Approximating Faceted Search for Graph Queries

Vidar Klungre and Martin Giese

University of Oslo

Abstract. We discuss the problem of implementing real-time faceted search interfaces over graph data, specifically the "value suggestion problem" of presenting the user with options that makes sense in the context of a partially constructed query. For queries that include many object properties, this task is computationally expensive. We show that good approximations to the value suggestion problem can be achieved by only looking at parts of queries, and we present an index structure that supports this approximation and is designed to scale gracefully to both very large datasets and complex queries. In a series of experiments, we show that the loss of accuracy is usually minor.

Keywords: Faceted Search · Visual Query Interface · Index Structure

1 Introduction

Faceted search [9] is a popular search and exploration paradigm which allows users to apply search filters to multiple orthogonal dimensions (facets) of the data. Filters can be added, removed or modified in any order, and every time this is done, the system immediately updates the list of results, giving the user instant feedback. To support this functionality, the system needs fast access to the underlying data. This is often provided by specialised software which provides better performance for the queries required by faceted search than standard triple stores and RDB-based implementations.

Faceted search has also been extended to semantics-based data, e.g. in SemFacet [1] and Rhizomer [2]. In these systems, datatype properties are treated as facets, but they also include some ability to query the graph structure via object properties. Combining faceted search with queries for graph patterns (called "graph queries") results in more expressive queries, but implementing faceted search also becomes computationally harder. The challenging task is to update the set of available values for the facets after each user interaction. The straightforward way of computing this set involves evaluating a query of similar complexity to the whole query built so far, and that needs to be done for each facet. For queries with large graph patterns, and large datasets, this can become too time consuming for an interactive system. The usual approach of using a search engine style index will not help, since these engines do not support graph queries.

Based on the visual query system OptiqueVQS [7], we have devised a system that combines faceted search with graph queries, and that uses an indexing structure for suggesting facet values that can easily be scaled out arbitrarily. In return, it compromises some accuracy in computing sets of available facet values, but in a highly configurable manner.[1]

In the present paper, we first discuss the combination of faceted search and graph queries, and introduce the concept of *adaptive value suggestion.* (Sect. 2). In Sect. 3 we summarise our formal framework and formally introduce the *value suggestion problem* that is the problem we attempt to solve. Several attempts at solving this problem are described in Sect. 4: one is optimal from the user's perspective but slow for large datasets and complex queries; the second is rather inaccurate, but allows fast implementation; the third is a configurable family of intermediate (good enough and fast enough) solutions to the problem, based on only looking at a part of the constructed query. In Sect. 6, we present a series of experiments that shows that

1. good approximations to the value suggestion problem can often be reached by taking into account only relatively small parts of the constructed query.
2. the accuracy of the approximations can often be improved dramatically by including only the presence of required object properties in the index.

2 Visual Query Systems and Faceted Search

2.1 VQSs

A *Visual query system* (VQS) presents a visual interface to users that allows them to extract information from a structured data source, based on some combination of filters and other requirements on the information to be retrieved. The intention is to provide data access to users without requiring them to learn a formal query language like e.g. SPARQL. VQSs need to find a balance between expressivity and usability: a system that covers the whole expressivity of SPARQL will hardly be more useful to lay users than a SPARQL editor. This trade-off differs depending amongst others on the user group, their information needs, and the complexity of the data [6]. 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 entities of different types (e.g. black shoes from a small company based in a democratic country). Examples of VQSs designed for underlying RDF data are Rhizomer [2], SemFacet [1], and OptiqueVQS [7]. We have used OptiqueVQS as the starting point for our work, which influenced some of our design choices.

[1] We first described the idea of this configurable index structure in a technical report [5] and presented our implementation in an ISWC demo [4].

2.2 Faceted Search

The term “Faceted Search” usually refers to search over datasets containing only one type of concept (like e.g. shoe), and where several orthogonal attributes (facets) are associated with each individual in the data. A typical faceted search interface has a pane on the left side or at the top, where all these facets are listed. For each facet, the interface allows the user to add or remove filters in any order. As this is done, the main part of the page updates immediately and displays a list of all instances satisfying the set of filters.

Many faceted search interfaces also display a list of *facet value suggestions* below each facet. If the user selects a value from this list, a new filter is added based on the corresponding facet and value. Displaying a good list of value suggestions is a challenging task: The user should not be overwhelmed by irrelevant values, but he should also find the values he expects to see and/or wants to filter on.

Simple faceted search systems will suggest a long and static list of values, containing all the different values appearing in the dataset. In this case, the user will always find the value he is looking for. However, it is not optimal, because the list will likely contain values that are incompatible with existing filters. By that we mean: selecting such a value will lead to a situation where the applied filters are too restrictive, and no results are returned. This kind of dead end is not desirable from a user experience perspective.

More advanced systems, on the other hand, remove or disable values (often indicated by a gray font color) that are not compatible with the existing filters – leaving a shorter list of suggestions to the user. We call this method *adaptive value suggestion*:

> **Adaptive Value Suggestion**: Calculate and suggest the complete set of values for a facet that are compatible with both the existing filters and underlying data, in order to not end up in a situation without any results.

Calculations needed to support adaptive value suggestion are quite intensive for large datasets. To solve this problem, it is common to use search engines like Lucene[2] or Sphinx[3] or similar software to index the data before use. These indices are known to scale to large datasets, e.g. by partitioning. This ensures fast response time, and no delay for the user.

2.3 Faceted Search for Graph Data

To accommodate complex information needs over structured data, VQSs need a way to support graph queries; in SPARQL parlance graph patterns with several variables connected by object properties. How this is supported varies between VQSs, but essentially, there will be some way of adding edges and nodes to a

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

graph query. Typically there is some menu of properties to choose from, either separate or merged with the facets. In an ontology-based system, this list may be generated from the ontology, from the data, or a combination of both.

It is desirable to offer users adaptive value suggestions as described in Sect. 2.2. Unfortunately, the usual indices used for faceted search, like Lucene, do not support querying graph data. The obvious way of achieving this over the original graph data (i.e. without an index) requires running the whole partial query once for every facet (see Sect. 4.1). For very large datasets and queries with many joins, this will be too slow. Some of the queries constructed with OptiqueVQS include up to 9 object properties, and are intended to be run over data stores of several PB. 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 custom-built solution is needed to calculate the adaptive value suggestions sufficiently fast.

It turns out that adaptive value suggestion is not possible to achieve for arbitrarily large graph data and complex queries sufficiently fast. Joining over large amounts of data is inherent in the problem. As we will see however, it is possible to compute *approximations* of the set of possible facet values, which may contain some irrelevant suggestions. The more irrelevant suggestions we tolerate, the faster we can compute them, so there is a trade-off between computation time and accuracy. Our proposal is a particular approximation that attempts to strike a good compromise between these two.

3 Formal Framework

For the purpose of this paper, we work with 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. See [4] for a more complete exposition including examples.

3.1 The Ontology and Navigation Graph

In some VQSs, there is a fixed ontology $\mathcal{O}$ that controls the behaviour of the system, while others analyse the data. In either case, the observed behaviour will be that based on the type of a variable in the query,

- some selection of datatype properties is available for filtering;
- some selection of object properties is available for extending the query graph;
- if the graph is extended with an object property triple, there will be some information about the type of the newly introduced query variable.

Instead of considering an ontology, we therefore work with a *navigation graph*, a directed graph $\mathcal{N}_{\mathcal{O}}$, where each node represents either a concept or datatype in $\mathcal{O}$, and where each edge represents a (object or datatype) property in $\mathcal{O}$.

Exactly how this navigation graph is built from the data and/or ontology differs between VQSs. The navigation graph used in OptiqueVQS is described in [8], Sect. 5.2.1.

3.2 Queries

When we refer to queries in this paper, we mean tree-shaped conjunctive queries where each variable is associated with a node in $\mathcal{N}_{\mathcal{O}}$, that specifies the type of the query variable. Furthermore, we assume that this typing is homomorphic, which means that the query obeys the type structure defined by $\mathcal{N}_{\mathcal{O}}$.

Queries can then be represented as trees, where each node represents a query variable, and the edges represent properties. We separate the query variables into two separate groups based on whether they are typed to a concept or a datatype in $\mathcal{N}_{\mathcal{O}}$. We call them concept variables and datatype variables respectively.

For each datatype variable in the query the user is allowed to add a filter, specifying which values it is allowed to take. The filter is specified by the filter function $\mathcal{F}$ which returns a set of values for each query datatype variable in Q. We do not include an "optional" operator, i.e. all variables of Q have to be bound.

3.3 Datasets and Query Answers

In addition to the ontology $\mathcal{O}$ and the corresponding navigation graph $\mathcal{N}_{\mathcal{O}}$, 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. When the query is finished, the user will be running it over $\mathcal{D}$ in order to retrieve the results of interest. However, the goal of this work is to use utilize the data in $\mathcal{D}$ during query construction as well, in order to provide adaptive value suggestions.

We will use $Ans(Q(\vec{x}), \mathcal{D})$ to denote the result we get by executing the query $Q(\vec{x})$ over a dataset $\mathcal{D}$. The result is a set of tuples, where the entries in each tuple corresponds to the variables in $\vec{x}$.

3.4 Focus Variable

Many VQSs have a concept of *focus variable*, which is the variable current facet filtering applies to, and the variable that will be the subject of new object property triples added to the query. It is usually possible to "refocus" during query construction, i.e. change to a different focus variable.

During a query session, both the constructed query and the focus variable changes frequently, but for any given state there is exactly one partial query, and a corresponding focus variable. Since the queries are tree-shaped, we assume that the focus variable is the root of this tree, possibly reversing the direction of some triples. For the reminder of this paper, we will use Q to denote the partial query with root (and focus variable) v, and for convenience we also let $\mathcal{C}$ denote the type of v.

3.5 Value Suggestion Problem

During query construction, the user is presented with a list of suggestions for every local datatype property t[4]. These suggestions are based on the current state of the session, i.e Q, in addition to the the underlying dataset $\mathcal{D}$. We formalize this by presenting what we call *suggestion functions*:

Definition 1. ***Suggestion function****: A suggestion function is a function $\mathcal{S}$ which takes a dataset $\mathcal{D}$, a query Q, and some datatype property t as input, and which returns a set of literal values:*

$$Sugg = \mathcal{S}(\mathcal{D}, Q, t).$$

By selecting values $X \subseteq Sugg$, the user modifies the filters related to t, i.e. Q is updated to $Q \wedge t(v, w)$, and $\mathcal{F}(w) = X$.

Notice that the definition above does not restrict $\mathcal{S}$ on neither its computation time nor the quality of suggested values. However, it should be clear by now that we are looking for functions that return adaptive value suggestions (described in Sect. 2.2) without spending so much time that it ruins the user experience of the VQS.

The problem we target in this work is the following:

Value suggestion problem Find a suggestion function $\mathcal{S}$ that

1. can be computed efficiently enough for interactive use, even for large $\mathcal{D}$ and complex Q
2. includes all values, that can be filtered on without making the answer set empty, i.e.

$$Ans(Q \wedge t(v, x), \mathcal{D}) \neq \emptyset \Longrightarrow x \in \mathcal{S}(\mathcal{D}, Q, t) \qquad \text{for all values } s \text{ in } \mathcal{D}$$

3. includes as few values as possible that will make the answer set empty, i.e. $\mathcal{S}(\mathcal{D}, Q, t)$ is as small as possible while satisfying condition 2.

Condition 1 is necessary because suggestions have to be calculated after every user interaction with the UI, and the user should not have to wait for the relevant suggestions. So they have to be calculated efficiently, and scale with respect to both $\mathcal{D}$ and Q.

The second condition formalizes the idea that all values that are compatible with the partial query should also be suggested to the user. Otherwise, some sensible queries could not be constructed. We say that the suggestion function should have *perfect recall*.

Finally, condition 3 reflects that we want to suggest as few options as possible to the user that are incompatible with the partial query, i.e. that would lead to an empty result set. We say that the suggestion function should be as *precise* as possible.

[4] A local datatype property is a datatype property related to the focus variable.

These three conditions are in conflict, in the sense that a more precise suggestion function with perfect recall will take more resources to compute. We consider perfect recall to be non-negotiable, so the problem amounts to finding a good compromise between accuracy and efficiency. In the next section we present three different suggestions functions. None of them perfectly satisfies all three conditions in the value value suggestion problem, but the family of approximate suggestion functions $\mathcal{S}_a$ are such compromises.

4 Suggestion Functions

We now introduce three different suggestion functions: The first ($\mathcal{S}_o$) is optimal from the user's perspective but slow for large datasets and complex queries; the second ($\mathcal{S}_r$) is rather inaccurate, but allows fast implementation; the third ($\mathcal{S}_a$) is a configurable family of intermediate (good enough and fast enough) solutions to the problem, where the idea is to only consider a limited part of the query Q.

4.1 Optimal Suggestion Function $\mathcal{S}_o$

Based on standard faceted search systems, we will now define what we consider to be the gold standard for the value suggestion problem (with respect to accuracy), namely the *optimal suggestion function* $\mathcal{S}_o$:

$$\mathcal{S}_o(\mathcal{D}, Q, t) = Ans(Q_o(x), \mathcal{D}) \text{ where } Q_o(x) = Q \wedge t(v, x).$$

It considers both the underlying dataset $\mathcal{D}$ and the partial query Q, and calculates suggestions that never lead the user into a combination of filters that are too restrictive, i.e. it returns adaptive value suggestions (see Sect. 2.2). Unfortunately, $\mathcal{S}_o$ does not scale for large Q and $\mathcal{D}$, because it has to calculate the answers to the query $Q_o(x) = Q \wedge t(v, x)$, which is more complex than Q itself.

Because $\mathcal{S}_o$ is not guaranteed to satisfy condition 1 of the value suggestion problem, it is not technically a solution. However, it is important because it satisfies both condition 2 and 3 perfectly. Furthermore, since it returns the optimal set of values, we will use it to define accuracy for the evaluation in Sect. 6.

4.2 Range-Based Suggestion Function $\mathcal{S}_r$

Another important suggestion function is the function that computes the suggestion set based only on the value range of t for instances of type $\mathcal{C}$ found in $\mathcal{D}$. We call this function the *range-based suggestion function* $\mathcal{S}_r$:

$$\mathcal{S}_r(\mathcal{D}, Q, t) = Ans(Q_r(x), \mathcal{D}) \text{ where } Q_r(x) = \mathcal{C}(v) \wedge t(v, x).$$

Notice that $\mathcal{C}$ and t are the only two parameters used by $\mathcal{S}_r$ that change during the query session: $\mathcal{D}$ is fixed during the session, and except for the focus concept $\mathcal{C}$, Q is just ignored. This means that we can calculate the suggestion set for each possible combination of $\mathcal{C}$ and t offline, and providing suggestions is then just a matter of fetching the correct pre-calculated set. The number of such combinations is limited, and Q_r is a very simple query, so any system based on $\mathcal{S}_r$ is efficient with respect to both time and space.

Since $Q \subseteq \mathcal{C}(v)$ holds for all possible Q[5], we know that $Q_o \subseteq Q_r$, and therefore $\mathcal{S}_o \subseteq \mathcal{S}_r$. In other words, $\mathcal{S}_r$ will always suggest at least all the values from $\mathcal{S}_o$, but it may also include more. This also makes sense intuitively, since $\mathcal{S}_o$ considers both $\mathcal{D}$ and Q, while $\mathcal{S}_r$ only considers $\mathcal{D}$.

We will show in Sect. 6 that this suggestion function often gives many irrelevant suggestions, that will lead to empty query results.

4.3 Approximate Suggestion Function $\mathcal{S}_a$

The optimal suggestion function $\mathcal{S}_o$ is too costly to compute in practice, while the range-based function, on the other hand, is quite inaccurate but can be pre-computed with reasonable effort. There is a gap between these two suggestion functions, and the following solution, the family of approximate suggestion functions $\mathcal{S}_a$, fills this gap.

Each member $\mathcal{S}_a^{\mathcal{Z}}$ of $\mathcal{S}_a$ is configured by what we call a *facet configuration* $\mathcal{Z}$, which is a function that returns one tree-shaped query (with root of type $\mathcal{C}$) $\mathcal{Z}_{\mathcal{C}}$ for every concept $\mathcal{C}$. We call $\mathcal{Z}_{\mathcal{C}}$ the *concept configuration* of $\mathcal{C}$.

Now, given Q and the corresponding focus concept $\mathcal{C}$, $\mathcal{S}_a^{\mathcal{Z}}$ computes a pruned version of Q: $Q_{pr} = \mathit{intersection}(Q, \mathcal{Z}_{\mathcal{C}})$. Q_{pr} is then used together with the underlying dataset $\mathcal{D}$ to calculate value suggestions, i.e.:

$$\mathcal{S}_a^{\mathcal{Z}}(\mathcal{D}, Q, t) = \mathit{Ans}(Q_a(x), \mathcal{D}) \text{ where}$$
$$Q_a(x) = Q_{pr} \wedge t(v, x) = \mathit{intersect}(Q, \mathcal{Z}_{\mathcal{C}}) \wedge t(v, x).$$

Intuitively the concept configuration $\mathcal{Z}_{\mathcal{C}}$ just defines which parts of the partial query to consider when calculating suggestions, and which parts to ignore. Every part of Q which is not covered[6] by $\mathcal{Z}_{\mathcal{C}}$ is simply ignored, and removed as the intersection between Q and $\mathcal{Z}_{\mathcal{C}}$ is calculated.

The most aggressive member of $\mathcal{S}_a$ is the suggestion function defined by concept configurations that only contains the root, i.e. $\mathcal{Z}_{\mathcal{C}} = \mathcal{C}(v)$ for all concepts $\mathcal{C}$. This member will actually return $Q_{pr} = \mathcal{C}(v)$ for any partial query given to it, regardless of its shape, size and focus concept. But this means that $Q_a(x) = \mathcal{C}(v) \wedge t(v, x) = Q_r(x)$, so this particular member of $\mathcal{S}_a$ returns exactly the same suggestions as $\mathcal{S}_r$. And in fact, the range-based suggestion function is just the special case of $\mathcal{S}_a$ where only the root of the partial query is considered, and everything else is ignored.

Now let us consider the opposite case: instances of $\mathcal{S}_a$ with very large concept configurations. If the partial query is completely covered by the tree defined

[5] Here, and in the following, $\subseteq$ between queries denotes query containment, defined like this: $Q_1 \subseteq Q_2$ if $\mathit{Ans}(Q_1, \mathcal{D}) \subseteq \mathit{Ans}(Q_2, \mathcal{D})$

[6] When writing that a query Q_1 covers another query Q_2, we mean $Q_2 \subseteq Q_1$. The intuitive interpretation of this is that the tree defined by Q_2 is smaller and included in the tree defined by Q_1.

by the concept configuration i.e. $\mathcal{Z}_\mathcal{C} \subseteq Q$, we get $Q_{pr} = Q$. Hence $Q_a(x) = Q \wedge t(v, x) = Q_o(x)$, which shows that $\mathcal{S}_o$ is the limit case of $\mathcal{S}_a$ for configurations that cover all possible queries.

In general, for every partial query Q and facet configuration $\mathcal{Z}$, the following holds:

$$Q \subseteq Q_{pr} \subseteq \mathcal{C}(v) \Rightarrow Q_o \subseteq Q_a \subseteq Q_r \Rightarrow \mathcal{S}_o \subseteq \mathcal{S}_a^{\mathcal{Z}} \subseteq \mathcal{S}_r. \qquad (1)$$

As we focus on a certain query, we can ignore large parts of the facet configuration $\mathcal{Z}$, since only one concept configuration $\mathcal{Z}_\mathcal{C}$ is needed to calculate value suggestions. In some cases, we will therefore use $\mathcal{S}_a^{\mathcal{Z}_\mathcal{C}}$ instead of $\mathcal{S}_a^{\mathcal{Z}}$, as short hand notation. Similarly, we may only use the word "configuration" if it is clear whether we mean "facet configuration" or "concept configuration".

5 Index Structure for $\mathcal{S}_a$

As seen above, $\mathcal{S}_a$ reduces the complexity of calculating suggestions by only considering Q_{pr} instead of Q. This will often reduce the query execution time, but it is not guaranteed to be good enough for our purpose: If Q_{pr} includes multiple different concepts, the UI response time will suffer due to the time consuming join operations it requires.

The solution to this problem is to pre-compute all joins covered by the facet configuration $\mathcal{Z}$, and store the results in an index structure. The system can then execute Q_a over this index structure instead of the original dataset $\mathcal{D}$, in order to retrieve answers fast enough. It is important though, that the final constructed query is executed over the original dataset. $\mathcal{S}_a$ and its index should only be used to support adaptive value suggestion, not to answer the user's final information need.

The index is guaranteed to contain all the data needed to answer Q_a since they both are limited by the variables defined by $\mathcal{Z}_\mathcal{C}$. Notice that constructing such an index would not be possible if we wanted a perfect system described by $\mathcal{S}_o$ – it is impossible to construct an index that fits all the data needed to cover any possible query, because there are infinitely many of them. By using $\mathcal{S}_a$, we only need to consider Q_{pr}, which is limited by $\mathcal{Z}_\mathcal{C}$, hence pre-computing and indexing is possible.

We now describe how to construct and use the index. It will consist of multiple tables – one for each concept $\mathcal{C}$. Each table is based on the corresponding concept configuration $\mathcal{Z}_\mathcal{C}$, and is constructed as follows:

1. One column is added for each variable in the query defined by $\mathcal{Z}_\mathcal{C}$.
2. One row is added for each *distinct* tuple of $\text{Ans}(\mathcal{Z}'_\mathcal{C}, \mathcal{D})$, where $\mathcal{Z}'_\mathcal{C}$ is a modified version of $\mathcal{Z}_\mathcal{C}$, where everything except for the root node is made optional.

The result is a large denormalized table containing all the data that is covered by $\mathcal{Z}_\mathcal{C}$. By using the optional version $\mathcal{Z}'_\mathcal{C}$ instead of $\mathcal{Z}_\mathcal{C}$ directly, we ensure that we also get the data that is just partly covered by $\mathcal{Z}_\mathcal{C}$.

Answering Q_a over this table is then just a simple table scan, and the query response time can be reduced to a satisfactory level by indexing the columns and/or parallellizing the storage and processing, similar to what state of the art search engines do. Such scaling out is much easier for a single pre-joined table than for relational or graph storage. But it requires the pruning defined by a fixed configuration, which is precisely the point of our approximate suggestion functions.

With data stored denormalized, we essentially have the same situation as for standard faceted search with only one concept: We just act like every column (i.e. variable of the configuration) is a facet. This means that we can achieve adaptive value suggestions (over the variables included in the configuration) with the same performance as standard faceted search systems by simply using the same underlying search engine technology.

Earlier we stated that $\mathcal{Z}_\mathcal{C} = \mathcal{C}(v)$ is the smallest possible concept configuration one can use for $\mathcal{S}_a$. This is true if we use the original dataset, but not if we use the index, because we need to answer $\mathcal{C}(v) \wedge t(v, x)$ for each datatype property t we want suggestions for, which cannot be done if a data column for t does not exist. We want to provide suggestions for each local datatype property $t \in T$, hence $\mathcal{Z}_\mathcal{C} = \mathcal{C}(v) \wedge \bigwedge_{t_i \in T} t_i(v, x_i)$ is the smallest configuration we allow.

5.1 Existential Concept Variables

With the index construction method described above, we get one column containing only URIs for each concept variable included in the concept configuration. This data is often just a waste of space in our context: 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, and that is why we introduce existential columns for concept variables.

Instead of storing the full URI, we use a boolean value to indicate whether an instance 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 the experiment we conducted, we explored how much the accuracy increase by adding another layer of these existential concepts nodes to the index, which is a comparatively cheap investment.

6 Evaluation

We have implemented a faceted search module for OptiqueVQS based on $\mathcal{S}_a$ and the index structure described in Sect. 5. Furthermore, we implemented both $\mathcal{S}_r$ and $\mathcal{S}_o$ in order to compare them to $\mathcal{S}_a$ in our experiment.

In Sect. 5 we argued that our system is at least as efficient (w.r.t.index access) as state of the art faceted search engines using only one concept, so we have not spent any effort on measuring the time our system uses. We have also not measured the performance of the index construction process, since it is not as time crucial as index access. In other words, we do not claim that our implementation is suited for systems that require real-time update. Instead, we explored how facet configurations of different size and shape affect the constructed index and the corresponding accuracy of value suggestions.

The goal of our experiment was to answer the following two questions about $\mathcal{S}_a$:

1. How does the accuracy of $\mathcal{S}_a$ increase as the size of $\mathcal{Z}_\mathcal{C}$ increases?
2. How much does the accuracy of $\mathcal{S}_a$ increase by adding existential concept variables to $\mathcal{Z}_\mathcal{C}$?

Section 6.1 gives a detailed description of how the experiment was done, while the results can be found in section 6.2.

6.1 Experiment setup

Dataset, Ontology and Queries
In the Optique project, user requirements led to queries with up to 9 object properties [3], but over proprietary data, so we could not use those queries directly for our experiment. Instead, we used the RDF version of the NPD Factpages[7] – 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 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 catalogue distributed with this dataset was not suited for our experiment: Just a few of the queries had the structure our system required, and none of them connected more than a few concepts together. So we constructed a new query catalogue consisting of complex queries of a more suitable size, with the goal to cover a wide set of possible cases. The catalogue 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.

The complete query catalogue is publicly available on Github[8].

Accuracy Measure
Providing the value suggestions is an information retrieval problem, where $\mathcal{S}_o$ defines the set of relevant values. We therefore use the well established measures

[7] https://gitlab.com/logid/npd-factpages
[8] https://github.com/Alopex8064/npd-factpages-experiments

of precision and recall to measure the accuracy of $\mathcal{S}_a$ (and $\mathcal{S}_r$). From Eq. 1 we know that $\mathcal{S}_o \cap \mathcal{S}_a = \mathcal{S}_o$ and $\mathcal{S}_o \cap \mathcal{S}_r = \mathcal{S}_o$ which gives:

$$pre(\mathcal{S}_a) = \frac{|\mathcal{S}_o \cap \mathcal{S}_a|}{|\mathcal{S}_a|} = \frac{|\mathcal{S}_o|}{|\mathcal{S}_a|} \qquad pre(\mathcal{S}_r) = \frac{|\mathcal{S}_o \cap \mathcal{S}_r|}{|\mathcal{S}_r|} = \frac{|\mathcal{S}_o|}{|\mathcal{S}_r|}$$
$$rec(\mathcal{S}_a) = \frac{|\mathcal{S}_o \cap \mathcal{S}_a|}{|\mathcal{S}_o|} = \frac{|\mathcal{S}_o|}{|\mathcal{S}_o|} = 1 \quad rec(\mathcal{S}_r) = \frac{|\mathcal{S}_o \cap \mathcal{S}_r|}{|\mathcal{S}_o|} = \frac{|\mathcal{S}_o|}{|\mathcal{S}_o|} = 1.$$

where $\mathcal{D}$, Q and t are all fixed (hence not included in the formulas).

Both $\mathcal{S}_a$ and $\mathcal{S}_r$ have perfect recall, which is expected from Eq. 1 and the fact that it was the most important condition to satisfy from the value suggestion problem. Hence, when evaluating these systems, only the precision matters, so in the reminder of this paper, we will use and mention precision instead of accuracy. Furthermore, since the user is exposed to several local datatype properties at the same time, and we want to do more high-level experiments on the system, we average the precision over all the local datatype properties.

$$pre(\mathcal{S}_a) = \frac{1}{|T|}\sum_{t \in T} pre(\mathcal{S}_a, t) \qquad pre(\mathcal{S}_r) = \frac{1}{|T|}\sum_{t \in T} pre(\mathcal{S}_r, t)$$

So given Q, $\mathcal{D}$, $\mathcal{Z}_\mathcal{C}$, we will use this average as the overall measure of precision. From Eq. 1, we can derive the following relationship between the precision of our three suggestion functions:

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

Test Cases and General Setup
In the experiment we ran multiple test cases, where each test case was based on one of the 29 queries from the query catalogue, and a generated concept configuration $\mathcal{Z}_\mathcal{C}$ covered by the tree defined by this query. Since each test case only considers one query and one concept configuration, the index we get based on the configuration will only contain one table. Value suggestions are then calculated by running Q_a over the table, and precision are calculated by comparing to $\mathcal{S}_o$ as described above.

Notice that a real world scenario would be more complex than this. The success of a concept configuration and its corresponding table index would not only depend on the success 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.

Configuration Generation

In our experiment we wanted to 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 "configuration cores" c for each query Q in the query catalogue. Each core consisted of one or more connected concept variables from Q, and was just used as a basis for generating two other concept configurations:

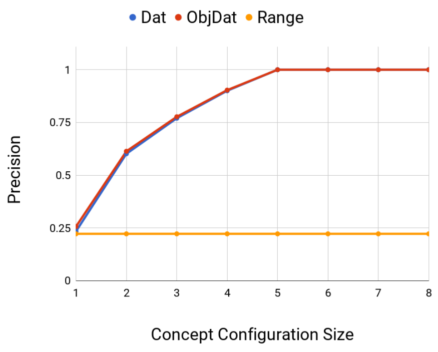

Fig. 1: Results for Query 2.6

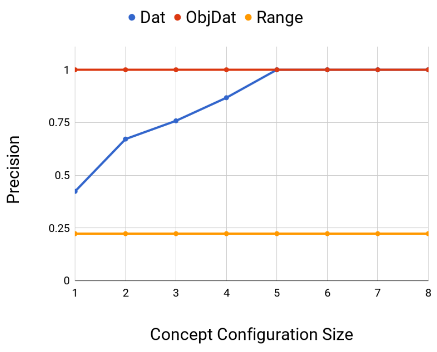

Fig. 2: Results for Query 2.8

- $Dat(c)$: 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. 5.1, 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, and thereby answering question 2.

Both $Dat(c)$ 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:

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

After running through every test case, the results was 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 visualized.

6.2 Results and Analysis

Below are several charts visualizing the results of our experiment. Fig. 1, 2 and 3 display charts for three selected individual queries, while Fig. 4 shows the average precision for all the queries of size 6 (15 of the 29 queries). Similar charts for queries of size 5, 7 and 8 are omitted from the paper, but can be found on Github[9] together with charts for every individual query used in the experiment.

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

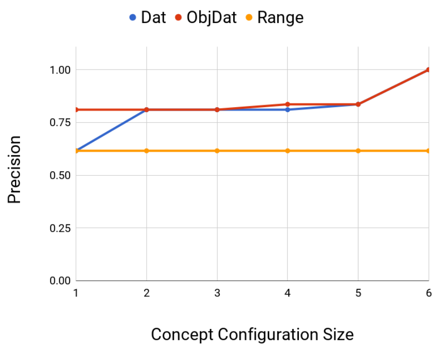

Fig. 3: Results for Query 3.5

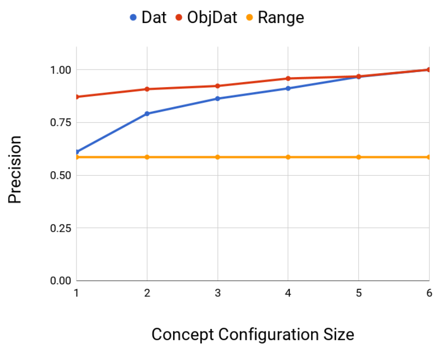

Fig. 4: Average precision of size 6 queries.

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 $pre(\mathcal{S}_a^{Dat}) \leq pre(\mathcal{S}_a^{ObjDat})$ for all configuration sizes.

The precision given by each of the three curves depends on how many of the important key restrictions of Q they are able to capture. A key restrictions is a restriction which reduces the number of instances one could assign to the root so much that it also causes a large reduction in the possible facet values. The query used in Fig. 1 for example has only one key restriction on the data property *name* of the *Field* concept variable in depth 2 of Q. 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 equals 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.

The query used in Fig. 2 has two key restrictions. The first restriction is associated with a datatype property filter on the root node (wellboreTemperature ≥ 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 Fig. 1.

The query used in Fig. 3 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 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, it will in fact always perform poorly if one or more such key restrictions exists. Fig. 1 and 2 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 its average precision of 0.58, displayed in Fig. 4.

The chart in Fig. 4 displays the average over all queries of size 6 used in the experiment, so it acts as a summary over all the queries.

The first thing to notice 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.

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 Fig. 2. The average difference between $\mathcal{S}_a^{ObjDat}$ and $\mathcal{S}_a^{Dat}$, shown in Fig. 4, 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).

7 Conclusion

We discussed the combination of visual query systems for graph queries with the adaptive value suggestions of faceted search. After defining the "value suggestion problem", we introduced three suggestion functions: an optimal one that is slow for large data sets 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

1. good approximations to the value suggestion problem 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.

In future work we intend to study alternative storage formats for the prejoined 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. Also suitable bucketing strategies can be determined from the query log and data.

References

1. M. Arenas, B. C. Grau, E. Kharlamov, Š. Marciuška, and D. Zheleznyakov. Faceted search over rdf-based knowledge graphs. *Web Semantics: Science, Services and Agents on the World Wide Web*, 37:55–74, 2016.
2. J. M. Brunetti, R. García, and S. Auer. From overview to facets and pivoting for interactive exploration of semantic web data. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 9(1):1–20, 2013.
3. E. Kharlamov, D. Hovland, M. G. Skjæveland, D. Bilidas, E. Jiménez-Ruiz, G. Xiao, A. Soylu, D. Lanti, M. Rezk, D. Zheleznyakov, M. Giese, H. Lie, Y. Ioannidis, Y. Kotidis, M. Koubarakis, and A. Waaler. Ontology based data access in statoil. *Web Semantics: Science, Services and Agents on the World Wide Web*, 44:3 – 36, 2017. Industry and In-use Applications of Semantic Technologies.
4. V. N. Klungre. A faceted search index for graph queries. Technical Report 469, Univ. of Oslo, Dept. of Informatics, 2017.
5. V. N. Klungre and M. Giese. A faceted search index for OptiqueVQS. In N. Nikitina, D. Song, A. Fokoue, and P. Haase, editors, *Proceedings of the ISWC 2017 Posters & Demonstrations and Industry Tracks co-located with 16th International Semantic Web Conference (ISWC 2017), Vienna, Austria, October 23rd - to - 25th, 2017.*, volume 1963 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2017.
6. A. Soylu, M. Giese, E. Jimenez-Ruiz, E. Kharlamov, D. Zheleznyakov, and I. Horrocks. Ontology-based end-user visual query formulation: Why, what, who, how, and which? *Universal Access in the Information Society*, 16(2):435–467, 2017.
7. A. Soylu, M. Giese, E. Jimenez-Ruiz, G. Vega-Gorgojo, and I. Horrocks. 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.
8. A. Soylu, E. Kharlamov, D. Zheleznyakov, E. Jimenez Ruiz, M. Giese, M. G. Skjaeveland, D. Hovland, R. Schlatte, S. Brandt, H. Lie, and I. Horrocks. OptiqueVQS: a visual query system over ontologies for industry. *Semantic Web*, (in press), 2017.
9. D. Tunkelang. Faceted search. *Synthesis lectures on information concepts, retrieval, and services*, 1(1):1–80, 2009.