

UiO : **Department of Informatics**  
University of Oslo

# A Faceted Search Index for Graph Queries

Research report no. 469

Vidar N. Klungre

28 July 2017

ISBN: 978-82-7368-434-9

ISSN: 0806-3036





# A Faceted Search Index for Graph Queries

Vidar N. Klungre

University of Oslo

**Abstract.** This report explains the details of a configurable index structure that allows to perform efficiently the kinds of filtering operations required to implement faceted search over RDF data. Unlike previous systems, it is designed to scale gracefully to both very large datasets *and* complex queries. In return, it compromises some precision in computing sets of available facet values, but it does so in a highly configurable manner.

**Keywords:** Faceted search, RDF, Index, Visual Query Interface

## 1 Introduction

Faceted search [5] is a popular search and exploration paradigm which allows users to apply search filters to multiple orthogonal dimensions (facets) of the data, often in combination with free text search, in order to find relevant information. Since the facets are independent, filters can be added, removed or modified in any preferred order, and every time this is done, the system immediately updates the list of results, giving the user instant feedback. Furthermore, the system counts for each facet the occurrences of different values appearing in the result set, telling the user which values it is interesting to filter on, and how large the result set will become if one such filter is added. To support this functionality, the system needs fast access to the underlying data. This is often provided by search engines like Lucene<sup>1</sup> or Sphinx<sup>2</sup> which provide better performance for the queries required by faceted search than RDB-based implementations.

Faceted search has also been extended to semantics-based data, e.g. in Sem-Facet [1] and Rhizomer [3]. 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 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,

---

<sup>1</sup> <https://lucene.apache.org/>

<sup>2</sup> <http://sphinxsearch.com/>

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.

We implemented this new functionality as part of OptiqueVQS [4], an ontology-based visual query system (VQS), intended for users with little IT expertise. OptiqueVQS combines graph querying with filtering on datatype properties. However, in previous versions of OptiqueVQS, the available values for each facet have been static, determined entirely by a suitably annotated ontology, and did not depend on the underlying data. Neither did they change in reaction to the filters on other facets. This new version adds a server side component that reads data from a SPARQL endpoint and stores information needed for efficient faceted search in a scalable index. This index, instead of the original SPARQL endpoint, is queried in reaction to user interactions to update the interface.

Even though the system has been implemented in OptiqueVQS, the theory of this paper can be applied to other similar VQSs.

The remaining part of this paper is structured as follows: Section 2 gives the prerequisites for all the work of the paper, Section 3 describes how the underlying resources and the configuration is represented, Sections 4 and 5 describe how the index is constructed and used respectively, Section 6 discuss the different ways of representing the index, Section 7 describes a data compression technique which can be used on the index, Section 8 analyzes the correctness of the system, Section 9 describes the implementation, while Section 10 wraps up the paper and presents future work.

## 2 Prerequisites

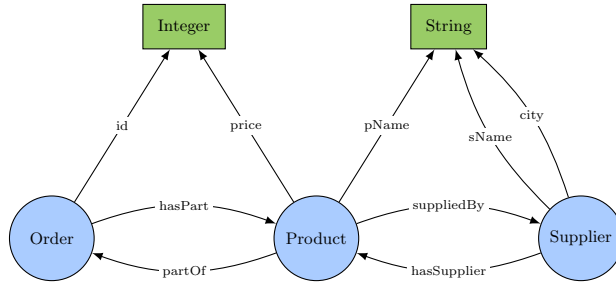
### 2.1 Graphs

As a prerequisite for our theoretical work, we are going to introduce a certain type of graphs, called RDF\*-graphs. These graphs are a slight simplification of RDF graphs that we will use to describe data, ontologies, queries, and system configurations to the extent needed in this work.

**Definition 1 (RDF\*-graph)** *An RDF\*-graph is a directed labeled multigraph with two types of nodes. It is defined by a triple  $\mathcal{G} = (C, D, P)$ , where  $C$  and  $D$  are the two disjoint sets of nodes and  $P$  is the set of edges (representing properties), and where none of the nodes in  $D$  have outgoing edges.*

**Example 1** *Fig. 1 shows an example of a navigation graph (a type of RDF\*-graph) where the set  $C$  is represented by blue circles, while  $D$  is represented by green rectangles, a color convention we will use for all graphs in this paper. Notice that none of the edges are leaving any rectangular node, as required by the definition of RDF\*-graphs.*

With this example, we get an intuition of what  $C$  and  $D$  aim to represent: The set  $C$  is meant to represent items related to concepts, like concepts and



**Fig. 1.** The navigation graph  $\mathcal{G}_N$  describing the sales domain with orders, products and suppliers together with relevant properties.

individuals, and it will be used to define graph structures in e.g. queries, while the set  $D$  on the other hand, aims to represent everything related to datatypes and data values. Unlike RDF, RDF\* does not have blank nodes.

The edges of the RDF\*-graph,  $P$ , are used to represent properties. Since properties are directional and have a unique URI, we use edges which are directional and labeled with the same URI as the property it is representing. Furthermore, since resources can have multiple properties between them, we define the RDF\*-graphs we use to be multigraphs.

Finally we have added the requirement that none of the nodes in  $T$  can have outgoing edges. This is a natural requirement related to the fact that subjects cannot be literals in RDF-graphs.

The RDF\*-graph does not specify explicitly whether an edge represents an object property or a datatype property, but we can easily detect the type by looking at the head of the edge. If the head is contained in  $C$ , the edge represents an object property, otherwise, a datatype property.

In the following sections we will define special types of graphs, like *paths* and *trees*, in addition to *homomorphism* in the context of RDF\*-graphs.

## 2.2 Paths

In general, a path in a graph is used to describe a sequence of nodes that can be visited in the order defined by the sequence. We use the same definition for paths in RDF\*-graphs, by ignoring the distinction between  $C$  and  $D$ .

**Definition 2 (Path)** A path in an RDF\*-graph  $\mathcal{G} = (C, D, P)$  is a sequence  $\mathcal{P} = (v_1, e_1 \dots v_{k-1}, e_{k-1}, v_k)$  where  $v_i \in C \cup D$ ,  $e_i \in P$  and  $e_i$  connects  $v_i$  to  $v_{i+1}$  for all  $i$ .

Def. 2 does not require the elements of the sequence to be distinct, so paths with repeating edges or nodes are allowed. This however, does not change the fact that  $\mathcal{P}$  defines a valid RDF\*-graph, made by the nodes and edges included in the path sequence after duplicates have been removed.

Since a path is defined to be a sequence, we can reuse the definition of sequence prefixes to define path prefixes.

**Definition 3 (Path Prefix)** Let  $\mathcal{P} = (v_1, e_1 \dots v_{k-1}, e_{k-1}, v_k)$  be a path in an RDF\*-graph. A path  $\mathcal{P}' = (v'_1, e'_1 \dots v'_{l-1}, e'_{l-1}, v'_l)$  is a prefix of  $\mathcal{P}$  if  $l \leq k$  and

$$\begin{aligned} v_i &= v'_i & \forall i \leq l \\ e_i &= e'_i & \forall i \leq l-1. \end{aligned}$$

Given any path  $\mathcal{P}$ , we let  $\text{prefix}(\mathcal{P})$  denote the set of all prefixes of  $\mathcal{P}$ .

**Example 2** The path

$$\mathcal{P} = (\text{Product}, \text{suppliedBy}, \text{Supplier}, \text{hasSupplier}, \text{Product}, \text{pName}, \text{String})$$

is an example of a path in the RDF\*-graph  $\mathcal{G}_N$  defined in Example 1. It has four prefixes, including itself, so  $\text{prefix}(\mathcal{P}) = \{\mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3, \mathcal{P}_4\}$  where

$$\mathcal{P}_1 = (\text{Product})$$

$$\mathcal{P}_2 = (\text{Product}, \text{suppliedBy}, \text{Supplier})$$

$$\mathcal{P}_3 = (\text{Product}, \text{suppliedBy}, \text{Supplier}, \text{hasSupplier}, \text{Product})$$

$$\mathcal{P}_4 = (\text{Product}, \text{suppliedBy}, \text{Supplier}, \text{hasSupplier}, \text{Product}, \text{pName}, \text{String}).$$

## 2.3 Trees

When we refer to trees in this paper and in the context of RDF\*-graphs, we mean rooted, directed trees.

**Definition 4 (Tree)** A tree is an RDF\*-graph  $\mathcal{G} = (C, D, P)$  which is connected and contains no cycles, even when direction of the edges is ignored. One specific node  $r \in C$  is called the root of the tree, and all edges must be directed away from  $r$ .

Since trees do not allow cycles, there is not more than one edge between each pair of nodes, and self-loops do not exist. Furthermore, since all of the edges are directed away from the root, each branch of the tree defines a path. Given a tree  $\mathcal{G}$  we let  $\text{branches}(\mathcal{G})$  denote the set of all branches of  $\mathcal{G}$ .

**Example 3** Let  $\mathcal{G}_{Q_1}$  and  $\mathcal{G}_{Q_2}$  be the two trees in Fig. 2 and Fig. 3 respectively. Then  $\text{branches}(\mathcal{G}_{Q_1}) = \{\mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3\}$ , where

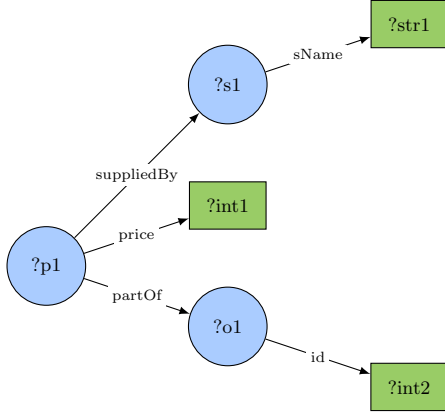
$$\mathcal{P}_1 = (?p1, \text{suppliedBy}, ?s1, \text{sName}, ?str1)$$

$$\mathcal{P}_2 = (?p1, \text{unitPrice}, ?int1)$$

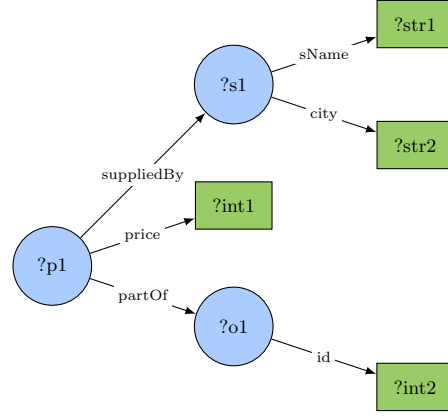
$$\mathcal{P}_3 = (?p1, \text{partOf}, ?o1, \text{id}, ?int2).$$

$\text{branches}(\mathcal{G}_{Q_2})$  contains all of the branches in  $\text{branches}(\mathcal{G}_{Q_1})$  in addition to

$$\mathcal{P}_4 = (?p1, \text{suppliedBy}, ?s1, \text{city}, ?str2).$$



**Fig. 2.** Query graph  $\mathcal{G}_{Q_1}$ .



**Fig. 3.** Query graph  $\mathcal{G}_{Q_2}$ .

## 2.4 Homomorphisms

We will introduce several different RDF\*-graphs in this paper, all of which will use the same set of properties, and have similar structure. We need to formally define a similarity relationship between them, and we will use homomorphisms to do so:

**Definition 5 (Homomorphism)** Let  $\mathcal{G}_1 = (C_1, D_1, P_1)$  and  $\mathcal{G}_2 = (C_2, D_2, P_2)$  be two RDF\*-graphs. A homomorphism from  $\mathcal{G}_1$  to  $\mathcal{G}_2$  is a function  $h: C_1 \cup D_1 \rightarrow C_2 \cup D_2$  where

$$h(c_1) \in C_2 \quad \forall c_1 \in C_1 \quad (1)$$

$$h(d_1) \in D_2 \quad \forall d_1 \in D_1 \quad (2)$$

and

$$v_1 \xrightarrow{l} w_1 \in P_1 \Rightarrow h(v_1) \xrightarrow{l} h(w_1) \in P_2 \quad (3)$$

If  $h$  is a homomorphism between  $\mathcal{G}_1$  and  $\mathcal{G}_2$ , we say that  $\mathcal{G}_1$  is homomorphic to  $\mathcal{G}_2$ .

Equations 1 and 2 from Def. 5 ensure that the two types of nodes are not interchanged, i.e., that  $h$  maps elements from  $C_1$  to  $C_2$ , and elements from  $D_1$  to  $D_2$ . Equation 3, on the other hand, ensures that each edge in the source graph has a corresponding edge in the target graph with the same label. Even though homomorphisms are only defined for nodes in the source graph, each edge will also have a corresponding edge in the target graph due to Equation 3. This means that we can apply homomorphisms to subgraphs of the source graph, and obtain a well-defined subgraph of the target graph (after removing duplicates).

**Example 4** Let  $\mathcal{G}_{\mathcal{Q}_1}$  be the graph in Fig. 2, and  $\mathcal{G}_{\mathcal{N}}$  be the navigation graph in Fig. 1. Then  $\mathcal{G}_{\mathcal{Q}_1}$  is homomorphic to  $\mathcal{G}_{\mathcal{N}}$ , because there is a homomorphism  $h_1: \mathcal{G}_{\mathcal{Q}_1} \rightarrow \mathcal{G}_{\mathcal{N}}$  defined by

$$\begin{aligned} h_1(?p1) &= Product & h_1(?str1) &= String \\ h_1(?s1) &= Supplier & h_1(?int1) &= Integer \\ h_1(?o1) &= Order & h_1(?int2) &= Integer. \end{aligned}$$

Since some of the homomorphisms are obvious from the context and the structure of our graphs, we will often omit to write them out explicitly.

Finally, if a homomorphism is bijective and its inverse is also a homomorphism, it is an isomorphism:

**Definition 6 (Isomorphism)** Let  $\mathcal{G}_1 = (C_1, D_1, P_1)$  and  $\mathcal{G}_2 = (C_2, D_2, P_2)$  be two RDF\*-graphs. An isomorphism from  $\mathcal{G}_1$  to  $\mathcal{G}_2$  is a bijective homomorphism whose inverse function is also a graph homomorphism

### 3 System Configuration

#### 3.1 Navigation graph

Instead of working directly with the ontology, we assume that the VQS has access to a simplified graph version of it, called the *navigation graph*. This graph is constructed by the VQS itself, based on a selected subset of the ontology axioms. It describes how to navigate between concepts during the query construction process in order to make meaningful queries, i.e., queries which do not conflict with the ontology or the data. Furthermore, since it defines which queries the system can construct, it defines how the data must be structured in order to be accessible by the VQS. There is no definite way to construct the navigation graph, different VQSs do it differently, but the domain and range axioms are usually central in the process, since they dictate the type of variables reached through properties.

OptiqueVQS [4], Semfacet [1] and Rizomer [3] are all VQSs which utilizes some kind of navigation graph. Another approach to selecting available navigation possibilities is described in [2].

In this paper we are not concerned with how the different VQSs construct their navigation graphs, we just assume that the graph exists.

**Definition 7 (Navigation graph)** A navigation graph  $\mathcal{G}_{\mathcal{N}} = (C_{\mathcal{N}}, D_{\mathcal{N}}, P_{\mathcal{N}})$  is an RDF\*-graph where  $C_{\mathcal{N}}$  and  $D_{\mathcal{N}}$  represent concepts and datatypes respectively.

The distinction between concepts and datatypes in the navigation graph is important since our goal is to combine faceted search with graphs. The concepts and the properties between them (object properties) allow us to define the graph structure, while the properties from concepts to datatypes (datatype properties) corresponds to the actual facets, which give access to the data values.

**Example 5** *Fig. 1 shows an example of a navigation graph  $\mathcal{G}_N$  with three concepts (Product, Supplier, Order) and two datatypes (String, Integer). If we look closer at the edges of this graph, we see that none of the edges leaves a datatype, which is required by the definition. Furthermore we see that each object property is displayed together with its inverse. It is not necessary from the point of view of our index to include the inverse of every object property, but adding them adds flexibility to the user during query construction, since it allows users navigate both back and forth between concepts during query construction.*

### 3.2 Data sources

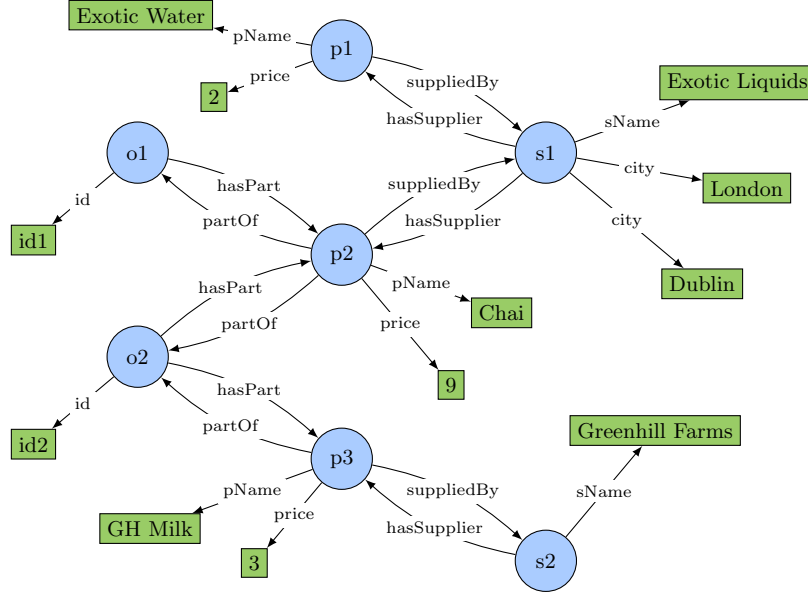
Each VQS setup must contain an underlying data source, which is used to answer queries constructed by the users. If querying over this is too time consuming to support the interactive UI, we need a faster data source, which is why we introduce the facet index. We will now define a concept of data source which covers both the underlying data source, and the facet index, and which works well with the navigation graph we just defined.

The actual data of the source is assumed to be stored in an RDF\*-graph (data graph) consisting of individuals and data values connected by relevant properties. Furthermore, in order to be queryable, it has to be homomorphic to the navigation graph.

**Definition 8 (Data source)** *Let  $\mathcal{G}_N = (C_N, D_N, P_N)$  be a navigation graph. A data source of  $\mathcal{G}_N$  is a pair  $\mathcal{D} = (\mathcal{G}_D, h_D)$  where  $\mathcal{G}_D = (C_D, D_D, P_D)$  is an RDF\*-graph (data graph) and  $h_D$  is a homomorphism from  $\mathcal{G}_D$  to  $\mathcal{G}_N$ . The sets  $C_D$  and  $D_D$  represent individuals and data values respectively.*

The homomorphism  $h_D$  maps individuals to concepts, and data values to data types, so it is in fact a typing function. In RDF/RDFS, individuals may be members of multiple concepts, but the homomorphisms we use do not support this. Extending the homomorphism definition to allow multiple types is future work.

**Example 6** *Fig. 4 shows an example of a data graph  $\mathcal{G}_D$  containing seven individuals and 12 data values. The data graph is homomorphic to the navigation graph  $\mathcal{G}_N$  from Example 5 as proved by the homomorphism  $h_D$  defined below, and hence the pair  $\mathcal{D} = (\mathcal{G}_D, h_D)$  is a valid data source of  $\mathcal{G}_N$ .*



**Fig. 4.** The data graph  $\mathcal{G}_D$  matching the navigation graph from Example 5.

$h_D(p1) = Product$	$h_D(Water) = String$
$h_D(p2) = Product$	$h_D(GH Milk) = String$
$h_D(p3) = Product$	$h_D(Chai) = String$
$h_D(s1) = Supplier$	$h_D(ExoticLiquids) = String$
$h_D(s2) = Supplier$	$h_D(GreenhillFarms) = String$
$h_D(o1) = Order$	$h_D(London) = String$
$h_D(o2) = Order$	$h_D(Dublin) = String$
$h_D(id1) = Integer$	$h_D(2) = Integer$
$h_D(id2) = Integer$	$h_D(3) = Integer$
	$h_D(9) = Integer$

### 3.3 Queries

In this paper, we consider SPARQL queries with only one tree-shaped graph pattern and typed query variables, i.e., each query variable must be mapped to exactly one specific concept or datatype in the given navigation graph. The main reason for this restriction is that OptiqueVQS only supports the construction of such queries for the time being.

Based on the types of the variables, we can distinguish *concept variables* and *datatype variables*, which are variables typed to concepts and datatypes respectively.

Datatype variables may also be restricted by filters, but we are not concerned with the details of these filters, only which values a certain variable can hold. To incorporate this information into a query, we include a *filter function*, which returns a set of allowed values for each datatype variable.

**Definition 9 (Filter Function)** *Let  $D$  be a set of datatype variables, and let each datatype variable  $d$  have a corresponding datatype  $t_d$ . A filter function  $\mathcal{F}$  of  $D$  takes a datatype variable  $d \in D$  as input, and returns a set of data values of type  $t_d$ .*

The special filter function  $\mathcal{F}^*$  returns the infinite set of all possible values, for every datatype variable. In other words, it is the filter function without any restrictions, which we will use for queries without filters.

**Definition 10 (Query)** *Let  $\mathcal{G}_N = (C_N, D_N, P_N)$  be a navigation graph. A query over  $\mathcal{G}_N$  is a triple  $\mathcal{Q} = (\mathcal{G}_Q, h_Q, \mathcal{F}_Q)$  where  $\mathcal{G}_Q = (C_Q, D_Q, P_Q)$  is a tree-shaped RDF\*-graph (query graph),  $h_Q$  is a homomorphism from  $\mathcal{G}_Q$  to  $\mathcal{G}_N$ , and  $\mathcal{F}_Q$  is a filter function of  $D_Q$ . The sets  $C_Q$  and  $D_Q$  represent concept variables and datatype variables respectively, while the root of  $\mathcal{G}_Q$ ,  $r$ , is called the root variable, and  $h_Q(r)$  is called the root concept.*

Note that the notions of root variable, concept variables and datatype variables are not defined in SPARQL, but something we introduce in our work.

**Example 7** *Fig. 2 shows the query graph  $\mathcal{G}_{Q_1}$ , consisting of three concept variables ( $?p1$ ,  $?s1$  and  $?o1$ ) and three datatype variables ( $?str1$ ,  $?int1$  and  $?int2$ ), with  $?p1$  as the root variable. Combined with the homomorphism from Example 4 and a suitable filter function  $\mathcal{F}_{Q_1}$ , we get a query  $\mathcal{Q}_1 = (\mathcal{G}_{Q_1}, h_{Q_1}, \mathcal{F}_{Q_1})$  over the navigation graph  $\mathcal{G}_N$  from Example 5.*

If we use the restrictionless filter function by setting  $\mathcal{F}_{Q_1} = \mathcal{F}^*$ , we get a query without any restrictions, so all the three datatype variables can technically take any value belonging to the datatype of the variable. However, we can also set query restrictions by defining a filter function for all datatype variables  $d \in D$  which restrict e.g.  $?int1$ :

$$\mathcal{F}(d) = \begin{cases} \{n \in \mathbb{N} \mid n < 5\} & \text{if } d = ?int1 \\ \mathcal{F}^*(d) & \text{otherwise} \end{cases}$$

If  $\mathcal{F}_{Q_1} = \mathcal{F}$ , the query  $\mathcal{Q}_1$  is only allowed to assign integers smaller than 5 to  $?int1$ .

Fig. 3 shows another query graph  $\mathcal{G}_{Q_2}$ , which together with the obvious homomorphism  $h_{Q_2}$  and a suitable filter function  $\mathcal{F}_{Q_2}$  makes the query  $\mathcal{Q}_2 = (\mathcal{G}_{Q_2}, h_{Q_2}, \mathcal{F}_{Q_2})$ .

Given a query  $\mathcal{Q}$  and a data source over a navigation graph, we can execute the query over the data source, i.e., find all possible assignments to the set of variables in  $\mathcal{Q}$  that respect the structure and filters of  $\mathcal{Q}$ .

**Definition 11 (Query Answers)** Let  $\mathcal{G}_N$  be a navigation graph,  $\mathcal{D} = (\mathcal{G}_D, h_D)$  a data source over  $\mathcal{G}_N$ , and  $\mathcal{Q} = (\mathcal{G}_Q, h_Q, \mathcal{F}_Q) = ((C_Q, D_Q, P_Q), h_Q, \mathcal{F}_Q)$  a query over  $\mathcal{G}_N$ . The answers of  $\mathcal{Q}$  over  $\mathcal{D}$ ,  $Ans(\mathcal{Q}, \mathcal{D})$ , is the set of all homomorphisms  $a$  from  $\mathcal{G}_Q$  to  $\mathcal{G}_D$  such that

$$h_Q(v) = h_D(a(v)) \quad (4)$$

and

$$a(v) \in \mathcal{F}_Q(v) \quad (5)$$

for each query variable  $v \in C_Q \cup D_Q$ .

Equation 4 and the requirement that  $a$  must be a homomorphism ensures that the assignment  $a$  preserves the correct structure and type of any query variable, while Equation 5 guarantees that the filters are also satisfied.

It is worth noting that the assignment function  $a$  may assign the same individual or data value in  $\mathcal{G}_D$  to multiple variables. This is exactly how SPARQL query answering is done, so this is a desirable property.

The answer function returns a set of homomorphisms, not the assigned values. To get data we can actually use, we need another query answer function that includes which of the query variables we want access to.

**Definition 12 (Projected Query Answers)** Let  $\mathcal{G}_N$  be a navigation graph,  $\mathcal{D} = (\mathcal{G}_D, h_D)$  a data source over  $\mathcal{G}_N$ ,  $\mathcal{Q} = (\mathcal{G}_Q, h_Q, \mathcal{F}_Q) = ((C_Q, D_Q, P_Q), h_Q)$  a query over  $\mathcal{G}_N$  and  $v \in C_Q \cup D_Q$  a query variable of  $\mathcal{Q}$ . The answers of  $\mathcal{Q}$  over  $\mathcal{D}$ , projected onto  $v$ , denoted  $Ans_v(\mathcal{Q}, \mathcal{D})$  is given by

$$Ans_v(\mathcal{Q}, \mathcal{D}) = \{h(v) \mid h \in Ans(\mathcal{Q}, \mathcal{D})\} \quad (6)$$

There may be several homomorphisms mapping  $v$  to the same individual or data value, but since we consider  $Ans_v(\mathcal{Q}, \mathcal{D})$  to be a set, duplicates are removed, and we are left with only distinct values.

**Example 8** If we execute  $\mathcal{Q}_1$  from Example 7 over the data source  $\mathcal{D}$  from Example 6 we get  $Ans(\mathcal{Q}_1, \mathcal{D}) = \{h_1, h_2, h_3\}$  where

$h_1(?p1) = p2$	$h_2(?p1) = p2$	$h_3(?p1) = p3$
$h_1(?s1) = s1$	$h_2(?s1) = s1$	$h_3(?s1) = s2$
$h_1(?o1) = o1$	$h_2(?o1) = o2$	$h_3(?o1) = o2$
$h_1(?int1) = 9$	$h_2(?int1) = 9$	$h_3(?int1) = 3$
$h_1(?int2) = id1$	$h_2(?int2) = id2$	$h_3(?int2) = id2$
$h_1(?str1) =$	$h_2(?str1) =$	$h_3(?str1) =$
<i>Exotic Liquids</i>	<i>Exotic Liquids</i>	<i>Greenhill Farms</i>

However, if we e.g. want to only get the distinct values  $?int1$  can be assigned to, we use the projected query answer function:

$$Ans_{?int1}(\mathcal{Q}_1, \mathcal{D}) = \{3, 9\}$$

The next thing we have to define is what a query looks like after we prune it with respect to another query, i.e., after we remove all the nodes which does not have a corresponding node in the second query. The result of this pruning is almost like a graph intersection where the variable names and filters from the first query is kept.

**Definition 13 (Query Pruning)** Let  $\mathcal{Q}_1 = (\mathcal{G}_{\mathcal{Q}_1}, h_{\mathcal{Q}_1}, \mathcal{F}_{\mathcal{Q}_1})$  and  $\mathcal{Q}_2 = (\mathcal{G}_{\mathcal{Q}_2}, h_{\mathcal{Q}_2}, \mathcal{F}_{\mathcal{Q}_2})$  be two queries over the navigation graph  $\mathcal{G}_{\mathcal{N}} = (C_{\mathcal{N}}, D_{\mathcal{N}}, P_{\mathcal{N}})$ . The pruning of  $\mathcal{Q}_1$  with respect of  $\mathcal{Q}_2$ , denoted  $\text{prune}(\mathcal{Q}_1, \mathcal{Q}_2)$  is given by

$$\text{prune}(\mathcal{Q}_1, \mathcal{Q}_2) = (\mathcal{G}, h_{\mathcal{Q}_1}, \mathcal{F}_{\mathcal{Q}_1})$$

where  $\mathcal{G}$  is the largest subtree of  $\mathcal{G}_{\mathcal{Q}_1}$  containing the root of  $\mathcal{G}_{\mathcal{Q}_1}$ , such that there exist an isomorphism  $f$  from  $\mathcal{G}$  to a subtree of  $\mathcal{G}_{\mathcal{Q}_2}$  which satisfies

$$h_{\mathcal{Q}_1}(v) = h_{\mathcal{Q}_2}(f(v)).$$

There may be cases where there are several subgraphs of  $\mathcal{G}_{\mathcal{Q}_1}$  which are largest, but of the same size. If so, we select the subgraph containing datatype variables with the most restrictive filters.

The last definition we need for queries is describing how new query variables can be appended already existing queries. Our query extension function adds a variable  $v$  of type  $c$  to the root variable  $r$ , by adding a property with label  $l$ :

**Definition 14 (Query extension)** Let  $\mathcal{G}_{\mathcal{N}} = (C_{\mathcal{N}}, D_{\mathcal{N}}, P_{\mathcal{N}})$  be a navigation graph and  $\mathcal{Q} = (\mathcal{G}_{\mathcal{Q}}, h_{\mathcal{Q}}, \mathcal{F}_{\mathcal{Q}}) = ((C_{\mathcal{Q}}, D_{\mathcal{Q}}, P_{\mathcal{Q}}), h_{\mathcal{Q}}, \mathcal{F}_{\mathcal{Q}})$  a query over  $\mathcal{G}_{\mathcal{N}}$  with root variable  $r$ . The extension of  $\mathcal{Q}$  where  $r$  is  $l$ -related to  $v$  of type  $c$  is denoted  $\text{extend}(\mathcal{Q}, l, v, c)$  and defined as:

$$\text{extend}(\mathcal{Q}, l, v, c) = (\mathcal{G}_{ex}, h_{ex}, \mathcal{F}_{ex}) = ((C_{ex}, D_{ex}, P_{ex}), h_{ex}, \mathcal{F}_{ex})$$

where

$$\begin{aligned} C_{ex} &= \begin{cases} C_{\mathcal{Q}} \cup \{v\} & \text{if } c \in C_{\mathcal{N}} \\ C_{\mathcal{Q}} & \text{if } c \in D_{\mathcal{N}} \end{cases} \\ D_{ex} &= \begin{cases} D_{\mathcal{Q}} \cup \{v\} & \text{if } c \in D_{\mathcal{N}} \\ D_{\mathcal{Q}} & \text{if } c \in C_{\mathcal{N}} \end{cases} \\ P_{ex} &= P_{\mathcal{Q}} \cup \{r \xrightarrow{l} v\} \\ h_{ex}(w) &= \begin{cases} c & \text{if } w = v \\ h_{\mathcal{Q}}(w) & \text{otherwise} \end{cases} \\ \mathcal{F}_{ex}(w) &= \begin{cases} \mathcal{F}^*(w) & \text{if } w = v \\ \mathcal{F}_{\mathcal{Q}}(w) & \text{otherwise} \end{cases} \end{aligned}$$

### 3.4 Configuration

The facet index we are going to build will consist of several concept indices, one for each concept in the given navigation graph. Given a concept  $c$ , its concept index will only include instances of  $c$ , and data located in a certain neighbourhood of it. In order to define this neighbourhood, we need a configuration structure in the form of a query. We call this the concept configuration of  $c$ , and it will be used both when building the index as an offline process, and when using the index during a query session.

**Definition 15 (Concept Configuration)** *Let  $\mathcal{G}_N = (C_N, D_N, P_N)$  be a navigation graph with a concept  $c \in C_N$ . A concept configuration of  $c$  over  $\mathcal{G}_N$  is a query  $\mathcal{C}_c = (\mathcal{G}_{\mathcal{C}_c}, h_{\mathcal{C}_c}, \mathcal{F}_{\mathcal{C}_c}) = ((C_{\mathcal{C}_c}, D_{\mathcal{C}_c}, P_{\mathcal{C}_c}), h_{\mathcal{C}_c}, \mathcal{F}_{\mathcal{C}_c})$  over  $\mathcal{G}_N$  satisfying the following three criteria:*

1.  $h_{\mathcal{C}_c}(r) = c$ , where  $r$  is the root of  $\mathcal{G}_{\mathcal{C}_c}$ .
2.  $\mathcal{F}_{\mathcal{C}_c} = \mathcal{F}^*$ .
3. For every pair of edges  $e_1 = v \xrightarrow{l} w_1 \in P_{\mathcal{C}_c}$  and  $e_2 = v \xrightarrow{l} w_2 \in P_{\mathcal{C}_c}$ ,

$$h_{\mathcal{C}_c}(w_1) = h_{\mathcal{C}_c}(w_2) \Rightarrow e_1 = e_2$$

The first requirement of Def. 15 states that the root of the query must be of type  $c$ , which makes sense since we want to define a neighbourhood around  $c$ . Then the branches of  $\mathcal{C}_c$  define how far the neighbourhood should go in each direction. The second requirement states that the filter function must accept all values, i.e., there can not be any restrictions to any of the query variables. Finally the third requirement limits the structure of the concept configuration graph by only allowing structurally different branches. I.e., none of the nodes can have two outgoing edges with the same label to two different other nodes.

**Example 9** *Fig. 5 shows two concept configuration graphs  $\mathcal{G}_{\mathcal{C}_{Prod}^1}$  and  $\mathcal{G}_{\mathcal{C}_{Prod}^2}$ , which together with their obvious homomorphisms and  $\mathcal{F}^*$  makes two possible concept configurations for the product concept:*

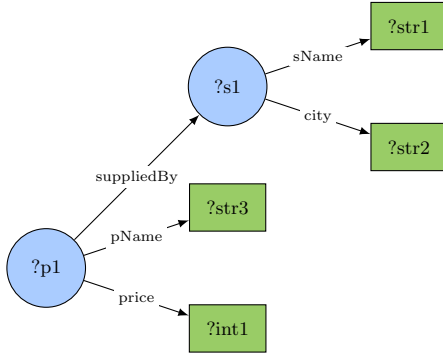
$$\mathcal{C}_{Prod}^1 = (\mathcal{G}_{\mathcal{C}_{Prod}^1}, h_{\mathcal{C}_{Prod}^1}, \mathcal{F}^*)$$

$$\mathcal{C}_{Prod}^2 = (\mathcal{G}_{\mathcal{C}_{Prod}^2}, h_{\mathcal{C}_{Prod}^2}, \mathcal{F}^*)$$

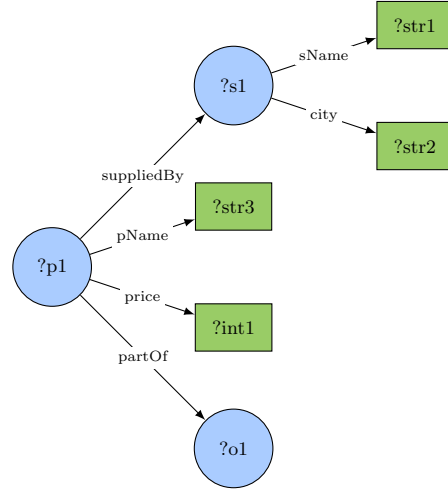
The concept configuration of  $c$  describes how the system should behave when  $c$  is in focus, so in order to describe how the system should be working as a whole, we need one concept configuration for each concept, which is why we introduce the index configuration:

**Definition 16 (Index Configuration)** *Let  $\mathcal{G}_N = (C_N, D_N, P_N)$  be a navigation graph. An index configuration over  $\mathcal{G}_N$  is a function  $\mathcal{Z}$  defined for all concepts  $c \in C_N$  which returns  $\mathcal{C}_c$ , the concept configuration of  $c$ .*

$$\mathcal{Z}: c \mapsto \mathcal{C}_c$$



**Fig. 5.** Concept configuration graph  $\mathcal{G}_{C^1_{Prod}}$ .



**Fig. 6.** Concept configuration graph  $\mathcal{G}_{C^2_{Prod}}$ .

The index configuration is just a collection of all concept configurations, defined as a function in order to give easy access to the configuration of any given concept.

## 4 Index Construction

Index construction is a process which has to be carried out every time the underlying data source changes. The process may be time-consuming, so ideally updates should be infrequent, and happen during time periods when the user traffic is low, e.g., during night. Incremental updates of the index are a topic for future work.

To construct the index, the system needs access to the navigation graph  $\mathcal{G}_N = (C_N, D_N, P_N)$ , a corresponding data source  $\mathcal{D}$ , and an index configuration  $\mathcal{Z}$ . Using this, it will create one concept index  $\bar{\mathcal{D}}_c$  for each concept  $c \in C_N$ , each one defined by the concept configuration  $\mathcal{C}_c = \mathcal{Z}(c)$ . Since the  $\mathcal{C}_c$  defines the relevant neighbourhood around  $c$ , we populate  $\bar{\mathcal{D}}_c$  with all the instances of  $c$  in addition all the data covered by  $\mathcal{C}_c$ , i.e., all nodes which can be reached by  $c$  through a path which corresponds to a branch in  $\mathcal{C}_c$ , or a prefix of it.

**Definition 17 (Concept Index)** Let  $\mathcal{G}_N = (C_N, D_N, P_N)$  be a navigation graph with a concept  $c \in C_N$ ,  $\mathcal{D} = (\mathcal{G}_D, h_D)$  a data source over  $\mathcal{G}_N$ , and  $\mathcal{C}_c = (\mathcal{G}_c, h_c, \mathcal{F}^*)$  a concept configuration of  $c$  over  $\mathcal{G}_N$ . The concept index of  $c$  defined by  $\mathcal{C}_c$  over  $\mathcal{D}$ , denoted  $buildIndex(\mathcal{C}_c, \mathcal{D})$  is defined by

$$\mathbf{buildIndex}(\mathcal{C}_c, \mathcal{D}) = \bar{\mathcal{D}}_c = (\mathcal{G}_{\bar{\mathcal{D}}_c}, h_{\mathcal{D}}),$$

where

$$\mathcal{G}_{\bar{\mathcal{D}}_c} = \bigcup_{\mathcal{P} \in \text{branches}(\mathcal{G}_{\mathcal{C}_c})} \left( \bigcup_{\mathcal{P}' \in \text{prefix}(\mathcal{P})} \left( \bigcup_{h \in \text{Ans}((\mathcal{P}', h_{\mathcal{C}_c}, \mathcal{F}^*), \mathcal{D})} h(\mathcal{P}') \right) \right). \quad (7)$$

The two first unions of Equation 7 range over all path prefixes  $\mathcal{P}'$  of each branch of  $\mathcal{C}_c$ , while the last union collects all the data we get when the query defined by  $\mathcal{P}'$  is executed over  $\mathcal{D}$ . The data returned by all the different versions of  $\mathcal{P}'$  overlaps a lot, but since we apply a graph union over the results, duplicates are removed. The final result is therefore the subgraph of  $\mathcal{D}$  consisting of all the instances of  $c$  together with their corresponding neighbourhoods.

**Example 10** *If we consider the data source  $\mathcal{D}$  from Example 6 and the concept configuration  $\mathcal{C}_{\text{Prod}}^1$  from Example 9, the resulting concept index  $\bar{\mathcal{D}}_{\text{Prod}}^1 = \mathbf{buildIndex}(\mathcal{C}_{\text{Prod}}^1, \mathcal{D})$  is displayed in Fig. 7.*

*The concept index shows three different products  $p1$ ,  $p2$  and  $p3$  and their corresponding neighbourhoods which are all isomorphic to either the full concept configuration graph or a pruned version of it. Product  $p3$  for example, does not include the city of its supplier, because this data is missing, however, its neighbourhood is still included in the concept index, just separated from the rest of the graph.  $p1$  and  $p2$  on the other hand both have fully populated neighbourhoods, but since they share a common supplier  $s1$ , their neighbourhoods are merged together.*

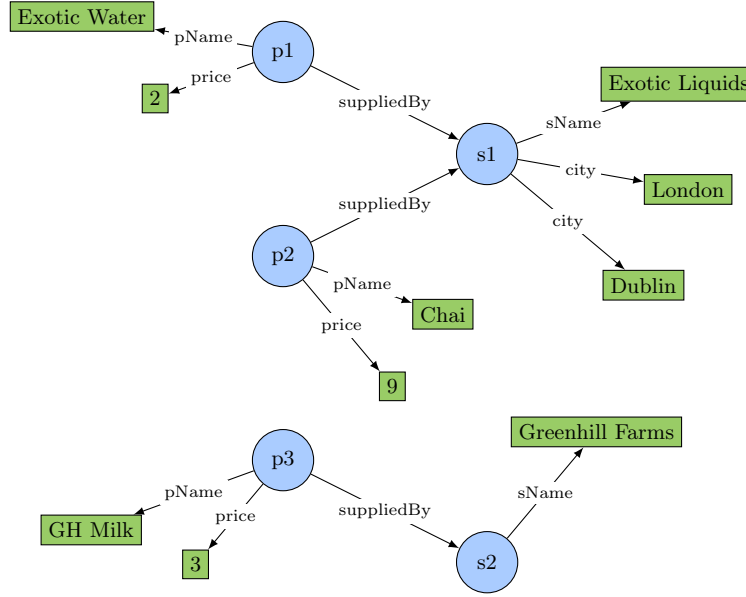
*Another thing worth noting is the fact that all properties labeled `hasSupplier` have been removed, since only the inverse `suppliedBy` is included in the configuration.*

*If we instead consider the concept configuration  $\mathcal{C}_{\text{Prod}}^2$  from Example 9, the resulting concept index  $\bar{\mathcal{D}}_{\text{Prod}}^2 = \mathbf{buildIndex}(\mathcal{C}_{\text{Prod}}^2, \mathcal{D})$  is displayed in Fig. 8. Notice how orders connected to products are now present in the graph, but not their ids.*

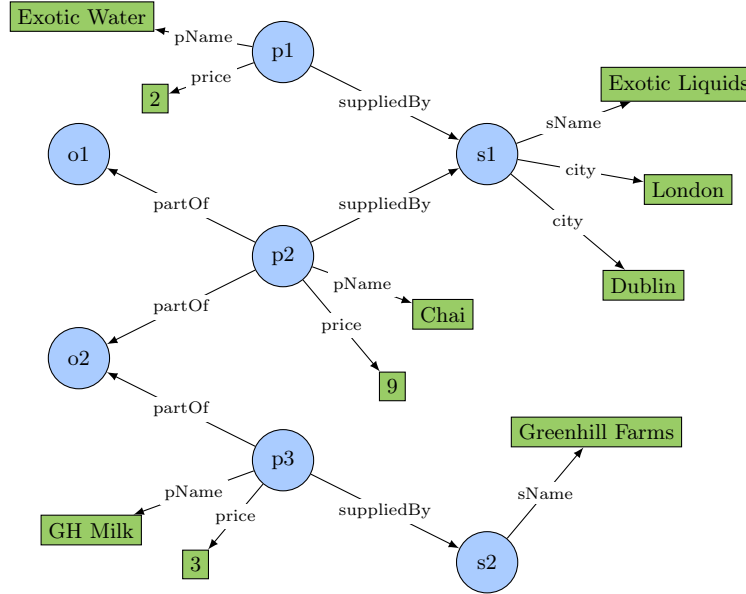
With one concept index for each concept, we can now combine them into the final facet index.

**Definition 18 (Facet Index)** *Let  $\mathcal{G}_N = (C_N, D_N, P_N)$  be a navigation graph,  $\mathcal{D}$  a data source of  $\mathcal{G}_N$ , and  $\mathcal{Z}$  a index configuration over  $\mathcal{G}_N$ . The facet index defined by  $\mathcal{Z}$  over  $\mathcal{D}$  is a function  $\bar{\mathcal{D}}$  defined for all concepts  $c \in C_N$  which returns  $\bar{\mathcal{D}}_c$ , the concept index of  $c$  defined by  $\mathcal{Z}(c)$  over  $\mathcal{D}$ .*

$$\bar{\mathcal{D}}: c \mapsto \bar{\mathcal{D}}_c \quad (8)$$



**Fig. 7.** Concept index  $\bar{D}_{Prod}^1$  for the product concept defined by the concept configuration  $\mathcal{C}_{Prod}^1$ .



**Fig. 8.** Concept index  $\bar{D}_{Prod}^2$  for the product concept defined by the concept configuration  $\mathcal{C}_{Prod}^2$ .

## 5 Index Usage

After the index has been created, it is ready to provide faceted search support during query sessions. The task we want to use the index for is to calculate facet suggestions for each relevant properties every time the user modifies the query or the focus variable, so we define a function **suggest** which does this.

**Definition 19 (Facet Value Suggestion)** *Let  $\mathcal{G}_N = (D_N, C_N, P_N)$  be a navigation graph,  $\mathcal{Q}$  a query over  $\mathcal{G}_N$  with root concept  $c$ ,  $\mathcal{D}$  a data source of  $\mathcal{G}_N$ ,  $\mathcal{Z}$  an index configuration over  $\mathcal{G}_N$ ,  $\bar{\mathcal{D}}$  the facet index of  $\mathcal{D}$  defined by  $\mathcal{Z}$  and  $p \in P_N$  a local property. Furthermore let  $\bar{\mathcal{D}}_c = \bar{\mathcal{D}}(c)$  and  $\mathcal{C}_c = \mathcal{Z}(c)$ . The list of suggested values for  $p$  given  $\mathcal{C}_c$ ,  $\bar{\mathcal{D}}_c$  and  $\mathcal{Q}$ , denoted  $\text{suggest}(\mathcal{C}_c, \bar{\mathcal{D}}_c, \mathcal{Q}, p)$  is given by*

$$\text{suggest}(\mathcal{C}_c, \bar{\mathcal{D}}_c, \mathcal{Q}, p) = \text{Ans}_v(\mathcal{Q}_{ex}, \bar{\mathcal{D}}_c)$$

where

$$\mathcal{Q}_{ex} = \text{extend}(\text{prune}(\mathcal{Q}, \mathcal{C}_c), p, v).$$

The standard way of calculating facet value suggestions is to extend the partial query with each relevant property by using the **extend** function, and run each extended query over the data source. **suggest** does the exact same thing, but it first prunes the query with respect to  $\mathcal{C}_c$  to avoid querying for data that is not included in the concept index.

**Example 11** *Assume we have created the concept index  $\bar{\mathcal{D}}_{Prod}^1$  from Example 10 based on the product concept configuration  $\mathcal{C}_{Prod}^1$  from Example 9. We will now calculate suggested values for the local property  $pName$ , given different queries from Example 7.*

$$\begin{aligned} \mathcal{Q} = (\mathcal{G}_{Q_1}, h_{Q_1}, \mathcal{F}) &\Rightarrow \\ \text{suggest}(\mathcal{C}_{Prod}^1, \bar{\mathcal{D}}_{Prod}^1, \mathcal{Q}, pName) &= \{\text{Exotic Water}, \text{GH Milk}\} \end{aligned} \quad (9)$$

$$\begin{aligned} \mathcal{Q} = (\mathcal{G}_{Q_1}, h_{Q_1}, \mathcal{F}^*) &\Rightarrow \\ \text{suggest}(\mathcal{C}_{Prod}^1, \bar{\mathcal{D}}_{Prod}^1, \mathcal{Q}, pName) &= \{\text{Exotic Water}, \text{GH Milk}, \text{Chai}\} \end{aligned} \quad (10)$$

$$\begin{aligned} \mathcal{Q} = (\mathcal{G}_{Q_2}, h_{Q_2}, \mathcal{F}^*) &\Rightarrow \\ \text{suggest}(\mathcal{C}_{Prod}^1, \bar{\mathcal{D}}_{Prod}^1, \mathcal{Q}, pName) &= \{\text{Exotic Water}, \text{Chai}\} \end{aligned} \quad (11)$$

However, if we instead used another concept configuration  $\mathcal{C}_{Prod}^2$  during index construction, resulting in the more extensive concept index  $\bar{\mathcal{D}}_c$ , we get different results for all of the queries.

$$\mathcal{Q} = (\mathcal{G}_{Q_1}, h_{Q_1}, \mathcal{F}) \Rightarrow \text{suggest}(\mathcal{C}_{Prod}^2, \bar{\mathcal{D}}_{Prod}^2, \mathcal{Q}, pName) = \{\text{GH Milk}\} \quad (12)$$

$$\mathcal{Q} = (\mathcal{G}_{Q_1}, h_{Q_1}, \mathcal{F}^*) \Rightarrow \text{suggest}(\mathcal{C}_{Prod}^2, \bar{\mathcal{D}}_{Prod}^2, \mathcal{Q}, pName) = \{\text{GH Milk}, \text{Chai}\} \quad (13)$$

$$\mathcal{Q} = (\mathcal{G}_{Q_2}, h_{Q_2}, \mathcal{F}^*) \Rightarrow \text{suggest}(\mathcal{C}_{Prod}^2, \bar{\mathcal{D}}_{Prod}^2, \mathcal{Q}, pName) = \{\text{Chai}\} \quad (14)$$

For all of the three first queries, the part of the query related to orders (?o1 and ?int2) is removed in the pruning process, since there are no orders in  $\mathcal{C}_{Prod}^1$ , hence the result includes Exotic Water even though that product is not included in any orders. Equation 9 returns all products except for Chai, since its price is too high for the filter, but when this filter is removed in Equation 10, all products are returned. In Equation 11, the query asks for suppliers with a city, so GH Milk is removed from the result since its supplier has no registered city. The three remaining equations display the results for the same three queries just with a more extensive concept index. In these cases only ?int2 is removed in the pruning process, while ?o1 is kept. Hence products must be part of an order in order to be returned.

## 6 Index Representation

Even though the facet index is stored as data graph in the description of our system, it is also possible to store the index in a tabular and more denormalized format. Doing this will in general require more space, but it will also reduce the data retrieval time, since time-consuming joins are not required anymore.

Instead of storing each concept index as a graph, each concept and its corresponding concept index is represented by a table. Each column of this table represents one specific query variable in the concept configuration, and each line represents one possible assignment to the query variables, showing combinations of values which can occur. The variables are also optional, so NULL values will occur if data is missing, i.e., if there is no possible assignment for a specific variable in the concept configuration.

**Example 12** Table 1 shows what the concept index  $\bar{\mathcal{D}}_{Prod}^1$  looks like in tabular format. This table is fairly straightforward, since each product gets one row in the table, however, it is interesting to see how data related to the supplier s1 is duplicated in column ?s1, ?str1 and ?str2, since both p1 and p2 are supplied by it.

?p1	?s1	?str1	?str2	?str3	?int1
p1	s1	Exotic Liquids	London	Exotic Water	2
p2	s1	Exotic Liquids	London	Chai	9
p3	s1	Greenhill Farms	NULL	GH Milk	3

**Table 1.** Table representation of the concept index  $\bar{\mathcal{D}}_{Prod}^2$

Table 2 shows what the concept index  $\bar{\mathcal{D}}_{Prod}^2$  looks like in tabular format. This concept index is larger than  $\bar{\mathcal{D}}_{Prod}^1$ , and if we compare to table 1 we see that the number of columns has increased by one, while the number of rows has increased by four. p1 is represented twice, since its supplier is located in two different cities.

The same is true for  $p2$ , but in addition it is also part of two different orders, so there are  $2 \times 2 = 4$  possible variable assignments rooted in  $p2$ . This gives an idea of how fast the size of the index grows when more than one value can take multiple data values.

The NULLs appearing in both tables indicate that values are missing, compared to what the concept configuration allows. To make sure that rows like these are not omitted, the SPARQL query which actually populates the tables must contain one *OPTIONAL* clause for each variable and its descendants. This can result in rows where large parts of the columns contain NULLs. In fact, if an individual is disconnected from all other nodes in the data graph, all columns except for the root columns will contain NULLs.

?p1	?s1	?str1	?str2	?str3	?int1	?o1
p1	s1	Exotic Liquids	London	Exotic Water	2	NULL
p1	s1	Exotic Liquids	Dublin	Exotic Water	2	NULL
p2	s1	Exotic Liquids	London	Chai	9	o1
p2	s1	Exotic Liquids	London	Chai	9	o2
p2	s1	Exotic Liquids	Dublin	Chai	9	o1
p2	s1	Exotic Liquids	Dublin	Chai	9	o2
p3	s2	Greenhill Farms	NULL	GH Milk	3	o2

**Table 2.** Table representation of the concept index  $\bar{\mathcal{D}}_{Prod}^2$

In order to use these table-based concept indices when calculating suggestions, the system must transform the pruned and extended SPARQL query into a query over the tables, using e.g. SQL. Filters applied to variables in the query are then transformed into filters over single columns, which is a common pattern in e.g. SQL.

The method presented, which describes how the data is flattened into tables is related to how dedicated search engines like Lucene stores the data in order to optimize data retrieval.

## 7 Existential Variables

As seen in Table 2, the memory footprint increases very quickly when the data contains multiple values or individuals which all can be assigned to the same variable. We can avoid this problem by introducing what we call existential variables in the configuration. The idea is to reduce the data size by not storing all possible assignment values of existential variables, but rather just store a boolean value which indicates whether such a value is present or not.

Table 3 shows what the tabular representation of the concept index  $\bar{\mathcal{D}}_{Prod}^2$  would look like if the variable  $?o1$  was set to be existential.  $p2$  now only spans over two rows, compared to the four in the original table. The number of rows

?p1	?s1	?str1	?str2	?str3	?int1	?o1
p1	s1	Exotic Liquids	London	Exotic Water	2	0
p1	s1	Exotic Liquids	Dublin	Exotic Water	2	0
p2	s1	Exotic Liquids	London	Chai	9	1
p2	s1	Exotic Liquids	Dublin	Chai	9	1
p3	s2	Greenhill Farms	NULL	GH Milk	3	1

**Table 3.** Table representation of the concept index  $\bar{\mathcal{D}}_{Prod}^2$  if ?o1 is an existential variable.

has been reduced by a factor of two, which equals number of orders p2 is a part of.

However, the introduction of existential variables limits the system: Users cannot add value filters to variables which are existential, nor can they get lists of suggested values for the variable. But what the users can do is to add existential filters and ask whether values for a variable exist.

We are still in the early stages of using existential variables, but so far it seems promising to use the feature on concept variables rather than on datatype variables. The reason for this is because the VQSs rarely support filters on concept variables, since that would require the user to know the URI of specific individuals. In fact, if we take a look at Table 3, it still contains all of the interesting data, i.e. data values from Table 2.

## 8 Correctness Analysis

The overall quality of the suggested values given to the user depends on the relationship between the facet index and the original data source, (defined by the index configuration,) and the partial query the user works on. If the concept configuration is large enough to cover the entire partial query, it means that the index contains enough data to compute the correct suggestion values, i.e. the same values one would get by querying the underlying data source. However, the interesting case is when the partial query is not fully covered by the concept configuration, so that the query has to be pruned in order to reduce it to something that fits the data. Will the values suggested by the system be useful?

Assume we have a partial query  $\mathcal{Q}$  with a concept variable  $?v$  of type  $c$  that is  $p$ -related to a datatype variable  $?d$ , and we want to add a filter on  $?d$ . We ask the system for suggested values, and expect it to return positive and negative suggestions. Positive suggestions are data values which if applied as a filter to  $?d$  return non-empty answers, while negative suggestions return nothing.

We assume that the system knows about all possible data values that are  $p$ -related to an instance of type  $c$ , i.e. all values  $a$  such that there exist an individual  $i$  of type  $c$  and  $i \xrightarrow{p} a$  is an edge in the underlying data source  $\mathcal{D}$ . The set of all such values, denoted  $S_{\mathcal{D}}$ , is the universe of values in the dataset, and hence no suggested value should be outside  $S_{\mathcal{D}}$ .

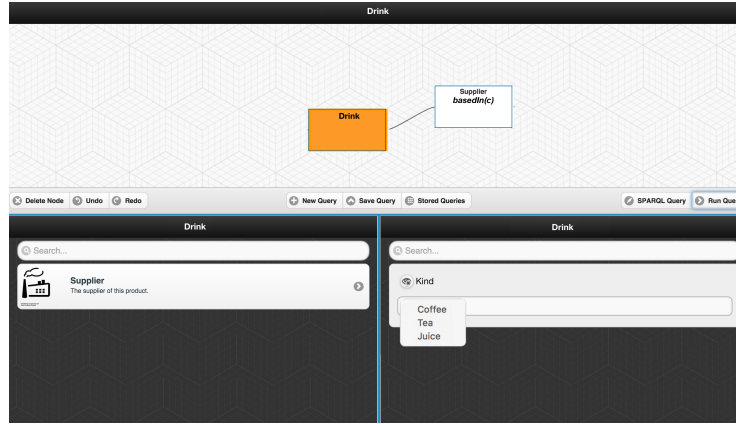
If the system had unlimited time, it could extend  $\mathcal{Q}$  directly to get  $\mathcal{Q}_{ex}$ , and execute it over  $\mathcal{D}$ , to get the set of positive suggestions  $S_p$ . However, this is not the case, and the system has to use the subindex  $\bar{\mathcal{D}}_c$  to calculate suggestion values. Even though  $\bar{\mathcal{D}}_c$  is smaller than the original data source, it is required that it contains all individuals and their  $p$ -relations to values in  $S_{\mathcal{D}}$ . Hence, since the pruned and extended query  $\mathcal{Q}'_{ex}$ , made by our system, is less restrictive than  $\mathcal{Q}_{ex}$ , our system will calculate a set of positive suggestions  $S'_p$  which is larger than  $S_p$ . However, since  $S_n = S_{\mathcal{D}} \setminus S_p$  and  $S'_n = S_{\mathcal{D}} \setminus S'_p$ , we know that  $S'_n < S_n$ .

To sum up, this means that any negative suggestion from our system is indeed going to be negative, while a positive suggestion may not necessary be so. This means that the user has to query over the original data source in order to figure out whether a positive suggestion is actually positive.

With a decent size of the concept configuration, it is also likely that users will accept the positive suggestions given to them, since it may require too much knowledge about the data, and too much reasoning to prove that the value is indeed negative.

## 9 Implementation

A prototype of the system described in this paper has been implemented as a module in the backend of OptiqueVQS. The system works as intended, and it aids the user during query construction by suggesting filter values for relevant properties. The facet index is stored in a relational database in the tabular format described in Section 6. A screenshot of the system in action is displayed in figure 9.



**Fig. 9.** Screenshot showing OptiqueVQS using the implemented faceted search functionality. Coffee and Tea are suggested values, while Juice is not.

## 10 Conclusions and Future Work

This paper defines a theoretical framework, and a detailed description of how to construct and use a scalable and highly configurable index structure to support faceted search over general RDF graphs.

The work is still ongoing, and several possible improvements/extensions have been suggested. Some of the improvements are implementation tasks, like e.g. moving the data over to a dedicated search engine like Lucene in order to speed up the system, while other tasks are larger scientific problems:

1. One limitation of our system is that individuals can only belong to one concept. We would like to extend the homomorphism definition in order to support individuals of more than one type.
2. Another interesting extension to the system is subclass axioms. Is it possible to add subclass relationships between concepts? Can one reuse the concept index of the subclasses instead of constructing a separate concept index for the superconcept.
3. Explore how the size of the indexes can be reduced without sacrificing too many features. The use of existential variables and bucketing techniques are possible solutions.
4. The configuration determines a compromise between index size and accuracy of facet value suggestions. User studies are needed to assess the impact of the accuracy of the suggestions on the perceived usability of the system.

## References

1. Marcelo Arenas, Bernardo Cuenca Grau, Evgeny Kharlamov, Šarūnas Marciuška, and Dmitriy Zheleznyakov. Faceted search over RDF-based knowledge graphs. *Web Semantics: Science, Services and Agents on the World Wide Web*, 37:55–74, 2016.
2. Sean Bechhofer and Ian Horrocks. Driving user interfaces from fact. In *DL2000, International Workshop on Description Logics*, 2000.
3. Josep Maria Brunetti, Roberto García, and Sören 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.
4. Ahmet Soylu, Martin Giese, Ernesto Jimenez-Ruiz, Guillermo Vega-Gorgojo, and Ian 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.
5. Daniel Tunkelang. Faceted search. *Synthesis lectures on information concepts, retrieval, and services*, 1(1):1–80, 2009.