# Fast Approximate Autocompletion for SPARQL Query Builders

Gabriel de la Parra and Aidan Hogan

DCC, Universidad de Chile & IMFD

**Abstract.** A number of interfaces have been proposed in recent years to help users build SPARQL queries, including textual editors with syntax highlighting and error correction, and visual editors that allow for drawing graph patterns using node and edge components. A common feature supported by such systems is autocompletion, which offers users suggestions for terms to insert into a query, potentially restricted by a keyword prefix. However, current systems either return irrelevant terms that will generate empty results, or return relevant terms but may time out while generating suggestions for complex queries. We propose an autocompletion technique based on a graph summary that aims to strike a balance by over-approximating relevant results in an efficient manner.

## 1 Introduction

The recent popularisation of knowledge graphs has introduced new users to the concept of representing and querying data through a graph abstraction [16]. Although a number of languages are now available for querying graphs – such as Cypher [11], Gremlin [23], SPARQL [15], etc. – users may not be familiar with such languages. Knowledge graphs are often used to represent diverse data that do not necessarily follow a schema [16], which makes querying them more difficult, even for users who are expert on the supported query language.

Prominent open knowledge graphs like Wikidata [27] already receive in the order of millions of queries per day from users over the Web [18]. A range of tools and techniques have thus been proposed to assist users to formulate queries over knowledge graphs more easily [24,1,17,9,7,19,2,2,13,22,3,8,18,25,26]. Among such systems, *autocompletion* is a key feature to improve usability. This feature allows users to select a term (often a property or a class) from a list of options, possibly matching a prefix that they have typed. Ideally, autocompletion should be *efficient*, enabling interactive query-building; and should return *relevant* results, avoiding terms that make no sense in the context of the current partial query.

We provide a real-world example of autocompletion in Figure 1 taken from the query editor provided by the Wikidata Query Service [18]. The user has created a variable `?x` and defined it to be an instance of (`wdt:P31`) film (`wd:Q11424`).

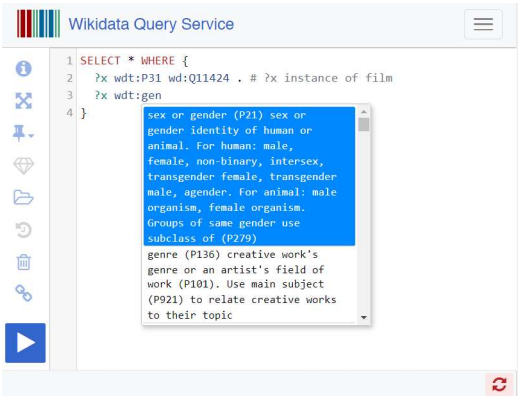



**Fig. 1.** Autocompletion for movie properties on the Wikidata Query Service's query editor (`https://query.wikidata.org`) based on the "gen" prefix

Next they wish to query for the genres of the film, and use the autocompletion feature typing "gen". However, the first suggestion is sex or gender (wdt:P21), which does not make much sense in the context of the current query. The second suggestion, genre (wdt:P136), is the one being sought.[1]

To avoid generating irrelevant suggestions, some query builder systems, such as Gosparqled [6] and RDF Explorer [26], only suggest terms that will generate non-empty results. In the context of the previous example, this is done by evaluating the following intermediate SPARQL query against the endpoint:

```
SELECT DISTINCT ?p WHERE { ?x wdt:P31 wd:Q11424 . ?x ?p ?o . }
```

in order to generate terms to replace ?p that will ensure non-empty results. However, this query is quite expensive to evaluate, as it will typically involve checking all instances of film in the knowledge graph, and extracting the unique set of properties used to describe them. Theoretically speaking, the problem of knowing whether or not a particular term is an answer to such a query (a query with projection and a single Basic Graph Pattern (BGP)) over a knowledge-graph is NP-complete [20]. In practice, such intermediate queries may lead to timeouts, meaning that no suggestions are generated. Even in cases where results are successfully returned, these intermediate queries generate load on the server.

There thus exists a fundamental trade-off between the *efficiency* and *relevance* goals for SPARQL autocompletion. In this context, we find two types of techniques for autocompletion used in practice: those that return suggestions independently of the query context, which are efficient, but may generate irrelevant results (as used by the Wikidata Query Service), and those that only return suggestions leading to non-empty results, which are costly, but only generate relevant results (as used by RDF Explorer). A number of works have further explored techniques that try to strike a better balance between efficiency and rel-

[1] Prefixes used herein can be found at `http://prefix.cc`.

evance [7,6,12,4]. However, these approaches address autocompletion by trying to solve a problem that remains NP-complete in the general case.

For autocompletion, we believe that *efficiency* should take priority, and that *relevance* is a secondary-goal *so long as all relevant suggestions are returned*. In other words, we believe that it is key for the autocompletion technique to return results in a predictable (ideally sub-second) amount of time, no matter what the current query, and that it should return all relevant options, but it may return some irrelevant results as a trade-off (ideally as few as possible). We thus propose a *tractable* technique that filters suggestions according to the current query context, but that may *over-approximate* the set of relevant suggestions.

This paper then discusses our ongoing work on Fast Approximate Autocompletion for SPARQL (FAAS), along with some preliminary experiments on Wikidata. Section 2 describes related works on the topics of query builders and autocompletion for RDF/SPARQL. Section 3 introduces preliminaries and notation for RDF/SPARQL. Section 4 describes the novel technique we propose for autocompletion. Section 5 presents some experiments to evaluate our technique with respect to efficiency and relevance over Wikidata. Section 6 concludes the paper with a discussion of strengths, limitations, and future work.

## 2 Related Work

A range of tools, techniques and interfaces have been proposed in recent years to help users to interact with RDF datasets, including search engines, faceted browsing systems, graph profilers, visualisation tools, question answering services, query-by example systems, query editors, query builders, and more besides [10,5]. The most expressive tools are those that allow for constructing SPARQL queries, namely query editors and query builders. Query editors offer text- and form-based interfaces with aids for writing SPARQL queries (see, e.g., Figure 1); such systems include include Konduit [1], SPARQL Assist [19], Assisted SPARQL Editor [7], QUaTRO2 [2], Gosparqled [6], YASGUI [22], and the Wikidata Query Service [18]. Query builders allow for constructing a query through a visual abstraction, e.g., drawing a graph pattern; such systems include NIGHTLIGHT [24], RDF-GL [17], Smeagol [9], QueryVOWL [14], OptiqueVQS [25], SPARQLing [3], ViziQuer [8], and RDF Explorer [26].

Many such tools rely on generating suggestions through autocompletion mechanisms to assist the user in selecting terms of interest. Such a feature is particularly useful in the context of language-agnostic knowledge graphs, such as Wikidata, where numeric identifiers are used for entities and properties (e.g., using `wd:Q11424` for film). Of these systems, Konduit [1], Smeagol [9], Assisted SPARQL Editor [7], SPARQL Assist [19], QUaTRO2 [2], QueryVOWL [14], YASGUI [22], OptiqueVQS [25], Wikidata Query Service [18], and RDF Explorer [26] support some form of autocompletion. Of these, Smeagol [9], QUaTRO2 [2], Gosparqled [6] and RDF Explorer [26] avoid empty results. Assisted SPARQL Editor [7] provides approximated suggestions.

Various works have addressed autocompletion in the context of RDF/SPARQL. Campinas et al. [7] use a graph summary consisting of three layers: a dataset layer, a node collection layer, and an entity layer. Each layer is based on a quotient graph, at the level of datasets, classes and entities. Autocompletion is then fulfilled by translating the SPARQL query into a higher-level query over the graph summary. Gombos and Kiss [12] present an autocompletion approach that takes into consideration two properties: `rdf:type` to extract the type of entity and the properties compatible with those types, and `rdfs:range` to extract the possible types of the object variables. Rafes et al. [21] propose an autocompletion technique for SPARQL based on patterns seen previously in other users' queries. Bast et al. [4] recently proposed an autocompletion method based on query templates for generating ranked suggestions of entities that may include the context of the current query (without the triple being constructed). Given that these queries can be expensive to compute, the authors propose optimisations based on characteristic sets and caching techniques.

The most similar approaches to that which we propose are the works by Campinas et al. [7], Gombos and Kiss [12], and Bast et al. [4]. The main difference between our approach and that of Campinas et al. [7] and Bast et al. [4] is that their approaches consider evaluating the graph pattern on either a summarised version of the graph, or using particular optimisations; as we later discuss, both techniques are thus based on the NP-complete problem of graph homomorphism, and thus cannot guarantee efficiency in certain cases.[2] Our approach is based on a tractable (over-approximated version of the original) problem, which offers some theoretical guarantees of efficiency. Compared with Gombos and Kiss [12], there are some similarities with our technique, but both approaches are fundamentally different; for example, their approach relies on explicit `rdfs:range` definitions to be provided, and is "unidirectional", not considering domains for outgoing properties. Our approach does not rely on such definitions being provided.

## 3 Preliminaries

Before presenting our technique, we first present some brief, necessary preliminaries for RDF graphs and for SPARQL queries.

An RDF graph is based on three pairwise disjoint sets of RDF terms: IRIs ($\mathbf{I}$), literals ($\mathbf{L}$) and blank nodes ($\mathbf{B}$). An RDF triple $(s, p, o) \in (\mathbf{I} \cup \mathbf{B}) \times \mathbf{I} \times (\mathbf{I} \cup \mathbf{B} \cup \mathbf{L})$ is a three-tuple of RDF terms, where $s$ is called subject, $p$ predicate and $o$ object. An RDF graph $G$ is then a finite set of RDF triples.

At the core of SPARQL queries lies the notion of a triple pattern, defined as $(s, p, o) \in (\mathbf{I} \cup \mathbf{B} \cup \mathbf{L} \cup \mathbf{V}) \times (\mathbf{I} \cup \mathbf{V}) \times (\mathbf{I} \cup \mathbf{B} \cup \mathbf{L} \cup \mathbf{V})$, where $\mathbf{V}$ denotes a set of variables disjoint from the RDF terms. Since blank nodes act as variables and subject literals cannot match any RDF triples, we will assume a simpler definition: $(s, p, o) \in (\mathbf{I} \cup \mathbf{V}) \times (\mathbf{I} \cup \mathbf{V}) \times (\mathbf{I} \cup \mathbf{L} \cup \mathbf{V})$. A basic graph pattern $B$ is then a set of triple patterns; equivalently we can think of a basic graph pattern

[2] In fact, the approach of Bast et al. [4] provides precise suggestions, and thus solves the original problem, which is known to be intractable [20].

as an RDF graph that permits variables in any position. We denote the set of variables appearing in a basic graph pattern $B$ as $\text{vars}(B)$.

The evaluation of a basic graph pattern over an RDF graph yields a set of solution mappings. A solution mapping $\mu$ is a partial mapping $\mathbf{V} \to (\mathbf{I} \cup \mathbf{B} \cup \mathbf{L})$ from variables to RDF terms. We denote by $\text{dom}(\mu)$ the domain of $\mu$, which is the set of variables for which $\mu$ is defined. We denote by $\mu(B)$ the image of $B$ under $\mu$; i.e., the result of replacing every variable $v \in \text{dom}(\mu) \cap \text{vars}(B)$ with $\mu(v)$ in $B$. The evaluation of a basic graph pattern $B$ on an RDF graph $G$ is then defined as $B(G) = \{\mu \mid \mu(B) \subseteq G \text{ and } \text{dom}(\mu) = \text{vars}(B)\}$; in other words, the evaluation returns all of the solution mappings that map the variables of $B$ to the RDF terms of $G$ such that the result is a sub-graph of $G$.

We are interested in generating suggestions for one variable at a time (what Campinas calls the "point of focus" [6]). Thus we introduce (single-variable) projection. Given a solution mapping $\mu$ and a variable $v \in \text{dom}(\mu)$, let $\mu_v$ denote the solution mapping such that $\text{dom}(\mu_v) = v$ and $\mu_v(v) = \mu(v)$, projecting (only) $v$ from $\mu$. Given a set of solution mappings $M$, we define $\pi_v(M) = \{\mu_v \mid \mu \in M\}$.

## 4 Fast Approximate Autocompletion for SPARQL

We now present our technique for fast approximate autocompletion in the context of SPARQL. We first show that autocompletion for SPARQL is a hard problem, which conflicts with the goal of efficiency in the general case, and motivates the need for approximate algorithms. We then introduce a relatively simple approach that over-approximates solutions for the problem, generating all relevant suggestions but possibly also some irrelevant suggestions.

### 4.1 Exact autocompletion is hard

The central problem of this paper is as follows:

| | |
|---|---|
| PROBLEM: | Exact autocompletion |
| INPUT: | BGP $B$, variable $v \in \text{vars}(B)$, RDF graph $G$ |
| OUTPUT: | $\pi_v(B(G))$ |

We begin by briefly showing that this problem cannot be solved efficiently. In particular, we show that the problem – a restricted case of the result presented by Pérez et al. [20] where one variable is projected – remains NP-hard.

**Proposition 1.** *Exact autocomplete is NP-hard.*

*Proof.* We present a polynomial-time reduction from the NP-complete problem of deciding graph homomorphism for directed graphs to the problem of exact autocompletion, which implies that the latter is NP-hard. Given the two directed graphs $D_1 = (V_1, E_1)$ and $D_2 = (V_2, E_2)$, our goal is to decide whether or not there is a homomorphism from $D_1$ to $D_2$ (NP-complete). Define the basic graph pattern $B = \{(\nu(u), \mathsf{p}, \nu(v)) \mid (u, v) \in E_1\}$ where $\nu : V_1 \to \mathbf{V}$ is an injective

mapping from the vertices of $V_1$ to variables and $\mathsf{p} \in \mathbf{I}$ is an arbitrary IRI. Define the RDF graph $G = \{(\iota(u), \mathsf{p}, \iota(v)) \mid (u, v) \in E_2\}$ where $\iota : V_2 \to \mathbf{I}$ is an injective mapping from the vertices of $V_2$ to IRIs. Taking any variable $v \in \text{vars}(B)$, then there exists a homomorphism from $D_1$ to $D_2$ if and only if $\pi_v(B(G))$ is non-empty. This concludes the proof. □

In the interest of efficiency, we thus propose to compute approximate solutions for the autocompletion problem.

### 4.2 Over-approximating autocompletions

In our approach, we propose to ensure that all relevant suggestions be returned, but allow non-relevant suggestions be returned as well in order to trade relevance for efficiency. Specifically, we thus address the following problem:

| | |
|---|---|
| PROBLEM: | Over-approximated autocompletion |
| INPUT: | BGP $B$, variable $v \in \text{vars}(B)$, RDF graph $G$ |
| OUTPUT: | $M$ such that $\pi_v(B(G)) \subseteq M$ |

Of course, there are some trivial solutions for $M$, such as mapping $v$ to every RDF term in $G$. However, our goal will be to try to (efficiently) minimise the set of solutions in $M \setminus \pi_v(B(G))$, i.e., the false positives that lead to empty results. These are the solutions $\mu \in M$ such that $\mu(B)(G) = \emptyset$.

To further reduce and minimise the irrelevant results, we include some practical features, whereby:

1. A user can specify a prefix, such as "gen", that will be matched with keywords associated with suggestions (e.g., though a label or alias property).
2. Ranking is used to prioritise the suggestion of candidates.

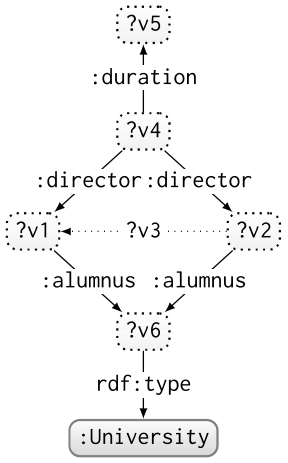
We will now describe the core of the technique we propose, which we call FAAS.

### 4.3 FAAS: core technique

From the RDF graph, we begin by extracting a summary to optimise suggestions generated by autocompletion. We assume here the definition of a type property IRI, which may be `rdf:type` or, in the case of Wikidata, `wdt:P31` (instance of). We will denote this IRI by the symbol $\varepsilon$ for brevity.

**Definition 1 (Autocompletion graph summary).** *Let $G$ be an RDF graph, $p$ be an IRI, $c$ and $x$ be IRIs or blank nodes and $y$ be an IRI, literal or blank node. We define the autocompletion graph summary as a tuple of six mappings:*

$$\begin{aligned}
\text{class}(x) &= \{c \mid (x, \varepsilon, c) \in G\} \\
\text{instance}(c) &= \{x \mid (x, \varepsilon, c) \in G\} \\
\text{incoming}(c) &= \{p \mid \exists x, y : \{(x, p, y), (y, \varepsilon, c)\} \subseteq G\} \\
\text{outgoing}(c) &= \{p \mid \exists x, y : \{(x, p, y), (x, \varepsilon, c)\} \subseteq G\} \\
\text{domain}(p) &= \{c \mid \exists x, y : \{(x, p, y), (x, \varepsilon, c)\} \subseteq G\} \\
\text{range}(p) &= \{c \mid \exists x, y : \{(x, p, y), (y, \varepsilon, c)\} \subseteq G\}
\end{aligned}$$




Autocompletion Graph Summary $\mathfrak{G}$ (sample)

| | |
|---|---|
| domain(:alumnus) | = {:Person} |
| range(:alumnus) | = {:School, :University} |
| domain(:director) | = {:Company, :Film, :Series} |
| range(:director) | = {:Person} |
| domain(:duration) | = {:Event, :Film, :Series} |
| range(:duration) | = {xsd:duration} |
| domain(:type) | = {rdfs:Resource} |
| range(:type) | = {rdfs:Class} |
| ... | = ... |
| incoming(:Person) | = {:director, :parent, :sibling} |
| outgoing(:Person) | = {:alumnus, :parent, :sibling} |
| ... | = ... |
| instance(:University) | = {:Aalborg, ..., :Zurich} |
| ... | = ... |
| class(:University) | = {rdfs:Class} |
| ... | = ... |

**Fig. 2.** Autocompletion example for BGP (left) and graph summary (right)

With respect to the previous definitions, we may note that $c \in \text{class}(x)$ if and only if $x \in \text{instance}(c)$, that $p \in \text{incoming}(c)$ if and only if $c \in \text{domain}(p)$, and that $p \in \text{outgoing}(c)$ if and only if $c \in \text{range}(p)$. Thus the summary contains some redundancy. Defining the mappings in this way, we then index on the argument, e.g., to quickly find instances by class, to find classes by instance, etc. If we wish to support autocompletion for datatype literals, we can consider adding a triple $(l, \varepsilon, d)$ to denote that literal $l$ is of type $d$. Furthermore, to ensure that all relevant results are returned, we assume that all nodes are declared to be an instance of some general class (e.g., `owl:Thing`, `rdfs:Resource`), or that all nodes can be returned if needed (akin to emulating a top-level class).

Given a basic graph pattern $B$, and a variable $v \in \text{vars}(B)$, the next step is to generate autocompleted suggestions for $v$ using the summary of $G$. Let:

$$\text{s}(B) = \{x \mid \exists p, y : (x, p, y) \in B\}$$
$$\text{p}(B) = \{p \mid \exists x, y : (x, p, y) \in B\}$$
$$\text{o}(B) = \{y \mid \exists x, p : (x, p, y) \in B\}$$

denote the subject, predicate and object terms in $B$, respectively.

Our first goal is to generate candidate types for the nodes (subjects and objects) and then candidates for variable predicates in $B$. We begin with the example shown in Figure 2 before defining the algorithm. We will denote by $B$ the basic graph pattern on the left, and by $\mathfrak{G}$ the autocompletion graph summary on the right. We begin by inferring the types of nodes, denoted with types[]. For constant nodes, we take the types from $\mathfrak{G}$.instance($\cdot$). For variables nodes, we intersect the types given in the basic graph pattern (if any), the domains of outgoing properties (if any), and the ranges of incoming properties (if any).

- types[?v1] = $\mathfrak{G}$.domain(:alumus) $\cap$ $\mathfrak{G}$.range(:director) = {:Person}
- types[?v2] = $\mathfrak{G}$.domain(:alumus) $\cap$ $\mathfrak{G}$.range(:director) = {:Person}
- types[?v4] = $\mathfrak{G}$.domain(:director)$\cap$$\mathfrak{G}$.domain(:duration) = {:Film, :Series}
- types[?v5] = $\mathfrak{G}$.range(:duration) = {xsd:duration}
- types[?v6] = {:University} $\cap$ $\mathfrak{G}$.range(:alumus) = {:University}
- types[:University] = $\mathfrak{G}$.class(:University) = {rdfs:Class}

Next we compute candidate properties for predicates in the basic graph pattern, denoted properties[]. In the case that a predicate is an IRI, its only candidate will be itself. Otherwise, it will be the intersection of all the incoming and outgoing properties for the subject and object classes; in case a subject/object has multiple classes as candidates, we take the union of their incoming/outgoing properties before intersecting as the subject/object may be *any* of the classes. This gives us the following candidates for predicates:

- properties[:alumnus] = {:alumnus}
- ...
- properties[?v3] = $\bigcup_{c \in \text{types[?v1]}} \mathfrak{G}.\text{incoming}(c) \cap \bigcup_{c \in \text{types[?v2]}} \mathfrak{G}.\text{outgoing}(c)$
  = {:director, :parent, :sibling} $\cap$ {:alumnus, :parent, :sibling}
  = {:parent, :sibling}.

We detail the process in Algorithm 1, which uses the following convention: we use $\{\star\}$ as syntax to represent the set of all classes/properties in a graph, and define the special intersection $\cap_\star$ such that $A \cap_\star \{\star\} = A$ and $\{\star\} \cap_\star \{\star\} = \{\star\}$. This avoids having to store all classes/properties in the initial set of candidates. The algorithm first establishes a set of candidate classes (denoted as the array types[]) for each node, and then a set of candidate values (denoted as the array properties[]) for each predicate, based on the information available in the basic graph pattern $B$ and the graph summary $\mathfrak{G}$. Note the following:

1. If a node is a constant, or is given multiple specific types in the basic graph pattern, then it would suffice to select one type as the set of candidates is considered a disjunction. However, to avoid non-determinism and simplify the algorithm, we add all known classes to the candidate types for the node.
2. The algorithm is potentially recursive in that we could use the candidate predicates to refine the possible types for nodes, which could then refine the set of predicates, and so on. We opt for a single pass to ensure efficiency.

Once we have these candidates, we can then generate suggestions for autocompletion on a variable $v \in \text{vars}(B)$ based on the candidate types or properties returned by Algorithm 1. If the variable is in a predicate position, $v \in \text{p}(B)$, then we simply return properties[$v$] (if properties[$v$] = $\{\star\}$, then all predicates in $G$ are suggested). Otherwise, we return the union of $\mathfrak{G}$.instance($c$) for all $c \in$ types[$x$] (if types[$v$] = $\{\star\}$, then all nodes in $G$ are suggested).[3]

[3] This implies that if a variable appears in a predicate and subject or object position, we will generate suggestions based on it appearing in the predicate position, as this will generate the most selective results.

We remark that the overall process is tractable: given an input RDF graph $G$ with $n$ triples, then $\mathfrak{G}$ can be computed in time $O(n)$ by creating six hashtables with $O(n)$ keys (assuming ideal hashing; note that the number of unique terms in $G$ is at most $3n$, i.e., $O(n)$). Similarly, let $m$ denote the number of triple patterns in $B$. Then we can use two hashtables with $O(m)$ keys to store the candidates for each term in $B$, where each term has at most $O(n)$ candidates. The candidates can themselves be stored as (nested) hashtables. Thus the intersections of candidate sets for each term takes $O(n)$ at each step, and there are at most three intersections (in the case of classes), giving us a complexity in the order of $O(mn)$ for type inference and autocompletion.[4] The key to tractability here is avoiding a recursive process when inferring classes and properties; otherwise we would be solving the basic graph pattern on a quotient graph, which is still NP-hard following the same reasoning as seen for Proposition 1.

We further remark that the technique returns all relevant results (and possibly more) due to the fact that the graph summary is computed from the data (rather than, e.g., relying on potentially incomplete or inaccurate `rdfs:domain` and `rdfs:range` definitions), that empty suggestions are returned only if the query yields no results, that classes for node variables are only ruled out if they have an incompatible incoming/outgoing property on an incident edge, and that candidate properties for predicate variables are only ruled out if they have an incident node whose possible classes are all incompatible with the domain/range of that property. However, we may additionally return irrelevant results.

### 4.4 Our technique: in practice

The procedure proposed in the previous section has some considerable practical weaknesses. For example, consider the query:

```
SELECT * WHERE { ?s ?p ?o . }
```

Autocompletions for `?s` or `?o` will return all nodes in the graph, while autocompletions for `?p` will return all predicates. In fact, such a query would not be uncommon in query editor and builder interfaces as a starting point for constructing a query. Hence we support two common heuristics used by systems that offer autocompletion: filtering by prefix/keyword, and ranking.

First, we allow users to type a prefix or keyword that will be matched against the text for a node (attached by label, alias or comment properties, as configured by the administrator); e.g., a user seeking autocompletions for `?s` may type "`aa`" and be suggested `:Aalborg`, `:Aardvark`, etc.; similarly if they seek autocompletions for `?p` with prefix "`d`", they will be suggested `:director`, `:duration`, etc. In cases where a fixed set of candidates can be found, prefixes can be combined with the previously discussed criteria to generate relevant suggestions.

Note that the user need not actually type a prefix, or may enter a prefix that still generates thousands of results. Thus, we also provide a ranking of

[4] In practice, one could expect better performance, taking time $O(n)$ to compute the autocompletion graph summary, and time $O(m+n)$ to compute candidates assuming that the number of classes and properties, in particular, is constant.

nodes and properties to prioritise results. For this, we currently apply PageRank on the directed graph induced by the RDF graph to rank node suggestions, and we count the number of triples in which predicates appear to rank property suggestions. Instead of generating all results, we rather paginate the suggestions.

### 4.5 Prototype implementation

We have developed a prototype implementation in C#. The implementation takes as input a set of type properties, and a set of textual properties used for prefix and keyword matching. It then uses custom scripts to extract the autocompletion graph summary (using external sorting rather than hashing, with $O(n \log n)$ performance but fewer random accesses on disk). The mappings are implemented as two inverted indexes in Lucene separated by nodes and properties (some elements may be indexed in both). The inverted indexes further include text indexing on the textual properties, which supports wildcards (e.g., "`aa*`" for prefix search), and ranking boosts based on PageRank and property frequency, respectively. Boosts are multiplied by TF-IDF-based relevance scores in case that a prefix or keyword is provided by the user.

### 4.6 Limitations

There are a number of limitations of our proposed technique and current prototype. First, we focus on autocompletion within a given basic graph pattern. Though we can support other query features such as `UNION`, `GROUP BY`, etc., by focusing on generating autocompletions for inner basic graph patterns, features such as property paths are not directly supportable in this manner. Second, our autocompletion framework depends on an index over a graph summary, meaning that it cannot be directly applied over an endpoint; while specialised indexes help improve performance, such an approach generates an additional cost in terms of downloading the dump and processing it locally, and introduces the question of keeping the graph summary up-to-date with respect to remote changes, which we currently do not consider. Third, our approach may generate irrelevant suggestions that lead to empty results, and may even be slower in *some* cases (in particular, involving selective queries) than computing the exact results for a variable over the remote endpoint; for this reason, we propose to evaluate autocompletions locally and over the endpoint in parallel, and in case the endpoint returns within a fixed amount of time, to use the exact results, otherwise deferring to the approximate result.

## 5 Experiments

We now present the results of some initial experiments for our autocompletion prototype. In particular, we present experiments for autocompletions over Wikidata. The baseline that we currently consider involves computing the exact

results for a variable on the public Wikidata Query Service (WDQS). We consider two main aspects: efficiency and relevance. For efficiency, we will measure response times in comparison to the baseline. For relevance, we will measure the precision of the suggestions generated as the ratio of results returned by our system that are also returned by the endpoint.

Our experiments are based on the May 23rd 2020 truthy dump of Wikidata, which we preprocessed to filter non-English labels, descriptions, etc., leaving a dataset of 1.098 billion triples. Experiments were run on a personal computer with an i7-4600M CPU @ 2.9 GHz and 16 GB of RAM, running on Windows 10. In terms of preprocessing, computing the ranking, extracting the autocompletion graph summary and indexing all of the data took 75 hours. The index size was 9.7 GB, describing 9.7 million nodes and 7,559 properties.

In terms of experiments, to the best of our knowledge there is no existing benchmark for autocompletion. Hence we created a benchmark considering four query templates, as shown in the following:

```
1 SELECT DISTINCT ?p WHERE { ?v1 :p ?v2 . ?v1 ?q ?v3 }
2 SELECT DISTINCT ?p WHERE { ?v1 :p ?v2 . ?v2 ?q ?v3 }
3 SELECT DISTINCT ?p WHERE { ?v1 :p ?v2 . ?v3 ?q ?v1 }
4 SELECT DISTINCT ?p WHERE { ?v1 :p ?v2 . ?v3 ?q ?v2 }
```

Here `:p` represents a constant property that we select based on a weighted sampling of 67 distinct properties in Wikidata, taking a mix of high and low values for the number of triples, number of domain classes, and the number of range classes associated with the property. These queries exhibit common types of joins, as are often encountered in intermediate queries under construction; they are also the types of queries that the endpoint struggles with in practice. Given that the WDQS endpoint frequently times out for node suggestions, where our local index rather paginates ranked results, to enable a fairer comparison we focus in these initial experiments on autocompleting `?q`.

We present the results in Figure 3, comparing the times for FAAS (our proposal) and WDQS, with the $x$-axis representing the percentile, and the $y$-axis representing the runtime, considering $67 \times 4 = 268$ queries. We see that FAAS is in general considerably faster than WDQS. The plateau seen for WDQS refers to queries that time out, where only 36% were processed successfully within the timeout. Given that it is difficult to see the times for FAAS, we present the results alone in Figure 4. We see that all autocompletions were processed within 3 seconds, while 35% of the queries were processed within 2 seconds, and 8% were processed within 1 second. While ideally the runtimes would be a bit lower (i.e., consistently sub-second), we consider that waiting up to 3 seconds is acceptable (particularly considering the alternative of generating no suggestions), and could be lowered with additional optimisations in future.

A key difference between WDQS and FAAS that enables the superior performance of the latter is that FAAS over-approximates suggestions while WDQS computes exact suggestions. For the queries that returned results from WDQS,

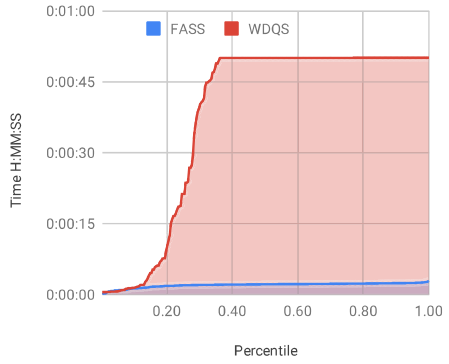



**Fig. 3.** Times: FAAS vs. WDQS

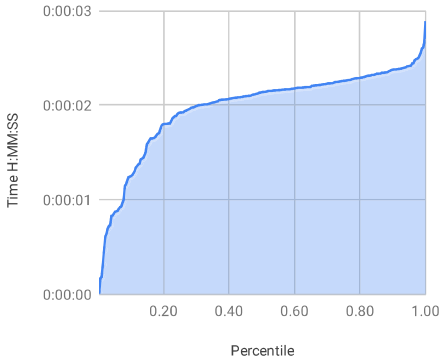



**Fig. 4.** Times: FAAS only

we thus measured the precision, where we found that across all such queries, the minimum precision was 0.04, the median precision was 0.21, and the maximum precision was 0.59. Thus in a typical case, we could expect roughly 1/5 of the suggestions returned by FAAS to return non-empty results when replacing the variable. We also measured the recall, which ranged from 0.92 to 1.00, with a median of 1.00; since we over-approximate results, we expect the recall to always be 1.00, where we confirmed that the cases with imperfect recall were caused by updates to the remote WDQS after the dump we use was published.

## 6 Conclusions

In this paper, we have described our ongoing work on a tractable technique for autocompletion of SPARQL query variables, which can be useful for query editors and query builders. Tractability is enabled by over-approximating the suggestions that lead to non-empty results. We have implemented a prototype based on Lucene, where initial experiments show promising results in terms of efficiency (all autocompletions are computed in less than 3 seconds, even without prefix or keyword search), at the cost of precision (0.21 in the median case, i.e., we return about 5 times more results than are actually relevant).

Key challenges for future work involve improving efficiency further (to get below 1 second, at least) while further increasing precision. Balancing the two appears tricky: for example, precision could be improved by adding a limited depth of recursion to Algorithm 1, but may lead to slower times. Another promising direction might be to look into ranking results by relevance to the query, which would reduce the impact of low precision on usability if the first results that appear will be relevant: though we do consider some ranking mechanisms, we do not consider the context of the query for ranking, which could be explored in future work. For instance, our algorithm will still return `sex or gender` (`wdt:P21`) as a suggestion for the motivating scenario posed in Figure 1: at the time of writing, four movies in Wikidata (presumably due to noise) have this property defined; in future, it could be ranked low based on being defined for few movies.

Another alternative would be to look at other variants of graph summary, for example, to include statistics that weight relations (e.g., to indicate how many instances of a class use a "domain" property, which might be helpful for query-specific ranking, as mentioned before), or to model direct relationships between properties that co-occur on nodes (e.g., to support additional features, such as property paths). Another challenge is keeping the graph summary up-to-date with respect to the knowledge graph, where computing the summary on a dump currently takes several days, and where we missed relevant suggestions due to changes in the live query service since the summary was computed; a possible solution would be to look into incremental updates of the summary.

We further plan to experiment with additional datasets, queries and baselines and have integrated our technique with RDF Explorer [26] for usability testing.

Please see `https://github.com/gabrieldelaparra/SPARQLforHumans` for code and additional material, including the queries used for experiments.

*Acknowledgements* This work was supported by ANID – Millennium Science Initiative Program – Code ICN17_002. Hogan was supported by Fondecyt Grant No. 1181896. We thank the reviewers for their helpful feedback; many of the interesting suggestions for future work were provided by them.

## References

1. O. Ambrus, K. Möller, and S. Handschuh. Konduit VQB: a Visual Query Builder for SPARQL on the Social Semantic Desktop. In *Visual Interfaces to the Social and Semantic Web (VISSW)*. ACM Press, 2010.
2. B. Balis, T. Grabiec, and M. Bubak. Domain-Driven Visual Query Formulation over RDF Data Sets. In *Parallel Processing and Applied Mathematics (PPAM)*, pages 293–301. Springer, 2013.
3. S. D. Bartolomeo, G. Pepe, D. F. Savo, and V. Santarelli. Sparqling: Painlessly Drawing SPARQL Queries over Graphol Ontologies. In *Visualization and Interaction for Ontologies and Linked Data (VOILA)*, pages 64–69, 2018.
4. H. Bast, J. Kalmbach, T. Klumpp, F. Kramer, and N. Schnelle. Efficient SPARQL autocompletion via SPARQL. *CoRR*, abs/2104.14595, 2021.
5. N. Bikakis and T. Sellis. Exploration and visualization in the web of big linked data: A survey of the state of the art. *arXiv preprint arXiv:1601.08059*, 2016.
6. S. Campinas. Live SPARQL Auto-Completion. In *ISWC 2014 Posters & Demos*, volume 1272 of *CEUR*, pages 477–480. CEUR-WS.org, 2014.
7. S. Campinas, T. Perry, D. Ceccarelli, R. Delbru, and G. Tummarello. Introducing RDF Graph Summary with Application to Assisted SPARQL Formulation. In *International Workshop on Database and Expert Systems Applications (DEXA)*, pages 261–266. IEEE Computer Society, 2012.
8. K. Cerans, A. Sostaks, U. Bojars, J. Ovcinnikova, L. Lace, M. Grasmanis, A. Romane, A. Sprogis, and J. Barzdins. ViziQuer: A Web-Based Tool for Visual Diagrammatic Queries Over RDF Data. In *European Semantic Web Conference (ESWC)*, pages 158–163, 2018.
9. A. Clemmer and S. Davies. Smeagol: a "specific-to-general" semantic web query interface paradigm for novices. In *Database and Expert Systems Applications (DEXA)*, pages 288–302. Springer, 2011.

10. A.-S. Dadzie and M. Rowe. Approaches to visualising Linked Data: A survey. *Semantic Web*, 2(2):89–124, 2011.
11. N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, and A. Taylor. Cypher: An Evolving Query Language for Property Graphs. In *SIGMOD International Conference on Management of Data*, pages 1433–1445. ACM, 2018.
12. G. Gombos and A. Kiss. Federated Query Evaluation Supported by SPARQL Recommendation. In S. Yamamoto, editor, *Human Interface and the Management of Information: Information, Design and Interaction*, volume 9734 of *LNCS*, pages 263–274. Springer, 2016.
13. P. Grafkin, M. Mironov, M. Fellmann, B. Lantow, K. Sandkuhl, and A. V. Smirnov. Sparql query builders: Overview and comparison. In *BIR Workshops*, 2016.
14. F. Haag, S. Lohmann, S. Siek, and T. Ertl. QueryVOWL: A Visual Query Notation for Linked Data. In *Extended Semantic Web Conference (ESWC)*, pages 387–402. Springer, 2015.
15. S. Harris, A. Seaborne, and E. Prud'hommeaux. SPARQL 1.1 Query Language. W3C Recommendation, 2013. `https://www.w3.org/TR/sparql11-query/`.
16. A. Hogan et al. Knowledge Graphs. *ACM Computing Surveys (CSUR)*, 54(4):1–37, 2021.
17. F. Hogenboom, V. Milea, F. Frasincar, and U. Kaymak. RDF-GL: A SPARQL-Based Graphical Query Language for RDF. In *Emergent Web Intelligence: Advanced Information Retrieval*, pages 87–116. Springer, 2010.
18. S. Malyshev, M. Krötzsch, L. González, J. Gonsior, and A. Bielefeldt. Getting the Most Out of Wikidata: Semantic Technology Usage in Wikipedia's Knowledge Graph. In *International Semantic Web Conference (ISWC)*, pages 376–394. Springer, 2018.
19. E. L. McCarthy, B. P. Vandervalk, and M. Wilkinson. SPARQL Assist language-neutral query composer. *BMC Bioinformatics*, 13(S-1):S2, 2012.
20. J. Pérez, M. Arenas, and C. Gutiérrez. Semantics and complexity of SPARQL. *ACM Trans. Database Syst.*, 34(3):16:1–16:45, 2009.
21. K. Rafes, S. Abiteboul, S. C. Boulakia, and B. Rance. Designing Scientific SPARQL Queries Using Autocompletion by Snippets. In *IEEE International Conference on e-Science*, pages 234–244. IEEE, 2018.
22. L. Rietveld and R. Hoekstra. The YASGUI family of SPARQL clients. *Semantic Web*, 8(3):373–383, 2017.
23. M. A. Rodriguez. The Gremlin graph traversal machine and language (invited talk). In *Symposium on Database Programming Languages*, pages 1–10. ACM, 2015.
24. P. R. Smart, A. Russell, D. Braines, Y. Kalfoglou, J. Bao, and N. R. Shadbolt. A Visual Approach to Semantic Query Design Using a Web-Based Graphical Query Designer. In *Knowledge Engineering and Knowledge Management (EKAW)*, pages 275–291. Springer, 2008.
25. A. Soylu, E. Kharlamov, D. Zheleznyakov, E. Jiménez-Ruiz, M. Giese, M. G. Skjæveland, D. Hovland, R. Schlatte, S. Brandt, H. Lie, and I. Horrocks. OptiqueVQS: A visual query system over ontologies for industry. *Semantic Web*, 9(5):627–660, 2018.
26. H. Vargas, C. B. Aranda, A. Hogan, and C. López. RDF Explorer: A Visual SPARQL Query Builder. In *International Semantic Web Conference (ISWC)*, pages 647–663. Springer, 2019.
27. D. Vrandečić and M. Krötzsch. Wikidata: a free collaborative knowledgebase. *Communications of the ACM*, 57(10):78–85, 2014.

**Algorithm 1:** Type and property inference

**Input:** Basic graph pattern $B$, graph summary $\mathfrak{G}$
**Output:** Candidate types for $\mathrm{s}(B) \cup \mathrm{o}(B)$, Candidate properties for $\mathrm{p}(B)$
**initialise** types[]; properties[];
**for** $x \in s(B) \cup o(B)$ **do**
  **if** $x \in \mathbf{I} \cup \mathbf{L}$ **then**
    types[$x$] $\leftarrow \mathfrak{G}$.class($x$);
  **else**
    $C_x \leftarrow \{\star\}$;
    **if** $\exists c \in \mathbf{I} : (x, \varepsilon, c) \in B$ **then**
      $C_x \leftarrow \{c' \in \mathbf{I} \mid (x, \varepsilon, c') \in B\}$;
    **end**
    **for** $(x, p, o) \in B : p \in \mathbf{I}$ **do**
      $C_x \leftarrow C_x \cap_\star \mathfrak{G}$.domain($p$);
    **end**
    **for** $(s, p, x) \in B : p \in \mathbf{I}$ **do**
      $C_x \leftarrow C_x \cap_\star \mathfrak{G}$.range($p$);
    **end**
    types[$x$] $\leftarrow C_x$;
  **end**
**end**
**for** $p \in p(B)$ **do**
  **if** $p \in \mathbf{I}$ **then**
    properties[$p$] $\leftarrow \{p\}$;
  **else**
    $C_p \leftarrow \{\star\}$;
    **for** $(s, p, o) \in B : \text{types}[s] \neq \{\star\}$ **do**
      $C_p \leftarrow C_p \cap_\star \bigcup_{c \in \text{types}[s]} \mathfrak{G}$.outgoing($c$);
    **end**
    **for** $(s, p, o) \in B : \text{types}[o] \neq \{\star\}$ **do**
      $C_p \leftarrow C_p \cap_\star \bigcup_{c \in \text{types}[o]} \mathfrak{G}$.incoming($c$);
    **end**
    properties[$p$] $\leftarrow C_p$;
  **end**
**end**
**return** types[], properties[]