in Section 1.1.

*SELECT ?entity (COUNT(DISTINCT ?x) AS ?score) WHERE {*
*  **%context%** . ?x ?entity ?object*
*} GROUP BY ?entity*

As explained for Example 2 of Section 1.1, existing SPARQL engines materialize all matches for **%context%** . *?x ?entity ?object* before computing the GROUP BY. If **%context%** constrains *?x* little or not at all, this is very expensive to compute.

We want to stress that these queries do not just occur in the context of autocompletion. For example, users of the SPARQL endpoint for the huge UniProt knowledge graph (94B triples) formulate many "discovery" or "statistics" queries that are of a similar form as our AC queries [10]. The rule miner from [18] is based on queries like the above, but has its own data structures to process them because existing SPARQL are too slow or time out.

To answer the kind of query shown above efficiently, we make the following preprocessing. It is based on the simple observation that in typical RDF data, the set of distinct predicates is the same for many subjects. This observation has been exploited before for other problems: In [20], it is used for join optimization. In [22], it is used for automatic schema extraction.

1. Let $\mathcal{S}$ be the set of all distinct subjects in the knowledge graph. For each $x \in \mathcal{S}$, compute the set of the distinct predicates from all triples that have $x$ as subject. This set is called the (predicate) *pattern* of $x$. From these sets, compute the set $\mathcal{P}$ of distinct patterns.

2. Give consecutive IDs to the patterns from $\mathcal{P}$ and store the map from IDs to patterns in an array of size $|\mathcal{P}|$.

3. Store the map from each subject to its pattern ID in an array of size $|\mathcal{S}|$.

The following table provides statistics of this pre-processing for our three knowledge graphs. The fourth column counts the total size of the patterns, where the size of a pattern is the number of predicates and each pattern is counted once. The fifth column specifies the total memory consumption of the result of the pre-processing.

| | $\lvert\mathcal{S}\rvert$ | $\lvert\mathcal{P}\rvert$ | $\sum_{\mathbf{P}\in\mathcal{P}} \lvert\mathbf{P}\rvert$ | **Mem** |
|---|---|---|---|---|
| **Fbeasy** | 60 M | 0.3 M | 3 M | 0.3 GB |
| **Freebase** | 476 M | 3.1 M | 95 M | 2.5 GB |
| **Wikidata** | 2068 M | 4.4 M | 160 M | 9.0 GB |

To process the above query, we make use of these precomputed patterns as follows, where steps 2 and 3 can be (and are) parallelized:

1. Let $S \subseteq \mathcal{S}$ be the set of subjects *?x* from **%context%** or $S = \mathcal{S}$ if **%context%** is empty.

2. Look up the pattern IDs from all $x \in S$ in the precomputed array and compute a map $c : \mathcal{P}_S \to \mathbb{N}$ that, for each pattern ID that occurs at least once, counts how many $x \in S$ have that pattern ID. This can be done in time linear in the size of $S$.

3. For each pattern $P \in \mathcal{P}_S$, retrieve the corresponding set of predicate IDs and for each $p$ in that set, increase a counter (initially 0) by $c(P)$. This takes time linear in $\sum_{P\in\mathcal{P}_S} |P|$.

4. Sort the encountered $p$ by the final counter values. This yields the result for the query above.

The worst case for this algorithm is that every subject has a different predicate pattern and exactly one triple for each predicate. Then $|\mathcal{P}_S| = |S|$, each $c(P)$ is 1, and Step 3 does exactly what the naive algorithm described in Section 1.1 would do. However, in realistic knowledge graphs, many subjects share the exact same set of predicates, so that $|\mathcal{P}_s| < |S|$, and $\sum_{P\in\mathcal{P}_S} |P|$ is much smaller than the total number of triples of all $x \in S$.

For example, consider the AC query above for Fbeasy, with **%context%** = *?x <is_a> <Person>*. In Fbeasy, there are 4.0 M persons with a total of 37 M triples. They have only $|\mathcal{P}_s| = 115$ K distinct patterns with $\sum_{P\in\mathcal{P}_S} |P| = 1.4$ M predicates. With QLever extended as described above, the query can be solved in under 0.1 s. With the standard query processing, QLever would take 6.6 s, of which 1.6 s are spent on sorting 37 M elements.

The graph summaries from [12] (see Section 2) realize the special case when the **%context%** is one triple that specifies the type.

## 4.2 Prefix filtering

Here is an AC query, similar to the one from Example 1 in Section 1.1, but on Wikidata (*wdt:P50* relates written works to authors):

```
1. SELECT ?entity (SAMPLE(?name) AS ?name)
2.                (SAMPLE(?score) AS ?score) WHERE {
3.   { SELECT ?entity (COUNT(?entity) AS ?score) WHERE {
4.     ?x wdt:P50 ?entity
5.   } GROUP BY ?entity }
6.   ?entity rdfs:label|skos:altLabel ?name
7.   FILTER REGEX(STR(?name), "^P")
8. } GROUP BY ?entity ORDER BY DESC(?score)
```

Blazegraph and Virtuoso both time out on this query because they materialize all *?name* strings and check the regular expression for each of them. In QLever, each IRI or literal has a unique internal ID and for each data type (string, integer, float), the order by ID is exactly the same as the natural order for that data type. A FILTER with a prefix regex can then be realized with two binary searches. Only for the final result are IDs translated to the strings they represent.

This trick also permits the following optimization for longer prefixes.[6] For object AC queries, which all have the form above (what differs is the **%context%** of line 4 and the prefix in line 7 by which we filter), we simply swap line 5 with lines 6 and 7. The effect is that when the triples of line 4 yield a large intermediate result, this is now significantly reduced before the GROUP BY.

### 4.3 Caching and pinned results

We have extended QLever by a thread-safe least-recently-used (LRU) query cache, which does not only store final results of a query, but also results from the intermediate operations. We made the query planner cache-aware: the cost estimate for computing the result of a cached query is zero and the exact size is known. This is crucial for the processing of sequences of similar SPARQL queries, as it naturally happens in our setting.

Our cache also allows *pinning* results. These results will not be removed by an LRU eviction (but there is a special command to clear the cache completely). In our evaluation, we pin the results of two queries: the query that provides the canonical name, score[7], and aliases for each entity that can occur as subject or object, and the query that provides the same information for each entity that can occur as predicate. We pin the first result in two orders: by entity (for efficient joins with the **%context%**) and by alias (for efficient filtering by prefix). Even for the large Wikidata, the size of these pinned results is just 6.7 GB.

## 5 EVALUATION

In this section, we describe how we evaluated our approach, and then present and discuss the results of this evaluation. Reproducibility materials are available on https://ad.cs.uni-freiburg.de/publications. In particular, a web-based analysis tool is provided that permits an interactive exploration of the performance details for *each* individual AC query.

### 5.1 SPARQL Engines and Hardware

We evaluate our own extension of QLever, described in Section 4, against Virtuoso and Blazegraph, briefly described at the beginning of that section. All experiments were performed on a standard PC with an AMD Ryzen 7 3700X CPU, 128 GB of DDR-4 RAM and 4 TB SSD storage (NVME, Raid 0).[8] For easy reproducibility, all experiments where run inside docker containers.

QLever was configured with a memory limit of 70 GB for query processing, of which 30 GB were available to the query cache; see Section 4.3. Before each experiment, the query cache was cleared and the results of the queries without context were pinned, as explained in Section 4.3. For Virtuoso, we use the latest release candidate of the open-source edition (7.2.6), configured using the largest memory preset for 64 GB of RAM.[9] For Blazegraph, we used the latest stable release (2.1.5), configured according to Blazegraph's own recommendations for running Wikidata [8]. In particular, Blazegraph gets 16 GB for the JVM heap, while the rest of the RAM is used for disk caching by the operating system.

We took great care to get the best query times for each engine, given its capabilities. To avoid bad query plans, we used slightly different formulations of the AC queries for each engine. As already discussed for Example 2 in Section 1.1, we dropped the DISTINCT in the predicate AC queries for Virtuoso and Blazegraph because otherwise almost all of these queries would fail for these engines. Apart from this, all AC queries give the same result for each engine.

### 5.2 Knowledge Graphs

We evaluate on the following three knowledge graphs, already introduced in Section 1. We deliberately chose three knowledge graphs with related content (general knowledge in this case), but different sizes and combinations of human-understandable vs. abstract IRIs.

**Fbeasy [4]**: 362 M triples, 50 M subjects, 2 K predicates. All IRIs are simple and understandable (e.g. *<Meryl_Streep>* or *<gender>*).

**Freebase [9]**: 1.9 B triples, 125 M subjects, 785 K predicates. Entity IRIs are abstract (e.g. *fb:m.05dfkg3* for Meryl Streep), but most predicate IRIs are understandable (e.g. *fb:people.person.gender*).

**Wikidata [25]**: 9.9 B triples, 1.8 B subjects, 41 K predicates (dump from 10.06.2021). Almost all IDs are abstract (e.g. *wd:Q873* for Meryl Streep and *wdt:P21* for gender). We removed all non-English language literals to help Virtuoso and Blazegraph, because these engines do not support efficient language filters.

### 5.3 Autocompletion (AC) queries

The basis for our evaluation were all example queries from the Wikidata Query Service [26]. They cover a wide spectrum of SPARQL queries: from simple to complex, using features like UNION, OPTIONAL, MINUS, predicate paths, subqueries, and covering the whole breadth of the content in the knowledge graph. We had to exclude some queries for technical reasons, which left 301 queries for Wikidata. We translated these queries as faithfully as possible and provided that an answer existed to Freebase (115 queries) and to Fbeasy (99 queries). All the details are provided under https://ad.cs.uni-freiburg.de/publications.

From these full SPARQL queries, we generate AC queries by conceptually "typing" the queries from left to right, top to bottom. Specifically, for a given mode and a given SPARQL query, we generate AC queries from the templates described in Section 3 as follows:

1. Consider each token (subject, predicate, or object) in the query that is either an IRI or a literal.[10] For each such token do:

[6] In our evaluation, we apply this when the prefix length is $\geq 3$.

[7] The score only matters for subject AC queries, which are rare, and for agnostic queries, which we use in our evaluation as a baseline (see Section 5.4).

[8] We also ran our experiments on HDD storage (Raid 5), and found little difference. However, indexing on HDD is much slower for Virtuoso and Blazegraph.

[9] When scaling this preset up to 128 GB we found no significantly different results, but frequently ran into problems with the out-of-memory killer.

[10] We exclude the special name predicates (*fb:type.object.name* for Freebase, *rdfs:label* for Wikidata) because they occur in almost every query and are trivial to suggest and would only distort our results. In the 301 Wikidata queries, there are 408 triples of the form *?x rdfs:label ?label* (similarly for Freebase). After having typed the subject variable, *rdfs:label* is then always among the most frequent suggestions.

| **Fbeasy (315 tokens)** | | **≤ 0.2s** | **≤ 1.0s** | **Max** | **Sensitivity** | **$MRR_7$** | **$KS_7$** |
|---|---|---|---|---|---|---|---|
| Unranked | Qlever | 100% | 100% | 281ms | 0: 3% 3: 24% 7: 61% | 0: 0% 3: 39% 7: 69% | 7.21 |
| Agnostic | Qlever | 100% | 100% | 362ms | 0: 48% 3: 44% 7: 56% | 0: 25% 3: 85% 7: 97% | 3.58 |
| Sensitive-Trunc | Blazegraph | 24% | 95% | 3% > 5s | 0: 100% 3: 100% 7: 100% | 0: 50% 3: 69% 7: 82% | 4.26 |
| Sensitive-Trunc | Virtuoso | 15% | 38% | 2% > 5s | 0: 100% 3: 100% 7: 100% | 0: 56% 3: 79% 7: 87% | 3.14 |
| Sensitive | Qlever | 93% | 99% | 2605ms | 0: 100% 3: 100% 7: 100% | 0: 68% 3: 96% 7: 98% | 1.69 |
| Mixed-Trunc | Blazegraph | 25% | 98% | 1000ms | 0: 95% 3: 72% 7: 73% | 0: 50% 3: 93% 7: 95% | 2.49 |
| Mixed-Trunc | Virtuoso | 14% | 100% | 1000ms | 0: 92% 3: 72% 7: 79% | 0: 52% 3: 93% 7: 97% | 2.35 |
| **Mixed** | **Qlever** | **93%** | **100%** | **1000ms** | **0: 99% 3: 100% 7: 100%** | **0: 67% 3: 96% 7: 98%** | **1.70** |

| **Freebase (479 tokens)** | | **≤ 0.2s** | **≤ 1.0s** | **Max** | **Sensitivity** | **$MRR_7$** | **$KS_7$** |
|---|---|---|---|---|---|---|---|
| Unranked | Qlever | 100% | 100% | 355ms | 0: 3% 3: 8% 7: 16% | 0: 0% 3: 24% 7: 55% | 8.93 |
| Agnostic | Qlever | 100% | 100% | 367ms | 0: 42% 3: 25% 7: 25% | 0: 12% 3: 83% 7: 94% | 4.41 |
| Sensitive-Trunc | Blazegraph | 25% | 92% | 5% > 5s | 0: 100% 3: 100% 7: 100% | 0: 43% 3: 76% 7: 76% | 3.73 |
| Sensitive-Trunc | Virtuoso | 35% | 50% | 13% > 5s | 0: 100% 3: 100% 7: 100% | 0: 44% 3: 76% 7: 77% | 3.98 |
| Sensitive | Qlever | 91% | 98% | 3074ms | 0: 100% 3: 100% 7: 100% | 0: 60% 3: 98% 7: 99% | 1.99 |
| Mixed-Trunc | Blazegraph | 23% | 99% | 1000ms | 0: 78% 3: 69% 7: 65% | 0: 43% 3: 94% 7: 97% | 2.55 |
| Mixed-Trunc | Virtuoso | 29% | 100% | 1000ms | 0: 87% 3: 73% 7: 73% | 0: 47% 3: 96% 7: 98% | 2.40 |
| **Mixed** | **Qlever** | **90%** | **100%** | **1000ms** | **0: 99% 3: 99% 7: 100%** | **0: 60% 3: 98% 7: 99%** | **2.00** |

| **Wikidata (1244 tokens)** | | **≤ 0.2s** | **≤ 1.0s** | **Max** | **Sensitivity** | **$MRR_7$** | **$KS_7$** |
|---|---|---|---|---|---|---|---|
| Unranked | Qlever | 99% | 100% | 445ms | 0: 0% 3: 3% 7: 23% | 0: 0% 3: 9% 7: 52% | 10.89 |
| Agnostic | Qlever | 99% | 100% | 476ms | 0: 27% 3: 28% 7: 34% | 0: 6% 3: 62% 7: 91% | 6.02 |
| Sensitive-Trunc | Blazegraph | 18% | 82% | 13% > 5s | 0: 100% 3: 100% 7: 100% | 0: 44% 3: 71% 7: 67% | 4.80 |
| Sensitive-Trunc | Virtuoso | 34% | 52% | 10% > 5s | 0: 100% 3: 100% 7: 100% | 0: 44% 3: 80% 7: 84% | 3.92 |
| Sensitive | Qlever | 57% | 85% | 4% > 5s | 0: 100% 3: 100% 7: 100% | 0: 44% 3: 93% 7: 97% | 2.93 |
| Mixed-Trunc | Blazegraph | 3% | 98% | 1000ms | 0: 89% 3: 36% 7: 46% | 0: 42% 3: 74% 7: 92% | 3.94 |
| Mixed-Trunc | Virtuoso | 30% | 100% | 1000ms | 0: 84% 3: 70% 7: 72% | 0: 36% 3: 81% 7: 96% | 3.63 |
| **Mixed** | **Qlever** | **55%** | **100%** | **1000ms** | **0: 90% 3: 94% 7: 96%** | **0: 43% 3: 93% 7: 98%** | **2.86** |

**Table 1: Query times, sensitivity, and relevance for three knowledge graphs, six completion modes, and three SPARQL engines. Sensitive-Trunc is an improved version of the best previous work. For each token, three AC queries were issued, for prefix lengths 0, 3, and 7. The columns for Sensitivity and $MRR_7$ show average results per prefix length. The "Max" column shows the maximum query time or the fraction of AC queries that timed out after 5s. For $MRR_7$ and $KS_7$, the timed-out queries are treated as if the desired token appeared at position $\infty$ and the number of keystrokes required is the length of the token plus 1.**

2. Compute **%context%** as described in Section 3 (only needed for the sensitive AC queries), and depending on the position of the token, also determine **%subject%** and **%predicate%**.

3. Choose a name from the **%name-path%** predicate path (canonical name and aliases) for that token uniformly at random.

4. From that name, compute three prefixes for **%prefix%**, of lengths 0 (the empty word), 3, and 7; the next but one paragraph explains why. For prefix lengths 3 and 7, if the name has fewer characters, we change the AC query such that it requires a full-word match.

5. For each prefix length, pick the AC query template from Section 3 according to the position of the token (subject, predicate, object) and the mode (unranked, agnostic, sensitive). Plug in **%context%** and **%prefix%**, and depending on the position also **%subject%** and **%predicate%**.

We deliberately did not evaluate AC queries after *every* keystroke, for the following reason: Ideally, a user does not have to type anything, and the desired token is suggested highly ranked already for prefix length 0. But if the suggestions for prefix length 0 are not good, the user needs an idea of what to type anyway and she might as well type a few letters instead of just one. We chose 7 as a representative for a prefix length that is not too long, yet should sufficiently narrow down the search for most tokens.

## 5.4 Modes

We evaluate the following modes. The exact AC queries are available on https://ad.cs.uni-freiburg.de/publications. It is important to note that we are comparing against an improved version of the best previous work: *Sensitive-Trunc* below is similar to [11], but the latter uses no ranking and only works for predicates and types.

*Sensitive:* These are the AC queries explained in Section 1.1. They are ideal regarding sensitivity, but a challenge regarding efficiency. While our extension of QLever can handle these queries, Blazegraph and Virtuoso often time out. We therefore evaluate these engines on modified AC queries explained next.

*Sensitive-Trunc:* All our AC queries have the same structure: an "inner" part, enclosed by a SELECT ... WHERE with a GROUP BY, followed by a prefix filter; see Sections 1.1, 4.1, 4.2, and 3. For Blazegraph, we truncate this inner part (before the GROUP BY is computed) by LIMIT 10.000 on Fbeasy and by LIMIT 100.000 on Freebase and Wikidata. The same approach was used in [11] for a less general class of AC queries; see Section 2. For Virtuoso, we use its "Anytime" feature with a timeout of 5 seconds. Virtuoso then produces a subset of the full result approximately within that time frame. This feature, which is unique to Virtuoso, gives slightly better results than mere truncation.

*Agnostic:* Agnostic AC queries completely ignore the context of the token. They return all entities where a name matches the prefix, ordered by a precomputed score.[11] These queries are always fast (see Section 4), but at the expense of sensitivity and relevance.

*Unranked:* Like Agnostic, but rank the suggestions alphabetically.

*Mixed:* Simultaneously issue an agnostic and a sensitive query. If the sensitive query finishes within 1s, take that result, otherwise take the result of the agnostic query.

*Mixed-Trunc:* Like Mixed, but using Sensitive-Trunc instead of Sensitive for prefix length 0. For Blazegraph and Virtuoso, we evalute this mode instead of Mixed for the following reason that will become clear in Section 5.6: For prefix lengths 3 and 7, the relevance of Agnostic is quite good, and better than Sensitive-Trunc. For prefix length 0, the relevance of Agnostic is very poor, and with Sensitive-Trunc, Blazegraph and Virtuoso at least have a chance to produce better results (whereas Sensitive often times out).

We set the timeout for all sensitive AC queries (for all knowledge graphs and all engines) to 5s. Note that the timeout for mixed mode is just 1s. We deliberately set this lower to also explore the effects of different timeouts on relevance. Also, 1s is more what a user would expect from an interactive experience.

## 5.5 Evaluation metrics

We evaluate both objectives from our definition in Section 1.1.

*Efficiency:* We report the percentage of AC queries that can be processed faster than 0.2s (this feels close to instantaneous) and faster than 1.0s (noticeable delay, but still acceptable). If no query times out, we also report the maximum query time; otherwise, we report the percentage of AC queries that timed out.

*Sensitivity:* For each AC query, we compute the percentage of suggestions that lead to a non-empty result; see Section 1.1. By definition, this is 100% for the sensitive and sensitive-trunc AC queries. Note that agnostic or unranked queries might also produce some sensitive suggestions, but cannot identify them.

*Relevance:* Of utmost importance to a user is the rank of the desired token in the list of suggestions We evaluate this as follows. We assume that suggestions are shown on "pages" of $k$ suggestions each. Ideally, the desired token is on the first page (which is displayed after each keystroke). In our evaluation, we take $k = 7$. We use the following two metrics:

**$\mathbf{MRR}_k$ (mean reciprocal rank):** For each AC query, the reciprocal rank is $1/r$, where $r$ is the index of the suggestion page on which the desired token occurs, that is, at a position in $(r-1) \cdot k \,..\, r \cdot k - 1$, with the first position being 0. We report the mean reciprocal rank of all AC queries with a particular prefix length (0, 3, and 7). The maximum value of $\text{MRR}_7$ is 100%; it is achieved when each token appears on the first page of suggestions.

Note that the reciprocal rank is a very natural measure in our setting: we only have one relevant item and the "gain" for the user indeed decreases sharply with the index of the page where the item occurs. A user would rather continue typing instead of scrolling down much further in the list of suggestions.

**$\mathbf{KS}_k$ (number of keystrokes):** For each token, the number of keystrokes is the minimal prefix length (out of 0, 3, and 7), for which the token appears on the first page of suggestions. If it is not on the first page even for prefix length 7, we take the number of keystrokes for that token as the length of the name of the token plus 1. This corresponds to typing the full name and indicating that it is not a prefix, but the full name.

## 5.6 Main results and discussion

Table 1 summarizes our main results on all AC queries with context[12], which we now discuss. We immediately see that for all benchmarks and modes, the better the relevance ($\text{MRR}_7$), the less a user has to type to find the desired token ($\text{KS}_7$). Concerning relevance, the following discussion therefore focuses on the $\text{MRR}_7$.

We want to emphasize that the explanation of many of the following observations required a deep understanding of the respective engines and would not have been possible without the interactive exploration capabilities of our analysis tool, available under https://ad.cs.uni-freiburg.de/publications.

**Sensitive AC queries help relevance a lot.** Compare the $\text{MRR}_7$ of Agnostic and Sensitive using QLever on Wikidata. The values at prefix length 0 are 6% vs. 44%. This shows that without typing anything, the desired token is hardly ever on the first pages of suggestions with Agnostic, but frequently on the first or second page for Sensitive. This case is particularly important because if you have to type something, then you already need an idea what you are looking for; see Figure 1 on Page 2 (examples at the bottom). After typing three letters, the result is almost always on the first page for Sensitive, and Agnostic is also becoming better.

**Sensitive AC queries are feasible with QLever, but require truncation for Virtuoso and Blazegraph.** Even on the very large Wikidata, our extension of QLever can compute fully sensitive suggestions in $\leq$ 1s for 85% of all AC queries. Only 4% of these AC queries time out after 5s on Wikidata, and none at all on the smaller knowledge graphs.

Without truncation, many AC queries time out for Blazegraph and Virtuoso (47% and 25%, respectively, on Wikidata). These number are not reported in Table 1, but on https://ad.cs.uni-freiburg.

[11] The details of this scoring are provided in Section 3. For example, we use the number of Wikimedia sitelinks as a score for subjects and objects on Wikidata.

[12] With context means that not included are subject AC queries or predicate AC queries with variable subject and empty **%context%** because context-sensitivity plays no role for those queries and our methods (Agnostic, Sensitive, Mixed) all give the same results.

de/publications. With truncation, the results are on par with those of QLever for prefix length 0. The reason is that when relevance is high for an empty prefix, this usually means that there are few suggestions, in which case truncation does not harm. If there are many possible suggestions, the truncation often eliminates the desired token, which then cannot be suggested even when a longer prefix is typed (because the truncation comes before the prefix filter). Also, our extension of QLever can make use of longer prefixes for more efficient query processing, leading to less timeouts; see Section 4.2. The result is that the MRR on longer prefixes is significantly better for QLever than for Blazegraph and Virtuoso.

**Agnostic AC queries are always fast; relevance is bad for prefix length 0 but quite good for longer prefix lengths.** All agnostic AC queries can be processed in well under one second because a large part of the results are pre-computed; see Section 4.3. The only non-trivial work to do at query time is to filter the precomputed results by the typed prefix.

For prefix length 0, the desired token will rarely be among the top suggestions because of the complete lack of contextual information. But a prefix length of 3 or even 7 is often enough to restrict the suggestions sufficiently, even without **%context%** (which by definition is empty for Agnostic). This is important in order to understand the results for the mixed AC queries, discussed below.

Virtuoso and Blazegraph both perform very poorly for agnostic AC queries, which is why we do not report them in Table 1. The reason is that both engines handle prefix searches on large lists of strings very inefficiently. See the discussions in Section 1.1 (after Example 1) and in Section 4.2.

**With agnostic suggestions, typing helps relevance more than it helps sensitivity.** When typing more letters, relevance increases much more than sensitivity. For example, when typing seven letters with Agnostic on Wikidata, the $MRR_7$ is 91% and almost as high as with Sensitive, but sensitivity is only 27%. That is, the desired token will often be on the first page of suggestions, but mixed with suggestions which do not lead to a non-empty result. In a best case, the non-sensitive suggestions are merely confusing because they have nothing to do with the part of the query already typed. In a worst case, there are multiple suggestions with the same name, and the user has no way to figure out which is the desired one. The latter is not unusual for very large knowledge graphs.[13]

**Mixed AC queries are a good compromise between sensitivity and performance.** Mixed always produces a result within 1s and so never times out. The reason is that agnostic AC queries can always be processed in under 1s. The price is that some of the suggestions may not be context-sensitive. But note that a user interface could indicate whether the suggestion came from the agnostic or from the sensitive AC query.

Our extension of QLever achieves essentially the same $MRR_7$ scores as in Sensitive mode, but always within 1s and with only a small sacrifice in sensitivity. The reason is that we can process most sensitive AC queries in $\leq$ 1s.

Blazegraph and Virtuoso show an improved $MRR_7$ for long prefixes, but at the price of a sigificantly reduced sensitivity (because for many queries, the sensitive AC query times out). Also note that since Blazegraph and Virtuoso cannot efficiently process agnostic queries by themselves (for our evaluation, these queries were computed via QLever), they cannot support mixed mode out of the box. Indeed, the Wikidata Query Service (realized using Blazegraph) [26] uses a separate service for its agnostic autocompletion.

**Unranked AC queries perform very poorly on large knowledge graphs.** Recall that the suggestions of Unranked are the same as those of Agnostic, but without ranking them by score. We include this mode in our evaluation to verify how important ranking is. On Wikidata, even for a prefix length of 3, the relevance of Unranked is very poor ($MRR_7$ = 9%). For a prefix length of 7, the $MRR_7$ rises to 52%, but it's still much worse than the 91% of Agnostic. Note that ranking for autocompletion is mainly an efficiency problem: often a very large number of suggestions has to be computed and sorted. In some of the previous work we discussed, ranking was omitted due to this reason; see Section 2.

[13] For example, Wikidata knows six entities with name *female*, but only one of them is used for *wdt:P21* (gender).

## 6 CONCLUSIONS

We showed how to perform context-sensitive SPARQL autocompletion with very good relevance and efficiency, for a large variety of queries on three different knowledge graphs. All suggestions were themselves provided via standard SPARQL queries, on the same knowledge graph. That way, our scheme can be used with any standard-conforming SPARQL engine.

We saw that on very large knowledge graphs (like Wikidata), many autocompletion queries are hard for existing SPARQL engines. We showed three ways out. First, we showed how to extend an existing open-source SPARQL engine to deal with most of these hard queries efficiently. Our extensions are useful also beyond autocompletion, since they speed up classes of SPARQL queries that occur frequently. Second, we introduced a mixed mode that sacrifices sensitivity for efficiency. Third, we showed how truncation helps slower engines.

Our code, benchmark, and all materials needed for reproducing our results are publicly available on https://ad.cs.uni-freiburg.de/publications . That page also links to a demo page, where one can try our context-sensitive autocompletion live on a number of knowledge graphs (including Fbeasy, Freebase, and Wikidata).

Interesting directions for future work are: improve the pattern processing in Section 4.1 to also benefit from patterns that are similar but not necessarily identical (this would further improve query times on Wikidata), improve the running time of the sensitive AC queries that still time out (see Section 5.6), compute approximate scores via sampling in order to be able to handle queries with a very large context, and extend the autocompletion mechanism to suggest more than just individual tokens (for example, predicate paths or predicate and object at the same time).

## 7 AUTHOR CONTRIBUTIONS

J.K. and H.B. were the driving forces behind this work. J.K. did around 80% of the implementation and the evaluation. H.B. implemented the evaluation web app. J.K. and H.B. wrote the paper together. T.K. created the benchmark queries and wrote an initial version of the related work section. N.S. worked on an early prototype. F.K. implemented the pattern pre-computation.

# REFERENCES


[1] Marcelo Arenas, Bernardo Cuenca Grau, Evgeny Kharlamov, Sarunas Marciuska, and Dmitriy Zheleznyakov. 2016. Faceted search over RDF-based knowledge graphs. *Journal of Web Semantics (J. Web Semant.)* 37-38 (2016), 55–74. http://www.cs.ox.ac.uk/files/8303/main.pdf

[2] Konstantine Arkoudas and Mohamed Yahya. 2019. Semantically Driven Autocompletion. In *Conference on Information and Knowledge Management (CIKM'19)*. 2693–2701. https://doi.org/10.1145/3357384.3357811

[3] Hannah Bast, Florian Bäurle, Björn Buchhold, and Elmar Haussmann. 2012. Broccoli: Semantic full-text search at your fingertips. *Computing Research Repository (CoRR)* (2012). https://arxiv.org/pdf/1207.2615.pdf

[4] Hannah Bast, Florian Bäurle, Björn Buchhold, and Elmar Haußmann. 2014. Easy access to the Freebase dataset. In *The Web Conference (WWW'14) (Companion Volume)*. 95–98. https://ad-publications.cs.uni-freiburg.de/WWW_FreebaseEasy_BBBH_2014.pdf

[5] Hannah Bast and Björn Buchhold. 2017. QLever: A Query Engine for Efficient SPARQL+Text Search. In *Conference on Information and Knowledge Management (CIKM'17)*. 647–656. https://ad-publications.cs.uni-freiburg.de/CIKM_qlever_BB_2017.pdf

[6] Hannah Bast and Elmar Haussmann. 2015. More accurate question answering on Freebase. In *Conference on Information and Knowledge Management (CIKM'15)*. 1431–1440. https://doi.org/10.1145/2806416.2806472

[7] Hannah Bast, Johannes Kalmbach, Theresa Klumpp, Florian Kramer, and Niklas Schnelle. 2021. Efficient SPARQL Autocompletion via SPARQL. *Computing Research Repository (CoRR)* (2021). https://arxiv.org/pdf/2104.14595.pdf

[8] Blazegraph [n.d.]. Blazegraph. https://blazegraph.com/docs/bigdata_architecture_whitepaper.pdf, retrieved 30.01.2021. Wikidata setup: https://www.mediawiki.org/wiki/Wikidata_Query_Service/User_Manual.

[9] Kurt D. Bollacker, Colin Evans, Praveen Paritosh, Tim Sturge, and Jamie Taylor. 2008. Freebase: a collaboratively created graph database for structuring human knowledge. In *International Conference on Management of Data (SIGMOD'08)*. 1247–1250. https://dl.acm.org/doi/10.1145/1376616.1376746

[10] Jerven Bolleman. 2021. The UniProt SPARQL endpoint. https://sparql.uniprot.org personal communication.

[11] Stéphane Campinas. 2014. Live SPARQL Auto-Completion. In *International Semantic Web Conference (ISWC'14) Posters & Demos*. 477–480. https://pdfs.semanticscholar.org/2628/15d156def72810bad221d1f2db1799f12daf.pdf

[12] Stéphane Campinas, Thomas Perry, Diego Ceccarelli, Renaud Delbru, and Giovanni Tummarello. 2012. Introducing RDF Graph Summary with Application to Assisted SPARQL Formulation. In *International Workshop on Database and Expert Systems Applications (DEXA'12, Workshops)*. 261–266. http://www.renaud.delbru.fr/doc/pub/webs2012-sparqled.pdf

[13] Gabriel de la Parra and Aidan Hogan. 2021. Fast Approximate Autocompletion for SPARQL Query Builders. In *Visualization and Interaction for Ontologies and Linked Data (VOILA at ISWC'21)*. http://ceur-ws.org/Vol-3023/paper10.pdf

[14] Ju Fan, Guoliang Li, and Lizhu Zhou. 2011. Interactive SQL query suggestion: Making databases user-friendly. In *International Conference on Data Engineering (ICDE'11)*. 351–362. https://doi.org/10.1109/ICDE.2011.5767843

[15] Sébastien Ferré. 2017. Sparklis: An expressive query builder for SPARQL endpoints with guidance in natural language. *Semantic Web* 8, 3 (2017), 405–418. https://pdfs.semanticscholar.org/e11f/644f0296f8c0a0342790a7ef20fc2ea94ae1.pdf

[16] Mustafa Jarrar and Marios D. Dikaiakos. 2012. A Query Formulation Language for the Data Web. *IEEE Transactions on Knowledge and Data Engineering* 24, 5 (2012), 783–798. http://www.jarrar.info/publications/JD10b.pdf

[17] Nodira Khoussainova, YongChul Kwon, Magdalena Balazinska, and Dan Suciu. 2010. SnipSuggest: Context-Aware Autocompletion for SQL. *Proceedings of the VLDB Endowment (PVLDB)* 4, 1 (2010), 22–33. https://doi.org/10.14778/1880172.1880175

[18] Jonathan Lajus, Luis Galárraga, and Fabian M. Suchanek. 2020. Fast and Exact Rule Mining with AMIE 3. In *European Semantic Web Conference (ESWC'20)*. 36–52. https://doi.org/10.1007/978-3-030-49461-2_3

[19] Jens Lehmann and Lorenz Bühmann. 2011. AutoSPARQL: Let Users Query Your Knowledge Base. In *Extended Semantic Web Conference (ESCW'11)*. 63–79. https://doi.org/10.1007/978-3-642-21034-1_5

[20] Thomas Neumann and Guido Moerkotte. 2011. Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In *International Conference on Data Engineering (ICDE'11)*. 984–994. https://doi.org/10.1109/ICDE.2011.5767868

[21] Eyal Oren, Renaud Delbru, and Stefan Decker. 2006. Extending Faceted Navigation for RDF Data. In *International Semantic Web Conference (ISWC'2006)*. 559–572. https://link.springer.com/content/pdf/10.1007%2F11926078_40.pdf

[22] Minh-Duc Pham, Linnea Passing, Orri Erling, and Peter A. Boncz. 2015. Deriving an Emergent Relational Schema from RDF Data. In *The Web Conference (WWW'15)*. 864–874. https://doi.org/10.1145/2736277.2741121

[23] Karima Rafes, Serge Abiteboul, Sarah Cohen Boulakia, and Bastien Rance. 2018. Designing Scientific SPARQL Queries Using Autocompletion by Snippets. In *International Conference on e-Science (e-Science'18)*. 234–244. https://ieeexplore.ieee.org/document/8588657

[24] Virtuoso [n.d.]. OpenLink Virtuoso. http://docs.openlinksw.com, retrieved 30.01.2021. RDF Index Scheme: http://docs.openlinksw.com/virtuoso/rdfperfrdfscheme.

[25] Denny Vrandecic and Markus Krötzsch. 2014. Wikidata: a free collaborative knowledge base. *Communications of the ACM (Comm. ACM)* 57, 10 (2014), 78–85. https://dl.acm.org/doi/10.1145/2629489

[26] Wikimedia [n.d.]. Wikidata Query Service (WDQS). https://query.wikidata.org. Example queries retrieved on 23.10.2020.