Evaluating a Faceted Search Index for Graph Data

Vidar Klungre(✉) and Martin Giese

University of Oslo, Oslo, Norway
vidarkl@ifi.uio.no

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 often minor, and additional accuracy can in many cases be achieved with a modest increase of index size.

1 Introduction

Faceted search [7] is a popular search and exploration paradigm (used by e.g. Ebay), which enables users to extract information from structured data sources without needing to know the relevant formal query language. Systems providing faceted search present multiple orthogonal dimensions (facets) of the data to the user, and allows him to apply or remove filters via an intuitive UI. As this is done, the system immediately updates the lists of results and new filter suggestions. To support this functionality, the system needs fast access to the underlying data. This is often provided by specialised software like e.g Lucene, Sphinx or Elasticsearch, which provides better performance for the queries required by faceted search than standard triple stores and RDB-based implementations.

In this paper we focus on ontology-based *Visual query systems* (VQSs) like Rhizomer [2], SemFacet [1], and OptiqueVQS [5]. The purpose of these systems is to allow non-experts to construct SPARQL queries and execute them over RDF-graphs. A good overview of VQSs and their target audience is described in [6]. Most VQSs provide an intuitive UI, and allow the user to apply or remove filters on different facets, similar to standard faceted search systems. For each variable in the SPARQL query, every datatype property is considered to be a facet. Furthermore, object properties are used to connect the variables, and they allow the user to construct complex *graph queries*. This is a difference from standard faceted search, where only one variable of exactly one type is considered.

In particular we look at one specific feature of faceted search: The ability to suggest reasonable filter values for each facet. One way of doing this, is to

© Springer Nature Switzerland AG 2018
H. Panetto et al. (Eds.): OTM 2018 Conferences, LNCS 11230, pp. 573–583, 2018.
https://doi.org/10.1007/978-3-030-02671-4_36

pre-compute and present all the values from the RDF-graph that are related to the given datatype property. This method is very straightforward, and since it does not depend on the current state of the query, all computations can be done outside the query session starts. However, some of these values may feel superfluous to the user, because they are incompatible with existing filters. So instead we aim for what we call *adaptive value suggestions*:

> **Adaptive Value Suggestion:** Calculate and suggest the complete set of filter values for a facet that are compatible with both the existing filters and underlying data, in order to avoid a query that returns no results.

Unfortunately, the indices used to acheive adaptive value suggestions 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 $\mathcal{S}_o$ in Sect. 2). 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 achieve our goal sufficiently fast.

Based on the visual query system OptiqueVQS [5], 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 the presented values, but in a highly configurable manner.

Formal Framework. For the purpose of this paper, we work with a number of simplified notions of 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 [3] and [4] for a more complete description and examples.

We assume that the VQS supports tree-shaped conjunctive queries Q that conforms to the given ontology $\mathcal{O}$. In addition each variable must be associated with either a concept in $\mathcal{O}$ (concept variables), or a data type (datatype variables). Filters are specified by a filter function $\mathcal{F}$ that returns a set of values for each datatype variable in Q. We do not include an "optional" operator, i.e. all variables of Q have to be bound. Furthermore, we assume that the system is given a dataset (RDF graph) $\mathcal{D}$, and when the query is finished, he will be running it over $\mathcal{D}$ in order to retrieve results. We use $Ans(Q, \mathcal{D})$ to denote the set of tuples we get by running Q over $\mathcal{D}$. We also assume that the user can select and focus on one specific variable v of Q, called the *focus variable*. It is convenient to define Q as a tree rooted in v (possibly reversing the direction of some triples), and for the reminder of this paper we assume this is the case. And for convenience we also let $\mathcal{C}$ denote the type of v.

During query construction, the user is presented with a list of suggestions for every relevant property t of v. These suggestions are based on the current state of the session, i.e Q, in addition to the underlying dataset $\mathcal{D}$. Before we continue, we need to formalize the idea of a *suggestion function*:

Definition 1. ***Suggestion function:*** *A suggestion function $\mathcal{S}$ takes as input a dataset $\mathcal{D}$, a query Q, and some datatype property t linked to the focus variable v of Q, and returns a set of literal values $Sugg = \mathcal{S}(\mathcal{D}, Q, t)$.*

By selecting values $X \subseteq Sugg$, the user modifies the filter related to w, where w is the datatype variable linked to v via t. I.e. Q is updated to $Q \wedge t(v, w)$, and $\mathcal{F}(w) = X$. Notice that the Definition 1 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 without spending so much time that it ruins the user experience of the VQS. In this work we target the following problem:

Value Suggestion Problem. Find a suggestion function $\mathcal{S}$ that

1. is 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. $Ans(Q \wedge t(v, x), \mathcal{D}) \neq \emptyset \implies x \in \mathcal{S}(\mathcal{D}, Q, t)$ for all values s 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 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. I.e. we want 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. the suggestion function should be as *precise* as possible. These three conditions are in conflict, and as indicated in the problem description, we consider conditions 1 and 2 to be non-negotiable.

2 Suggestion Functions and Indexing

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. 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.

Range-Based Suggestion Function $\mathcal{S}_r$. Another important suggestion function is the function that computes suggestions 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 w.r.t. both time and space.

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 we now present a family of approximate suggestion functions $\mathcal{S}_a$ to fill 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 returning 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: $Q_{pr} = Q \cap \mathcal{Z}_{\mathcal{C}}$. Q_{pr} is then used together with the underlying dataset $\mathcal{D}$ to calculate value suggestions:

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

Intuitively the concept configuration $\mathcal{Z}_{\mathcal{C}}$ just defines what to consider when calculating suggestions. Every part of Q which is not covered by $\mathcal{Z}_{\mathcal{C}}$ is simply ignored, and removed as the intersection $Q \cap \mathcal{Z}_{\mathcal{C}}$ 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

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:

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

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".

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} combines several concepts, it will result in bad user experience 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 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.

With the index construction method described above, we get one column containing only URIs for each concept variable included in the concept configuration. But this is just a waste of space, since only filtering on datatype variables are allowed. So 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. In our first experiment, we explored how much the accuracy increases by adding another layer of these existential concepts nodes to the index, which is a comparatively cheap investment.

3 Evaluation

We have implemented a faceted search module for OptiqueVQS based on $\mathcal{S}_a$ and the index structure described in Sect. 2. Furthermore, we implemented both $\mathcal{S}_r$ and $\mathcal{S}_o$ in order to compare them to $\mathcal{S}_a$ in our experiments.

In Sect. 2 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 how accurately they can suggest values for different kinds of queries. In total we conducted two experiments with the goal of answering the following three questions about $\mathcal{S}_a$:

1. How does the accuracy increase as the size of $\mathcal{Z}_{\mathcal{C}}$ increases?
2. How much does the accuracy rise by adding existential concept variables to $\mathcal{Z}_{\mathcal{C}}$?
3. How much does the index size have to be increased in order to obtain a given increase in accuracy?

Dataset, Ontology and Queries. In the experiments we used the RDF version of the NPD Factpages[1] – 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[2] we used for the experiments consists of complex queries covering a wide set of possible cases. It 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.

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 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: $pre(\mathcal{S}_a) = \frac{|\mathcal{S}_o|}{|\mathcal{S}_a|}$, $pre(\mathcal{S}_r) = \frac{|\mathcal{S}_o|}{|\mathcal{S}_r|}$ and $rec(\mathcal{S}_a) = rec(\mathcal{S}_r) = 1$.

Both $\mathcal{S}_a$ and $\mathcal{S}_r$ have perfect recall. 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.

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 both of our experiments 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 based on the configuration will only contain one table, so we use the number of cells in the table as a measure for index size. Value suggestions are then calculated by running Q_a over the table, and precision is calculated by comparing to $\mathcal{S}_o$ as explained 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

[1] https://gitlab.com/logid/npd-factpages.
[2] https://github.com/Alopex8064/npd-factpages-experiments.

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 large set of queries.

Exp. 1 - Configuration Type/Size vs Precision. In Experiment 1 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 "configurations 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:

– $Dat(c)$: 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. As explained earlier, it is relatively cheap (w.r.t. storage usage) to add these concept variables, 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, and in general the following holds: $pre(\mathcal{S}_r) \leq pre(\mathcal{S}_a^{Dat(c)}) \leq pre(\mathcal{S}_a^{ObjDat(c)}) \leq pre(\mathcal{S}_o) = 1$. After running through every test case, the results 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 visualized.

Figure 1 displays 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[3] together with charts for each individual query.

The yellow line 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 average 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.

It is worth noting the relatively high precision of the range-based function. In our experiment, its precision ranged from 0.22 to 0.96 (depending on the query), with an average of 0.56. This does not sound too bad, but user studies on OptiqueVQS show that the users are not always satisfied with $\mathcal{S}_r$.

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

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, which was the case for many of our individual queries. The average difference between $\mathcal{S}_a^{ObjDat}$ and $\mathcal{S}_a^{Dat}$, shown in Fig. 1, 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).

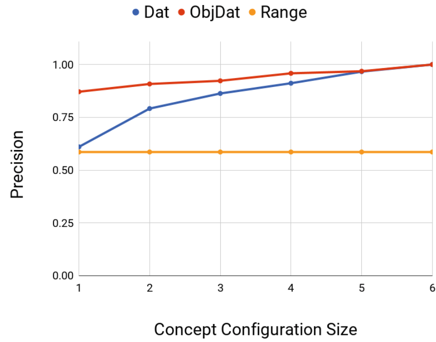

Fig. 1. Average precision of size 6 queries. (Color figure online)

Exp. 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, and each possible configuration $\mathcal{Z}_\mathcal{C}$ covered by it. Then, for each such test case, we calculated both the size of the table generated by $\mathcal{Z}_\mathcal{C}$, and the precision of $\mathcal{S}_a^{\mathcal{Z}_\mathcal{C}}$. Finally we analysed and visualized the results.

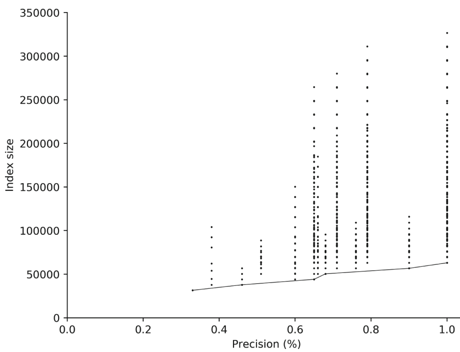

Fig. 2. Scatter plot for Query 6.2. Pareto optimal configurations are connected. Index size is not normalized.

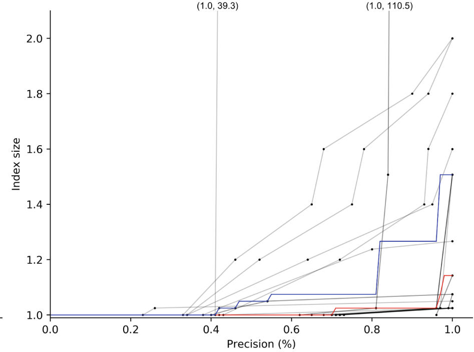

Fig. 3. Pareto optimal configurations for all queries with median (red) and upper quartile (blue). Index size is normalized. (Color figure online)

Figure 2 shows the results for Query 6.2 visualized 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.

We cannot *expect* to achieve results like this consistently, but 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: For any configuration $\mathcal{Z}_{\mathcal{C}}'$ with branches outside Q, there exists another concept configuration $\mathcal{Z}_{\mathcal{C}}$ 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 $\mathcal{Z}_{\mathcal{C}}^{min}$ and $\mathcal{Z}_{\mathcal{C}}^{max}$ denote the configurations used for the first and last of these points. $\mathcal{Z}_{\mathcal{C}}^{max}$ is the configurations that is isomorphic to Q. I.e. it fully covers Q, 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. $\mathcal{Z}_{\mathcal{C}}^{min}$ on the other hand contains only the root and all local datatype properties. This is the smallest configuration that can provide value 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 $\mathcal{Z}_{\mathcal{C}}^{min}$ 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 normalized the index size by dividing by the index size of $\mathcal{Z}_{\mathcal{C}}^{min}$. The y-axis then becomes just a factor, where e.g 2.0 means that the index is twice as large as the index constructed from $\mathcal{Z}_{\mathcal{C}}^{min}$. The pareto-optimal points for all the 29 queries are displayed in Fig. 3, together with the median (red) and upper quartile (blue).

The overall results from Fig. 3 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.

From Experiment 1 and Fig. 1 we know that the average precision of $\mathcal{S}_a$ when using the smallest possible configuration for each query ($\mathcal{Z}_{\mathcal{C}}^{min}$) is 0.61. Figure 3 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 queries that are orders of magnitude higher. This is caused by their highly restrictive filters on branches far away from the root.

4 Conclusion and Future Work

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 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

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.
3. modest increases in index size often leads to a significant increase in accuracy.

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.

Acknowledgement. This project is partially funded by NFR through the SIRIUS center.

References

1. Arenas, M., Grau, B.C., Kharlamov, E., Marciuška, Š., Zheleznyakov, D.: Faceted search over RDF-based knowledge graphs. J. Web Semant. **37**, 55–74 (2016)
2. Brunetti, J.M., García, R., Auer, S.: From overview to facets and pivoting for interactive exploration of semantic web data. IJSWIS **9**(1), 1–20 (2013)
3. Klungre, V.N.: A faceted search index for graph queries. Technical report 469, University of Oslo, Department of Informatics (2017). https://www.duo.uio.no/handle/10852/56755
4. Klungre, V.N., Giese, M.: Approximating faceted search for graph queries. In: 12th Scalable Semantic Web Systems (SWSS) (2018)
5. Soylu, G., Giese, M., et al.: Experiencing OptiqueVQS: a multi-paradigm and ontology-based visual query system for end users. UAIS **15**(1), 129–152 (2016). https://doi.org/10.1007/s10209-015-0404-5
6. Soylu, A., Giese, M., et al.: Ontology-based end-user visual query formulation: why, what, who, how, and which? UAIS **16**(2), 435–467 (2017). https://doi.org/10.1007/s10209-016-0465-0
7. Tunkelang, D.: Faceted search. Synthesis lectures on information concepts, retrieval, and services, **1**(1), 1–80 (2009)