Query Extension Suggestions for Visual Query Systems through Ontology Projection and Indexing

Vidar N. Klungre · **Ahmet Soylu** · **Ernesto Jimenez-Ruiz** · **Evgeny Kharlamov** · **Martin Giese**

Abstract Ontology-based visual query formulation is a viable alternative to textual query editors in the Semantic Web domain for extracting data from structured data sources in terms of the skills and knowledge required. A visual query system is at any moment responsible for providing the user with query extension suggestions; however, **suggestions leading to empty results are often not useful**. To this end, in this article, we first present an approach for projecting OWL 2 ontologies into navigation graphs to be used for query formulation and then a solution where an efficient finite index is used to calculate **non-ranked** approximated extension suggestions for ontology-based visual query systems using navigation graphs. The results of our experiments suggest that one can efficiently project an ontology into a navigation graph, query it for running an interactive user interface, and suggest query extensions **that do not lead to dead-ends**.

1 Introduction

Ontology-based visual query formulation is a viable alternative to textual query editors in the Semantic Web domain for extracting data from structured data

Vidar N. Klungre
University of Oslo, E-mail: vidarkl@ifi.uio.no

Ahmet Soylu
NTNU, E-mail: ahmet.soylu@ntnu.no

Ernesto Jimenez-Ruiz
University of Oslo, E-mail: ernestoj@ifi.uio.no

Evgeny Kharlamov
University of Oxford, E-mail: evgenykh@ifi.uio.no

Martin Giese
University of Oslo, E-mail: martingi@ifi.uio.no

sources in terms of the skills and knowledge required given the increasing use of ontology-based data access (OBDA) [13,29] approach in various domains [9,10]. A *visual query system* (VQS) presents a visual interface to users allowing them to extract information from a structured data source, based on some combination of filters and other requirements on the information to be retrieved [5,27]. The intention is to provide data access to users without requiring them to learn a formal query language such as SPARQL. Each VQS needs to find a balance between expressivity and usability, that is a system that covers the whole expressivity of SPARQL will hardly be more useful to lay users than a textual query editor [4]. This trade-off differs depending amongst others on the user group, their information needs, and the complexity of the data [23]. Simple information needs will be met by filtering on some attributes of a single class (e.g. black shoes of size 42), but more advanced use often involves multiple entities of different types (e.g. black shoes from a small company based in a democratic country). Examples of VQSs designed for RDF data are Rhizomer [3], SemFacet [1], and OptiqueVQS [24].

As the user interacts with the VQS, a query is constructed in the background, and a visual representation is usually displayed to the user. The VQS is at any moment responsible for providing the user with query extension suggestions. This can be a list of datatype property filters, or object properties connecting to new concepts. Simple systems may in this case present long, static lists of suggestions containing all the different values appearing in the underlying data source (e.g., [31]). This will ensure that the user finds the suggestion he is looking for somewhere in the list, but it is not optimal, because the list will likely contain suggestions that are incompatible with other parts of the partial query. In other words, selecting such a value will lead to an underlying query which is too restrictive, and hence no results are returned. This kind of dead-end is not desirable from a user experience perspective, and more advanced systems solve this by removing, disabling or down-ranking suggestions (often indicated by a grey font colour) that are not compatible with the existing query – leaving a shorter, more manageable list to the user (e.g., [22]). We call this technique *adaptive extension suggestion* in general, where the goal is to calculate and suggest the complete set of query extensions that are compatible with both the existing query, underlying data and the goal of user, **while we call techniques particularly designed for avoiding queries leading to no results** *dead-end elimination*.

Calculations needed to support adaptive extension suggestion are quite intensive for large datasets. In essence, it requires answers to multiple queries that are all at least as complex as the partial query itself. For queries with many variables, which require joins, this will be too slow. Even with very fast hardware, these queries cannot be executed within tenths of seconds as required for interactive systems. It becomes clear that some kind of index structure is needed to calculate the adaptive extension suggestions sufficiently fast. If the query only contains one variable of a given class, it is possible to achieve desirable performance by using search engines like Lucene[1] or Sphinx[2] or similar software to index the data before use. These indices are known to scale to large datasets, e.g. by partitioning, which ensures fast response time, and no delay for the user. This setup is quite common

---

[1] `https://lucene.apache.org`
[2] `http://sphinxsearch.com`

in e-commerce systems like PriceSpy[3], and a core feature of what is often referred as *faceted search* [30]. These search engines require a fixed number of attributes to index on, which is the case with queries of only one concept. However, we want to support more complex queries with an arbitrary number of connected concepts, where no such static list exists. Ensuring good enough performance for such queries is a challenging task, not something supported by these standard search engines.

In fact, it is impossible to achieve perfect suggestions for arbitrary complex queries and large datasets with good performance. Some kind of index is needed, but it would have to be infinitely large in order to support arbitrary complex queries and large datasets. However, we can support complex queries in an efficient way if the user tolerates some irrelevant suggestions. To this end, in this article, **we focus on dead-end elimination** by first presenting an approach for projecting OWL 2 ontologies into navigation graphs to be used for query formulation and then presenting a solution where an efficient finite index is used to calculate **non-ranked** approximated extension suggestions for ontology-based visual query systems using navigation graphs. The accuracy of these suggestions depends on the size of the index – a larger index gives equal or better accuracy. We take a closer look at this trade-off, and search for concrete approximations that attempts to strike a good compromise between these two. The results of our experiments suggest that one can efficiently project an ontology into a navigation graph, query it for running an interactive user interface, and suggest query extensions **that do not lead to empty results sets. The work presented here provides basis for further refinements, such as fine-grained ranking algorithms and pagination, since the list of possible extensions may still be overwhelmingly long after eliminating the dead-ends from the user-experience point of view**.

The rest of the article is structured as follows. In Section 2, we present the formal framework describing the preliminary knowledge such as on navigation graph and query extensions. In Section 3, we present our contribution on ontology projection and adaptive query extensions, while we present our evaluations in Section 4. Finally, we conclude the article and discuss future work in Section 5.

2 Formal Framework

In the followings, we use a number of simplified notions of schema/ontology, dataset, and query. These are less general than OWL, RDF, and SPARQL, respectively, but they cover the essential notions for VQSs that we require in this article.

2.1 Ontology and Navigation Graph

It is essential for end users to be able to navigate or browse through an ontology $\mathcal{O}$, to get a big picture of what classes are there, and what they have in common in terms of other related classes and properties [8, 28, 15]. This allows users to effectively formulate queries and perform domain exploration tasks.
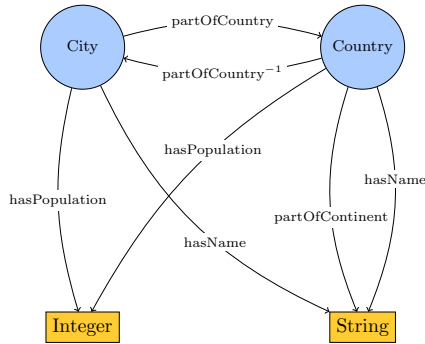
---

[3] `http://pricespy.co.uk`

Fig. 1: Example navigation graph $G_\mathcal{O}$; blue and yellow nodes are concepts and datatypes respectively.
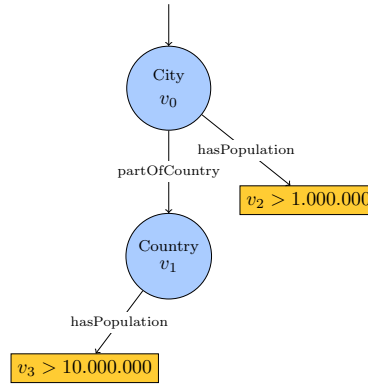
Fig. 2: Example query Q conforming to $G_\mathcal{O}$; blue and yellow nodes are concept and datatype variables respectively.

Based on an underlying ontology, the VQS has to set up rules to control which queries the user is allowed to make. We assume that all these rules are summarised into a *navigation graph* $G_\mathcal{O} = (V, E)$, where each vertex is associated with either a concept or a datatype from $\mathcal{O}$, while the directed edges are associated with property names from $\mathcal{O}$. Furthermore we assume that each edge $e = C_1 \xrightarrow{p} C_2 \in E$ of $G_\mathcal{O}$ has an inverse $e^{-1} = C_2 \xrightarrow{p^{-1}} C_1 \in E$. These inverse edges allow connections between two related concepts regardless of which one is the starting point. In essence $G_\mathcal{O}$ acts like a schema for the whole system, by stating which concepts and/or datatypes we are allowed to connect via which properties. In fact we require that all graph structures in our work conform with $G_\mathcal{O}$, including queries and underlying data.

Figure 1 shows an example of a navigation graph containing two concepts (City, Country), two datatypes (Integer, String), five datatype properties (edges from concepts to datatypes) and two object properties (partOfCountry and its inverse).

2.2 Queries

Based on the navigation graph $G_\mathcal{O} = (V, E)$, we can now define the type of queries we allow. If we represent queries as graphs, where the nodes are query variables, and the edges are the properties connecting them. We only allow tree-shaped conjunctive queries Q, since the literature suggests that majority of end-user queries are in this form [19,27]. We also require that each variable v of Q is typed to either a concept or a datatype in $G_\mathcal{O}$, that is there is exactly one $v \in V$ such that $type(v) \in V$, where $type$ is the typing function. Furthermore $type$ must be a homomorphism from Q to $G_\mathcal{O}$, i.e. for each edge $v_1 \xrightarrow{p} v_2$ of Q, there must exist a corresponding edge $type(v_1) \xrightarrow{p} type(v_2)$ in $G_\mathcal{O}$. For convenience we separate the query variables into two separate groups based on whether they are typed to a concept or a datatype in $G_\mathcal{O}$. We call them *concept variables* and *datatype vari-*

ables respectively. We also allow filters on variables v in Q. This is denoted $v \in \mathcal{F}_v$, where $\mathcal{F}_v$ is the set of data values v can take. By default, there are no filters on any of the variables. We do not include an optional operator, i.e. all variables of Q have to be bound.

During query construction, the user can at any point select which concept variable of the partial query Q_p he wants to extend from. This variable is called the *focus variable* v_f, and the corresponding concept $C_f = type(v_f)$ is called the *focus concept*. During a query session, both Q_p and v_f changes frequently as the user interacts, but at the moment when extension suggestions supposed to be calculated, they can be considered to be fixed. In order to calculate extension suggestions, it is crucial to know which variable is in focus. To support this, we represent the partial query Q_p as a rooted tree where v_f is the root, and where each edge points away from v_f. We can always do this reorientation because the query is tree-shaped and all property inverses exists in $G_{\mathcal{O}}$.

Figure 2 displays the tree representation of the query

$$
\begin{aligned}
City(v_0) \wedge partOfCountry(v_0, v_1) \wedge Country(v_1) \wedge \\
hasPopulation(v_0, v_2) \wedge (v_2 > 1M) \wedge hasPopulation(v_1, v_3) \wedge (v_3 > 10M).
\end{aligned}
\tag{1}
$$

The query conforms to the navigation graph in Figure 1, its focus variable is v_0, and focus concept is City.

2.3 Datasets and Query Answers

In addition to the ontology $\mathcal{O}$ and the corresponding navigation graph $G_{\mathcal{O}}$, we assume that the VQS has access to an underlying dataset (RDF graph) $\mathcal{D}$. This RDF graph should adhere to the OWL2 DL restrictions of keeping instances, classes, object properties, and datatype properties separate, in other words it is a proper description logic ABox. In addition it must conform with $\mathcal{O}$, i.e. $\mathcal{D}$ must be homomorphic to $\mathcal{O}$. When the partial query is complete, the user will be running it over $\mathcal{D}$ in order to retrieve the results of interest. Our goal however, is to utilise the data in $\mathcal{D}$ to compute and present useful query extensions during the query construction phase.

Given a query Q and a data graph $\mathcal{D}$ that are both homomorphic to $G_{\mathcal{O}}$, we let $Ans_{\vec{v}}(Q, \mathcal{D})$ denote the results we get by executing Q over $\mathcal{D}$ and projecting the results onto the vector of variables $\vec{v}$. $Ans_{\vec{v}}(Q, \mathcal{D})$ is a multi-set of tuples, where the entries in each tuple corresponds to an assignment of the variables in $\vec{v}$.

Given two queries Q_1 and Q_2 we can now define query containment:

$$
Q_1 \sqsubseteq Q_2 \iff \forall \mathcal{D}, Ans(Q_1, \mathcal{D}) \subseteq Ans(Q_2, \mathcal{D})
\tag{2}
$$

If $Q_1 \sqsubseteq Q_2$ holds, it means that Q_1 is *more restrictive* than Q_2. We will also use the phrase Q_1 *covers* Q_2 since the tree representing Q_1 fully covers the tree representing Q_2. Furthermore $Q_1 \cap Q_2$ represents the query we get by intersecting the rooted trees represented by Q_1 and Q_2 modulo query variable names.

Table 1 shows an example dataset $\mathcal{D}$ describing four cities, their corresponding countries and related properties. It is represented as a table, and not as a data

City	City-name	City-pop.	Country	Country-pop.	Country-continent
OS	Oslo	0.6M	NOR	5M	Europe
VI	Vienna	1.7M	AUT	8.7M	Europe
RO	Rome	2.9M	ITA	60.6M	Europe
NY	New York	8.5M	USA	323M	North America

Table 1: Example dataset $\mathcal{D}$ describing four cities and their corresponding countries.

graph for convenience. The example is quite simple, since it does not include any one-to-many relationships between cities and countries and no data is missing. This is done by purpose to show how our method works without making the examples too complex.

We can now execute Q from Figure 2 over $\mathcal{D}$ and project over v_0 to get all cities with population higher than 1M and a corresponding country with population higher than 10M:

$$Ans_{(v_0)}(Q, \mathcal{D}) = \{NY, RO\}$$

2.4 Query Extensions

We assume that the VQS supports three possible types of query extensions: *object properties*, *datatype properties* and *datatype filters*. For each of them, the goal is to provide a ranked list of suggestions $\mathcal{S} = (s_1, s_2, \ldots, s_k)$, where each tuple s_i represents a concrete suggestion. If the user selects $s_i \in \mathcal{S}$, the partial query Q_p is updated to $Q_p \wedge Q_{s_i}^{ext}$. Table 2 presents each of the three query extension types, together with the general form of a suggestion s, and the general extension Q_s^{ext}.

Extension type	s	Q_s^{ext}
Type 1. Object property	(p, C)	$p(v_f, v) \wedge C(v)$
Type 2. Datatype property	(p)	$p(v_f, v)$
Type 3. Datatype filter	(p, x)	$p(v_f, v) \wedge (v \in \{x\})$

Table 2: Table showing the three supported query extension types, the general structure of a suggestion s, and the resulting general extension Q_s^{ext}.

All three extension types depend on the property p to connect v_f to a new variable v, hence p is included in each of the three suggestion tuples. If p is an object property (Type 1), then v must be a concept variable of type C. If however p is a datatype property and v a datatype variable (Type 2), then the type of v can be inferred from $G_{\mathcal{O}}$, hence it is not included as a part of the suggestion tuple or updated query. The two first extensions are what we call existential filters: They require a new variable v connected to v_f, but they do not put any additional restrictions on it. The third type of extension on the other hand, adds filters to v, but this can only be done if v is a datatype variable. In theory it would also be

possible to add filters on concept variables, but in real life this is not something users need, because then they would have to know which URIs to filter on. A better solution is then to filter on a data property related to the concept, such as its label or id.

Among the three presented extension types, the third is the hardest one to calculate. In fact, if we can provide adaptive extension suggestions for type 3, we have also done it for type 2, that is a given property p should only be suggested if there are no possible datatype filters left. Extension type 1 and 2 are essentially the same, so they are equally hard to make adaptive suggestions for.

3 Adaptive Extension Suggestions

In this section, we first present our approach for projecting a given ontology into a navigation graph and then present our solution for adaptive extension suggestions **not leading to any empty results**.

3.1 Ontology Projection

Our goal for ontology projection is, given an ontology, to create a directed labelled graph, called a navigation graph [1, 26], whose nodes correspond to the named classes and datatypes in the ontology and edges between nodes to the object properties and datatype properties. Let C_1, C_2, and C_3 be classes, r_1, r_2, and r_3 object properties, d_1 a datatype property, i_1 and i_2 individuals, and dt_1 a data type. First, each class and datatype in the ontology is translated to a node in the navigation graph $G_{\mathcal{O}}$. Then we add edges of the form $C_1 \xrightarrow{r_1} C_2$ and $C_1 \xrightarrow{d_1} dt_1$ into the navigation graph derived from the axioms of the ontology. The types of axioms resulting in an edge are presented with examples in the followings using description logic (DL) [2].

Ontologies have a propagative effect on the amount of information to be presented. This case is considered in two forms, namely the top-down and bottom-up propagation of property restrictions [6, 23]. The first form emerges from the fact that, in an ontology, explicit restrictions attached to a class are inherited by its subclasses. The second form is rooted from the fact that the interpretation of an OWL class also includes the interpretations of all its subclasses. Therefore, for a given class, it may also make sense to derive edges from the (potential) object and datatype properties of its subclasses and superclasses.

3.1.1 Edges Through Object Properties

Domains and Ranges: Domain and range axioms using named classes are translated to an edge. For instance, example given in Axiom 3 maps to edge $C_1 \xrightarrow{r_1} C_2$.

$$\exists r_1.\top \sqsubseteq C_1 \text{ and } \top \sqsubseteq \forall r_1.C_2 \tag{3}$$

$$\exists r_1.\top \sqsubseteq C_1 \text{ and } \top \sqsubseteq \forall r_1.(C_2 \sqcup C_3) \tag{4}$$

If a complex class expression, formed through intersection ($\sqcap$) or union ($\sqcup$), appears as a domain and/or range, then an edge is created for each pair of domain and range classes. For instance, example given in Axiom 4 maps to edges $C_1 \xrightarrow{r_1} C_2$ and $C_1 \xrightarrow{r_1} C_3$.

Object Property Restrictions: Object property restrictions used in class descriptions, formed through existential quantification ($\exists$), universal quantification ($\forall$), individual value restriction, max ($\geq$), min ($\leq$), and exactly ($=$), are mapped to edges. For instance, examples given in Axiom 5 to 7 map to $C_1 \xrightarrow{r_1} C_2$. Note that in Axiom 7, there is a complex class expression on the left-hand-side.

$$C_1 \sqsubseteq \exists r_1.C_2 \tag{5}$$

$$C_1 \equiv \leq_n r_1.C_2 \tag{6}$$

$$\forall r_1.C_1 \sqsubseteq C_2 \tag{7}$$

Example given in Axiom 8 includes an individual value restriction and an edge is created with the type of individual, that is $C_1 \xrightarrow{r_1} C_2$.

$$C_1 \sqsubseteq \exists r_1.\{i_1\}, \text{ and } i_1 : C_2 \tag{8}$$

Example given in Axiom 9 includes a complex class expression. In this case, an edge is created for each named class, that is $C_1 \xrightarrow{r_1} C_2$ and $C_1 \xrightarrow{r_1} C_3$.

$$C_1 \sqsubseteq \exists r_1.(C_2 \sqcup C_3) \tag{9}$$

Given an enumeration of individuals, an edge is created for each individual's type. For instance, example given in Axiom 10 maps to two edges, that is $C_1 \xrightarrow{r_1} C_2$ and $C_1 \xrightarrow{r_1} C_3$.

$$C_1 \sqsubseteq \exists r_1.\{i_1\} \sqcup \{i_2\}, \ i_1 : C_2, \text{ and } i_2 : C_3 \tag{10}$$

Inverse Properties: Given an edge in the navigation graph such as $C_1 \xrightarrow{r_1} C_2$ and an inverse property axiom for the corresponding object property such as given in Axiom 11, a new edge is created for the inverse property, that is $C_2 \xrightarrow{r_1^{-1}} C_1$.

$$r_1 \equiv r_1^- \tag{11}$$

Role Chains: Given two edges $C_1 \xrightarrow{r_1} C_2$ and $C_2 \xrightarrow{r_2} C_3$ n the navigation graph, and a role chain axiom between r_1, r_2, r_3 such as given in Axiom 12, a new edge is created for r_3, that is $C_1 \xrightarrow{r_3} C_3$.

$$r_1 \circ r_2 \sqsubseteq r_3 \tag{12}$$

Top-down Propagation: Given an edge $C_1 \xrightarrow{r_1} C_2$ in the navigation graph and a subclass axiom such as as given in Axiom 13, a new edge is added to the graph, that is $C_3 \xrightarrow{r_1} C_2$. **Analogous edges** could be created for subproperties.

$$C_3 \sqsubseteq C_1 \tag{13}$$

Bottom-up Propagation: Given an edge $C_1 \xrightarrow{r_1} C_2$ in the navigation graph and a subclass class axiom such as given in Axiom 14, a new edge is added to the graph, that is $C_3 \xrightarrow{r_1} C_2$. **Analogous edges** could be created for superproperties.

$$C_1 \sqsubseteq C_3 \tag{14}$$

3.1.2 Edges through Datatype Properties

Domains and Ranges: Domain and range axioms using datatype properties are translated to an edge. For instance, example given in Axiom 15 maps to an edge, that is $C_1 \xrightarrow{d_1} dt_1$.

$$\exists d_1.DatatypeLiteral \sqsubseteq C_1 \text{ and } \top \sqsubseteq \forall r_1.dt_1 \tag{15}$$

Datatype Property Restrictions: Datatype property restrictions, formed through existential quantification ($\exists$), universal quantification ($\forall$), max ($\geq$), min ($\leq$), exactly ($=$), and value are mapped to edges. For instance, the example given in Axiom 16 maps to $C_1 \xrightarrow{d_1} dt_1$.

$$C_1 \sqsubseteq \exists d_1.dt_1 \tag{16}$$

Top-down Propagation: Given an edge $C_1 \xrightarrow{d_1} dt_1$ in the navigation graph and a subclass axiom such as as given in Axiom 17, a new edge is added to the graph, that is $C_2 \xrightarrow{d_1} dt_1$. **Analogous edges** could be created for subproperties.

$$C_2 \sqsubseteq C_1 \tag{17}$$

Bottom-up Propagation: Given an edge $C_1 \xrightarrow{d_1} dt_1$ in the navigation graph and a subclass class axiom such as given in Axiom 18, a new edge is added to the graph, that is $C_3 \xrightarrow{d_1} dt_1$. **Analogous edges** could be created for superproperties.

$$C_1 \sqsubseteq C_3 \tag{18}$$

3.2 Suggestion Functions

As a minimum requirement the VQS should only allow suggestions leading to legal queries with respect to $G_{\mathcal{O}}$. However, we can increase the user experience by also considering the underlying dataset $\mathcal{D}$ and the partial query Q_p. In this article, we will consider several different *suggestion functions* $\mathcal{S}$ that takes $\mathcal{D}$ and Q_p as input and returns a set of suggestions [11]:

$$\mathcal{S}(\mathcal{D}, Q_p) = \{s_1, s_2, \ldots, s_k\}$$

If it is clear from the context what $\mathcal{D}$ and Q_p are, we may omit the input and just write $\mathcal{S}$.

3.2.1 The optimal suggestion function $\mathcal{S}_o$

We will now formally define the suggestion function that returns the adaptive extension suggestions we described in Section 1. It is what we consider to be the gold standard with respect to accuracy, and we call it the *optimal suggestion function $\mathcal{S}_o$*. The idea is simply to execute the generic query $Q_p \wedge Q_s^{ext}$ over $\mathcal{D}$, and then project the result onto the variables in the suggestion tuple:

$$\mathcal{S}_o = Ans_{(s)}(Q_o(s), \mathcal{D}) \text{ where } Q_o(s) = Q_p \wedge Q_s^{ext} \tag{19}$$

By selecting extensions from $\mathcal{S}_o$, the user is guaranteed to not end up with a too restrictive query, which is exactly what our goal is.

By replacing s with the given suggestion tuples from Table 2, we get the concrete formulas for each of the three supported query extension types:

1: $\mathcal{S}_o = Ans_{(p,C)}(Q_o(p, C), \mathcal{D})$ where $Q_o(p, C) = Q_p \wedge p(v_f, v) \wedge C(v)$

2: $\mathcal{S}_o = Ans_{(p)}(Q_o(p), \mathcal{D})$ where $Q_o(p) = Q_p \wedge p(v_f, v)$

3: $\mathcal{S}_o = Ans_{(p,x)}(Q_o(p, x), \mathcal{D})$ where $Q_o(p, x) = Q_p \wedge p(v_f, v) \wedge (v \in \{x\})$

As already indicated. $\mathcal{S}_o$ does not scale very well. The problem is that Q_p (and hence also Q_o) is arbitrary large in size and complexity, so there is no way to guarantee efficient results. Running it directly over $\mathcal{D}$ requires too many joins, and since Q_o is arbitrary, it is also impossible to pre-calculate all possible joins and store them in an index.

We will now show an example of how the optimal suggestion function works. If we assume that the partial query Q_p equals the query Q from Figure 2, and we want the calculate optimal datatype filter suggestions for the city names of the focus variable $v_f = v_0$, we need to evaluate the query of type 3 from above. In general, this calculates suggestions for all properties p, but we are now only interested in the names, since this is the only property of v_f without any filters yet. We know that Q_p only returns two cities: RO and NY, hence the relevant suggestions are $\mathcal{S}_o = \{(hasName, NewYork), (hasName, Rome)\}$.

3.2.2 Accuracy measure

Since $\mathcal{S}_o$ is the desired set of suggestions, we will use it to define the accuracy of any other suggestion function $\mathcal{S}$. To do this we use the well-established measures of precision and recall, which gives us the following two equations:

$$precision(\mathcal{S}) = \frac{|\mathcal{S}_o \cap \mathcal{S}|}{|\mathcal{S}|} \qquad recall(\mathcal{S}) = \frac{|\mathcal{S}_o \cap \mathcal{S}|}{|\mathcal{S}_o|} \tag{20}$$

Among these two accuracy measures, the recall is by far the most crucial one for our purpose. **Imperfect precision may lead to cases where the user sees extensions leading to no results, while imperfect recall, on the other hand, may completely block the user from making valid queries. In fact, since the recall is so crucial, we will in this article only consider suggestion functions with perfect recall. It is important to understand that these metrics only indicate of how well a suggestion function removes dead-ends. It must not be confused with precision and recall related to the final selection of the user.**

3.2.3 The range-based suggestion function $\mathcal{S}_r$

An alternative to the optimal solution which is used by many systems today because of its simplicity is what we call the *range-based suggestion function $\mathcal{S}_r$*. This function aims to gather the full range of suggestions defined by the data, regardless of the state of the partial query. To do this, it ignores all parts of Q_p except for the focus variable and its type:

$$\mathcal{S}_r = Ans_s(Q_r(s), \mathcal{D}) \text{ where } Q_r(s) = C_f(v_f) \wedge Q_s^{ext} \tag{21}$$

Since we know that $C_f(v_f)$ is one of the conjunctions in Q_p (i.e. Q_p is more restrictive), it is possible to establish a relationship between the two suggestion functions $\mathcal{S}_o$ and $\mathcal{S}_r$:

$$Q_p \sqsubseteq C_f(v_f) \Rightarrow Q_o \sqsubseteq Q_r \Rightarrow \mathcal{S}_o \subseteq \mathcal{S}_r$$

From this we can update the formulas for precision and recall:

$$precision(\mathcal{S}_r) = \frac{|\mathcal{S}_o \cap \mathcal{S}_r|}{|\mathcal{S}_r|} = \frac{|\mathcal{S}_o|}{|\mathcal{S}_r|} \qquad recall(\mathcal{S}_r) = \frac{|\mathcal{S}_o \cap \mathcal{S}_r|}{|\mathcal{S}_o|} = \frac{|\mathcal{S}_o|}{|\mathcal{S}_o|} = 1$$

It makes sense that $\mathcal{S}_o$ returns fewer suggestions than $\mathcal{S}_r$ since it considers all the restrictions given by Q_p. This leads to the fact that $\mathcal{S}_r$ has perfect recall, which is important. The precision of $\mathcal{S}_r$ however, is not perfect, and depends on how close $\mathcal{S}_r$ is to $\mathcal{S}_o$.

Even though the precision of $\mathcal{S}_r$ is not perfect, it is still a powerful suggestion function, because it can be computed very efficiently. The suggestions given by $\mathcal{S}_r$ only depend on the focus concept C_f, which is limited to a relatively small and finite set of concepts. This means that we can calculate the set of suggestions for each possible focus concept offline, and index the results. Now the VQS can easily fetch suggestions during a query session by simply looking up the static set corresponding to the given focus concept.

Since $\mathcal{S}_r$ is well-known, and the default solution for many systems, we consider this the baseline with respect to accuracy. We will also use $\mathcal{S}_r$ as a fallback solution for the method we present in the following section.

If we assume that the partial query Q_p equals Q from Figure 2, we only consider the concept type City and the corresponding names. This gives us the following set of suggestions

$$\mathcal{S}_r = \{(hasName, Oslo), (hasName, Vienna),$$
$$(hasName, NewYork), (hasName, Rome)\}$$

and $precision(\mathcal{S}_r) = \frac{|\mathcal{S}_o|}{|\mathcal{S}_r|} = \frac{2}{4} = 0.5$.

3.3 The Query Extension Index

In this section, we describe our main contribution: a method to efficiently calculate **dead-end free** suggestions for all the three possible query extension types in Section 2.4 with high accuracy. The method requires a *query extension index $\mathcal{I}$* in order to ensure sufficient performance, and we will use what we call a *configuration*

query $\mathcal{Z}$ to configure/represent the content of this index. The idea is then to make suggestions based on just the parts of Q_p that are included in $\mathcal{Z}$, and hence $\mathcal{I}$. This gives basis to the suggestion function $\mathcal{S}_a^{\mathcal{Z}}$, which is one of many possible functions in the family of *approximate suggestion functions* $\mathcal{S}_a$.

3.3.1 The Configuration Query $\mathcal{Z}$

Before we can construct or use the query extension index $\mathcal{I}$, we need a way to represent the data it contains. To do this we will use a special query without any filters called a *configuration query* $\mathcal{Z}$.

In order to make our system work, it is important that the configuration query we use has a root of type C_f. This requires a setup with multiple configuration queries – one for each possible focus concept. However, given a particular partial query Q_p, there will only be one corresponding configuration query $\mathcal{Z}$, so for now we focus only on this one.

Intuitively $\mathcal{Z}$ works as a configuration for our system, by deciding which parts of $\mathcal{D}$ to include in the generated index, and hence which parts of Q_p it has consider/ignore when making suggestions (see Section 3.3.3). A large $\mathcal{Z}$ will in general result in a large index, but a corresponding suggestion function $\mathcal{S}_a^{\mathcal{Z}}$ with high precision. A small $\mathcal{Z}$, on the other hand, will in general result in a cheaper index with lower precision. It is also important to consider the structure of $\mathcal{Z}$: Best results are achieved by including properties and concepts that users are likely to use in their queries, while making sure that the size of the index does not explode.

In this article, we assume that the configuration query is made in advance by a human or algorithm with knowledge about the users, the domain and the dataset. It is impossible for the configuration maker to know exactly what the partial queries will look like, but based on for example a query log of the user it will often be possible to estimate it. This together with the dataset can be used to make a configuration that leads to a useful but relatively small index. The results we can achieve depends on properties of the dataset like size and branching degree, but also on how similar the new query is to the queries in the query log. In general, we have a trade-off between quality and index size. Datasets with high branching degree will potentially lead to exponentially growth in index size, and in these cases the configuration query has to be relatively small and not very complex.

We are currently working towards an algorithm that can automatically search for the optimal configurations given a threshold on the index size. This is quite challenging due to the large search space of all possible configuration queries, but also due to the fact that we need to execute queries over possibly very large datasets in order to evaluate them. We believe it is possible to overcome this problem by estimating the number of answers a query returns, but this is part of another study.

Above we stated that the root of $\mathcal{Z}$ must be of the same type as the focus variable of Q_p, which is C_f. This is necessary in order to be able to compare and intersect $\mathcal{Z}$ with Q_p. This becomes clear in Section 3.3.3 when we describe the approximate suggestion function $\mathcal{S}_a^{\mathcal{Z}}$.
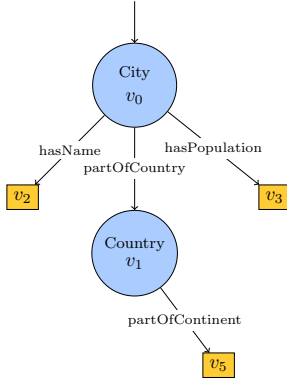
Fig. 3: Example configuration query $\mathcal{Z}$ including a city's name and population, and the corresponding country's continent.
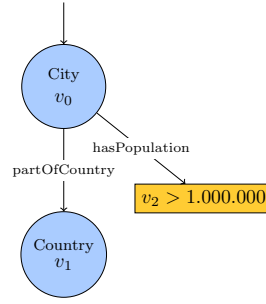
Fig. 4: The pruned query we get by intersecting Q from Figure 2 and $\mathcal{Z}$ from Figure 3.

3.3.2 Index Generation

Since the performance is so crucial when making online suggestions, we need an index to support this task. This index must be constructed offline, and it is supposed to serve multiple (online) user sessions. To do this well, it is important to select a suitable subset of $\mathcal{D}$ to index, which is achieved by using a good configuration query $\mathcal{Z}$.

Given a configuration query $\mathcal{Z}$, the idea is to include all data from $\mathcal{D}$ that is fully or partially covered by it. To do this we first need to construct the modified version of $\mathcal{Z}$ where every branch and subbranch is optional. We call this query $\mathcal{Z}_{opt}$. Now we get the index by executing $\mathcal{Z}_{opt}$ over $\mathcal{D}$:

$$\mathcal{I} = genIndex(\mathcal{Z}, \mathcal{D}) = Ans(\mathcal{Z}_{opt}, \mathcal{D}) \tag{22}$$

One can represent $\mathcal{I}$ in two different ways: either as a denormalised table with one column for each variable in $\mathcal{Z}$, where each row represents a possible assignment to these variables, or as a data graph, i.e. the subset of $\mathcal{D}$ which is covered by $\mathcal{Z}_{opt}$. Which one of these we use is irrelevant with respect to precision. However, if we consider performance, the tabular representation is preferred for the type of queries we have, so we use this in our actual implementation.

Table 3 gives an example of an index table generated from $\mathcal{Z}$ in Figure 3 and the dataset $\mathcal{D}$ from Table 1. In this simple example the number of rows is very small, but in a larger more realistic case, the number of rows will increase rapidly if many-to-many or one-to-many relationships exists.

3.3.3 The Approximate Suggestion Function $\mathcal{S}_a$

Given a configuration query $\mathcal{Z}$, we have what we need to define the corresponding suggestion function $\mathcal{S}_a^{\mathcal{Z}}$:

$$\mathcal{S}_a^{\mathcal{Z}} = Ans_{(s)}(Q_a^{\mathcal{Z}}(s), \mathcal{D}) \text{ where } Q_a^{\mathcal{Z}}(s) = (Q_p \cap \mathcal{Z}) \wedge Q_s^{ext} \tag{23}$$

City	City-name	City-pop.	Country	Country-continent
OS	Oslo	-	NOR	Europe
VI	Vienna	1.7M	AUT	Europe
RO	Rome	2.9M	ITA	Europe
NY	New York	8.5M	USA	North America

Table 3: The resulting index table $\mathcal{I} = genIndex(\mathcal{Z}, \mathcal{D})$ with $\mathcal{Z}$ from Figure 3 and $\mathcal{D}$ from Table 1.

Here $Q_p \cap \mathcal{Z}$ is the pruned version of Q_p we get by intersecting the trees defined by Q_p and $\mathcal{Z}$. If Q_p contains filters on any of the datatype variables, they are kept.

If we assume that $\mathcal{Z}$ contains all possible properties related to the root, i.e. the root is fully saturated, then $Q_a^{\mathcal{Z}}$ is completely covered by $\mathcal{Z}$. This means that all data from $\mathcal{D}$ that is relevant for $Q_a^{\mathcal{Z}}$ is also included in $\mathcal{I}$, i.e.

$$\mathcal{S}_a^{\mathcal{Z}} = Ans_{(s)}(Q_a^{\mathcal{Z}}(s), \mathcal{D}) = Ans_{(s)}(Q_a^{\mathcal{Z}}(s), \mathcal{I})$$

In other words, we get the same result if we run $Q_a^{\mathcal{Z}}(s)$ over $\mathcal{I}$ instead of $\mathcal{D}$. The advantage of using $\mathcal{I}$ instead of $\mathcal{D}$ directly is of course that suggestions are returned fast enough.

However, if the root of $\mathcal{Z}$ is not fully saturated, then our approach will not return any suggestions related any property p missing. In that case, the system can always fall back on the range-based solution $\mathcal{S}_r$ for p, or simply not give any suggestions related to it.

We have now considered three different suggestion functions. If we compare the formulas each of them uses, and focus on a fixed property, we get the following relationship between them:

$$Q_p \sqsubseteq (Q_p \cap \mathcal{Z}) \sqsubseteq C_f(v_f) \Rightarrow Q_o \sqsubseteq Q_a^{\mathcal{Z}} \sqsubseteq Q_r \Rightarrow \mathcal{S}_o \subseteq \mathcal{S}_a^{\mathcal{Z}} \subseteq \mathcal{S}_r \qquad (24)$$

And from this we can derive the full relationship between the precision and recall of the functions:

$$recall(\mathcal{S}_r) = recall(\mathcal{S}_a^{\mathcal{Z}}) = recall(\mathcal{S}_o) = 1$$

$$0 \leq precision(\mathcal{S}_r) \leq precision(\mathcal{S}_a^{\mathcal{Z}}) \leq precision(\mathcal{S}_o) = 1$$

Given a partial query Q_p, and a fixed property p, each of the three functions will give us a set of suggestions. $\mathcal{S}_o$ returns the optimal set by considering the whole structure of Q_p, $\mathcal{S}_a^{\mathcal{Z}}$ returns a larger less precise set by ignoring everything not covered by $\mathcal{Z}$, and $\mathcal{S}_r$ returns an even larger set of suggestions by not considering the structure of Q_p at all.

We will now calculate approximate suggestions using the same input as we used with $\mathcal{S}_o$ and $\mathcal{S}_r$. The intersection $(Q_p \cap \mathcal{Z})$ can be seen in Figure 4, and it only includes the filter on the city's population, which has to be higher than 1M. We are then left with three city individuals: VI, RO and NY, which gives the following suggestions for the name property:

$$\mathcal{S}_o = \{(hasName, Vienna), (hasName, Rome), (hasName, NewYork)\}$$

The corresponding precision of the approximate function is then $precision(\mathcal{S}_a^{\mathcal{Z}}) = \frac{|\mathcal{S}_o|}{|\mathcal{S}_a^{\mathcal{Z}}|} = \frac{2}{3} = 0.66$.

3.3.4 Existential Concept Variables

With the index construction method described in Section 3.3.2, the columns representing concept variables will be filled with only URIs. This data is wasted space: Users do not need to filter on URIs, and suggested values of URIs are therefore not needed. However, it is often interesting to know whether an assignment to the concept variable exists or not, so instead of removing the column completely, we replace the URIs with boolean values indicating whether an assignment exists or not. This reduces the index size considerably, compared to the case where all URIs are stored, because multiple rows where only one URI differs can now be collapsed into only one row.

By using existential concept variable columns, it becomes quite cheap to include concept variables in the configurations, since it only requires one more column of boolean values, while the number of rows stays fixed. In Experiment 1, we explore how much the accuracy increase by adding another layer of these existential concept nodes to the index, which is a comparatively cheap investment.

4 Evaluation

We implemented our ontology projection approach and adaptive extension suggestion solution and conducted a series of experiments. The results and findings are presented in what follows.

4.1 Ontology Projection

The evaluation of ontology projection approach includes its use in practical systems and a performance evaluation checking its feasibility for use in interactive applications without any significant delay in a query interface.

4.1.1 Practical Use

The variants of ontology projection approach has been implemented in OptiqueVQS [27], a visual query formulation tool, and SemFacet [1], a faceted search tool. Both interfaces support tree-shaped conjunctive queries and their source codes are available online in GitLab [4] [5].

OptiqueVQS (see Figure 5) is a visual query system. It allows users to navigate the conceptual space and each traversal from a class to another adds a typed variable-node and object property connecting it to the query graph. OptiqueVQS was deployed and evaluated in different use cases, including Siemens' case for sensor data [25,10], Statoil's case for oil and gas [27,9], and on generic datasets [24]. In Figure 5, there is an example query asking for all trains with a turbine named "Bearing Assembly" and their journal bearing temperature readings in the associated generator.

---

[4] `https://gitlab.com/ernesto.jimenez.ruiz/OptiqueVQS`

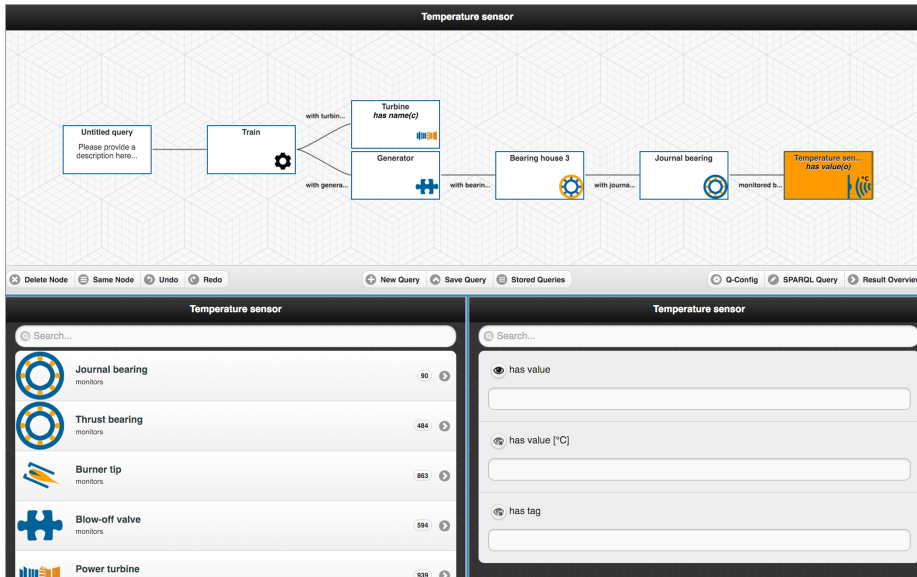[5] `https://github.com/OxfordSemTech/SemFacet`

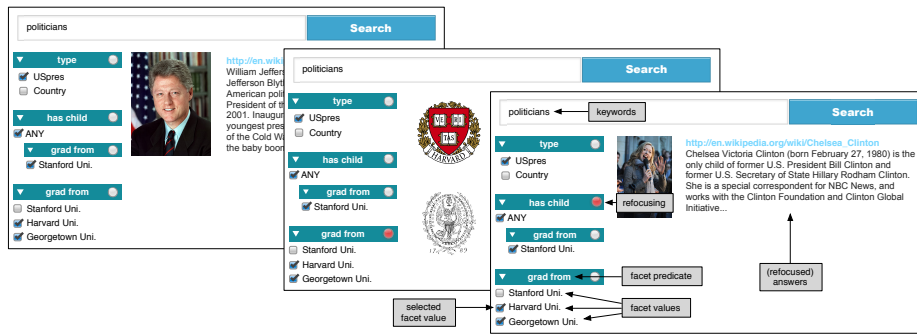Fig. 5: OptiqueVQS over a use case provided by Siemens.

Fig. 6: SemFacet over Yago Knowledge Graph.

SemFacet (see Figure 6) is full-fledged general-purpose faceted search interface. In typical faceted search, users are presented with facet-values organised in groups according to facet-names and it is often not allowed to navigate between classes. SemFacet allows end users to navigate between classes and browse data sets at the same time. The interface was deployed and evaluated over a slice of Yago database [1]. In Figure 6 there is an example search for U.S. presidents who graduated from Harvard or Georgetown, and whose children graduated from Stanford. All these conditions are combined conjunctively and their constraints apply simultaneously. One can see that changing the focus of the query, one can either see the presidents (left part of the figure), or their universities (centre part of the figure), or their children (right part of the figure).

4.1.2 Performance

Our current implementation of the ontology projector is written in Java, and used by both OptiqueVQS and the SIRIUS Geoscience image annotation and

Ontology	Axioms	Classes	SubSlassOf
TMO	1152	225	235
NPD[14]	4110	142	594
MI	13645	1494	1474
IDOMAL	31101	3159	3556
ENM	105983	12533	17191

Table 4: Key metrics of the ontologies used in the performance evaluation of the projection algorithm. The columns provide the name of the ontology, the number of axioms in total, the number of distinct classes and the number of subclass axioms in the ontology.

classification project [6]. It uses the HermiT Reasoner [7] for classification, and RDFox to do the propagation of properties/edges in the resulting graph. The full source code is publicly available in GitLab [8].

When evaluating this implementation, we focused on the time spent on constructing the navigation graph, and querying over it. The first task is to construct the navigation graph from a given ontology using the described approach. This only needs to be done once each time the ontology changes and it is an offline process, so it is in practice possible to run this on a remote server, and the timing of this is not very crucial. The second task is to query $G_{\mathcal{O}}$ in order to determine which actions the user is allowed to make. In practice this means to find all outgoing datatype properties and object properties for a given concept in $G_{\mathcal{O}}$. Since this is done frequently during a query session, it should ideally finish so fast that the user does not even notice the delay.

We considered 5 different ontologies: Translational Medicine Ontology[9] (TMO), Norwegian Petroleum Directorate Factpages[10] (NPD), Molecular Interactions[11] (MI), Malaria Ontology[12] (IDOMAL) and eNanoMapper[13] (ENM). NPD is an ontology covering the Oil&Gas domain, while the remaining four are from the biology domain. We used the newest available versions of the ontologies at the time we conducted the experiment (28. January 2019). All of them are listed in Table 4 together with relevant metrics about them. For each of them we performed the two evaluation tasks, and all results are presented in Table 5.

The results show that querying over the navigation graph is lightning fast. In fact, among the ontologies we tested, it never took more than 5ms to fetch the relevant properties with an average of about 1ms. Note that 100ms is the suggested limit for having the user feel that the system is reacting instantaneously [17]. Regarding the task of the constructing the navigation graph, it is much more

---

[6] https://github.com/Sirius-sfi/geoscience-image-classification

[7] http://www.hermit-reasoner.com/

[8] https://gitlab.com/ernesto.jimenez.ruiz/ontology-services-toolkit/tree/master

[9] https://bioportal.bioontology.org/ontologies/TMO

[10] https://gitlab.com/logid/npd-factpages/blob/develop/ontology/npd-db.ttl.owl

[11] https://bioportal.bioontology.org/ontologies/MI

[12] https://bioportal.bioontology.org/ontologies/IDOMAL

[13] https://bioportal.bioontology.org/ontologies/ENM

Ontology	Projection creation	Datatype properties	Object properties
TMO	4001	0.42	0.49
NPD	4573	0.37	0.31
MI	3878	0.3	0.3
IDOMAL	4391	0.3	0.3
ENM	57641	0.3	0.3

Table 5: Performance of the ontology projection for the five given ontologies. The columns present the name of the ontology, the time it takes to create the navigation graph, time spent querying for all datatype properties and object properties on average. All times are given in milliseconds.

complex and hence slower. However, it is still within the reasonable time frame of one minute for all the tested ontologies, given that this is an offline task.

4.2 Extension Suggestion

In this section we describe the two experiments done on our implementation of the query extension index. Each of them focuses on the third type of suggestions, datatype filters, since this is the hardest suggestion task.

4.2.1 Dataset, Ontology and Queries

We used the RDF version of the NPD Factpages[15] – a dataset covering details about oil and gas drilling activities in Norway. This dataset contains 2.342.597 triples, and it has a corresponding OWL ontology containing 209 concepts and 375 properties. The NPD Factpages is actually a RDB, containing information that all oil companies in Norway are legally required to report to the authorities. This means that the RDF version, which is generated from this RDB, is fairly complete and homogeneous. This is optimal for persons who want answers to complex queries. Among the different concepts we considered in our queries, each have on average of 14.1 different outgoing datatype properties, and 6.4 outgoing object properties in NPD Factpages. The number of distinct individuals/literals each such property leads to is 572 on average (with a median of 12).

The query log distributed with this dataset was not suited for our experiment, since only a few of the queries had the structure our system required, and none of them connected more than a few concepts together. Therefore, we constructed a new query log consisting of complex queries of a more suitable size, with the goal to cover a wide set of possible cases. The log consists of 29 queries ranging from 5 to 8 concept variables and 0 to 12 datatype variables, and the corresponding result sets over the NPD dataset range from just 12 tuples, to over 5 million tuples. **Furthermore Table 6 gives the query size distribution.** The complete query log is publicly available on GitHub[16].

---

[15] https://gitlab.com/logid/npd-factpages
[16] https://github.com/Alopex8064/npd-factpages-experiments

Query size	5	6	7	8
Frequency	3	15	2	9

Table 6: Frequency of query size in the query log[17] used in the experiments.

4.2.2 Test Cases and General Setup

In both of our experiments we ran multiple test cases, where each test case used a query Q_p from the query catalogue, and a generated concept configuration $\mathcal{Z}$. By testing multiple $\mathcal{Z}$ together with each Q_p we got a fairly complete picture of how our system behaves. For each test case we first constructed the corresponding index $\mathcal{I}$ based on $\mathcal{Z}$ and the dataset. Then we calculated suggestions for all of the three defined suggestion functions, and the corresponding precision they achieved. We also calculated the *cost* associated with each choice of $\mathcal{Z}$. This cost differed between the two experiments, but was related to the size of either $\mathcal{Z}$ or $\mathcal{I}$. (See the individual experiments for details.)

Since we only considered one query Q_p at a time, it was pointless to consider parts of $\mathcal{Z}$ which was not covered by Q_p itself. Every addition to $\mathcal{Z}$ that is outside Q_p will not affect the resulting suggestions. So we focused on the more interesting cases where $\mathcal{Z}$ was fully covered by Q_p.

Notice that a real-world scenario would be more complex than our setup with simple test cases. The success of a concept configuration and its corresponding index would not only depend on the precision of one single query, but rather a large set of possibly very different queries. One of our future goals is to develop methods for finding configurations that works well for a whole catalogue of queries.

4.2.3 Experiment 1: Configuration Type/Size vs Precision

In Experiment 1, we show how the accuracy of $\mathcal{S}_a$ changes as configurations of different size and shape are used. To do this, we first generated a set of random "configurations cores" c for each query Q_p in the query catalogue. Each core consisted of one or more connected concept variables from Q_p, and was just used as a basis for generating two other concept configurations:

- $Dat(c)$: Every possible datatype property is added to the concept variables in c.
- $ObjDat(c)$: Every possible datatype property *and* object property is added to the concept variables in c.

The only difference between these two configurations, is that $ObjDat(c)$ contains one extra layer of concept variables. It is relatively cheap (w.r.t. storage usage) to add these concept variables, as described in Sect. 3.3.4, but the precision will (potentially) increase by doing it. So the split between $Dat(c)$ and $ObjDat(c)$ was created in order to measure how much the precision increases.

Both of the two configurations $Dat(c)$ and $ObjDat(c)$ were used in one test each, where suggestion values for each of the four different suggestion functions of interest were calculated. They are given below, and they satisfy:

$$precision(\mathcal{S}_r) \leq precision(\mathcal{S}_a^{Dat(c)}) \leq precision(\mathcal{S}_a^{ObjDat(c)}) \leq precision(\mathcal{S}_o) = 1.$$

```
1   PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
2   PREFIX npd: <http://sws.ifi.uio.no/vocab/npd-v2#>
3
4   SELECT * WHERE {
5       ?c1 rdf:type npd:ExplorationWellbore.
6       ?c2 rdf:type npd:Field.
7       ?c5 rdf:type npd:ProductionLicence.
8       ?c3 rdf:type npd:Company.
9       ?c4 rdf:type npd:FieldStatus.
10      ?c8 rdf:type npd:Discovery.
11      ?c6 rdf:type npd:ProductionLicenceStatus.
12      ?c7 rdf:type npd:ProductionLicenceArea.
13
14      ?c1 npd:explorationWellboreForField ?c2.
15      ?c1 npd:explorationWellboreForLicence ?c5.
16      ?c2 npd:currentFieldOperator ?c3.
17      ?c4 npd:statusForField ?c2.
18      ?c6 npd:statusForLicence ?c5.
19      ?c7 npd:isGeometryOfFeature ?c5.
20      ?c8 npd:includedInField ?c2.
21
22      ?c1 npd:wellboreBottomHoleTemperature ?a7.
23      ?c2 npd:name ?a8.
24      ?c2 npd:status ?a9.
25      ?c3 npd:name ?a6.
26      ?c4 npd:status ?a5.
27      ?c5 npd:isActive ?a10.
28      ?c5 npd:name ?a11.
29      ?c5 npd:originalAreaSize ?a12.
30      ?c6 npd:status ?a4.
31      ?c7 npd:isStratigraphical ?a1.
32
33      FILTER(?a7 >= 150).
34      FILTER(regex(?a8, "TAMBAR", "i")).
35  }
```
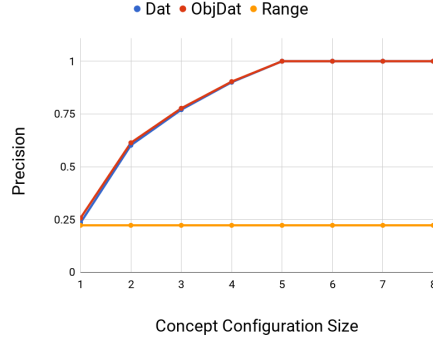

Fig. 7: Results for Query 2.6

After running through every test case, the results were grouped by both the configuration type (Dat or ObjDat) and the size of the configuration, where the size of a configuration is defined by the number of concept variables in the configuration core c. Finally the average precision of each group was calculated and the results visualised.

Results and Analysis: **This section contains results from experiment 1. First we present individual results for three selected queries (Query 2.6, Query 2.8 and Query 3.5) in Figure 7, Figure 8 and Figure 9. Then follows the resulting averages for queries of size between 5 and 8 in Figure 10-Figure 13. For a the complete set of queries and corresponding results, we refer to GitHub[18].**

The yellow line in each chart shows the precision of the range-based function $\mathcal{S}_r$, which is always constant. Since this is the suggestion function with the lowest precision we consider, it acts as a baseline – marking the worst case scenario for $\mathcal{S}_a$. The blue and red curves show the precision of $\mathcal{S}_a^{Dat}$ and $\mathcal{S}_a^{ObjDat}$ respectively. As expected, these two curves are non-decreasing and $precision(\mathcal{S}_a^{Dat}) \leq precision(\mathcal{S}_a^{ObjDat})$ for all configuration sizes.

The precision given by each of the three curves depends mostly on how many of the important key restrictions of Q_p they are able to capture, where a key restriction is a restriction that reduces the number of instances one could assign

---

[18] https://github.com/Alopex8064/npd-factpages-experiments

```
1    PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
2    PREFIX npd: <http://sws.ifi.uio.no/vocab/npd-v2#>
3
4    SELECT * WHERE {
5        ?c1 rdf:type npd:ExplorationWellbore.
6        ?c2 rdf:type npd:Field.
7        ?c3 rdf:type npd:Company.
8        ?c4 rdf:type npd:FieldStatus.
9        ?c5 rdf:type npd:ProductionLicence.
10       ?c6 rdf:type npd:ProductionLicenceStatus.
11       ?c7 rdf:type npd:ProductionLicenceArea.
12       ?c8 rdf:type npd:Discovery.
13
14       ?c1 npd:explorationWellboreForField ?c2.
15       ?c1 npd:explorationWellboreForLicence ?c5.
16       ?c2 npd:currentFieldOperator ?c3.
17       ?c4 npd:statusForField ?c2.
18       ?c6 npd:statusForLicence ?c5.
19       ?c7 npd:isGeometryOfFeature ?c5.
20       ?c8 npd:includedInField ?c2.
21
22       ?c1 npd:wellboreBottomHoleTemperature ?a7.
23
24       FILTER (?a7 >= 190).
25   }
```
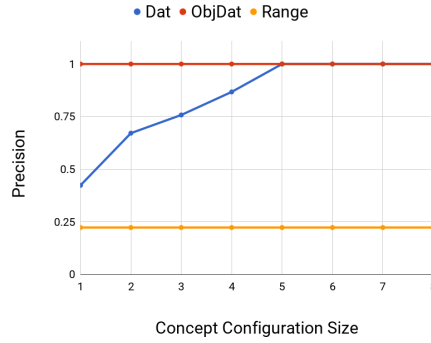

Fig. 8: Results for Query 2.8

to the root so much that it also causes a large reduction in the possible facet values. The query used in Figure 7 (Query 2.6) for example has one important key restriction on the data property *name* of the *Field* concept variable in depth 2 of Q_p. Since this key restriction is associated with a datatype variable, both $\mathcal{S}_a^{Dat}$ and $\mathcal{S}_a^{ObjDat}$ perform about equally well. The slight difference between $\mathcal{S}_a^{Dat}$ and $\mathcal{S}_a^{ObjDat}$ is caused by other much less important restrictions, which $\mathcal{S}_a^{ObjDat}$ manages to capture, but $\mathcal{S}_a^{Dat}$ does not. If this chart had shown the best case scenario, the precision would have been perfect already at size 2, because that is the point it would reach the *Field* concept node. But since we average over multiple differently shaped configurations, and the branching factor of Q_p is close to 2, the two lines moves steadily upwards until they reach size 5. At this point the configuration is guaranteed to cover the key restriction regardless of its shape.

Query 2.8 in Figure 8 has two key restrictions: the first restriction is associated with a datatype property filter on the root node (wellboreTemperature $\geq$ 190). This is captured by all the configurations we used in the experiment, and the difference between $\mathcal{S}_r$ and $\mathcal{S}_a^{Dat}$ at size 1 shows the effect of capturing it. The other key restriction is associated with the *Field* concept variable in depth 2. Since $\mathcal{S}_a^{ObjDat}$ includes one additional layer of concept variables, it captures this already from size 1, while $\mathcal{S}_a^{Dat}$ on the other hand, needs to be of the correct shape in order to capture it, hence the steadily rising curve, similar to Query 2.6 in Figure 7.

Query 3.5 in Figure 9 is a linear query (the tree has only one branch), so there is one possible configuration core for each configuration size. Hence, the resulting curve only shows that one case of growing configuration. This query also has two key restrictions. The first one is an object property restriction in depth 2 of the query – the effect of capturing this restriction is shown by the precision increase of $\mathcal{S}_a^{ObjDat}$ between size 1 and 2. The second restriction is a data property restriction associated with the only concept variable in depth 6 of the query. This restriction

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX npd: <http://sws.ifi.uio.no/vocab/npd-v2#>

SELECT * WHERE {
    ?c1 rdf:type npd:ExplorationWellbore.
    ?c2 rdf:type npd:Field.
    ?c3 rdf:type npd:FieldOperator.
    ?c4 rdf:type npd:Company.
    ?c5 rdf:type npd:BAA.
    ?c7 rdf:type npd:BAAArea.

    ?c1 npd:explorationWellboreForField ?c2.
    ?c3 npd:operatorForField ?c2.
    ?c3 npd:fieldOperator ?c4.
    ?c5 npd:baaOperatorCompany ?c4.
    ?c7 npd:isGeometryOfFeature ?c5.

    ?c7 npd:areaSize ?a3.

    FILTER(?a3 >= 300).
}
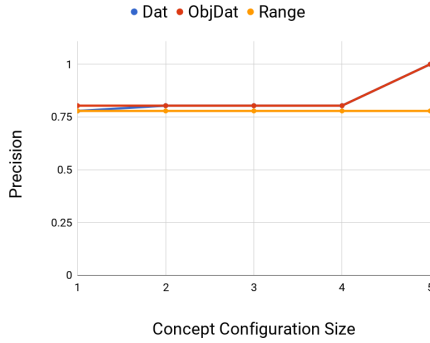```
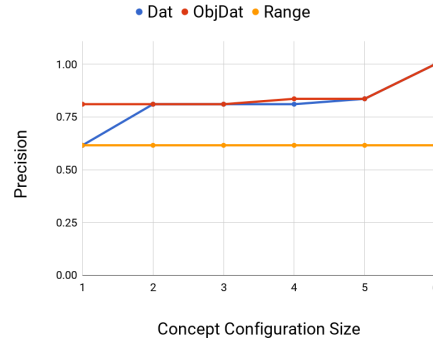
Fig. 9: Results for Query 3.5
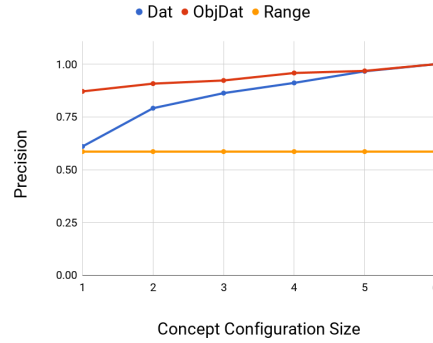
Fig. 10: Average precision of size 5 queries.

Fig. 11: Average precision of size 6 queries.

is very hard to capture for both $\mathcal{S}_a^{ObjDat}$ and $\mathcal{S}_a^{Dat}$, but when the configuration reaches size 6, and the whole query is covered by each of their configurations, the resulting precision becomes perfect.

The rules that control $\mathcal{S}_a^{ObjDat}$ and $\mathcal{S}_a^{Dat}$ also apply to $\mathcal{S}_r$. It only performs well if it is able to capture all of the important key restrictions. But since $\mathcal{S}_r$ never considers $Q_p$, it will in fact always perform poorly if one or more such key restrictions exists. Figure 7 and Figure 8 both show examples where this happens. For each of those cases the precision of $\mathcal{S}_r$ is only 0.22. This is quite low compared to 0.50, which is the average precision of size 8 queries given by Figure 13.

The charts in Figure 10-Figure 13 display the average over all queries grouped by query size. The relation $\mathcal{S}_r \leq \mathcal{S}_a^{ObjDat} \leq \mathcal{S}_a^{Dat}$ still holds over the averages. The first thing to notice from the average results is the relatively high precision of the range-based function. In our experiment, its precision ranged from 0.22 to 0.96, with an average of 0.56. This does not sound too bad, but user studies done with OptiqueVQS show that the users are not always satisfied with $\mathcal{S}_r$, which actually motivated us to start exploring $\mathcal{S}_a$ as an alternative.
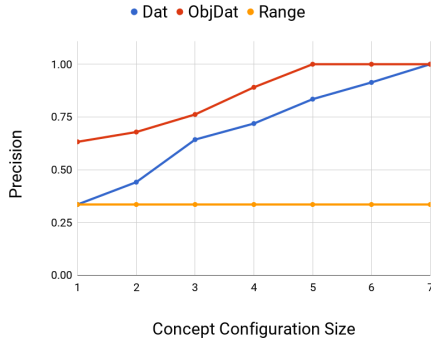
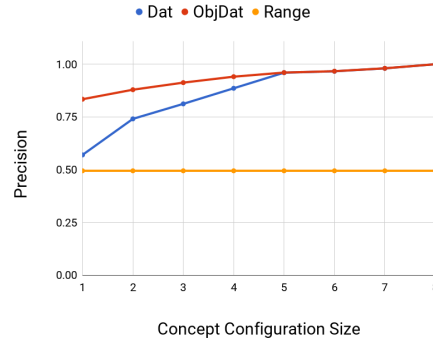Fig. 12: Average precision of size 7 queries.



Fig. 13: Average precision of size 8 queries.

In the cases where key restrictions are associated with object properties, $\mathcal{S}_a^{ObjDat}$ performs much better than $\mathcal{S}_a^{Dat}$. In fact, it quite often returns suggestions with perfect precision, as shown in Figure 8. The average difference between $\mathcal{S}_a^{ObjDat}$ and $\mathcal{S}_a^{Dat}$, shown in Figure 10-13, indicates that it is worth adding this extra layer of object properties to the configuration, especially since the resulting increase in the index size is relatively small (one extra boolean column).

**The average results in Figure 10-13 are highly influenced by the individual queries of the relevant size, especially for queries of size 5 and 7 where the average is based on only 3 and 2 queries respectively. Hence, we cannot conclude anything about how the query size affects the precision.**

### 4.2.4 Experiment 2: Index Size vs Precision

In Experiment 2 we made a direct comparison between the index size and the precision. We did this by first making one test case for every query $Q_p$, and each possible configuration $\mathcal{Z}$ covered by it. Then, for each such test case, we calculated both the size of the table generated by $\mathcal{Z}$, and the precision of $\mathcal{S}_a^{\mathcal{Z}}$. Finally, we analysed and visualised the results.

*Results and Analysis:* Figure 14 shows the results for one of the tested queries (Query 6.2), visualised as a scatter plot, where each point represents a test case/-concept configuration/index table. Some of the points are *pareto-optimal*, which means that neither of the two dimensions (precision and index size) can be improved without weakening the other. These points are located in the bottom right part of the plot (smaller index and higher precision are better), and are connected by line segments. The frontier of pareto-optimal points shows how large the index must be in order to achieve a given precision in a *best-case scenario*, i.e. when the configuration is chosen optimally.

There are two reasons for using the best-case scenario:

1. The configuration is a part of the setup process of our system, and is supposed to be optimised by experts or an algorithm.
2. The number of possible configurations in total is infinite, so using the average or something similar would be impossible.
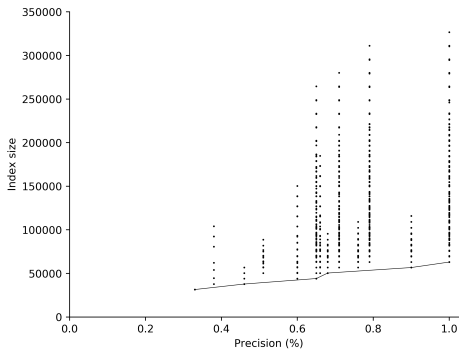
Fig. 14: Scatter plot for Query 6.2. Pareto optimal configurations are connected. Index size is not normalised.
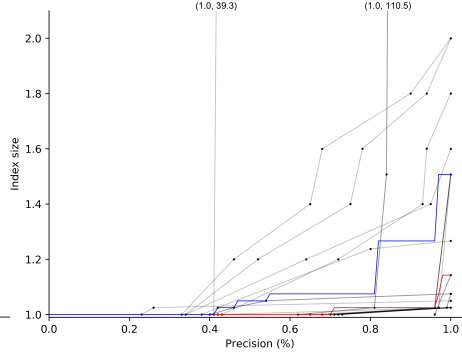
Fig. 15: Pareto-optimal configurations for all queries with median (red) and upper quartile (blue). Index size is normalised.

So while we cannot *expect* to achieve results like this consistently, it does give an indication of what might be achieved with an optimal choice of configuration. The fact that we investigate the best-case scenario also explains why it is sufficient to only consider the configurations covered by $Q_p$. For any configuration $\mathcal{Z}'$ with branches outside $Q_p$, there exists another concept configuration $\mathcal{Z}$ which leads to the same precision, but a smaller index. Visually, the set of all such test cases would appear as points above the already existing points, and hence not be candidates for pareto-optimality.

The set of pareto-optimal points for each query defines a monotonically increasing curve. Let $\mathcal{Z}^{min}$ and $\mathcal{Z}^{max}$ denote the configurations used for the first and last of these points. $\mathcal{Z}^{max}$ is the configurations that is isomorphic to $Q_p$. I.e. it fully covers $Q_p$, but it has no branches outside of it. The precision given by this configuration is perfect, but it also uses the largest index of the pareto-optimal configurations. $\mathcal{Z}^{min}$ on the other hand contains only the root and all local datatype properties. This is the smallest configuration that can provide suggestions for each of the local datatype properties.

When we look at the pareto-optimal configurations for all the different queries, we see that the index size of $\mathcal{Z}^{min}$ differs depending on the focus concept of the query. We can't expect the index to become smaller than a table of the instances of the class along with their attributes, which mostly depends on the number of instances in the dataset. So in order to compare them under equal conditions, we normalised the index size by dividing by the index size of $\mathcal{Z}^{min}$. The index size then becomes just a factor, where e.g 2.0 means that the index is twice as large as the index constructed from $\mathcal{Z}^{min}$. The pareto-optimal points for all the 29 queries are displayed in Figure 15 (normalised index size), together with the median (red) and upper quartile (blue).

The overall results from Figure 15 seems promising, as most of the transitions between pareto-optimal points (black line segments) are more horizontal than vertical. This means that with clever selection of configuration branches, one can transition to a much higher precision without having to increase the index very much. The median and upper quartile have similar horizontal profiles, but with a slight increase as they approach 100% precision, resulting in a more convex curve. **In other words, the last 10% precision will cost us more than any**

**other 10% increase. In a real-life scenario it will also never be possible to guarantee 100% precision because the users may construct queries not seen by the system before, so aiming for 100% precision is not a reasonable option anyway.**

From Experiment 1 and Figure 11 we know that the average precision of $\mathcal{S}_a$ when using the smallest possible configuration for each query ($\mathcal{Z}^{min}$) is 0.61. Figure 15 shows that this precision can be increased to 100% with an index that is less than 2.1 times larger, with the exception of three[19] queries that are orders of magnitude higher. This is caused by their highly restrictive filters on branches far away from the root. **The median goes up to 90% precision with about 2.5% increase in index, while going from 90% to 100% precision costs us an additional 10% increase in index size.**

**How well a configuration works, and what the optimal configuration is, depends to a high degree on the actual dataset and queries constructed by the users. Some datasets have a high branching degree, which causes the index to grow faster than for other datasets, and/or some query catalogues may have queries of very similar shape, (possibly) resulting in higher precision for configurations including these shapes. Therefore, we should be careful about generalising the results of this experiment to other datasets and query catalogues.**

## 5 Related Work

Regarding ontology projection, visualisations for different aspects of the Semantic Web such as ontology visualisation, query formulation, and search are relevant for the work presented here, since they mainly require end users to examine and interact with the elements of a given ontology. However, to best of our knowledge, none of the existing works deal with projecting navigation graphs from ontologies, although the inverse exists such as for ontology axiomatization through diagramming [20]. Among others [8], the graph paradigm is often used to depict the structure of ontological elements and relationships as they reflect the interconnected nature of ontology classes. There are various approaches using graphs for ontology visualisation and exploration such as GrOWL [14] and KC-Viz [16]. Similarly, tools for visual query formulation also often use the graph paradigm to depict the information needs and domain exploration such as gFacet [7] and NITELIGHT [21]. In a graph-based approach, classes are often represented as nodes and properties as edges.

Non graph-based approaches, such as form-based, still use a navigation approach for browsing through ontology classes. Examples include Rhizomer [3], a faceted search tool, and PepeSearch [31], a form-based query formulation tool. Typically, form-based approaches are meant to operate on a single class level; however, as in the case of Rhizomer and PepeSearch, navigation between classes is an essential instrument. OptiqueVQS and SemFacet represent these two different paradigms, that is graph-based and form-based respectively. In OptiqueVQS, the navigation graph is used to explore the domain, while a constrained tree-shaped

---

[19] There are two queries pointing towards (1.0, 110.5) and one pointing towards (1.0, 39.3).

representation is used for query visualisation instead of a graph for usability purposes, while SemFacet allows navigation between classes and employs form elements rather than graphical visualisations. We refer interested readers to related publications [1, 27] on these tools including end user experiments.

Regarding data-driven adaptive suggestions, there are plenty of systems that suggest filters on the facets of a single class. In fact, this a core feature of faceted search, which is quite common on websites like e.g. Ebay [20] and PriceSpy [21]. Popular implementations of faceted search includes e.g. Apache Solr [22] and Elasticsearch [23]. **Since these systems only consider one class at a time, they can afford to calculate dead-end extensions with both perfect precision and recall, which distinguish them from our system.**

All existing systems that support multiple connected classes while aiming to provide adaptive extension suggestions has some kind of weakness. SemFacet [1] is one of these systems. It relies on a highly scalable in-memory RDF triple store (RDFox) in order to get sufficient performance, but even this does not help if the queries are very complex. **Other systems like e.g. DISQOVER [18] restricts the user by only allowing extensions leading to query with result count under a given threshold. Many of these systems are both mature and feature-rich, and provides more than the dead-end elimination our system delivers. One example of this is ranking, which is useful when the number of valid extensions is so large that one must prioritise what to display to the user. The dead-end elimination we provide can be considered to be a (binary) ranking method in this respect.** To our knowledge, no previous work has considered the particular query extension index we present, or the approximation of suggestions that comes with it.

## 6 Conclusion

In this article, we focused on ontology-based VQSs from an end-user perspective and explored means for using ontologies for the query formulation task, that is how one can navigate through the concepts of a given ontology and how elements of an ontology could be efficiently and effectively suggested to an end user **without leading to any empty results**. We first presented an approach for projecting ontologies into navigation graphs for the purpose of supporting query formulation and ontology exploration tasks. However, one should note that such an approach is useful in general for supporting ontology-based user interfaces. Ontology to graph projection approach is implemented and tested in two different VQSs and experiment results suggest that we can efficiently project a given ontology into a navigation graph and query it. Secondly, we introduced three query extension suggestion functions **for eliminating dead-ends**: an optimal one that is slow for large datasets and complex queries; a range based one that is rather inaccurate, but allows fast implementation; and a configurable family of intermediate (precise enough and fast enough) solutions to the problem, based on only looking at a part of the constructed query. We conducted a series of experiments to conclude that

---

[20] https://www.ebay.com/

[21] https://pricespy.co.uk/

[22] http://lucene.apache.org/solr/

[23] https://www.elastic.co/

1. good approximations to the best set of suggestions can often be reached by taking into account only relatively small parts of the constructed query.
2. the precision of the approximations can often be improved dramatically by including the presence of required object properties in the configuration, rather than only connected datatype properties.
3. modest increases in index size will (in many cases) lead to a significant increase in accuracy.

**In future work we intend to further improve the suggestions given to users by providing a ranking on the extensions that are not dead-ends. This ranking could be based on either the underlying data and/or a given query log [12].** Furthermore, we would like to consider alternative storage formats for the pre-joined index. In particular a document database like MongoDB could be suitable. A related question is how to share storage space between indices for sub- and super-classes in the type hierarchy. The viability of our approach depends on a good choice of the facet configuration: it should be possible to determine an optimal configuration given a log of previous user queries. Another approach to reducing the index size is to work with buckets that combine ranges of facet values. Suitable bucketing strategies can also be determined from the query log and data.

## References

1. Arenas, M., Grau, B.C., Kharlamov, E., Marciuska, S., Zheleznyakov, D.: Faceted search over RDF-based knowledge graphs. Journal of Web Semantics **37-38**, 55–74 (2016). DOI 10.1016/j.websem.2015.12.002
2. Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F. (eds.): The Description Logic Handbook: Theory, Implementation, and Applications. Cambridge University Press, New York, NY, USA (2003)
3. Brunetti, J.M., García, R., Auer, S.: From Overview to Facets and Pivoting for Interactive Exploration of Semantic Web Data. International Journal on Semantic Web and Information Systems **9**(1), 1–20 (2013). DOI 10.4018/jswis.2013010101
4. Catarci, T.: What Happened When Database Researchers Met Usability. Information Systems **25**(3), 177–212 (2000). DOI 10.1016/S0306-4379(00)00015-6
5. Catarci, T., Costabile, M.F., Levialdi, S., Batini, C.: Visual Query Systems for Databases: A Survey. Journal of Visual Languages and Computing **8**(2), 215–260 (1997). DOI 10.1006/jvlc.1997.0037
6. Grau, B.C., Giese, M., Horrocks, I., Hubauer, T., Jiménez-Ruiz, E., Kharlamov, E., Schmidt, M., Soylu, A., Zheleznyakov, D.: Towards Query Formulation, Query-Driven Ontology Extensions in OBDA Systems. In: Proceedings of the 10th International Workshop on OWL: Experiences and Directions (OWLED 2013), *CEUR Workshop Proceedings*, vol. 1080. CEUR-WS.org (2013)
7. Heim, P., Ertl, T., Ziegler, J.: Facet Graphs: Complex Semantic Querying Made Easy. In: Proceedings of the 7th Extended Semantic Web Conference (ESWC 2010), *LNCS*, vol. 6088, pp. 288–302. Springer (2010). DOI 10.1007/978-3-642-13486-9_20
8. Katifori, A., Halatsis, C., Lepouras, G., Vassilakis, C., Giannopoulou, E.G.: Ontology visualization methods - a survey. ACM Computing Surveys **39**(4) (2007). DOI 10.1145/1287620.1287621
9. Kharlamov, E., Hovland, D., Skjæveland, M.G., Bilidas, D., Jiménez-Ruiz, E., Xiao, G., Soylu, A., Lanti, D., Rezk, M., Zheleznyakov, D., Giese, M., Lie, H., Ioannidis, Y.E., Kotidis, Y., Koubarakis, M., Waaler, A.: Ontology Based Data Access in Statoil. Journal of Web Semantics **44**, 3–36 (2017). DOI 10.1016/j.websem.2017.05.005

10. Kharlamov, E., Mailis, T., Mehdi, G., Neuenstadt, C., Özçep, Ö.L., Roshchin, M., Solo-makhina, N., Soylu, A., Svingos, C., Brandt, S., Giese, M., Ioannidis, Y.E., Lamparter, S., Möller, R., Kotidis, Y., Waaler, A.: Semantic access to streaming and static data at Siemens. Journal of Web Semantics **44**, 54–74 (2017)

11. Klungre, V., Giese, M.: Evaluating a Faceted Search Index for Graph Data. In: Proceedings of the On the Move to Meaningful Internet Systems (OTM 2018), *LNCS*, vol. 11230, pp. 573–583 (2018). DOI 10.1007/978-3-030-02671-4_36

12. Klungre, V., Soylu, A., Giese, M., Waaler, A., Kharlamov, E.: On Enhancing Visual Query Building over KGs Using Query Logs. In: The Proceedings of the 8th Joint International Conference on Semantic Technology (JIST 2018), *LNCS*, vol. 11341, pp. 77–85. Springer (2018). DOI 10.1007/978-3-030-04284-4_6

13. Kogalovsky, M.R.: Ontology-Based Data Access Systems. Programming and Computer Software **38**(4), 167–182 (2012)

14. Krivov, S., Williams, R., Villa, F.: GrOWL: A Tool for Visualization and Editing of OWL Ontologies. Journal of Web Semantics **5**(2), 54–57 (2007). DOI 10.1016/j.websem.2007.03.005

15. Lohmann, S., Negru, S., Haag, F., Ertl, T.: Visualizing ontologies with VOWL. Semantic Web **7**(4), 399–419 (2016). DOI 10.3233/SW-150200

16. Motta, E., Mulholland, P., Peroni, S., d'Aquin, M., Gomez-Perez, J.M., Mendez, V., Zablith, F.: A Novel Approach to Visualizing and Navigating Ontologies. In: Proceedings of the 10th International Conference on The Semantic Web (ISWC 2011), *LNCS*, vol. 7031, pp. 470–486. Springer (2011). DOI 10.1007/978-3-642-25073-6_30

17. Nielsen, J.: Usability Engineering. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1993)

18. Pattyn, F., Vermaere, S., Van Huffel, P., Knecht, K., Constandt, H.: Semantic Linking and Integration of Researchers and Research Organizations in DISQOVER. In: Proceedings of the 9th International Conference Semantic Web Applications and Tools for Life Sciences (SWAT4LS 2016), *CEUR Workshop Proceedings*, vol. 1795. CEUR-WS.org (2016)

19. Picalausa, F., Vansummeren, S.: What Are Real SPARQL Queries Like? In: Proceedings of the International Workshop on Semantic Web Information Management (SWIM 2011), pp. 7:1–7:6. ACM (2011)

20. Sarker, M.K., Krisnadhi, A.A., Hitzler, P.: OWLAx: A Protege Plugin to Support Ontology Axiomatization through Diagramming. In: Proceedings of the Posters & Demonstrations Track co-located with 15th International Semantic Web Conference (ISWC 2016), *CEUR Workshop Proceedings*, vol. 1690. CEUR-WS.org (2016)

21. Smart, P.R., Russell, A., Braines, D., Kalfoglou, Y., Bao, J., Shadbolt, N.R.: A Visual Approach to Semantic Query Design Using a Web-Based Graphical Query Designer. In: Proceedings of the 16th International Conference on Knowledge Engineering and Knowledge Management (EKAW 2008), *LNAI*, vol. 5268, pp. 275–291. Springer (2008). DOI 10.1007/978-3-540-87696-0_25

22. Soylu, A., Giese, M., Jiménez-Ruiz, E., Kharlamov, E., Zheleznyakov, D., Horrocks, I.: Towards Exploiting Query History for Adaptive Ontology-based Visual Query Formulation. In: Proceedings of the 8th Metadata and Semantics Research Conference (MTSR 2014), *CCIS*, vol. 478, pp. 107–119. Springer (2014)

23. Soylu, A., Giese, M., Jimenez-Ruiz, E., Kharlamov, E., Zheleznyakov, D., Horrocks, I.: Ontology-based end-user visual query formulation: Why, what, who, how, and which? Universal Access in the Information Society **16**(2), 435–467 (2017). DOI 10.1007/s10209-016-0465-0

24. Soylu, A., Giese, M., Jimenez-Ruiz, E., Vega-Gorgojo, G., Horrocks, I.: Experiencing OptiqueVQS: a multi-paradigm and ontology-based visual query system for end users. Universal Access in the Information Society **15**(1), 129–152 (2016). DOI 10.1007/s10209-015-0404-5

25. Soylu, A., Giese, M., Schlatte, R., Jiménez-Ruiz, E., Kharlamov, E., Özçep, Ö.L., Neuenstadt, C., Brandt, S.: Querying industrial stream-temporal data: An ontology-based visual approach. Journal of Ambient Intelligence and Smart Environments **9**(1), 77–95 (2017). DOI 10.3233/AIS-160415

26. Soylu, A., Kharlamov, E.: Making Complex Ontologies End User Accessible via Ontology Projections. In: Proceedings of the 8th Joint International Conference on Semantic Technology (JIST 2018), *LNCS*, vol. 11341, pp. 295–303. Springer (2018). DOI 10.1007/978-3-030-04284-4_20

27. Soylu, A., Kharlamov, E., Zheleznyakov, D., Jimenez Ruiz, E., Giese, M., Skjaeveland, M.G., Hovland, D., Schlatte, R., Brandt, S., Lie, H., Horrocks, I.: OptiqueVQS: a Visual Query System over Ontologies for Industry. Semantic Web **9**(5), 627–660 (2018). DOI 10.3233/SW-180293
28. Soylu, A., Modritscher, F., De Causmaecker, P.: Ubiquitous web navigation through harvesting embedded semantic data: A mobile scenario. Integrated Computer-Aided Engineering **19**(1), 93–109 (2012)
29. Spanos, D.E., Stavrou, P., Mitrou, N.: Bringing Relational Databases into the Semantic Web: A Survey. Semantic Web **3**(2), 169–209 (2012)
30. Tunkelang, D.: Faceted search. Synthesis lectures on information concepts, retrieval, and services. Morgan & Claypool Publishers (2009)
31. Vega-Gorgojo, G., Giese, M., Heggestøyl, S., Soylu, A., Waaler, A.: PepeSearch: Semantic Data for the Masses. PLoS ONE **11**(3) (2016). DOI 10.1371/journal.pone.0151573