University of Freiburg

Department of Computer Science

Master's Thesis

Efficient SPARQL Autocompletion on Large Knowledge Bases

presented by Johannes Kalmbach

Matriculation Number: 4012473

Supervisor and First Reviewer:
Prof. Dr. Hannah Bast

Second Reviewer:
Prof. Dr. Fang Wei-Kleiner

August 2, 2021

Contents

Abstract

We describe an autocompletion system, that is able to help users formulate complex SPARQL queries on large knowledge bases. As the user types a SPARQL query, they are presented with suggestions for the entity that is currently under the cursor. These suggestions are *context-sensitive*. This means that they form a useful continuation to the part of the query previously typed. We obtain these suggestions by executing special SPARQL queries which we call *autcompletion queries (AC queries)*. We formalize this setting as the *SPARQL Autocompletion via SPARQL* problem. We describe a template-based approach that allows the automatic generation of the AC queries. This approach can easily be adapted to a wide range of knowledge bases. We provide technical improvements to the QLever SPARQL engine that allow us to efficiently process the AC queries. We also provide an extensive evaluation of our setup using multiple knowledge bases and query engines. This evaluation shows, that context-sensitive autocompletion leads to significantly better suggestions than alternative approaches which do not consider the context. We also show that our improved version of QLever is able to process 90% of a diverse set of context-sensitive AC queries in less than 1 second even on the large Wikidata knowledge base (6.9 Billion statements). This makes our approach feasible for a pratical user interface because the high-quality context-sensitive suggestions can be presented to the user interactively as they type.

1 Preliminaries

Knowledge bases, as defined by the *RDF* standard (*Resource Description Framework*) are one of the most important methods for storing structured data. The standard way of retrieving information from such knowledge bases is the *SPARQL query language*. In this section we will first introduce RDF and SPARQL. We limit ourselves to the aspects of these standards that are necessary to understand the remainder of this thesis. We then introduce Wikidata, Freebase and Fbeasy, three general-purpose knowledge bases. We will then explain, why it is in practice often hard to manually formulate useful SPARQL queries on large knowledge bases. This discussion forms the motivation for the context-sensitive autocompletion system which we will introduced in the subsequent sections.

1.1 The Resource Description Framework

The Resource Description Framework (RDF) is a standardized way to formulate structured data or knowledge.[1] Pieces of information are stored as triples of the form *Subject Predicate Object*. A set of triples is referred to as a *knowledge base*. Consider for example the following simple knowledge base which consists of three triples:

<Cillian Murphy> <profession> <Actor> .
<Cillian Murphy> <played-role> <Scarecrow> .
<Cillian Murphy> <played-role> <Thomas Shelby> .

In the first triple, the subject is *<Cillian Murphy>*, the predicate is *<profession>*, and the object is *<Actor>*. Triple elements enclosed in angle brackets <> are *Internationalized Resource Identifiers*

[1] The full standardization documents can be found at `https://www.w3.org/RDF/`.

(IRIs) that uniquely identify an *entity*. To enforce this uniqueness, real-world IRIs are typically much longer and start with an URL prefix that is controlled by the party "responsible" for this entity. If we consider the above triples to be part of a dataset called "exampleKB" that is maintained at the chair of Algorithms and Data Structures in Freiburg, the first triple of the small knowledge base above realistically could look as follows:

<http://ad.cs.uni-freiburg.de/exampleKB/Cillian_Murphy>
<http://ad.cs.uni-freiburg.de/exampleKB/profession>
<http://ad.cs.uni-freiburg.de/exampleKB/Actor>

To make this very verbose form more readable, we can equivalently abbreviate the common prefix of the entities using a *PREFIX* declaration:[2]

PREFIX kb: <http://ad.cs.uni-freiburg.de/exampleKB/>
kb:Cillian_Murphy kb:profession kb:Actor

In addition to entities identified by IRIs, which were just described, there is only one other type of triple elements, the socalled *literals*. Literals can only appear as the object of a triple. They can be annotated by a language tag that links the literal to a certain language, or by a datatype to denote e.g a numeric or date literal. This is best shown by a few examples:

PREFIX xsd:<http://www.w3.org/2001/XMLSchema#>
<Cillian Murphy> <name> "Cilian Murphy"@en .
<Cillian Murphy> <name> "Killian Mörfi"@az .
<Cillian Murphy> <date of birth> "1976-05-25T00:00:00"^^xsd:dateTime .

Here the language tag *@az* stands for the Azerbaijani language. RDF knowledge bases typically use datatypes defined by the *XML Schema Definition (XSD)* (e.g. *xsd:dateTime*).

In this thesis we will always use the above definition of a knowledge base as a set of triples. Note that it is equivalently possible to define a knowledge base as a graph, where the subjects and objects are nodes and the predicates are labeled edges. For this reason, a knowledge base is sometimes also referred to as a *triple graph* or *knowledge graph*. Some technical terms from the SPARQL language, which will be described in the next section, also refer to this graph terminology.

1.2 The SPARQL Query Language

This section describes SPARQL, the standard query language for RDF data. SPARQL was standardized by the W3C consortium.[3] The general idea behind SPARQL is simple: SPARQL uses a syntax for querying a knowledge base, that is very similar to the syntax for specifying the knowledge base. A SPARQL query also contains triples, but some triple elements are replaced by *variables* that can assume the value of an arbitrary but fixed entity or literal (The variable is then *bound* to this value). The query result consists of all variable bindings, for which a subset of the

[2]Using *PREFIX* declarations when specifying a knowledge base is only allowed when storing the knowledge base using the *RDF Turtle (.ttl)* format and not in the simpler *RDF NTriples (.nt)* format, but this distinction is not important for this thesis, as it is of a purely syntactical nature.

[3]The full standard can be found at `https://www.w3.org/TR/sparql11-query/`.

knowledge base matches the triples in the query. Consider the following simple SPARQL query (it refers to the example knowledge base from the previous section):

SELECT ?actor ?role WHERE {
  ?actor <profession> <Actor> .
  ?actor <played-role> ?role .
}

The inner part of the query where the pattern is specified (between the curly braces) is called the *query body*. The result of a SPARQL query is a table with one column per selected variable (*?actor* and *?role* in the example) and one row for each result (one row for each pair of actor and role that match the query body). In addition to this basic triple format, SPARQL also supports more complex features. In the following we will deliver a short description of several of these features that are important for understanding this thesis.

Order By

An Order By clause stands after the query body and sorts the result according to the values of the specified variables. For example appending the clause *ORDER BY DESC(?actor) ASC(?role)* to our example query would sort the result in descending order by the value of the *?actor* variable. If this variable has the same value for some rows, these rows will be sorted in ascending order by the *?role* variable.

Group By and Aggregates

A Group By clause specifies one or more variables which are *grouped*. This means that all result rows that have the same value for these variables are merged into one row for the final result. For each variable that is selected, but not grouped, a so-called *aggregate* has to be specified that defines, how the multiple different values for these variables are merged in the final result. Consider the following example:

SELECT ?actor (COUNT(?role) AS ?role_count) WHERE {
  ?actor <profession> <Actor> .
  ?actor <played-role> ?role .
} *GROUP BY ?actor ORDER BY DESC(?role_count)*

The result will have exactly one row for each actor that has at least one *<played-role>* triple. It will have two columns, *?actor* and *?role_count*. The second column will contain the number of *<played-role>* triples for the corresponding *?actor*. (This is what the *COUNT* aggregate does). Actors with more played roles will appear higher up in the result because of the ORDER BY clause.

SPARQL supports a wide range of aggregates, from simple ones like *COUNT, MIN, MAX, SUM, etc.* to almost arbitrary arithmetic expressions and string functions. Other than *COUNT*, the only aggregate that is relevant for this thesis is *SAMPLE*, which chooses one of the different values that have to be aggregated.

OPTIONAL

We can think of each triple in a SPARQL query as a constraint. Possible results (aka variable bindings) are removed from the final result when they can't fulfill these constraints because one of the variables can't be bound. This behavior can be relaxed by an OPTIONAL clause. Triples that occur inside this clause do not eliminate possible results when the variables in these triples cannot be bound, but these variables remain unbound. One can also think of this as assigning a special *NULL* or *UNDEFINED* value to this column of the result. This behavior is best explained by an example. In the query

```
SELECT ?actor ?role WHERE {
  ?actor <profession> <Actor> .
  ?actor <played-role> ?role .
}
```

we only get actors in the result, that have played at least one role. If we now put the second triple into an OPTIONAL clause, we obtain the following query:

```
SELECT ?actor ?role WHERE {
  ?actor <profession> <Actor> .
  OPTIONAL {?actor <played-role> ?role}
}
```

We now still get a result with two columns *?actor* and *?role* and the entities in the *?actor* column always have to be actors (the first triple in the query is outside of the OPTIONAL clause and thus mandatory). For all actors, that have played at least one role, we get exactly the same results as in the "original" query without the OPTIONAL. Actors that haven't played any role are now included with a single result row in which the *?role* variable is undefined.

In addition to this simple case (A single triple inside a single OPTIONAL clause) there are some more complex behaviors of the OPTIONAL feature, e.g. when there are multiple optional clauses in the same query that share common variables. These rules are beyond the scope of this thesis, the full specification of OPTIONAL clauses can be found in the SPARQL standard.

UNION

By using a UNION clause it is able to form the concatenation of two intermediate results. Consider the following example that returns all actors who have won at least one Oscar or one Golden Globe, together with the total number of such awards per actor:

```
SELECT ?actor (COUNT(?actor) as ?num_awards) WHERE {
  ?actor <profession> <Actor> .
    {?actor <won-award> <Academy Award>}
  UNION
    {?actor <won-award> <Golden Globe>}
} GROUP BY ?actor
```

MINUS

With a MINUS clause, results that match certain triples are eliminated instead of added to the result. For example the following query yields all actors who have not won any Academy Award:

SELECT ?actor WHERE {
  ?actor <profession> <Actor> .
  MINUS { *?actor <won-award> <Academy Award>* }
}

Subqueries

Subqueries are complete SPARQL queries that are nested inside the body of another SPARQL query. Although all queries that contain subqueries can be equivalently rewritten without this feature, using subqueries often helps the readability of more complex queries. Sometimes using subqueries can also help a query engine to perform a query faster (see section 6.1 for details).

1.3 Real-World Knowledge Bases

In this section we will introduce the three knowledge bases which are relevant for this thesis and on which we have run our evaluation. This introduction stands at such an early point in this thesis because we need some of the properties of these knowledge bases to motivate our problem definition. We have chosen three knowledge bases of different sizes: *Wikidata*, *Freebase* and *Fbeasy* (ordered from largest to smallest). All three store general knowledge. We have deliberately chosen knowledge bases with similar contents but different sizes. This allows us to evaluate the influence of a knowledge base's size on our evaluation results.

Wikidata

Wikidata [29] is the knowledge base of the Wikimedia project. It is thereby a sister project of Wikipedia. For instance the information boxes in many Wikipedia articles are automatically generated using structured knowledge from Wikidata. Like all Wikimedia projects, Wikidata is community-based, which means that everyone can easily contribute data. While this approach helps the Wikidata project to grow quickly, it also poses some problems: Sometimes the way that information is stored in triples (e.g. which predicates are being used) is inconsistent between similar pieces of information. There are also many mistakes in this dataset, which are not always quickly eliminated. We will later see an example for such a mistake that motivates the use of our autocompletion system. In Wikidata, all IRIs are abstract IDs like *Q42* (the author Douglas Adams) or *P106* (The predicate for the occupation of a person). The dump of Wikidata we used contains ca. 6.9 billion triples, ca. 1.2 billion subjects, and ca. 32000 predicates.

Freebase

Freebase [4] also is a community-based knowledge base. It was originally maintained by the american company Metaweb and purchased by Google in 2010. In May 2016 it was discontinued and

its contents were integrated into the Wikidata project.[4] Freebase uses abstract IDs for most subjects and objects (e.g. *fb:m.02_286* for New York City) and human-readable names for predicates and some common subjects or objects (e.g. *fb:people.person.place_of_birth* or *fb:people.person*). Freebase has ca. 1.9 billion triples, ca. 125 million subjects and ca. 785000 predicates. Although Freebase is discontinued, we have decided to integrate it into our evaluation, since it has a different size characteristic than Wikidata: Although it has less than one third of Wikidata's triples, it has many more predicates.

Fbeasy

Fbeasy [8] (short for *Freebase Easy*) was created at the chair of Algorithms and Data Structures at the University of Freiburg. It is basically a subset of Freebase where all IRIs are human-readable names, e.g. *<New_York_City>* or *<Person>*. Freebase has ca. 362 million triples, ca. 50 million subjects and ca. 2000 predicates.

1.4 Motivation: Formulating Proper Queries is Hard

In the previous sections we have discussed, how structured knowledge can be stored in an RDF knowledge base and how the SPARQL language can be used to retrieve certain information from a given knowledge base. In this section we will discuss, why in practice it is often hard to manually formulate "appropriate" SPARQL queries on a large knowledge base. By appropriate we mean queries that deliver results that the user expected (this will become clearer with the examples below). Some of the aspects we discuss refer to concrete knowledge bases, namely *Wikidata*, *Freebase* and *Fbeasy* (see section 1.3).

SPARQL is Conceptually Easy, but Tricky in Practice

In general, formulating syntactically correct SPARQL queries is relatively simple. There are relatively few keywords, and these are very similar to those used in other common query languages like SQL. The triple-based syntax for the query body is straight-forward to learn. One could also argue, that SPARQL is easier than SQL, since it does not include table or column identifiers which have to be incorporated into the query. This fact on the other hand makes it often harder to formulate good queries, because the data format (RDF) provides hardly any external structuring to the data: There are only triples of subjects, predicates and objects. Additionally, each entity can in principle occur in each of those positions. Because of this very limited structuring, RDF is also sometimes referred to as a *semi-structured* format. On the one hand this format is very flexible and allows capturing also complex and heterogeneous relations between entities that can't easily be represented in a fixed, table-based scheme like in relational databases. On the other hand, this lack of structure makes it often hard to formulate good queries: When formulating a triple for a query, one has to find the proper entities and additionally has to find out, in which position (subject, predicate, object) they have to be placed to match the structure of the knowledge base. In the following we will discuss some detailed aspects of query formulation whose difficulty often

[4] `https://web.archive.org/web/20190320195748/https://plus.google.com/109936836907132434202/posts/bu3z2wVqcQc`

can be related to the general lack of external structure in RDF knowledge bases that was just described.

Matching Labels to IRIs can be Tedious

Consider the example from section 1.2: We are given a general-purpose knowledge base and want to retrieve a list of all actors and the roles they played. From the general structure of the SPARQL language, an experienced user might immediately get the idea, that a triple that looks somewhat like this should be part of the query:

?x <profession> <Actor>

To formulate this triple for a concrete knowledge base, we have to find the appropriate entities from the knowledge base, which correspond to the concepts of "profession" and "actor" (we will limit ourselves to the predicate "profession" in the following example). This task is easiest on the knowledge base *Fbeasy*, where the IRIs of entities directly are readable names enclosed in angled brackets. Even here we have to find out if the predicate that connects people to their professions is called *<profession>* or *<occupation>*, only to find out, that *<is-a>* would be the appropriate choice, because in Fbeasy, professions are part of the more general type predicate *<is-a>*.

The task of finding the correct predicate gets even harder for Freebase, where the predicates still have somewhat readable names. Here the correct profession predicate is *fb:people.person.profession*. Here *fb:* is the general prefix for all Freebase entities. From this entity's name we can see, that the predicates in freebase are hierarchically structured: From the name *people.person.profession* we could deduce, that this predicate does not only denote the profession, but that it is also only to be used with people, and that there might be other predicates that start with *people.person* that could be useful for queries about people. This is true indeed, e.g. there also is a *fb:people.person.place_of_birth* predicate on Freebase. This structuring of the predicates however only occurs on the level of entity names and is not represented in the RDF syntax.

Our task of finding a predicate that connects people with their professions becomes the hardest on *Wikidata*. Here all the IRIs are abstract IDs. The predicate for professions is called *wdt:P106*. The *wdt:* prefix is reserved for so-called *direct* or *truthy* predicates (see the third example in section 2.2 for details). *P106* is an abstract ID that stands for the profession of a person. The connection between this abstract ID and the human readable-names "profession" and "occupation" is also contained in the Wikidata knowledge base. We will later make use of this connection in our autocompletion system which resolves abstract IDs of Wikidata (and also Freebase) to human-readable names which are then presented to the user.

In Conclusion we have seen in this paragraph, that it is hard in practice to find a specific entity in a knowledge base, even if we already have a natural language description of this entity.

Entity Labels are Ambiguous

The following problem mostly occurs on Wikidata: There are many different entities that have similar or even identical labels. In many cases only one of these entities is the correct one when formulating a query. When we for example want to retrieve information about women from Wikidata, the correct way to do this would be by retrieving humans with female gender, s.t. in our "pseudo-triples" this part of the query could look as follows:

?x <is-a> <Human> . ?x <gender> <Female>

Here at some point we have to find the correct IRI for the *<Female>* object. In Wikidata there are five entities that have "female" as a label[5] and many more that have a label that starts with "female". Only one of these is correct in our context (when using it in connection with humans), another one is used for plants and animals, and some of the others can be even considered mistakes: In the version of Wikidata we used for our experiments, there was a female film maker that had "female" as a label, which is probably not correct. To choose the correct entity among these candidates we could for example try all of them out and check, for which of them the query result becomes useful (which typically means non-empty). However, performing this step manually quickly becomes tedious and even infeasible if we have to do so for many entities in a longer query. It is however useful to keep this manual approach in mind: We will later use an automated version of it in our context-sensitive autocompletion system (see section 4 for details).

Complex N-Ary Relations are Hard to Guess

This problem again mostly occurs on Wikidata. Consider the following request, that seems to be simple at first: For the actress Meryl Streep obtain all Oscars which she has won, as well as the corresponding movies. The correct query to do this on Wikidata is the following:

```
SELECT ?award ?film WHERE {
  wd:Q873 p:P166 ?m .
  ?m ps:P166 ?award_id .
  ?m pq:P1686 ?film_id .
  ?award_id wdt:P31 wd:Q19020 .
  ?award_id rdfs:label ?award .
  ?film_id rdfs:label ?film }
```

We will now explain this query: Meryl Streep is *wd:Q873* and *P166* stands for "won award". The *p:P166* predicate is used to connect Meryl Streep to an *abstract statement node* (*?m*) that stands for all the information connected to a single award that she has won. The *ps:P166* predicate connects the abstract node to the ID of the concrete award (e.g. the *Academy Award for Best Actress*) and the *pq:P1686* predicate connects the statement node to the ID of the film for which the award was won. The triple *?award_id wdt:P31 wd:Q19020* means that the award *is-a* (*wdt:P31*) *Academy Award* (*wd:Q19020*). The two triples with the *rdfs:label* predicate connect the IDs of the films and awards to a human readable label. The result of this query is then the following:

Academy Award for Best Actress	*Sophie's Choice*
Academy Award for Best Supporting Actress	*Kramer vs. Kramer*
Academy Award for Best Actress	*The Iron Lady*

The structure this query can also be visualized as a graph as seen figure 1. The reason for the complexity of this query lies in the *n-ary* nature of the information : The information which award was won for which movie at which point in time has to be connected via an abstract statement

[5]Wikidata distinguishes between primary labels, of which there is only one per entity, and alternative labels or aliases. We use the term "label" to denote all of those labels and aliases.

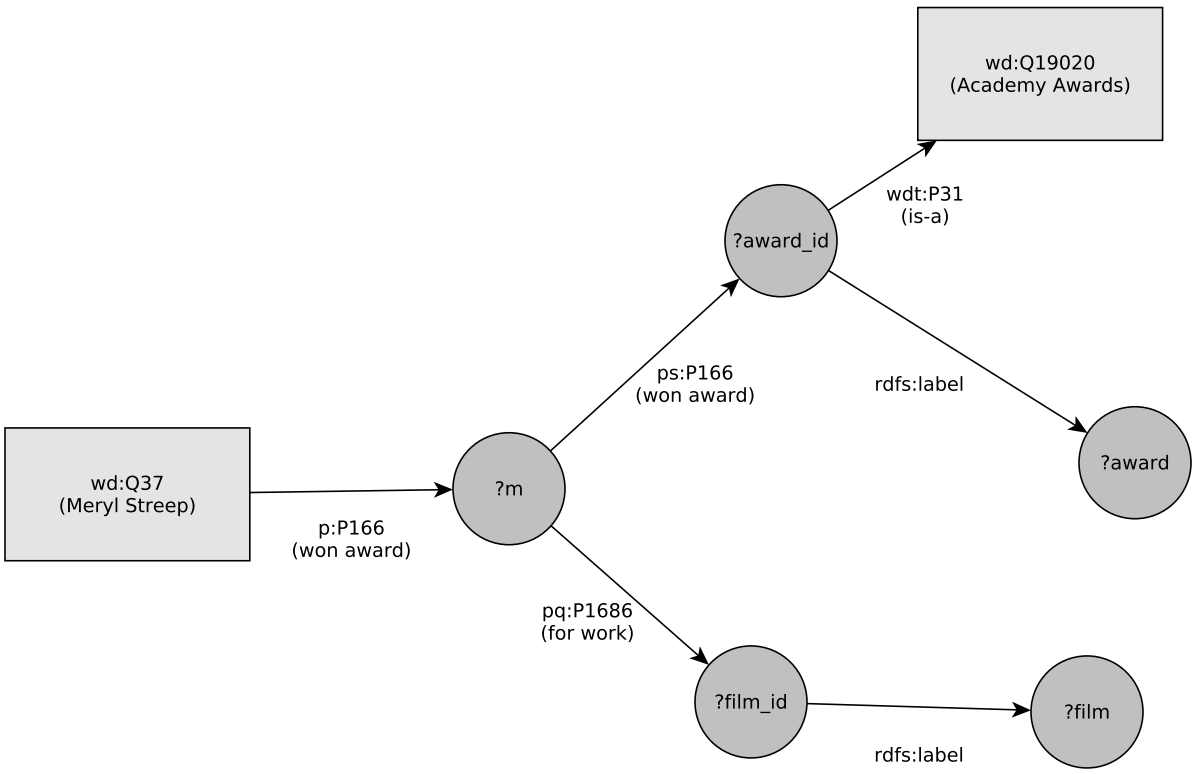

Figure 1: Visualization of the SPARQL query "All Oscars of Meryl Streep and the films she won them for" as a graph. Subjects and Objects of triples are nodes in the graph, predicates are labeled edges. Variables are visualized as circles, graph nodes with fixed IRIs are visualized as rectangles.

node (*?m* in the query). It would not suffice to store the information in the format "Meryl Streep won an oscar for the move Sophie's Choice", "Meryl Streep won an Academy Award for Best Actress", etc. because the important information which award was won for which movie would be lost. Unfortunately the resulting structure makes the SPARQL queries more complex. Even for people with experience with SPARQL and Wikidata it is tedious to correctly formulate such queries.

Summary

We have discussed that it is in practice often hard to find the appropriate entities to be used in a SPARQL query. For instance it is hard to find entities when we are already given a name for that entity, because names are often ambiguous. It is also hard to formulate queries which include n-ary information, because the formulation requires detailed knowledge about how this information is stored in the concrete knowledge base. In the following we will introduce a formal problem definition that helps us tackle these issues. Afterwards we will introduce an efficient and effective solution to that problem which can in practice be used to mitigate the difficulties discussed in this section.

2 Problem Definition

In this section we will formally define the *SPARQL Autocompletion Via SPARQL* problem. We will discuss, how solving this problem can help in formulating SPARQL queries. We will also discuss the scope and limitations of our problem definition.

2.1 The SPARQL Autocompletion via SPARQL Problem

Let q be a SPARQL query, called the *target query*. Let s be a prefix of the query body of q. We require that s only consists of complete tokens. The term "token" denotes the subject, predicate, or object of a triple. In the following we often say that "s has already been typed" to describe this setup. Let t be the token in the target query q that stands directly after s. We call t the *target token*. Let p be a prefix of a name or alias of t. The *SPARQL Autocompletion via SPARQL* problem is then defined as follows:

We are given s and p and want to construct and process a SPARQL query called the *autocompletion query* (often abbreviated by *AC query* in the following). This AC query must have the following properties:

- The result of the AC query has three columns: The first contains an entity, the second one a human-readable name for that entity and the third one contains a score. The query is sorted in descending order of the scores. In the following we often call the results of AC queries *suggestions*.
- All the names in the second result column start with the prefix p.
- One of the result rows contains the target token t in the first column.

To assess the quality of the constructed AC queries, we consider the following goals:

- **Efficiency**: The AC query shall be processed as quickly as possible.
- **Relevance**: The row containing t stands as early as possible in the AC query's result.
- **Sensitivity**: When choosing an entity $u \neq t$ from the first result column, then it is possible to formulate a SPARQL query, whose query body starts with $s + u$ (string contatenation) and which leads to a non-empty result. We say that u *sensitively continues* the partial query s.

2.2 Examples

In this subsection, we will illustrate our formal problem definition with some examples. These examples will also show, how this definition relates to the setting of a user typing a SPARQL query. First consider the following simple query on Fbeasy:

SELECT ?x WHERE {
  ?x <is-a> <Person> .
  ?x <gender> <Female> .
}

This is our target query q from the definition. Assume now, that a user wants to type this query, and that the user has already entered the following part of the query body into an editor:

*?x <is-a> P_*

The underscore denotes the current cursor position when typing. The first two tokens *?x <is-a>* are then the s from our definition and the prefix p is *"P"*. This means that the user is looking for an entity that is a useful continuation of the *<is-a>* predicate and whose label starts with "P". The target token t is *<Person>*. On Fbeasy, we can formulate a corresponding AC query as follows:

```
1. SELECT ?entity ?name (COUNT(?x) AS ?score) WHERE {
2.     ?x <is_a> ?entity .
3.     BIND (STR(?entity) AS ?name) .
4.     FILTER REGEX(?name, "^P")
5. } GROUP BY ?entity ?name ORDER BY DESC(?score)
```

Line 3 obtains labels for the *?entity* variable by converting an IRI of the form *<Entity>* to the literal *"Entity"*.[6] The result of this conversion is written to the variable *?name*. Line 4 allows only results where the *?name* starts with "P". The *?score* of an entity is the number of triples in the *<is-a>* relation for this entity, calculated by the GROUP BY and COUNT operation. The first lines of the result of this AC query look as follows:

<Person>	*"Person"*	*3970825*
<Politician>	*"Politician"*	*127809*
<Plant>	*"Plant"*	*60459*

[6] Recall, that the IRIs are directly human-readable on Fbeasy, see section 1.3. The conversion from IRIs to literals via STR() is required by the SPARQL standard, because string operations like regex/prefix filters are not supported directly on IRIs.

We immediately see, that our target token *<Person>* is in the first result row, so the relevance is perfect. Due to the way our AC query was designed, the sensitivity is also perfect: We only obtain results that have at least one triple of the form *?x <is-a> ?result*. For a discussion of the efficiency of such AC queries, we refer to our evaluation in section 6.

As a second example, consider the same target query. Assume that we have completed the first triple and the subject (a variable) of the second triple. The situation now looks as follows:

?x <is-a> <Person>
?x _

This is the s from the definition, the prefix p is empty, the target token t is *<gender>*. The AC query now needs to suggest predicates which are useful in connection with persons (from what the user has typed, the variable *?x* is already restricted to "being a person"). Such an AC query can be formulated as follows:

```
SELECT ?entity ?name (COUNT(DISTINCT ?x) AS ?score) WHERE {
  ?x <is_a> <Person> .
  ?x ?entity [] .
  BIND (STR(?entity) AS ?name) .
} GROUP BY ?entity ?name ORDER BY DESC(?score)
```

The score we chose here for a certain predicate in the *?entity* column is the number of persons (*?x*) that occur as the subject of a triple with this predicate. Persons that have multiple triples for the same predicate are counted only once, hence the DISTINCT. The design choices for the ranking functions will be discussed in detail in section 4. The first result rows of this AC query are as follows:

<is_a>	*"is_a"*	*3970825*
<gender>	*"gender"*	*2276146*
<birth_date>	*"birth_date"*	*1915167*

In this case, we get our target token *<gender>* as the second suggestion, which is again very good. Note that in this case we did not have to type a single letter, so we did not even have to know a label for the entity we were looking for. The sensitivity is also again perfect by design of the AC query.

2.3 Discussion

In this subsection we will discuss some important aspects and also limitations of the problem definition.

- We assume that a hypothetical user is typing a SPARQL query. Even with good entity suggestions, this scenario still requires some knowledge about the syntax of SPARQL and some intuition about the way, structured information is stored as triples. There are other approaches towards even easier accessibility of knowledge bases. These include graphical UIs that hide the SPARQL syntax or systems that transform natural language questions into SPARQL queries. We do not consider such approaches in this thesis and refer to the related work section (section 3) as a starting point for readers, who are interested in them.

- Our definition assumes, that the user knows a label or alias for the target token. In practice this works well for knowledge bases where entities have a lot of aliases and where it is thus likely to correctly guess one of them. This is for example true for Wikidata. When using this autocompletion setup with a knowledge base that only has few aliases per entity, one could consider extending the autocompletion system with a synonym database to increase the user's chances to find the correct entity. This could be especially helpful for knowledge bases like Fbeasy, where every entity only has exactly one label. Such external synonym finders are also beyond the scope of this thesis. It is also worth noting that this problem only arises when the user has to type anything at all. In the second example in section 2.2 the target token was among the first results of the AC query without typing anything. In these cases the user does not have to know an alias of the entity. In our evaluation (section 6) we will explicitly evaluate the quality of the autocompletion results when no prefix is typed.
- In our problem definition and evaluation, we assume that a concrete target query, which the user wants to complete, exists. This scenario is somewhat artificial. In our experience, users typically start with a natural language description for a desired piece of information, which they want to retrieve from a knowledge base (e.g. "All female mayors of cities with more than 1 million inhabitants"). However it is hard to quantitatively evaluate such a scenario. It could be evaluated by conducting a user study, where users are first given a general introduction into SPARQL and then are given such snippets of information which they are supposed to retrieve from the knowledge base. Such a user study could be also used to compare our autocompletion system to approaches that make knowledge bases accessible without exposing the SPARQL language to the user. However, such a user study is also beyond the scope of this thesis.
- We only provide the technical background for an autocompletion-based SPARQL-UI. When designing such a UI in practice, a lot of aspects that are not part of this thesis have to be considered. These include for example helping users with the syntactic aspects of the SPARQL language like the correct usage of keywords. There exists a SPARQL UI at `https://qlever.cs.uni-freiburg.de/`. This UI was not developed by the author of this thesis, but uses the ideas and technical contribution for this thesis to provide autocompletion.

2.4 Contributions

Many of the contents of this thesis are also contained in a scientific paper where the author of this thesis is a co-author and which is currently under review [9]. The contributions described in the technical section (section 5) have solely been developed by the author of this thesis. In that section we will also document in detail, where we relied on already existing software, that was only used in a novel way, and where new features where actually implemented by the author. The autocompletion templates (section 4) were mostly developed by the author. Before the work on this thesis started, a first draft of such templates was already pre-existing but these were rather a general proof of concept and did not run on all knowledge bases and query engines which are considered in this thesis. In addition, the careful optimization of these templates to make them perform as efficiently as possible on all combinations of knowledge bases and query engines is also a contribution of this thesis. The formalization of the problem definition (section 2), the

evaluation setup (metrics and evaluation modes, section 6) and the discussion of the results (later in the same section) have been developed in a team effort among the authors of the paper. The experimental evaluation was performed by the author of this thesis. This included implementing the experimental setup as well as running it. The writeup of this thesis was performed by the author completely independent from the writeup of the paper. In addition, this thesis includes several additions in comparison to the paper contents, which are all contributions by the author of this thesis.

3 Related Work

In this section we will give a brief overview over related work in the field of SPARQL autcompletion in particular and easier access to complex knowledge bases in general.

SPARQL Autocompletion

Campinas et al. [13] describe an autocompletion system that also uses AC queries. This system is only able to create suggestions for predicates and types (i.e. objects of dedicated *type* predicates which have to be manually specified). In contrast, our system is able to create suggestions for all subjects, objects, and predicates. Campinas et al. do not run their AC queries on the complete knowledge base, but on a preprocessed *graph summary*, where subjects and objects that are connected to the same predicates are merged together. This helps the efficiency, since the AC queries become much easier on the smaller graph summary, but harms the relevance of the suggestions.

In a follow-up paper, Campinas [12] introduces *Gosparqled*. This system uses AC queries that run on the complete knowledge base. The suggestions are sensitive but, in contrast to our suggestions, unranked. This greatly helps the efficiency, since it suffices to calculate some suggestions using a LIMIT clause, whereas calculating a ranking requires looking at all possible sensitive suggestions. In general, unranked queries with a LIMIT clause are much easier to process for many query engines (see section 5.1). Campinas evaluates this system on a knowledge base which is less than half the size of the smallest knowledge base we consider in this thesis (Fbeasy). On this small dataset, Campinas compares the suggestion relevance of *Gosparqled* to that of fully ranked sensitive AC queries similar to those introduced in this thesis. He observes, that dropping the ranking of the suggestions significantly reduces the relevance.

Jarrar and Dikaiakos [17] perform autocompletion for a query language called *MashQL*, which is similar to SPARQL. Their suggestions are only context-sensitive if the context is *linearly shaped*. This means that that if the user has already typed *?x <place of birth> ?y . ?y <country> ?z . ?z _* the system will only suggest predicates that are connected to countries in which at least one entity has been born. But after typing a *star-shaped* context like *?x <place of birth> <Berlin> . ?x <gender> <Female> . ?x _* the suggestions will not be context-sensitive. This limitation does not exist in the autocompletion system introduced in this thesis, where the full context is always taken into account, independent of its structure. For efficiency reasons, Jarrar and Dikaiakos also use graph summaries. The structure of these graph summaries is the technical reason for the restriction on linearly-shaped contexts.

Bast et al. [7] present *Broccoli*. This system creates suggestions for *tree-shaped queries* (other

structures of queries are not supported at all). Broccoli uses a graphical user interface that does not expose the underlying query language to the user. The main focus of *Broccoli* is combining the search on structured data from knowledge bases with unstructured data from full-text corpora, and autocompletion is also performed on the full-text data.

Ferré [15] presents *SPARKLIS*. In this system, users formulate queries as natural language, e.g. "Give me every person whose gender is female and who works as a computer scientist". SPARKLIS computes context-sensitive autocompletion suggestions for these queries. The suggestions are created by SPARQL queries that are similar to our AC queries from section 4, but are again unranked (see above, where unranked suggestion have been discussed in connection with Campinas' *Gosparqled* system). Ferré uses the cached results of previous AC queries for the same target query to speed up the computation of the suggestions. We use a similar approach to caching, see section 5.4.

Other Assisted Formulation Systems

There are many approaches towards query formulation that do not rely on autocompletion queries. In this paragraph we will shortly introduce some of these approaches together with a representative system.

Log-Based Approaches

These approaches do not create suggestions based on the actual contents of the knowledge base or database, but on query logs. They thus assume, that queries that have been formulated in the past are representative for future queries. An example is found in Rafes et al. [25]. Their system suggest *basic graph patterns* (parts of SPARQL queries that consist of at least one triple). These suggestions are created using a hierarchical clustering based on basic graph patterns from the query log.

Semantic Parsing

In this approach, the query is formulated in natural language and then automatically transformed into SPARQL or another structured query language. An example is the *Aqqu* system presented by Bast et al. [6], which transforms natural language questions into SPARQL queries. This system is based on machine-learning and requires no ground-truth SPARQL queries but is trained from question-answer pairs.

Example-Based Approaches

A user of the *AutoSPARQL* system presented by Lehmann and Bühmann [22] has to first enter a simple query that yields at least one result of their "actual" target query. The user is then repeatedly asked by the system, whether certain entities should be included in the final query result and the query is automatically refined based on the user's answers. This system only works for tree-shaped queries.

4 Autocompletion (AC) Queries

In this section, we will describe how we can formulate AC queries that can be used to solve the *SPARQL Autocompletion via SPARQL* problem (see section 2). We will introduce *sensitive* AC queries that completely fulfill this problem definition. We will also introduce *agnostic* and *unranked* AC queries that partially sacrifice the relevance and sensitivity of the results for faster query times. We will also introduce *mixed* autocompletion that combines sensitive and agnostic queries, forming a compromise between those two completion modes (In the following we will often use the term *completion mode* for the usage of sensitive, agnostic, unranked or mixed AC queries). All AC queries will be presented in the form of *query templates*. These templates still contain placeholders. To obtain a concrete AC query, these placeholders have to be substituted depending on what has already been typed by the user. At the end of this section, we will discuss how the query templates can be adapted to work with almost arbitrary knowledge bases.

4.1 Calculating the Context

Sensitive AC queries necessarily have to depend on the previously typed partial query (the s from the definition). But it might occur, that not all of s is connected to the currently typed triple. Consider for example the following query:

SELECT ?a ?b WHERE {
  ?a <is-a> <Human> .
  ?b <is-a> <Human> .
  ?a <married-to> ?b .
}

The first two triples are not directly connected to each other, because they don't share a common variable. They only become connected by the third triple which connects $?a$ and $?b$. Now assume that the user has already typed the following in our autocompletion scenario:

?a <is-a> <Human> .
?b <is-a> _

We see, that at this point the first triple is completely irrelevant for the object suggestions of the second triple. It can thus be completely ignored when formulating the AC query. Note that keeping this triple would not change the suggestions, but we have found that doing so severely slows down some of the query engines we evaluated.[7] For this reason we have decided to only include the part of s into our AC queries that is actually connected to the currently typed triple. We will call this connected part of s the **%context%**. In the following we will describe how to calculate it:

Let tr be the currently completed partial triple (e.g. if we are currently completing an object, then tr only contains the subject and predicate immediately before the current token). Set the initial context c_i to be the part of the query body that occurs before tr and thus has already been typed. Let T be all triples and FILTER clauses that are contained in c_i. Note that this

[7] This happens, because large cartesian products are materialized for the unconnected triples

also includes triples that occur inside SPARQL constructs like UNION, MINUS, OPTIONAL, etc. Construct an undirected graph G with the node set $T \cup \{tr\}$. Two nodes in G are connected if and only if they share a variable. We compute the connected component on G that contains tr by performing a breadth-first search. **%context%** is then c_i without all triples that are **not** part of this connected component. We additionally have to deal with the following special case: subqueries are either contained in the context fully or not at all. A subquery is contained in the context when one of its selected variables occurs in the above-mentioned connected component in G. Please note also that this definition of the context and all the following query templates only make sense, if we assume that the current triple tr is embedded directly into the query body and not nested inside an OPTIONAL, MINUS, ... clause. In section 4.5 we will show how we deal with these constructs in a way that enforces this constraint. To illustrate this algorithm, recall our example from the beginning of this paragraph:

?a <is-a> <Human> .
?b <is-a> _

Here the partial triple tr is "*?b <is-a>*", and $T = c_i$ is only the first triple "*?a <is-a> <Human>*". This triple does not share a variable with tr, thus the **%context%** will be empty. Note that the tokens in tr (tokens from the same triple as the current cursor positions) are also part of the relevant context. These are not included in the **%context%** placeholder, but will be included into the AC query manually, see below.

4.2 AC Queries for Subjects

When the target token t is the subject of a triple, it is easy to see, that the **%context%** is always empty, meaning that everything that was previously typed is not relevant for the AC query: Either the subject is a variable, then we do not have to emit an AC query at all, or it is a fixed IRI. In the last case it is also never connected to the previous triples (see section 4.1). We can obtain suggestions for subjects on Wikidata using the following template:

```
1.  SELECT ?entity (SAMPLE(?name) AS ?name)
2.                 (SAMPLE(?score) AS ?score) WHERE {
3.    ?subject rdfs:label|skos:altLabel ?name .
4.    ?subject ^schema:about/wikibase:sitelinks ?score .
5.    FILTER REGEX(?name, "^%prefix%")
6.  } GROUP BY ?entity ORDER BY DESC (?score)
```

Line 3 gets the canonical name and all aliases for all possible subjects. Line 4 retrieves the number of Wikimedia sitelinks[8] of the possible subjects. This is the score used for ranking the suggestions. Note that because of these two lines we will only get suggestions for subjects that have at least one name or label and for which the number of wikimedia sitelinks is specified. This is a design decision which has to be made for every knowledge base. In section 4.5 we will discuss alternatives to this decision and how the retrieval of labels and the ranking of suggestions can

[8]Wikimedia sitelinks for an entity include for example all Wikipedia articles about this entity in different languages.

be realized on other knowledge bases. Line 5 eliminates all subjects that don't have at least one name or alias that starts with **%prefix%** (this is the prefix p from the definition, we will use this more verbose formulation in the following for the sake of readability). We remove this line from the query if the prefix is empty. The *GROUP BY* clause in line 6 in combination with the *SAMPLE* clauses in line 1 and 2 leads to only one result line per remaining *?subject* (in case a possible subject has multiple names starting with **%prefix%**). The result is sorted by the score, meaning that entities with a higher number of sitelinks will be ranked higher in the result.

4.3 AC Queries for Predicates

When the target token is a predicate, we can use the following AC query template to retrieve suggestions (it uses the definition of **%context%** in section 4.1).

```
1.  SELECT ?predicate (SAMPLE(?name) AS ?name)
2.                    (SAMPLE(?score) AS ?score) WHERE {
3.    { SELECT ?entity (COUNT(DISTINCT ?x) AS ?score) WHERE {
4.        %context% %subject% ?predicate []
5.    } GROUP BY ?predicate
6.    ?predicate (^(<> |! <>)/(rdfs:label|skos:altLabel))? ?name .
7.    FILTER REGEX(?name, "%prefix%")
8.  } GROUP BY ?predicate ORDER BY DESC (?score)
```

The subquery from line 3 through 5 calculates all the predicates that sensitively continue the current partial query body which consists of **%context%** and **%subject%**. **%subject%** is the already completed subject of the currently typed triple. This subquery additionally counts, for how many different values of **%subject%** a given predicate forms a sensitive continuation and binds this value to the variable ?*score*. We then retrieve the labels and aliases for these predicates (line 6). This is done via the *rdfs:label* and *skos:altLabel* relation. On Wikidata, these relations are not connected to the predicate directly but via an intermediate node which we find by the property path ^(<> |! <>) (this means "follow any predicate in inverse direction"). We alternatively take the IRI of the possible ?*predicate* itself as a label (the question mark around the whole property path makes it optional). This is useful for several predicates which do not have a label, but are human-readable IRIs themselves, e.g. *rdf:type*. In Line 7 we again filter out all predicates whose name doesn't match the already typed prefix. If the prefix is empty, this line is removed. Note that we do not have a prefix regex here (which would start with ^) but also find matches inside the label. This is again a design decision for Wikidata, because it enables us to find the human-readable prefix IRIs by e.g. typing "type" for *rdf:type*. It is also feasible to compute, since typical knowledge bases and namely Wikidata only contain relatively few predicates (see section 1.3), so the expensive regex filtering will only be performed on few elements. The outer *GROUP BY* clause in connection with the *SAMPLE*s again gives us only one result row per possible predicate in case a predicate has multiple matching labels.

4.4 AC Queries for Objects

When the currently typed token t is an *object*, we can use the following AC query template:

```
1.  SELECT ?object (SAMPLE(?name) AS ?name)
2.                 (SAMPLE(?score_2) AS ?score) WHERE {
3.    { SELECT ?entity (COUNT(?entity) AS ?score_2) WHERE {
4.        %context% %subject% %predicate% ?object
5.    } GROUP BY ?object }
6.  ?object rdfs:label|skos:altLabel ?name .
7.    FILTER REGEX(?name, "^%prefix%")
8.  } GROUP BY ?object ORDER BY DESC (?score)
```

The subquery in lines 3 through 5 computes the objects of all possible triples that match the previously typed partial query body (This includes the **%subject%** and **%predicate%** that immediately precede the current cursor position). We again count the number of such triples for each possible object. Again we attach the labels/aliases and filter by the prefix if this prefix is present. (For details see section 4.2). In case the target token object is completely unconstrained, because **%context%** is empty and **%subject%** as well as **%predicate%** are variables, we do not use this template, but we use the query template for subjects from section 4.2 instead, because it provides a reasonable ranking for unconstrained entities.

4.5 Adapting AC Templates

We aim to formulate the autocompletion via SPARQL as a generic approach, that can be used with any combination of query engine and knowledge base. However some adaptations have to be made when changing the knowledge base. In this subsection we discuss, which design decisions lie behind the above query templates, in which aspects they could be formulated differently, even when using the same knowledge base (Wikidata in this case), and how they can be adapted to different knowledge bases (Freebase and Fbeasy in particular).

Which Entities Are Suggested at All?

When we look at the template for subjects (section 4.2) we notice, that we only get suggestions for entities that have at least one label or alias connected via the *rdfs:label* and *skos:altLabel* relations. This eliminates a lot of entities that occur as subjects in Wikidata. For example, we never get suggestions for the intermediate *statement nodes* (see the discussion on n-ary relations on Wikidata in section 1.4). This is a useful decision because there is no feasible way to get a human-readable description for those, and since they typically do not appear als IRIs directly in queries, but only appear as values of variables in (intermediate) results. In general we suppose that one should only suggest entities for which a human-readable representation can be easily created from the knowledge base. In Freebase this can be done via the *fb.type.object.name* relation. In Fbeasy, the IRIs are human-readable themselves, so names can be retrieved via *BIND(STR(?subject) AS ?name)* (see the examples in section 2.1).

In the template for predicates we consider all predicates that sensitively continue the previously typed partial query. If **%context%** is empty and **%subject%** is a variable, we suggest all predicates from the knowledge base. This approach is totally independent of the used knowledge base. In the template for objects we consider all possible objects that sensitively continue the current triple and can be assigned a readable name (this constraint is similar to the subject

template). Most importantly we never get suggestions for *literals* this way. If suggestions for literals are desired, one can replace lines 6 and 7 in the object template by the following:

6. OPTIONAL { ?object rdfs:label|skos:altLabel ?name } .
7. OPTIONAL { BIND(STR(?object) AS ?name) } .
8. FILTER REGEX(?name, **"^%prefix%"***)*

This would include **all** connected objects. The *?name* of objects that have no label via the *rdfs:label* or *skos:altLabel* relation will be the IRI/literal itself. Note that the decision to include literals in the suggestions might not always be helpful. If the user almost exclusively wants to complete queries that don't contain literals, showing many unnecessary literal suggestions might confuse them. But typically this problem is not so relevant in practice, because most predicates in real life queries are fixed IRIs (no variables) and because in most knowledge bases (including Wikidata) each predicate has either only IRIs or only literals as objects. This means, that the object suggestions will typically either be all IRIs or all literals.

How Do We Rank The Suggestions?

All the AC query templates are ranked. This means that a score is assigned to each possible suggestion and the suggestions are sorted descending by the score function. This is an important element for a high-quality autocompletion system: In section 3 we have seen examples of *sensitive* but unranked autocompletion systems, whose performance is worse than that the performance of sensitive **and** ranked systems like ours (also see the evaluation in section 6).

Our approach to ranking is as follows: For constrained tokens (this means that either the **%context%** is not empty or one of the already typed tokens from the same triple is not a variable) we count for each suggestion the number of triples that sensitively continue the query (see sections 4.3 and 4.4). For unconstrained predicates we choose the same approach, namely ranking them by their frequency in the whole knowledge base. This approach can be used in any knowledge base, since it does not rely on any special characteristics of the knowledge base. In particular, we have also found it to work well for Freebase and Fbeasy. In the following we will call this score the *sensitive score*.

For unconstrained subjects and objects we have chosen the number of Wikimedia sitelinks as a score (see section 4.2). This has proven to be a useful measure for most cases. We have found that this works better than also ranking these suggestions by their frequency: There are entities in Wikidata that appear in a high number of triples but are not very relevant. A similar approach (manually calculating a score for each entity using predicates from the respective knowledge base) can also be applied for other knowledge bases: For the score of subjects and unconstrained objects on Freebase and Fbeasy we have chosen the number of *types* an entity has. As type relation we used *fb:type.object.type* in Freebase and *<is-a>* on Fbeasy.[9] In the following we will call this method of score calculation the *unconstrained score*.

In a future work we could consider more complex score calculations and evaluate their impact on the relevance and efficiency of the AC queries. Such advanced scores could be created e.g. by

[9]On Freebase and Fbeasy this works, because entities typically have multiple types, and more popular entities have a higher number of types. On Wikidata this would not work, because there an entity typically has at most one type. This holds for the "logical" type relation *wdt:P31* ("is-a") as well as for the more technical *rdf:type*.

combining the sensitive and the unconstrained score for objects. We could for example take a weighted sum of both scores, multiply the sensitive score with the (possibly normalized) unconstrained score, or use the unconstrained score of the *subjects* of each possible continuation triple as a weighting factor for the object suggestions.

How Do We Obtain Human-Readable Names For Entities?

During the discussion of the templates for Wikidata (sections 4.2 through 4.4) we have seen, how we can obtain human-readable names for entities using the *rdfs:label* and *skos:altLabel* predicates. On Freebase this can be done similarly using the predicate *fb:type.object.name*. On Fbeasy, the entities are human-readable themselves but have to be converted to literals using a *BIND(STR(...))* construct (see section 2.2 for details).

An additional question that one has to answer when adapting the AC query templates to a new knowledge base is, whether simply adding any label to the suggestion suffices, or whether additional information should also be shown. For example on Wikidata, some aliases are strange or even misleading. For example, the literal "rdf:type" is an alias for the relation *wdt:P31* ("instance of") which is not equivalent to the IRI *rdf:type*. In these cases it is beneficial to also add the canonical label (e.g. via the *rdfs:label* relation) of the suggestions in a separate column. Another problem might arise when two entities have exactly the same name. Consider the two entities *wd:Q388320* and *wd:Q2898115* which both have the label "Avishai Cohen". Both stand for very famous Israeli jazz musicians (the first one is a bassist, the second one a trumpeter), so even context-sensitive autocompletion might not suffice here. To make a human user correctly disambiguate between these entities we could also annotate the suggestions with the result of a description predicate (e.g. *schema:description*). This can be done efficiently, because these additional columns do not influence the suggestion result or the ranking. Thus it suffices to only resolve those columns for the highest-ranked suggestions that are actually sent back to the user.

4.6 Agnostic and Unranked AC Queries

The sensitive AC query templates introduced above yield high quality results (see the evaluation in section 6), but are often hard to compute, mainly because the **%context%** can become arbitrarily large. We therefore also evaluated *agnostic* AC queries which ignore the context and only consider the unconstrained score of the entities. This means that for the same token position (subject, predicate, object) and the same typed prefix we will always get the same suggestions, independent from what was previously typed in the query. Note that the AC query for subjects from section 4.2 is already an agnostic query, because subjects never have context (see there for an explanation). In principle, we don't require a dedicated SPARQL engine for agnostic autocompletion. We could also precompute a list of all entities together with their scores and names, and then just filter this list by the typed prefix and order it by score.[10] However, later in this thesis we also describe improvements for the SPARQL engine QLever, that allow us to answer all agnostic AC queries in less than one second (see section 5).

As a baseline, and to demonstrate that not only the sensitivity but also the ranking are

[10] In a previous work the author of this thesis indeed implemented a data structure supporting such agnostic suggestions as part of a simple SPARQL UI [18].

important for good suggestions, we also evaluate so-called *unranked* AC queries. These are similar to the agnostic queries, but do not sort the final result by the score. This means that all entities that match the typed prefix are suggested in an unspecified[11] order. We will see that this approach performs poorly, even when the set of possible entities is highly constrained by a very long prefix.

4.7 Semi-Sensitive AC Queries

So far we have discussed *sensitive* AC queries that consider the complete context of the target token, but might be expensive to compute, and *agnostic* AC queries which ignore this context, leading to less relevance of the suggestions, but higher efficiency. In this section we will briefly discuss ways to formulate AC queries that lie between these two extremes.

Only Considering Intra-Triple Context

One possibility is to discard the **%context%** placeholder but keep the "intra-triple context" from the placeholders **%subject%** and **%predicate%**. The most common use-case of this method is the following: When completing an object, and when the predicate immediately before this object is a fixed IRI (no variable), we will only get suggestions for objects that occur in this predicate relation. Consider for example the following situation:

?x <is-a> <Human> .
?x <profession> _

The sensitive templates would suggest all professions that are performed by humans. The agnostic templates would suggest all possible objects. Typically, professions will not be high up in this list, because in typical knowledge bases, e.g. countries have a higher unconstrained score than professions. The semi-sensitive approach we just introduced would suggest all professions. Note that in this example this will be almost as good as the fully sensitive completion, because almost all professions are held by humans.[12] But we still might get suggestions that lead to empty results with this method: Imagine we have previously constraint our *?x* from the example to "Humans born before the year 1500", then the semi-sensitive templates would still suggest the profession *computer scientist*. But overall the suggestions should be much better than with the agnostic AC queries, because the short-term context is correct (Typing *profession* and then getting *USA* as a highly-ranked suggestion for the following object is highly irritating, but typically happens with the agnostic templates).

Sampling The Context

A different approach towards semi-sensitive autocompletion is sampling from the context. In the related work section (section 3) we have discussed several approaches that use a LIMIT clause to reduce the number of intermediate results that are used for calculating the suggestions. We have also seen that this significantly reduces the relevance of the returned suggestions. The main problem here is that the LIMIT clause does not enforce a representative sample of the whole result. We hypothesize, that the quality of this approach would greatly benefit from implementing

[11] More exactly, the order depends on implementation details of the query engine.
[12] Depending on the knowledge base, several fictional characters and animals also have professions.

a LIMIT variant that leads to a uniform random sample of the unlimited result. This could e.g. be done by carefully sampling in all intermediate calculations. This approach would however require heavy changes in the internal query processing of the different engines. Thus, the question how efficiently this idea can be implemented and how well the achieved autocompletion results are is beyond the scope of this thesis and is left open for future work.

4.8 Mixed Autocompletion

As a combination between the high-quality, but computationally expensive sensitive AC queries and agnostic AC queries with exactly opposite characteristics, we also consider *mixed* autocompletion. In mixed autocompletion for each step (when a new token starts, or when the user has typed something) we start processing the corresponding sensitive *and* agnostic AC query at the same time. If the sensitive query can be processed within a certain time (in our experiments we choose 1 second) then the sensitive suggestions are shown. Otherwise the agnostic suggestions are shown. In our evaluation we will see that this can improve the quality of the results especially when many sensitive AC queries take too long to be feasible for autocompletion. There is however one downside: Sometimes the user gets sensitive suggestions and thus knows, that they can safely choose one of the suggestions as a meaningful continuation of the previously typed partial query. But sometimes they get agnostic suggestions and can not rely on this property. As a workaround in a concrete UI these different types of suggestions could be shown in different colors to make the user aware of the difference in sensitivity. Another possibility is using *certificate queries* (also called *certificates* in the following) to check the sensitivity of the agnostic suggestions. A certificate is a SPARQL query, that checks if a single (agnostic) suggestion is able to continue the previously typed context in a sensitive way. A template for a certificate for objects could look like this:

*SELECT * WHERE* {
  %context%
  %subject% %predicate% %candidate% .
} *LIMIT 1*

This template will lead to an empty result, if and only if the agnostic suggestion **%candidate%** is not a sensitive continuation of the context. A certificate template for predicates could be derived in a similar way.

4.9 Adapting AC Query Templates to Complex SPARQL Constructs

The AC query templates we have considered this far had one serious limitation: We always assumed that the currently typed triple that contains the target token t is embedded directly into the query body, and not part of a nested SPARQL construct[13] like OPTIONAL, UNION, MINUS or subqueries, and that these triples don't contain property paths (in section 1.2 there is a short introduction of these features). We will now discuss, how our previously defined templates can be used to complete triples inside such constructs. In section 4.1 we have already seen how we deal with these constructs when calculating the **%context%**.

[13]The SPARQL standard calls these constructs (among others) *Graph Patterns*.

OPTIONAL

When completing a triple that is contained inside an OPTIONAL clause, we act as if this triple and all previous triples from the same OPTIONAL clause were outside the OPTIONAL clause. Consider the following partially typed query:

```
1.  SELECT ?actor ?award WHERE {
2.    ?actor <profession> <Actor> .
3.    OPTIONAL { ?actor <award-won> ?award .
4.               ?award _
```

Our token t (line 4) is a predicate whose triple is inside an OPTIONAL clause. The triple in line 3 is inside the same OPTIONAL clause. We then formulate our AC query as if our partial query was actually the following:

```
1.  SELECT ?actor ?award WHERE {
2.    ?actor <profession> <Actor> .
3.    ?actor <award-won> ?award .
4.    ?award _
```

This has the effect that we only get suggestions for predicates that create at least one variable binding. For our example this means that we only are suggested predicates which are actually connected to any award that was one by an actor. The more triples with this predicate exist that fulfill this condition, the higher this predicate will be ranked.

UNION

When completing a token from a triple inside a UNION clause, we act as if this triple and all previous triples from the same branch of this UNION were outside of the UNION. The other branch of the same UNION is completely discarded, even if it stands before our current triple. Consider the following partial query:

```
1.  SELECT ?actor ?award WHERE {
2.    ?actor <profession> <Actor> .
3.    { ?actor <award-won> <Academy_Award> .
4.    } UNION {
5.     ?actor <award-won> Gold_
```

Here the current partial triple in line 5 is inside a UNION and thus moved out of this UNION. The triple in line 3 is in the other branch of the same UNION and is thus ignored for the current token (The current token is an object whose name starts with "Gold"). So we formulate an AC query that actually corresponds to the following partial query:

```
1.  SELECT ?actor ?award WHERE {
2.    ?actor <profession> <Actor> .
3.    ?actor <award-won> Gold_
```

This way we maximize the number of possible results in each branch separately. We also enforce that each branch of the UNION separately leads to a non-empty result.

MINUS

When completing a token that is inside a MINUS clause, the current triple as well as all previous triples inside the same MINUS clause are moved outside the MINUS clause. This has the effect, that these triples would be "added" to the query result instead of "removed". For our autocompletion this means, that we get suggestions, that remove at least one row from the result and rank suggestions higher, the more rows they remove. We therefore suggest the content of the MINUS clause that makes this clause the most "active". Consider the following example:

```
1.  SELECT ?actor WHERE {
2.    ?actor <profession> <Actor> .
3.    MINUS { ?actor _
```

Here we would create an AC query that corresponds to the following situation:

```
1.  SELECT ?actor WHERE {
2.    ?actor <profession> <Actor> .
3.    ?actor _
```

Subqueries

Subqueries are SPARQL queries that are nested inside the body of another SPARQL query. When our current token is inside a subquery, we perform autocompletion on the subquery and ignore the outer query. Note that this technically violates our goal of *sensitive* results: We might get suggestions for subqueries that themselves yield results, but are not compatible to what was previously typed in the outer query. In our experiments we have found this to work sufficiently well and thus stuck with this simplified approach.

Property Paths

In the *predicate* position of each triple the SPARQL standard does not only allow variables or single IRIS (e.g. <*is-a*>) but also so-called *Property Paths*. These consist of possibly nested *sequences* (denoted by "/" and *alternatives* (denoted by "|") of IRIS and variables. Additionally each property path may be annotated by a modifier like "$*$" (0 or more occurences of the same element in sequence) or "+"(1 or more occurences in sequence), similar to regular expressions. A typical property path is <*is-a*> / <*subclass-of*>*, which enforces that the subject and object of the triple are connected by a path of one *is-a* property and an arbitrary number of *subclass* properties.[14] All sequences and alternatives can be almost equivalently reformulated in a more verbose way by using intermediate variables (for sequences) and UNION clauses (for alternatives).[15] For autocompletion

[14]This property path actually does not exist in Fbeasy but its equivalent *wdt:P31/wdt:P279** plays an important role in Wikidata queries. We have translated this example to Fbeasy for simplicity.

[15]The only difference is that using the shorter Property Path syntax eliminates duplicate result rows which the extended version does not. This behavior has however no significant effect on our autocompletion scenario.

we use this reformulation which allows us to use the templates described above. When the current token has a modifier, we ignore this modifier and "manually" add it after the token has been completed. This corresponds to our assumption, that queries are typed from left to right, so the autocompletion system only learns about this modifier after the token has been completed. This approach also enforces, that there exist at least one match, even if the modifier would allow the empty path. This means that we don't allow property paths, that have no effect on the query.

Conclusion

We have shown how our query templates from section 4 can be used to complete queries that contain more complex SPARQL constructs. By this we have implicitly fulfilled an extended definition of the *sensitivity* goal: We only allow continuations that actually affect the query result. Namely we don't allow OPTIONAL clauses that add no bindings, UNION branches that have no results, MINUS clauses that remove no triples and property paths that only bind to the empty path.

5 Technical Contributions

For the evaluation of the AC queries, we have chosen three query engines: *Blazegraph*, *Virtuoso*, and *QLever*. As a contribution of this thesis, we have implemented several improvements for the QLever engine, which make the execution of AC queries faster on this engine. In this section, we will first describe the basic architecture of all three query engines. We will then describe several features of QLever that are especially important for processing AC queries, and how we have improved these features for this thesis. In the following, we will use the term "original version of QLever" to denote a version of QLever without the contributions from this thesis.

5.1 Basic Architecture of QLever, Blazegraph and Virtuoso

QLever

QLever[5] is a SPARQL+Text engine, that is being developed at the chair of Algorithms and Data Structures at the university of Freiburg. It is licensed under the Apache License 2.0 (open source). In addition to standard SPARQL features, QLever also supports loading a set of text snippets that are linked to a knowledge base and supports combined queries on the structured (knowledge base) and unstructured (text) data. In this thesis we will only consider the standard SPARQL features of QLever. QLever assigns a unique integer ID to each token (IRI or literal) of a knowledge base. The set of all tokens that occur in a knowledge base is called the *Vocabulary* of the knowledge base (Sometimes we will also use the term *Vocabulary* for the mapping from this set into the ID space). The order of the IDs corresponds to the lexicographical or numerical order of the corresponding tokens.[16] This allows to perform range filters like *prefix filters* completely in the ID space, without actually looking at the tokens (see section 5.2 for details). The disadvantage of this ordering is, that the complete knowledge base has to be loaded into QLever at once. Therefore, QLever currently does not support SPARQL UPDATE operations. In practice however, this is

[16]IRIs and string literals are sorted by their lexicographical value, numeric literals (including date and time literals) according to their numeric value

not much of an issue, since even the full Wikidata can be loaded in less than 24 hours using a machine like in our evaluation (see section 6.2). QLever stores the complete knowledge base in the ID space in the six permutations PSO, POS, SPO, SOP, OPS, OSP (e.g. the PSO permutation is primarily sorted by the predicate, then by the subject and last by the object). Each of these permutations is stored as one large array on disk. Each query is first mapped to the ID space by looking up the ID for each query token in the Vocabulary. The complete processing of the query is then performed in the ID space.[17] Only the final result is then mapped back to the string space and sent to the user. The time needed for the first mapping can typically be neglected, because most queries contain very few tokens. In the autocompletion scenario this also holds for the final mapping, because we are actually only interested in a small number of suggestions (100 in our experiments), so the number of strings that has to be resolved is limited. An important characteristic of QLever is, that all intermediate results are fully materialized in the ID space. This is an advantage for queries that require sorting/ranking, because these operations cannot be done without looking at the complete result, and because QLever is optimized for this use-case. This is especially true for all of our (ranked) AC queries.

Blazegraph and Virtuoso

In this subsection we will shortly introduce Blazegraph and Virtuoso and their general architecture. Blazegraph [10] is the engine behind the *Wikidata Query Service*[18], Wikimedia's official SPARQL endpoint for Wikidata. Blazegraph is licensed under the GPL 2.0 license (open-source). Blazegraph stores the knowledge base in three permutations (POS, SPO and OSP) in a *B+ tree*.

Virtuoso [28] is a combined SQL and SPARQL database. It has a dual-licensing system with a commercial and an open-source version. The open-source version has slightly limited functionality, e.g. it only supports literals with a limited length. Virtuoso stores all triples in a SQL database with four columns: The subject, object, predicate and the graph name (this last column is not relevant for us, since we only load one graph/knowledge base into one instance of Virtuoso). SPARQL queries are transformed into SQL queries and then handed over to Virtuoso's SQL engine. Virtuoso builds the following indices for the triples table: *PSO, POS, SP, OP*. The SP and OP indices are distinct.

Both Virtuoso and Blazegraph also seem to use some internal mapping from tokens to IDs/integers. We found this information partially by looking at their documentations and sources, and partially by closely monitoring the performance characteristics of these engines. However, these IDs do not seem to reflect the lexicographical order of the tokens. This makes UPDATE operations like the insertion of triples easier, because new IDs can just be created at the end of the ID space[19]. This approach allows the efficient execution of basic operations like joins, which don't require a specific ordering of the ID space. However it does not help with efficient prefix filtering. In contrast to Qlever, Virtuoso and Blazegraph both are able to only partially materialize intermediate result when full materialization is not needed. This means, that for example a query like "Give me ten female mayors of large cities" can be efficiently answered without materializing all female humans

[17] There are only few string options, that require checking of the full string value of a token and thus form an exception to this rule. This holds for example for general regex filters.

[18] `query.wikidata.org`

[19] However, the developers of Virtuoso also recommend reading the whole knowledge base at once, because adding triples requires at least partial rebuilding of the indices, which may take quite some time.

or all cities. As soon as we need a ranked result (e.g. "Ten largest cities of the world", or all of our AC queries), their performance often gets very poor, because the full materialization is needed to compute the ranking (see the evaluation in section 6).

5.2 Efficient Prefix Filtering

In the previous subsection we have seen, that the order of the IDs in QLever is equal to the lexicographical ordering of the corresponding tokens. This directly implies, that given a prefix, the IDs of all tokens in the knowledge base, that start with this prefix, form a consecutive range. We can make use of this fact to compute a *prefix filter*, which in our AC query templates (section 4) was formulated as a SPARQL construct like this:

*FILTER REGEX(?variable, "^%***prefix***%")*

To compute these filters, we use the following operation: As input we are given an intermediate SPARQL result r in ID space, that is a table of IDs, where the columns stand for the already bound variables and one row represents a possible binding. We are also given c, one of the columns/variables and a prefix p (a string). We want to create a new result that contains exactly the rows from r where the ID in column c corresponds to a token that starts with p. To solve this, we first get the IDs f and l of the first and last token in the Vocabulary that starts with p via binary search on the (sorted!) Vocabulary. We then check for each row in r if the ID in column c is in the range $[f, l]$ and only add the row to the result, if it fulfills this condition. This immediately gives us an algorithm that is linear in the number of rows in r ($\mathcal{O}(|r|)$)[20]. If the input r is sorted on column c, this means that the rows, that will end up in our result, also form a consecutive range. We can identify this range easily by finding the first row where column c is $\geq f$ and the last row where $c \leq l$ by performing binary search. Our complexity is then $\mathcal{O}(\log(|r|) + |o|)$ where $|o|$ is the number of rows in the result that have to be written.[21] Especially when the prefix is longer and thus many rows of r are eliminated this is very efficient.

Improvements of the Prefix Filter

The general approach to prefix filtering in QLever was already implemented in the original version of QLever. We still considered a detailed description of this mechanism necessary since it is crucial for understanding, why the AC queries can be executed efficiently by QLever. Additionally, for this thesis the prefix filter was improved in the following ways:

- We have fixed several bugs in the original prefix filter implementation. This included the correct detection, when to use it. We can use the prefix filter to compute a SPARQL regex filter, when the regex starts with a circumflex and uses no other "special" regex characters like ".", "*", etc. If all of these characters are correctly escaped, we can also use the prefix filter. The detection of this case was originally broken. This bugfix is relevant for the AC queries, because on all three knowledge bases there exist labels that include these characters.

[20] Here and in the following we neglect the cost of looking up l and r which is logarithmic in the size of the Vocabulary.

[21] Currently, QLever always copies all intermediate results. It would be possible to implement the prefix filter result as a "view" into its sorted input.

It is crucial that we never accidentally trigger QLever's filter for arbitrary regexes in the AC queries, since this filter has to look at the full strings and is thus very slow.

- We improved the size estimate for the prefix filter operation, which is used by QLever's query planner. Originally the size of the filtered result was severly overestimated. This lead to inefficient query plans, because the prefix filter was executed too late in the query execution. For the autocompletion we found the assumption, that the result of a prefix filter is $10^{-k} \cdot n$ where n is the size (number of rows) of the input and k is the length of the prefix to work sufficiently well. This assumes, that typing a single character eliminates 90% of the intermediate result.

- The lexicographic order of the strings that is used to determine the IDs originally was determined by the ASCII value of the strings. This had the effect, that the prefix filters were *case-sensitive*, because the small letters *a-z* and the capital letters *A-Z* form separate ranges in the ASCII encoding. A similar problem occured with characters that lie outside the ASCII range, but are typically considered variants of characters within the ASCII range. This is true e.g. for German umlauts like "ä" or letters with diacritics like "á". Originally, those were not sorted close to the letter "a". For the prefix filters in the AC queries, this had the consequence, that all diacritics etc. had to be exactly typed. For example, when typing the prefix *"Ang"*, one could get suggestions for *Angelina Jolie* but not for *Ángela Molina*.[22] We fixed this by integrating proper unicode support into QLever using the *ICU (International Components for Unicode)* library[23], the de facto standard for unicode encoding.[24] We then perform the ordering of the strings according to a set of so called *unicode collation rules*. These specify, how strings that use unicode characters are ordered. For example, when comparing two strings, the letter case and all diacritics are first ignored, and only if the strings compare equal, these aspects are used as a tie break. Using this rule, *"an"*, *"An"* and *"án"* would be sorted closely together, and all three would stand before *"ant"*. Additionally we can determine prefix matches that ignore these tie breaks, such that all these four strings match the typed prefix *"an"* in an AC query.[25] Integrating this feature is actually not relevant for our evaluation, since we there assume that the user always knows an exact label which includes proper casing and usage of diacritics etc. For a real life scenario this feature however is crucial, because eliminating results because of a missing uppercase or because of a missing diacritic, would be highly irritating.

[22] In Wikidata this problem sometimes was not relevant, because many entities had a simplified alias with all the diacritics removed from the "actual" label. However we did not want to rely on special properties of a knowledge base.

[23] `http://site.icu-project.org/`

[24] ICU provides the most complete set of features among the competitors in this field, but has some maintainability issues, because it is a relatively old code base, that was originally ported from Java. The C++ community is still struggling to provide a unicode library that is (at least nearly) feature-complete and has an interface that interacts well with modern C++.

[25] This is an a strictly simplified description of the unicode collation. In reality the concrete rules depend on the chosen language (e.G. in swedish, *å* comes after *z*, but in other languages user would expect it to be a variant of *a*. Sometimes we can even chose between multiple sets of collation rules per language. For example, in German dictionaries *ü* is treated as a variant of *u*, while in phone books it is treated like *ue*, making "Müller" and "Mueller" appear next to each other. For details see `https://unicode.org/reports/tr10/`.

5.3 Efficient Predicate Completion Using Patterns

In this subsection we will shortly describe QLever's *pattern trick*, an efficient mechanism to calculate predicate suggestions. We will then show how we have improved this mechanism's efficiency by parallelizing it.

Motivation and Description of the Pattern Trick

In section 4.3 we have seen, that AC queries for predicates have the following form (we have omitted the retrieval of labels since it is not relevant in this section):

```
1.  SELECT ?predicate (COUNT(DISTINCT ?subject) AS ?score) WHERE {
2.    ?subject <is-a> <human> .
3.    ?subject ?predicate []
3.  } GROUP BY ?predicate
```

This is a concrete example, that corresponds to the following situation in our assumed SPARQL editor:

```
?subject <is-a> <human> .
?subject _
```

Note that the trivial approach towards processing the AC query includes materializing all humans from the knowledge base as well as all triples from the knowledge base that have one of the humans as subject. This typically takes a long time. For example on Fbeasy there are 37 Million triples of this form, but there are only 4M distinct matches for *?subject* (humans), and the final result has only 620 rows. To process queries like this, that simply count how often all predicates occur for a given set of entities (in our example: all humans), QLever makes use of the following preprocessing:

1. For each subject s in the knowledge we compute the set of predicates p that have s as a subject, meaning that there exists a triple of the form *s p someObject*. This set of predicates is called the *pattern* of s.
2. We compute the set of distinct patterns: We have found, that typically, many subjects have exactly the same pattern. We store each pattern only once.
3. We assign a unique ID to each pattern, and store the mapping from all subjects to their pattern IDs, and the mapping from pattern IDs to the actual pattern.

To efficiently answer predicate AC queries as stated above, we make use of this preprocessing in the following way:

1. We are given a set S of subjects. For each predicate p in the knowledge base we want to find out, how many $s \in S$ have at least one triple of the form *spo*, where o is an arbitrary object.
2. We first count, how often each pattern occurs for our set S. This can be efficiently done using a hash map $M_{pattern}$ that maps pattern IDs to their occurence count and using our precomputed mapping from subjects to their pattern ID.

3. We then initialize a hash map that maps predicate IDs to their occurence. For each pattern pr in $M_{pattern}$ and for each predicate p that is contained in pr we increase the count of p by $M_{pattern}(pr)$ (this is the number of occurences for the pattern in our input S).

We call this algorithmic scheme (the preprocessing as well as its usage during query processing) the *pattern trick*. Note that the pattern trick is, more generally spoken, a dictionary compression scheme with a very simple heuristic, namely choosing the patterns as dictionary entries. We have found this to work well in practice, since in typical knowledge bases, many entities share the exact same set of predicates (aka pattern). In a future work, we are planning to improve this scheme by using a more advanced heuristic, e.g. by also compressing the predicate set across subjects which have *almost* the same pattern.

Parallel Implementation of the Pattern Trick

For this thesis, we have improved the pattern trick that was just described by providing a parallelized implementation of its query processing part, which we will shortly describe here:

- In step 2 (counting, how often each pattern occurs with the set of input subjects) we split up the input set S into several parts. Each thread can then independently calculate the pattern counts for one of the parts. This results in one hash map per thread. We merge those hash maps by adding the counts for each pattern.
- Step 3 can be parallelized in exactly the same way, by letting each thread calculate the total predicate count for some of the patterns, and afterwards merging the results.

We have implemented the parallelization using the OpenMP instruction set. We have used the `omp taskloop` construct from OpenMP 4.5. This ensures that the parallelization also works efficiently if there are other loads on the CPUs.

We have found that the pattern trick together with this parallelization allows efficient processing of most predicate AC queries. In our evaluation (section 6) we will see, that QLever is significantly better than other query engines, especially with respect to predicate AC queries.

5.4 Caching of Subresults

QLever's caching mechanism is one of the main reasons why this engine is well-suited for executing AC queries. In this section we will describe, how caching works in QLever, how QLever's caching mechanisms help with processing AC queries, and which improvements for the cache we have implemented for this thesis.

5.4.1 Description of QLever's Cache

As described in section 5.1, QLever fully materializes all (sub-)results during query processing. QLever uses a *least-recently-used* (LRU) cache to store these subresults. This caching of subresults is especially beneficial when many similar queries must be processed soon after each other: Similar queries typically have common subresults, which QLever can compute only once and then cache them. Such similarities between queries often occur when processing AC queries: If the user for example starts typing a query with the triple *?x <is-a> <human>* then the list of all humans is

part of the **%context%** for all following AC queries. QLever caches this list and thus only has to calculate this first triple once. Even more important is the following usage of QLever's cache for autocompletion: All AC queries need to add labels to the relevant entities, because the labels have to be shown to the user and QLever possibly has to perform prefix filtering on the labels. Using QLever's cache we can make this process much easier by by loading all entities and their labels into the cache. Then all AC queries can reuse these pre-loaded results that are needed for every AC query.

5.4.2 Comparison to Blazegraph and Virtuoso

Because autocompletion can greatly benefit from caching (see above) we have also tried to find out, to which extent our competitors Blazegraph and Virtuoso have a cache for query results. Unfortunately we could not find any useful information in their documentations.[26] By manually experimenting we found, that there does not even seem to be a cache for final query results in these engines. Sometimes queries get faster when they are executed repeatedly, but the times are still way too long for a simple read from a cache. We hypothesize that these speedups come from caching parts of the knowledge base in RAM instead of caching intermediate or final query results. Both engines seem to rely heavily on such mechanisms. Blazegraph's documentation explicitly recommends leaving a lot of free RAM on the machine for the operating system's disk cache. Virtuoso implements its own mechanism for caching the knowledge base and lets the user configure, how much RAM it may use for this. Note that such caching mechanisms are often useful in practice since typically some parts of a knowledge base are much more often needed than others for typical queries. This is also true for our evaluation dataset. But we have also experienced that QLever's more advanced mechanism is even more suited for autocompletion (see the evaluation and discussion in section 6). The claim that QLever's caching is more advanced is justified by the fact, that Blazegraph's and Virtuoso's behavior can be implemented as a special case of QLever's caching mechanism, namely by caching only the so-called *Index Scans*, where a part of the knowledge base is simply read into RAM for further processing.

5.4.3 Improvements for QLever's caching

In this subsection we describe the improvements to QLever's caching mechanism that were implemented for this thesis to make QLever more suitable for autocompletion.

Pinned Queries

We implemented *pinned* queries. The results of these queries are never evicted from the cache by the LRU mechanism. There is however the functionality to clear the cache completely (including the pinned results) or to just clear the non-pinned results. When pinning a query result, the user either has the option to pin all intermediate results that occur during query processing, or to just pin the final result. For the autocompletion we pin the following building blocks: All relevant subjects/objects together with their labels and aliases and all predicates with their labels and

[26] The only aspect related to caching we could find was, that Virtuoso uses a cache to speed up the query planning. This is not relevant for our work, because the AC queries in our evaluation are relatively cheap to plan, because they do not contain many triples.

aliases (see section 4 for details on the AC queries). Each of those building blocks is stored twice. One of the copies is ordered by the entities (sub-/objects or predicates) and is used when nothing has been typed. Then we can efficiently join[27] the cached building block with all the possible entities from the **%context%**. The other copy is sorted by the label/alias column. It is used for the prefix filtering when the user has already typed a prefix of the next entity. This operation can be performed more efficiently if its input is sorted (see section 5.2 for details).

Making the Query Planner Cache-Aware

We improved QLever's query planner in the following way: The query planner knows which subresults are cached and considers these results with a cost of zero and knows their exact size (Estimating the size of subresults is one of the trickiest aspects in query planning for SPARQL and other query languages). In combination with the pinned queries we can use this for example as follows: When considering a single AC query for an object, the fastest query plan includes only retrieving the labels for the object suggestions that are actually relevant in the concrete AC query. But when we have all possible objects and their labels in the cache, then reusing this result is much faster. We therefore pin this precomputed object-label table (see above) and our query planner implementation makes sure, that the query planner indeed chooses this plan which is only cheaper when considering the cache.

Limiting the Cache's Memory

Originally, QLever's cache was only able to limit the number of subresults that could be held in the cache simultaneously. This was problematic since these subresults can have very different sizes. When there were many small subresults in the cache, the cache would start evicting elements, although the memory consumption of the cache was low. When there were many large subresults in the cache, the memory consumption could be enormous without triggering the eviction mechanism. Because of this, we had to manually clear the cache regularly in our initial experiments with SPARQL autocompletion to not run into memory problems. We have removed this problem by actually measuring and limiting the total size of all elements contained in the cache. This allows us to completely run our evaluation without manipulating the cache in the middle of the experiments.

Improving the Software Architecture

Originally, QLever's cache was implemented as a C++ class that was highly specialized for the LRU structure: It was based on a *linked list* where new elements and elements that were accessed were moved to the front and the elements at the back of the list were the first to be evicted. Additionally the datatype of the cached values was fixed to the datatype QLever uses for subresults. The cache class additionally handled the thread-safe access to the cache and made sure, that subresults that were required by two or more query computations at the same time were computed exactly once. From the perspective of a clean software architecture, this original design had several problems: It violated the *single responsibility principle*, which states that each class or module should have exactly one purpose. Additionally the cache could not be used for different caching strategies (e.g.

[27] Here we refer to the standard database JOIN operation which can be performed more efficiently when the inputs are sorted on the join column, for details see [5].

least *frequently* used) and could not be used for other software projects because it was fixed to QLever's *subresult* datatype. We overcame these issues in the following way:

- To reach single responsibility, we split the original functionality of the cache into three separate classes[28]: The actual cache class, a class that handles the thread-safe access to an arbitrary data structure and a class that makes sure, that each subresult is calculated exactly once. The last class uses the cache to check if a requested subresult currently is cached, but additionally has to keep track of subresults that are currently being computed and therefore do not yet reside in the cache.
- We made the cache class a class template where the cached datatype is a template parameter. This allows reusing the cache for different software projects. It also increases the testability, because tests can be run with simple datatypes like integers.
- To make the cache compatible with other strategies than LRU, we implemented a *priority queue* that supports the *change key* operation. Caching can be implemented using a priority queue in the following way: Each element in the cache has a *key* that decides, when the element is to be evicted from the cache. When the cache is full, we remove element with the lowest key from the cache. This is exactly the *pop* or *delete-min* operation of a priority queue. At some point in time, the keys of existing elements have to be updated. For typical caching strategies the key of a cache element is updated when that element is accessed. This key update can be implemented using the *change key* operation of a priority queue. Implementing a priority queue ourselves was necessary, because the C++ standard library only provides a priority queue that is implemented as a binary heap and therefore does not support the change key operation.

5.4.4 Conclusion

We have described QLever's approach to caching all subresults, and why this approach is more suited for autocompletion than that of Blazegraph and Virtuoso. We have presented several improvements for the cache, some of which immediately were helpful for the autocompletion scenario and some of which improved the software architecture to make QLever ready for future extensions.

5.5 Implementation of a Memory Limit for QLever

Originally, the total amount of RAM which QLever could use was not limited. Thus, when a query required more memory than was currently available on the machine, QLever would frequently be killed by the operating system's Out-Of-Memory-Killer (OOM-Killer).[29] This problem also occured during initial experiments with the autocompletion evaluation, where we had to manually identify the problematic queries and remove them from the evaluation. To avoid that QLever is

[28] Some of these classes were technically *class templates*.

[29] On Linux systems, programs are allowed to allocate more memory than is physically available. Problems only arise once this memory is actually used. Once the amount of memory that is actually used regularly exceeds the actual capacity, the system gets unusably slow, because it starts swapping out memory pages to much slower file systems. To avoid this freezing, Linux starts killing the processes that are responsible for the high memory usage.

killed by the operating system's OOM-killer, we have implemented a global memory limit which has the following functionality and interface:

- Before starting QLever, the user decides how much RAM QLever is allowed to use. This quantity is then passed to QLever as a commandline parameter.
- QLever keeps track of the total amount of memory that was dynamically allocated. By *dynamically allocated* we mean heap allocations as they are performed by C++'s `new` operator.[30] It is sufficient to track dynamic allocations, as the vast majority of QLever's memory consumption lies on the heap. This especially holds for all (sub-)results of queries.
- Before each dynamic allocation, QLever checks whether the sum of the already allocated memory and the newly requested memory is still smaller or equal to the global memory limit. If this is not the case, the allocation fails and throws an out-of-memory exception.
- When such an out-of-memory exception is thrown, the user is informed that their query could not be processed due to the memory limit.
- Optionally (depending on QLever's configuration), when failing an allocation because of the limit, QLever checks first whether the requested memory can be allocated if the cache is (partially) cleared first. In this case, the cache is cleared and then the allocation is performed. This functionality is especially relevant if the cache is allowed to take up a large percentage of the total specified memory limit.

Description of the Implementation

The memory limit was implemented by C++'s allocator interface. All containers of the STL (Standard Template Library, C++'s standard library) can be templated on a specific allocator type. This allocator type specifies how the container handles dynamic memory allocations and deallocations. The default allocator type is `std::allocator`. It internally uses the global `new` and `delete` operators. We have implemented a custom allocator called the *LimitedAllocator*. The *LimitedAllocator* uses a `std::allocator` under the hood to perform the actual (de-)allocations. The *LimitedAllocator* additionally stores the amount of memory m that is currently in use and the memory limit l. When a *LimitedAllocator* is used to allocate n bytes, we first check if there is enough space left, meaning that $m + n \leq l$. If yes, we increase the memory counter by n bytes ($m := m + n$) and use the `std::allocator` to actually allocate n bytes. If there is not enough space left, we throw an out-of-memory exception. Deallocation of n bytes works in the opposite way: We first deallocate the memory and then decrease the memory counter m by n bytes. We have additionally implemented the *LimitedAllocator* in a threadsafe way.

Evaluation and Further Work

We have found that the memory limit completely prevents the QLever engine from being terminated by the OOM-killer. Note, that this only works, if the amount of memory that QLever is allowed to use is physically available on the machine. The user manually has to make sure that this

[30] There are other ways of dynamically allocating memory, e.g. the `malloc` function from C, but those do not play a role in QLever.

property holds. The memory limit implementation only adds a very small time overhead to each (de-)allocation, namely a comparison and addition to the memory counter m. Since the actual allocations themselves are relatively expensive operations, we have found no measurable performance difference after introducing the memory limit. However, with the current implementation there still are some caveats:

- It is still possible to crash QLever when the user allows it to use more memory than is actually available. This typically happens when QLever is allowed to use all the memory that is left on the machine at its starting time. When other memory-intensive software is started on the same machine, the invariant "memory limit for QLever $\leq$ available memory on machine" is violated and QLever might again use too much memory and thus might be killed. This is not a big problem in production, where the QLever instance should be exclusively run on a dedicated machine. But it would still be nice, if QLever additionally respected the total amount of RAM available on a machine as an additional limit. However, to the author's knowledge there exists no possibility to obtain this information in a way, that is portable across different platforms and operating systems.
- Currently, QLever only has a global memory limit. This limit includes all queries that are currently being processed and the contents of the cache. For some scenarios it could also be useful to limit the memory for a specific query. Consider for example a QLever instance which is accessed by many users at the same time. If one of the users inputs a query that requires 98% of the total memory, then the other users are much more likely to trigger an out-of-memory exception, even if their queries require a rather low amount of memory. A per-query limit could prevent this unfair behavior. Such a limit could also be implemented using the allocator-based approach by including multiple memory counters into the *LimitedAllocator*, e.g. one for the global limit and one for the per-query limit.
- The allocator-based implementation of the memory limit requires changes in every place, where dynamic allocations are performed. This bears the risk that some of those places were missed during the implementation. However we took great care to track the places where QLever allocates memory. C++'s allocator interface is helpful with this task, as it suffices to pass the allocator to a container (data structure) when it is created, and the container will then use this allocator for all of its allocations. This problem of manually passing the LimitedAllocators around could be solved by overloading the global operators `new` and `delete`. This would in fact track all allocations. However, this mechanism would not allow per-query limits and also would make it harder to flexibly change QLever's allocation mechanisms in general.

In summary we can say that the global memory limit implementation introduced for this thesis efficiently eliminated one of the most common sources of QLever's crashes especially in load-heavy scenarios like sensitive autocompletion.

5.6 Implementation of a Timeout for QLever

Originally, QLever did not support the cancellation of queries. This means that after a query was sent to Qlever, it would either run to completion or until Qlever was shut down or crashed.

Additionally, the total number of queries that could be processed simultaneously by QLever was limited by a user-defined constant (the total number of operating system threads that QLever's HTTP server was allowed to use). The properties just described frequently lead to the following problems:

- There are queries, which QLever is able to process, but which take a very long time (hours or even days). Typically a user does not want to wait this long and would like to cancel such a query and accept that it has failed. But since there was no way to cancel a query, QLever would keep on processing such a query until it was completed or QLever was shut down or crashed. During this long time, QLever would require resources (CPU time and memory) for the query processing. Additionally, such a query would block one of QLever's server threads for a long time.
- After issuing several of the "infinite" queries just described it could happen, that all of QLever's server threads were used by such queries. This meant, that no new queries could be accepted by QLever until one of the long-running queries had completed or until QLever was manually restarted. This problem could be tackled by allowing QLever to use a higher number of threads. But the problem of the wasted system resources would remain.

The problems just described occured often during initial experiments with the context-sensitive autocompletion, because in this scenario many similar queries are issued within a short time. This section will describe the implementation of a timeout mechanism for QLever. The desired behavior of this mechanism is the following: The user inputs a query and a maximum time T. QLever then starts processing the query. If this processing has not finished after time T, all computations belonging to this query are stopped, the RAM that was used by the query is freed, and a timeout error is reported to the user.

Different Approaches to Canceling Computations

When implementing a computation that can time out, there are two general approaches which will be called **cooperative** and **non-cooperative** in the following. In the **cooperative** approach, the computation itself regularly checks whether it has timed out. If it has timed out, the computation stops itself and returns an error value indicating the timeout. In the **non-cooperative** approach we run a second program or function in parallel to our main computation. This second function is typically called a *watchdog*. If the maximum time for the main computation has elapsed and it has not yet finished, the watchdog aborts the computation and reports the error.

At a first glance the non-cooperative, watchdog-based approach seems to be easier and cleaner since it requires no changes within the "actual" computation. However, this approach requires, that the computation can be safely canceled from outside, which is not always possible. For example it is typically possible to cancel *processes*, e.g. by sending them a `kill` signal on Linux systems. But there is no way to cancel computations that run in a thread that was created using the C++ standard library. On Linux systems one could use the native `pthreads` (Posix Threads) library that supports killing and canceling threads in an uncooperative way. This approach is very likely to cause serious problems, since it does not interact well with possible cleanup operations that the canceled routine has to perform. Herb Sutter, the head of the ISO-C++ standardization

committee writes the following about the non-cooperative canceling of threads: "Every major platform has reinvented this trap because it seems like a simple idea at first, until you realize it's nearly impossible to write correct code whose execution can be abruptly killed at arbitrary and unpredictable points."[31] Sutter's further argumentation can be summarized as follows: It is not possible to safely cancel threads without the the routine running in the thread cooperating to at least some amount. The canceled routine at least has to define some *cancellation points* where it may safely be stopped.

Description of the Implementation

For the reasons that we just discussed it was necessary to choose a cooperative approach for QLever's timeout functionality. We implemented a function `checkTimeout()` that checks whether the current computation has run out of time. If this happens, a timeout exception is thrown which leads to the canceling of the complete query and to an appropriate message to the user. We then had to decide, in which places we should call the `checkTimeout()` function. In general we should call it often enough to make sure that a computation is canceled quickly after it has run out of time. But if `checkTimeout()` is called too often, this might hurt the performance: Although the `checkTimeout()` function is computationally very cheap, its overhead might still be measurable when called too often in a tight loop.

For most operations that can occur when processing a query, it was relatively straightforward to integrate the timeout checks. In loops for example it often suffices to identify a suitable constant k depending on how computationally expensive each iteration is. We then call `checkTimeout()` in every k-th iteration of the loop. Another case that is a little bit more difficult is when QLever uses a possibly long-running call to an external library like C++'s STL. In many of those cases the library call could be split up into multiple shorter calls with timeout checks in between. One example for this scenario are large reads from disk, which can trivially be replaced by multiple smaller reads.

Timeout for Sort Operations

The only relevant operation that does not fall under one of the two cases we just discussed is sorting. Efficient sorting algorithms neither consist of loops with roughly equally expensive iterations nor can they trivially be split up into smaller operations. We first experimented with a custom *cancelable sort* which we implemented by adding timeout checks to an open-source sort implementation. While this approach worked in general, we have decided against integrating it into QLever, because it introduces a high maintainability burden: We would no longer automatically benefit from improvements of external sorting libraries, but would have to adapt our code manually when an update for the used sort implementations is released. For a standard operation like sorting, where there is a lot of specialized research, this is undesirable. For this reason we have decided to solve the timeout for sorting differently: We estimate the time, that the sort will take, and only allow starting it, if this estimate is not much longer than the remaining time for the query. This approach is feasible, because we are only sorting integers in QLever and thus the required time is highly predictable. We have implemented a *SortEstimator*. This module sorts inputs of different

[31] `https://www.drdobbs.com/parallel/interrupt-politely/207100682`

sizes when QLever is started and measures the required time. These measurements are then used to estimate the processing time for other sorting operations.

6 Evaluation

In this section we will first describe the experimental setup. We will then present and discuss our results.

6.1 Tuning AC Queries

Many SPARQL queries can be formulated equivalently in different ways. For example most queries that contain subqueries can be reformulated without using this feature. One reason for choosing between these equivalent reformulations is, that some forms might be more understandable for humans that write and read these queries. But unfortunately many SPARQL engines perform differently depending on which equivalent form we choose for the query. In section 4 we have described AC query templates in the form we consider most readable to humans. However when we use these templates directly with our three query engines Virtuoso, Blazegraph and QLever, they often perform poorly. We therefore have tried to slightly adapt the templates for each engine to get the maximum performance. Some of this changes are equivalent, some of them slightly change the semantics of the query without having too much of an impact concerning the autocompletion scenario. In the following we will describe the most important aspects that played a role in this process.

How to Perform the Prefix Filtering

In our query templates in section 4 we have used the following form of prefix filters:

*FILTER REGEX(?name, "^***%prefix%***")*

QLever automatically uses the efficient prefix filters described in section 5.2 for this construct. For Blazegraph and Virtuoso the performance of this SPARQL construct is very slow. We alternatively tried SPARQL's *STRSTARTS* function which makes it easier for the query planners to recognize, that only a prefix match is needed. Unfortunately this did not help the performance. Additionally, Blazegraph as well as Virtuoso support additionally building a full-text index to perform more efficient keyword search using special predicates (*bdfs:search* on Blazegraph and *bif:contains* on Virtuoso. But even when we removed all spaces and other deliminators from the labels, which makes the keyword search basically a prefix search[32], we could not find a significant speedup in our scenario.

When to Perform the Prefix Filtering

The AC query template for objects (section 4.4) roughly works as follows: First we get all possible objects that sensitively continue the context, as well as their score. In a second step, after having grouped these objects, we retrieve their labels and apply the prefix filter. For Virtuoso and

[32]Both engines require for their fulltext search, that a fixed prefix of the searched keyword is specified. This indicates that their fulltext indices use a data structure that should be optimized for prefix search.

Blazegraph this is indeed the fastest order of operations. In QLever however, when we have already typed some letters, we can use the following order: First retrieve **all** possible objects for which the prefix filter matches, ignoring the context. This can be done efficiently if we cache a table with all possible objects and their labels and use the efficient prefix filter (see section 5). When the prefix is sufficiently long, this already restricts the set of possible objects by a lot. For example, of the 70M entities in Wikidata which have at least one label, only 2.3M entities have a label that starts with "cat". Then we only join this restricted set with the context. This indeed helps the efficiency: In QLever, object AC queries are less likely to time out, if they contain a filter with a longer prefix. For Virtuoso and Blazegraph this reformulation does actually worsen the performance, because it leads to a higher number of expensive prefix filters that have to be computed. Note, that even though the queries yield identical results, none of the engines is capable of moving the joining of the names and the prefix filtering into or out of the grouping operation. Therefore we had to create an additional template for QLever for the case of objects, where the typed prefix is sufficiently long (we have found that 3 characters are a good threshold).

How to Retrieve Labels

In section 4.5 we have already discussed how we can obtain human-readable names for entities on Wikidata, Freebase and Fbeasy. When using the version of Blazegraph that is used by the Wikidata query service[33], there is an alternative way to obtain labels on Wikidata, namely the so-called *label service*. It is activated by adding the following special triple to the query body:

SERVICE wikibase:label bd:serviceParam wikibase:language "en"

If this triple is present, then for each *?variable* that is present in the query the new variable *?variableLabel* will be bound to the canonical label of this entity. For typical queries, which do not filter on the labels, this way of retrieving them is more efficient than joining with the label relation via a "normal" triple. We have experimented with this label service on the combination of Blazegraph+Wikidata but have found no significant changes in the query times. We hypothesize that the expensive prefix filter operation dominates any possible speedup in the retrieval of the labels.

6.2 SPARQL Engines and Hardware

This subsection is identical to the corresponding section in [9]. It was written by the author of this thesis.

We evaluate our own extension of QLever, described in Section 5, against Virtuoso and Blazegraph, the basic architecture of which was already described in Section 5.1.

All experiments were performed on a standard PC with an AMD Ryzen 7 3700X CPU (8 cores + SMT), 128 GB of DDR-4 RAM and 4 TB SSD storage (NVME, Raid 0). We also ran our experiments on HDD storage (Raid 5), and found little difference.[34]

[33] `https://github.com/wikimedia/wikidata-query-rdf/`. The label service described in this section is not included in the "vanilla" version of Blazegraph from their official github repository.

[34] However, indexing on HDD is much slower for Virtuoso and Blazegraph, but that is not the focus of this paper.

QLever was configured with a memory limit of 70 GB for query processing, of which 30 GB were available to the query cache. Before each experiment, the query cache was cleared and the basic building blocks of every AC query were pinned to the cache (see section 5.4 for details).

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.[35] For Blazegraph, we used the latest stable release (2.1.5), configured according to Blazegraph's own recommendations for running Wikidata [10]. 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.

6.3 Autocompletion Queries

As a basis for our evaluation we used the example queries for Wikidata which are provided by the Wikidata query service [30]. These cover a wide range from rather simple to very complex queries and use almost all features of the SPARQL query language. We had to exclude some queries from this dataset because they cannot be processed using the Wikidata knowledge base alone (Some of them use the additional *Lexeme* dataset which was not part of our evaluation, and others query an external SPARQL endpoint using the *SERVICE* keyword). Some other queries had to be removed because they heavily rely on SPARQL features which are not supported by one of the query engines we used. For the same reason, some other queries had to be slightly changed. These changes never made the query easier since we mostly removed *FILTER* clauses with arithmetic expressions, that one of the engines did not support. This left us with 301 target queries for Wikidata. For Freebase we manually translated all of the Wikidata queries for which the content was also part of Freebase. The same approach was taken for Fbeasy. This lead to 115 target queries for Freebase and 99 target queries for Fbeasy.

From these target queries we derived the autocompletion queries which we used in our evaluation according to the following procedure:

1. For each token (subject, predicate, object) in a target query which is not a variable, sample a label from the knowledge base and calculate the **%context%** (see section 4.1).
2. From the label calculate prefixes of length 0, 3 and 7.
3. For each of the three prefixes create an AC query by plugging in the context and the prefix into the appropriate AC query template (see section 4). Run each of these AC queries, measure the execution time and determine, at which position in the result the token appears.
4. If the sampled label is shorter than 3 or 7 characters respectively, we modify the AC queries, such that they enforce a full match with the label instead of a prefix match. This corresponds to the user of an UI indicating (e.g. via entering a special character) that they have reached the end of a label they want to type.

The following details of our experimental setup are also worth noting:

- The queries were transmitted using the HTTP SPARQL endpoint that all engines provide. We measured the total roundtrip time of the HTTP request. Using QLever, which also

[35] 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.

reports the pure query execution time, we found that the overhead of the HTTP connection was less than 50ms.

- We did not issue AC queries for predicates that retrieve the label of an entity (e.g. *rdfs:label* or *fb:type.object.name*). Those appear in almost every query and would have distorted our results. Additionally we assumed, that a well-designed UI would make using these predicates in a query unnecessary, because resolving entities to human-readable names should be performed automatically by the UI instead of manually via a SPARQL triple.
- We evaluate the following completion modes: unranked, agnostic, sensitive, mixed (see section 4). The unranked and agnostic modes are only evaluated on QLever, since they only are a baseline that in principle doesn't even require a SPARQL engine (see section 4.6). If a completion query takes longer than 5 seconds, we count it as failed. In the mixed mode we issue the agnostic and the sensitive query at the same time. If the sensitive query returns a result within 1 second we use this result, else the agnostic result (we are able to consistently calculate agnostic AC queries in 1 second, see below).

6.4 Evaluation Metrics

In this section we describe how we measure our goals of *efficiency* and *relevance*.

Measures for Efficiency

We report for each experiment the percentage of queries that were faster than 0.2s (this is typically considered to be interactive) and 1s respectively (here the user notices the wait, but this is typically still acceptable). We also report the percentage of timeouts after 5s.

Measures for Relevance

For measuring the relevance we assumed the following scenario: Each time a user types something, an AC query is started. The results of these queries are shown on pages of 7 suggestions each. It is less important, on which exact position the target token is, but on which page it appeares. This motivates the following metric:

$\mathbf{MRR_k}$ (Mean Reciprocal Rank)

The *Reciprocal Rank* for an AC query is 1 divided by the page number on which the target token appears. This means that we get 100% for a token on the first page, 50% for a token on the second page, etc. When the token does not appear at all in the AC query's result, the Reciprocal Rank is 0. More formally it is $1/((r \ mod \ k) + 1)$, where r is the position of the target token in the result and k is the page size. The *Mean Reciprocal Rank* ($\mathbf{MRR_k}$) is the average of all the Reciprocal Ranks of one experiment. We report it for $k = 7$ separately for each of the prefix lengths 0, 3 and 7.

$\mathbf{KS_k}$ (Keystrokes)

In this metric we measure the number of *keystrokes* aka the required prefix length until the token is shown on the first page for pages of size k. Note that due to our experimental setup, the KS_k for a single token is either 0, 3, 7 or the length of the typed label plus 1. We use the last case if the token does not appear after 7 characters. This assumes that by specifying the full match with the label (see section 6.3) we can always get the token on the first page. We report the average over the KS_7 for all tokens of an experiment.

Sens. (Sensitivity)

In the *Sens.* column of our evaluation results we report the percentage of AC queries for each experiment for which the results were sensitive and which did not time out. We also count a result as non-sensitive if the user has no way of knowing whether the results are sensitive. For the agnostic and unranked experiments, the sensitivity is then 0% by this definition; for sensitive experiments it is the percentage of AC queries that did not time out. For mixed results it is the percentage of sensitive AC queries that could be answered in under $1s$, so where we did not use the agnostic fallback. Note that the difference between sensitive and mixed evaluations in this column comes from the different timeout (5s for sensitive, 1s for mixed).

6.5 Results

In table 1 on page 46 you can see our main results. This table only reports results for tokens where the context was not empty, because those are the tokens for which sensitive completion can actually make a difference. An interactive webapp to explore the evaluation in more detail, as well as detailed instructions to reproduce the results can be found at `https://ad-publications.cs.uni-freiburg.de/ARXIV_sparql_autocompletion_BKKKS_2021.materials/`. In the following we will discuss some important aspects of these results.

Sensitive Autcompletion Improves the Quality of Suggestions

When comparing the MRR with 0 letters typed between the agnostic completion and the sensitive completion using QLever (which is consistently the best) we get the following values: 25% vs. 67% on Fbeasy, 12% vs. 60% on Freebase and 6% vs. 50% on Wikidata. This means that especially on the larger knowledge bases, the target token is almost never among the first results when using agnostic autocompletion. This is totally expected because the agnostic autocompletion will always suggest the same entities and because there is no prefix which constrains the suggestions. For the sensitive completion the average MRR of at least 50% on all knowledge bases means, that many of the target tokens appear on the first or second page of the result (Recall that a target token on the first result page has a MRR of 100% and 50% on the second page). When typing a prefix of length 3 or 7 the results get much better for agnostic as well as sensitive, the difference between these modes becomes smaller, and sensitive stays ahead. But we consider the prefix length of 0 to be the most important case because the user doesn't need to know a label of the entity which they are looking for. When getting good suggestions without typing anything the user can be "inspired" by the suggestions on how the query should be continued.

Fbeasy (314 tokens)		**$\leq$ 0.2**s	**$\leq$ 1.0**s	**Max**	**Sens**	**MRR$_7$**	**KS$_7$**
Unranked	Qlever	100%	100%	444ms	0%	0: 0% 3: 33% 7: 72%	8.04
Agnostic	Qlever	100%	100%	470ms	0%	0: 25% 3: 86% 7: 96%	3.75
Sensitive	Blazegraph	26%	42%	45% > 5s	55%	0: 38% 3: 53% 7: 53%	5.74
Sensitive	Virtuoso	37%	61%	25% > 5s	75%	0: 58% 3: 65% 7: 65%	3.79
Sensitive	Qlever	91%	97%	1% > 5s	99%	0: 67% 3: 96% 7: 97%	1.77
Mixed	Blazegraph	26%	97%	1234ms	45%	0: 50% 3: 94% 7: 97%	2.52
Mixed	Virtuoso	47%	100%	1000ms	72%	0: 57% 3: 96% 7: 91%	2.07
Mixed	**Qlever**	**90%**	**100%**	**1000ms**	**98%**	**0: 66% 3: 96% 7: 98%**	**1.75**

Freebase (478 tokens)		**$\leq$ 0.2**s	**$\leq$ 1.0**s	**Max**	**Sens**	**MRR$_7$**	**KS$_7$**
Unranked	Qlever	100%	100%	645ms	0%	0: 0% 3: 24% 7: 55%	9.05
Agnostic	Qlever	100%	100%	621ms	0%	0: 12% 3: 83% 7: 94%	4.52
Sensitive	Blazegraph	29%	46%	41% > 5s	59%	0: 39% 3: 57% 7: 58%	5.32
Sensitive	Virtuoso	41%	59%	19% > 5s	81%	0: 43% 3: 79% 7: 80%	3.79
Sensitive	Qlever	87%	97%	0.5% > 5s	100%	0: 60% 3: 97% 7: 99%	2.06
Mixed	Blazegraph	28%	100%	1041ms	49%	0: 42% 3: 93% 7: 98%	2.68
Mixed	Virtuoso	42%	100%	1000ms	62%	0: 43% 3: 97% 7: 99%	2.42
Mixed	**Qlever**	**73%**	**100%**	**1000ms**	**82%**	**0: 59% 3: 96% 7: 99%**	**2.15**

Wikidata (1258 tokens)		**$\leq$ 0.2**s	**$\leq$ 1.0**s	**Max**	**Sens**	**MRR$_7$**	**KS$_7$**
Unranked	Qlever	100%	100%	635ms	0%	0: 0% 3: 8% 7: 53%	10.87
Agnostic	Qlever	100%	100%	696ms	0%	0: 6% 3: 64% 7: 92%	5.88
Sensitive	Blazegraph	3%	27%	59% > 5s	41%	0: 26% 3: 36% 7: 37%	7.93
Sensitive	Virtuoso	35%	53%	24% > 5s	76%	0: 38% 3: 67% 7: 67%	5.26
Sensitive	Qlever	71%	90%	6% > 5s	94%	0: 50% 3: 92% 7: 95%	2.91
Mixed	Blazegraph	0%	98%	1066ms	25%	0: 22% 3: 71% 7: 94%	4.71
Mixed	Virtuoso	35%	100%	1002ms	59%	0: 36% 3: 76% 7: 91%	4.22
Mixed	**Qlever**	**68%**	**100%**	**1000ms**	**88%**	**0: 47% 3: 93% 7: 98%**	**2.76**

Table 1: Evaluation Results. Each row represents one experiment consisting of a knowledge base (Fbeasy, Freebase, Wikidata), a query engine(QLever, Virtuoso, Blazegraph), and a mode (Unranked, Agnostic, Mixed, Sensitive). We report the percentage of AC queries that can be computed in $\leq 0.2s$ and $\leq 1.0s$. The *Max* column shows the percentage of timeouts, or the maximum computation time, of there were no timeouts. The *Sens* column shows the percentage of *sensitive* AC queries in an experiment. The MRR_7 (average mean reciprocal rank) is separately reported for each of the prefix lengths 0, 3 and 7. The KS_7 column reports the average number of keystrokes until a token is on the first suggestion page. For details on the metrics see section 6.4.

QLever is Able to Compute Sensitive AC Queries Efficiently

We now compare the sensitive results between Blazegraph, Virtuoso and QLever. We see, that on all three knowledge bases, QLever is able to deliver the result of at least 90% of the sensitive AC queries in at most 1 second which is still feasible for a system that feels interactive. The closest competitor is Virtuoso which only delivers 50 to 60 percent of the sensitive results within a second, depending on the knowledge base. This would be infeasible for an interactive autocompletion system, since the user would have to wait too long for roughly every second query. Blazegraph performs even worse. Even with a rather generous timeout of 5 seconds we see that Virtuoso and Blazegraph have many timeouts. We have found that, especially on Blazegraph, many of these queries even fail with a much longer timeout of 60 seconds. In summary we can say that our improved version of QLever (including the improvements describe in section 5) is the only one of the systems we compared, on which computing sensitive AC queries is feasible. We could unfortunately not evaluate our improved version of QLever against the original version of QLever, because the original version was lacking the ability to run many of the AC queries due to missing features or crashes under heavy load.

Agnostic is an Interactive and Useful Baseline

The agnostic AC queries can be consistently computed in at most 1 second by QLever and have a decent quality as soon as the user types a prefix of the next entity. This means that, at least when we have some idea of what the next entity in the target query is supposed to be, an agnostic autocompletion system already can be of great help to a user. As already stated in section 4.6 such a system can also be implemented without a complete SPARQL engine and such an approach has already been taken by several SPARQL UIs, for example by the *Wikidata Query Service*[36]. We however argue, that agnostic autocompletion is a good and feasible baseline, but that our sensitive approach performs significantly better. This has already been discussed in the previous paragraph and will be further elaborated below.

Mixed is a Good Compromise

The mixed autocompletion uses the sensitive result if it can be computed in at most 1 second and the agnostic result else. Thus it is feasible as an interactive autocompletion system. For all knowledge bases and engines, the mixed MRR is higher than the agnostic MRR, because the sensitive results almost always have a higher MRR than the agnostic results and all engines are able to compute at least some sensitive AC queries in 1 second. QLever again outperforms Virtuoso and Blazegraph. This is a direct consequence of the better sensitive performance (see above). The raw MRR values even suggest that mixed generally performs better than sensitive in many cases. There is however the following caveat that is connected to our goal of *sensitivity*: With a sensitive result, the user knows, that all the suggestions are useful continuations to the partial query that was already typed. This is not the case for agnostic results. From our own experience we consider this difference in sensitivity to be highly relevant, but it is hard to quantify. Please note, that the *Sens* column in the result does only measure how many sensitive results were used in the experiment. This is only a proxy for the actual sensitivity, because some of the suggestions

[36] `http://query.wikidata.org`

from an agnostic result might also be sensitive according to our definition in section 2. In the mixed mode, the user sometimes sees sensitive suggestions and sometimes agnostic suggestions, which might be confusing. In a practical UI one could e.g. use a different color for the sensitive and agnostic results when the mixed autocompletion is applied.

Ranking is Important, Especially for Agnostic Results

As a baseline we have also integrated unranked AC queries (see section 4.6) which are like agnostic AC queries but without ranking. We observe, that this unranked mode performs much worse than the agnostic mode. This holds even when a prefix of length 3 or 7 is typed and the difference between these modes becomes bigger for larger knowledge bases. This means that the choice of an appropriate ranking function is an essential ingredient of an agnostic autocompletion system. We have not evaluated, which influence the ranking function has on the sensitive autocompletion. But examples for sensitive but unranked SPARQL autocompletion systems have previously been evaluated by other authors were they performed significantly worse than sensitive and ranked systems (see for example [13] and [12] and the discussion of these papers at the beginning of section 3).

7 Conclusion

In this thesis we have shown, how the formulation of SPARQL queries (target queries) on large knowledge bases can be made easier by processing other SPARQL queries (AC queries), which are automatically generated using templates and which compute context-sensitive suggestions for the continuations of the target queries. We have introduced several improvements to the QLever query engine that allow the efficient processing of fully sensitive and ranked AC queries. This is a novelty in comparison to previous work where the suggestions were either not fully sensitive (e.g. because suggestions were computed based on a summary of the knowledge base) or not fully ranked (e.g. because suggestions were based on a non-uniform sampling of query results). We have formally defined the *SPARQL autocompletion via SPARQL* problem and introduced several metrics that allow the empirical analysis of SPARQL-based autocompletion systems. We have conducted a thorough evaluation of different autocompletion modes (fully sensitive, fully agnostic, and combinations of these extremes) on different query engines (Virtuoso, Blazegraph, and QLever) and knowledge bases (Fbeasy, Freebase, and Wikidata). This evaluation has shown, that using our template-based autocompletion approach and our improved version of QLever it is possible to build an autocompletion-based user interface for SPARQL engines that interactively provide the user with high-quality suggestions for entities when formulating a complex query on a large knowledge base.

Further Work

In this final section we will discuss, how the work of this thesis could be used as a basis for further research towards highly efficient and effective SPARQL autocompletion systems.

Implementing and Evaluating Semi-Sensitive Autocompletion

In sections 4.7 and 4.8 we have discussed several approaches towards *semi-sensitive* autocompletion, which we expect to be computationally cheaper than the sensitive autocompletion and to yield better results than agnostic autocompletion. Due to time constraints we did not integrate these ideas into our evaluation. It could be interesting to experiment with such approaches in the future, to see where their potential lies especially for cases, where sensitive AC queries are conceptually too expensive because of a huge intermediate **%context%** (see section 4.1 for details).

Building a Fully Featured UI

This thesis only describes the technical foundation of an autocompletion system. This can be used as a basis for a graphical user interface for SPARQL engines which includes suggestions based on autocompletion. At `http://qlever.cs.uni-freiburg.de` there already exists a prototype of such a UI which was implemented by Daniel Kemen and Julian Bürklin, and which greatly benefits from the interactive autocompletion mechansim described in this thesis. It remains to further improve this UI and to systematically evaluate its quality (see also the next paragraph).

Conducting a User Study For Autocompletion Systems

Our evaluation in section 6 was based on a rather artificial setting: We used a set of ground truth target queries and evaluated, how far up the "correct" token was in the autocompletion suggestions when typing the query from top to bottom. In our experience, this is a good proxy for the actual suggestion quality for a real user, who doesn't know the correct query in advance. To verify this experience, one could conduct a user study to assess how fast users are able to extract a certain piece of information from a knowledge base using different autocompletion mechanisms. This user study could even include completely different approaches to making knowledge bases accessible (e.g. graphical UIs or systems that transform natural language into SPARQL query), and compare these approaches to our autocompletion system.

Improved Ranking Functions

In this thesis we have used rather simple functions to rank the autocompletion suggestions, that were solely based on a single property of the suggested entities, like the number of distinct subjects in the context that have a certain predicate for sensitive predicate suggestions (see section 4). In our evaluation (section 6), we have found these to be rather effective. It could how ever be possible to try out more complex rating functions, which for example take multiple features of the context and the suggestion into account and calculate a weighted average over them. In general, the quality of a given rating function highly depends on the structure of the knowledge base and on the type of queries a "typical" user wants to formulate on this knowledge base. This relation between specific knowledge bases and suitable rating functions for autocompletion on them could be an interesting subject for further research.

More Efficient Pattern Compression

In section 5.3 we have described QLever's pattern compression and our parallel implementation of this mechanism, which plays an important role in autocompletion for predicates. This compression is currently based on a very simple approach, which always takes the set of all predicates of an entity as a pattern. This works well in practice for the knowledge bases we tested, but is far from perfect. Especially it performs poorly if no pair of entities have exactly the same pattern, but if there are great overlaps between the patterns. This compression of predicate patterns could be formalized as a combinatorial problem to evaluate more sophisticated approaches and their implications on the memory usage and speed for the practical application of the pattern compression inside QLever.

Further Improvements For QLever

From its start as a combined SPARQL + Text engine with a big emphasis on the full text features, QLever has come a long way and is to our knowledge currently the fastest existing SPARQL engine for complex queries on large knowledge bases like Wikidata. However, QLever still is missing several features of the SPARQL standard, like UPDATE clauses. Additionally there are some types of queries where other engines like Blazegraph or Virtuoso currently outperform QLever. This mostly includes "simple" queries which don't require ordering the result and which limit the number of desired results. Here QLever, due to the full materialization of all intermediate results, is sometimes slower than other engines which are able to compute results "one by one" (see section 5.1 for details). A next step here would be implementing the full SPARQL standard in QLever and systematically assessing and mitigating QLever's current performance issues.

8 Acknowledgements

I want to thank Prof. Dr. Hannah Bast and Theresa Klumpp for the long and insightful discussions on the theoretical and practical aspects of SPARQL autocompletion. I want to thank Prof. Dr. Fang Wei-Kleiner who agreed to act as a second reviewer for this thesis. I want to thank my dear wife Regina and my parents for their continuous support during my prolonged years of university studies.

References

[1] Katrin Affolter, Kurt Stockinger, and Abraham Bernstein. "A comparative survey of recent natural language interfaces for databases". In: *VLDB J.* 28.5 (2019), pp. 793–819.

[2] Oszkár Ambrus, Knud Möller, and Siegfried Handschuh. "Konduit vqb: a visual query builder for sparql on the social semantic desktop". In: *Workshop on visual interfaces to the social and semantic web*. 2010.

[3] Marcelo Arenas et al. "Faceted search over RDF-based knowledge graphs". In: *J. Web Semant.* 37-38 (2016), pp. 55–74.

[4] Konstantine Arkoudas and Mohamed Yahya. "Semantically Driven Auto-completion". In: *Proceedings of the 28th ACM International Conference on Information and Knowledge Management, CIKM 2019, Beijing, China, November 3-7, 2019*. ACM, 2019, pp. 2693–2701. DOI: 10.1145/3357384.3357811. URL: https://doi.org/10.1145/3357384.3357811.

[5] Hannah Bast and Björn Buchhold. "QLever: A Query Engine for Efficient SPARQL+Text Search". In: *CIKM*. ACM, 2017, pp. 647–656.

[6] Hannah Bast and Elmar Haussmann. "More accurate question answering on freebase". In: *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*. 2015, pp. 1431–1440.

[7] Hannah Bast et al. "Broccoli: Semantic full-text search at your fingertips". In: *arXiv preprint arXiv:1207.2615* (2012).

[8] Hannah Bast et al. "Easy access to the Freebase dataset". In: *WWW (Companion Volume)*. ACM, 2014, pp. 95–98.

[9] Hannah Bast et al. *Efficient SPARQL Autocompletion via SPARQL*. 2021. arXiv: 2104.14595 [cs.DB].

[10] *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.

[11] Kurt D. Bollacker et al. "Freebase: a collaboratively created graph database for structuring human knowledge". In: *SIGMOD Conference*. ACM, 2008, pp. 1247–1250.

[12] Stéphane Campinas. "Live SPARQL Auto-Completion". In: *ISWC Posters & Demos*. Vol. 1272. CEUR Workshop Proceedings. 2014, pp. 477–480. URL: https://pdfs.semanticscholar.org/2628/15d156def72810bad221d1f2db1799f12daf.pdf.

[13] Stéphane Campinas et al. "Introducing RDF Graph Summary with Application to Assisted SPARQL Formulation". In: *DEXA Workshops*. IEEE Computer Society, 2012, pp. 261–266. URL: http://www.renaud.delbru.fr/doc/pub/webs2012-sparqled.pdf.

[14] Ju Fan, Guoliang Li, and Lizhu Zhou. "Interactive SQL query suggestion: Making databases user-friendly". In: *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*. IEEE Computer Society, 2011, pp. 351–362. DOI: 10.1109/ICDE.2011.5767843. URL: https://doi.org/10.1109/ICDE.2011.5767843.

[15] Sébastien Ferré. "Sparklis: An expressive query builder for SPARQL endpoints with guidance in natural language". In: *Semantic Web* 8.3 (2017), pp. 405–418.

[16] Marie Le Guilly, Jean-Marc Petit, and Vasile-Marian Scuturici. "SQL Query Completion for Data Exploration". In: *CoRR* abs/1802.02872 (2018). URL: http://arxiv.org/abs/1802.02872.

[17] Mustafa Jarrar and Marios D. Dikaiakos. "A Query Formulation Language for the Data Web". In: *IEEE Trans. Knowl. Data Eng.* 24.5 (2012), pp. 783–798. URL: http://linc.ucy.ac.cy/publications/pdfs/2012-TKDE-mashQL.pdf.

[18] Johannes Kalmbach. "Efficient and Effective Search on Wikidata Using the QLever Engine". Bachelor's Thesis. University of Freiburg, 2018.

[19] Nodira Khoussainova et al. "SnipSuggest: Context-Aware Autocompletion for SQL". In: *Proc. VLDB Endow.* 4.1 (2010), pp. 22–33. DOI: 10.14778/1880172.1880175. URL: http://www.vldb.org/pvldb/vol4/p22-khoussainova.pdf.

[20] Mikko Koho, Erkki Heino, and Eero Hyvönen. "SPARQL Faceter - Client-side Faceted Search Based on SPARQL". In: *ESWC*. Vol. 1615. CEUR Workshop Proceedings. CEUR-WS.org, 2016.

[21] Kasjen Kramer, Renata Queiroz Dividino, and Gerd Gröner. "SPACE: SPARQL Index for Efficient Autocompletion". In: *ISWC Posters & Demos*. 2013, pp. 157–160. URL: http://ceur-ws.org/Vol-1035/iswc2013_demo_40.pdf.

[22] Jens Lehmann and Lorenz Bühmann. "AutoSPARQL: Let Users Query Your Knowledge Base". In: *The Semantic Web: Research and Applications - 8th Extended Semantic Web Conference, ESWC 2011, Heraklion, Crete, Greece, May 29-June 2, 2011, Proceedings, Part I*. Vol. 6643. Lecture Notes in Computer Science. Springer, 2011, pp. 63–79. DOI: 10.1007/978-3-642-21034-1_5. URL: https://doi.org/10.1007/978-3-642-21034-1%5C_5.

[23] Eyal Oren, Renaud Delbru, and Stefan Decker. "Extending Faceted Navigation for RDF Data". In: *International Semantic Web Conference*. Vol. 4273. Lecture Notes in Computer Science. Springer, 2006, pp. 559–572.

[24] Ana-Maria Popescu, Oren Etzioni, and Henry A. Kautz. "Towards a theory of natural language interfaces to databases". In: *IUI*. ACM, 2003, pp. 149–157.

[25] Karima Rafes et al. "Designing Scientific SPARQL Queries Using Autocompletion by Snippets". In: *eScience*. IEEE Computer Society, 2018, pp. 234–244.

[26] Thibault Sellam and Martin L. Kersten. "Meet Charles, big data query advisor". In: *CIDR 2013, Sixth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*. www.cidrdb.org, 2013. URL: http://cidrdb.org/cidr2013/Papers/CIDR13%5C_Paper94.pdf.

[27] Zhao Sun et al. "Efficient Subgraph Matching on Billion Node Graphs". In: *PVLDB* 5.9 (2012), pp. 788–799. URL: http://vldb.org/pvldb/vol5/p788_zhaosun_vldb2012.pdf.

[28] *OpenLink Virtuoso*. http://docs.openlinksw.com, retrieved 30.01.2021. RDF Index Scheme: http://docs.openlinksw.com/virtuoso/rdfperfrdfscheme.

[29] Denny Vrandecic and Markus Krötzsch. “Wikidata: a free collaborative knowledgebase”. In: *Commun. ACM* 57.10 (2014), pp. 78–85.

[30] *Wikidata Query Service (WDQS).* `https://query.wikidata.org`. Example queries retrieved on 23.10.2020.

DECLARATION

I hereby declare, that I am the sole author and composer of my thesis and that no other sources or learning aids, other than those listed, have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work.
I also hereby declare that my thesis has not been prepared for another examination or assignment, either in its entirety or excerpts thereof.

Rastatt, August 2, 2021
Place, date


Signature