Schema-agnostic SPARQL-driven Faceted Search Benchmark Generation

Claus Stadler[a], Simon Bin[a], Lisa Wenige[a], Lorenz Bühmann[a], Jens Lehmann[a,b,c]

[a] *Institute for Applied Informatics, Goerdelerring 9, Leipzig, Germany*
[b] *University of Bonn, Endenicher Allee 19a, Bonn, Germany*
[c] *Fraunhofer IAIS, Zwickauer Straße 46, Dresden, Germany*

Abstract

In this work, we present a schema-agnostic faceted browsing benchmark *generation framework* for RDF data and SPARQL engines. Faceted search is a technique that allows narrowing down sets of information items by applying constraints over their properties, whereas facets correspond to properties of these items. While our work can be used to realise real-world faceted search user interfaces, our focus lies on the construction and benchmarking of faceted search queries over knowledge graphs. The RDF model exhibits several traits that seemingly make it a natural foundation for faceted search: all information items are represented as RDF resources, property values typically already correspond to meaningful semantic classifications, and with SPARQL there is a standard language for uniformly querying instance and schema information.

However, although faceted search is ubiquitous today, it is typically not performed on the RDF model directly. Two major sources of concern are the complexity of query generation and the query performance. To overcome the former, our framework comes with an intermediate domain-specific language. Thereby our approach is *SPARQL-driven* which means that every faceted search information need is intensionally expressed as a single SPARQL query. In regard to the latter, we investigate the possibilities and limits of real-time SPARQL-driven faceted search on contemporary triple stores. We report on our findings by evaluating systems performance and correctness characteristics when executing a benchmark generated using our generation framework.

All components, namely the benchmark generator, the benchmark runners and the underlying faceted search framework, are published freely available as open source.

Keywords: Faceted Search, Benchmark, SPARQL, RDF, Benchmark Generator, Triple Store

1. Introduction

Faceted browsing is ubiquitous on the Web today. Most if not all major online shops and media platforms provide at least some faceted browsing features to navigate their products or – more specifically – the data records about them. Typical examples include support for filtering videos by length, music by genre, or more generally, products by relevant features. Faceted search is a technique that facilitates exploratory search by allowing for narrowing down sets of information items by applying constraints over their property values, whereas facets correspond to properties of these items. However, many of these faceted search interfaces are based on systems that require tailoring of datasets – i.e. manual specification of what facets and values to show to users and how the input data relates to them. Such approaches circumvent flexible ad-hoc exploration of datasets.

In contrast, the RDF model exhibits several traits that seemingly make it a natural foundation for faceted search: all information items are represented as RDF resources, property values typically already correspond to meaningful semantic classifications, and with SPARQL there is a standard language for uniformly querying instance and schema information. Furthermore, RDF was designed to enable the construction of knowledge graphs (KG) that capture relations between items of arbitrary type thereby exploiting web technology.

The idea of Semantic Faceted Search (SFS) systems is exactly to utilise the flexibility of the RDF model for faceted search. However, although several SFS systems with different features and degrees of expressivity have been proposed, there are only few works on benchmarking faceted search performance on RDF. Among the work concerned with benchmarking, to the best of our knowledge, each of the existing approaches is tied to a specific schema. Conversely, none is *schema-agnostic*, i.e. can operate on an arbitrary given schema. However, w.r.t. usability, it is beneficial to know in advance whether faceted search can be interactively performed on a given dataset, which limits acceptable response times to roughly one second.

Additionally, most of the available SFS systems are primarily designed as applications in contrast to libraries,

Email addresses:
cstadler@informatik.uni-leipzig.de (Claus Stadler),
jens.lehmann@iais.fraunhofer.de (Jens Lehmann)

which makes re-use and evaluation of existing tools difficult. Furthermore, SFS are generally not interoperable due to the lack of common APIs or intermediate languages. Yet, SPARQL as a query language facilitates interoperability between RDF stores and is suitable to express the information needs of faceted search (cf. Section 4). Hence, in this work we focus on the *generation of benchmarks* for assessing performance and correctness of triple store performance w.r.t. given datasets and SPARQL query loads generated from simulated interaction with a real-world SFS engine. In contrast to other benchmarks that assess triple stores, our goal is to specifically study the performance of triple stores w.r.t. workloads of SPARQL queries tied to the faceted search paradigm.

This work builds upon the ideas presented in [1] which describes several types of possible interactions with a SFS. There, the outcome was a set of manually crafted query templates for simulating a faceted search user session on a specific schema. In *this* work we present significant advances featuring a comprehensive automatic benchmark generator that explores a dataset in a schema-agnostic way based on a library of functions that advance the state of a faceted search session in various ways. The state of such a session determines the set of SPARQL queries that are generated. Thus, the sequences of SPARQL queries obtained by repeatedly advancing the session state form a generated benchmark.

For this purpose, we built a comprehensive framework for SPARQL-driven faceted search named *Facete*. The most essential components are the *framework core* and the *benchmark generator*. To test the validity of our framework, a faceted search framework *application for end-users* is also available. The latter features a text-based user interface. The framework's core features a model for faceted search queries together with several translations to SPARQL queries in order to satisfy essential information needs of the faceted search paradigm. The model and the translations are detailed in Section 4. The benchmark generator and the user application are both built on the same core and thus make use of the same model for faceted search queries and the corresponding SPARQL query generation capabilities. As a result, in the context of this work the user application serves as a demonstrator that our system indeed allows for real-world SPARQL-driven faceted search and thus testifies to the relevance of the described system. Furthermore, the user application not only enables a user to browse facets, facet values and matching values of a given dataset but it also allows for viewing the underlying SPARQL query strings which are the same ones the benchmark generator emits. Although the focus lies on the *Facete* benchmark generator, the *Facete* user application can be seen as a complementary interactive verification and debugging tool that allows one to manually inspect the generated queries. Note that the framework's core is independent of any user interface. Whereas many related works on faceted search have strong ties to graphical user interfaces, in this work we describe a model-driven approach to faceted search. This model is intended to enable (SPARQL-driven) exploratory search over RDF data also for machines. Our benchmark generator is one such implementation.

In detail, our contributions are as follows:

- A formal description of a model for faceted search with corresponding translations to SPARQL queries that satisfy faceted search information needs. Most notably, we detail the construction of SPARQL queries that intensionally capture facet counts, facet value counts and matching values under a given set of constraints.
- Implementation of these techniques in the core of the SPARQL-driven faceted search framework *Facete* which is used as a building block to realise the benchmark.
- Design and implementation of a schema-agnostic benchmark generation framework within *Facete*[1], which allows for highly configurable query generation based on customisable distributions of transition types on arbitrary datasets.
- Performance and correctness evaluation of contemporary triple stores with regard to the faceted browsing paradigm.
- As a side contribution, we also present a text mode user interface for faceted search which is also built on the *Facete* framework's core. This demonstrates that the engine is suitable for real-world applications and the generated SPARQL queries actually conform to the faceted search paradigm.

The remainder of the paper is structured as follows: First, in Section 2 we present related work and position our approach in it. Afterwards, in Section 3 we introduce RDF and SPARQL and on this basis formalise fundamental notions for faceted search query generation as used in our benchmarking framework. Subsequently, in Section 4 we first propose a model for faceted search and detail the generation of SPARQL queries from it. The actual benchmark generation is described in Section 5, where we first present the conceptual grounding followed by a description of the implementation. Our findings when executing an exemplary benchmark generated by our system are reported on in Section 6. Finally, we conclude in Section 7 and also point out directions for future work.

2. Related Work

There are two lines for evaluating faceted search systems in general: Performance benchmarking and usability studies. The latter requires a user interface and the former

[1] `https://github.com/hobbit-project/facete3`

is typically tied to a specific user interface and/or a specific dataset and is thus difficult to generalise.

There is a considerable amount of benchmarks available to test the general performance of triple stores (e.g., LUBM [2], SP2 [3], BSBM [4], WatDiv [5] and Geographica [6]). However, specifically for benchmarking faceted search performance on triple stores, to date, there exists only FacetBench[2] and [1].

Our work is based on [1], which discusses fundamental considerations for faceted search benchmarking. However, the benchmark that resulted from that work is a set of manually written SPARQL query templates that resemble faceted search queries and that are tied to a specific schema. In this work, we have fully automated the process of benchmark generation, as detailed in Section 5.

Performance benchmarking aside, there is a large body of research for faceted search systems in general and the more recent semantic faceted search systems, especially w.r.t. user interfaces and query formulation. We do not consider user interfaces in this work other than grounding our benchmark in hypothetical user interactions that could be done through a user interface.

The first faceted search approaches were developed in the 1990s for information retrieval systems. They combined the paradigms of *structured retrieval* and *similarity-based ranking* by leveraging the benefits of constraint-based querying over large data sources. Since then, the approach has proven to be tremendously useful for search applications. Hence, faceted browsing is ubiquitous on the Web today. Most if not all major online applications provide at least some faceted browsing features to navigate the metadata records of their items [7]. This is why this approach has not only been applied for unstructured text documents, but also for knowledge graph data.

Two high-level approaches can be identified for Semantic Faceted Search (SFS) systems: whether SPARQL is used as a protocol to communicate with a backend, and whether pre-computed indexes are used.

Furthermore, a distinction between *SPARQL-based* and *SPARQL-driven* can be made: We refer to an approach that answers a faceted search information need as SPARQL-driven if that approach's result is intensionally specified using a single SPARQL query. As a consequence, SPARQL-driven approaches allow indirectly operating on the faceted search result sets by means of SPARQL query transformations such as slicing, filtering or the addition of extra joins.

A selection of SFS user interfaces, including our *Facete* user application, is depicted in Figure 1. Interesting approaches in the category of index-based systems are *Broccoli* [8] and *Grafa* [9]. However, they do not use SPARQL to communicate with the database system. Instead, the former uses a custom engine, and the latter is based on setting up a Lucene[3] index. Both systems allow for very fast response times even when large portions of the data match the facet selection. Some further prototypes facilitate indexing and faceted browsing of RDF data [10, 11, 12, 13, 14, 15, 16, 17] as well. But since these systems require an index at runtime, they offer limited flexibility regarding the kinds of queries that can be posed. Hence, entity types or facet combinations are hard-wired into the application and can neither be easily adapted to changing user needs nor different kinds of data models [18].

As faceted search is concerned with the construction of intensional descriptions that match items by their (indirect) properties, it is in close relation with description logic and first-order logic (FOL), which provide well studied varying degrees of expressivity. Likewise, query-based faceted search is the paradigm introduced with [19] where an intermediate language is used to express faceted search information needs, and LISQL is a proposed language for that purpose. Note, that although SPARQL is comparatively powerful, certain constructs that can be concisely expressed in FOL, such as "for-all" quantifications are cumbersome (yet possible) to express in SPARQL. Another interesting proposal for an intermediate language is SQUALL [20] which introduces a controlled natural language with a translation to SPARQL. Building upon this work, *Sparklis* [21] adds additional functionalities to the faceted search architecture. Besides providing out-of-the-box SPARQL endpoint support it uses a controlled language to combine query builder capabilities with faceted search to produce SPARQL queries. This approach facilitates an impressive trade-off between expressivity and usability, as on the one hand, it was capable of solving complex tasks of the QALD *Statistical question answering over RDF datacubes* challenge[4] as well as providing an intuitive user interface. Sparklis and *SemFacet* [22, 23] both support aggregation over attributes of the matching values which is a prerequisite for many data analytics tasks. We emphasise that the approach to query construction that underpins the computation of facet and facet value counts under a given set of constraints vastly differs from the one for aggregation of an attribute's values, since the query pattern has to be assembled in a different way. We give one possible construction in Section 4.

Furthermore, [22] introduces a conceptual decoupling of a graphical user interface from query generation by introducing an abstract model for a (basic) faceted search interface. Our work follows the same spirit, however, we put more emphasis on faceted search information needs: In our conceptual model a hypothetical user interface answers faceted search information needs from a given faceted search session state.

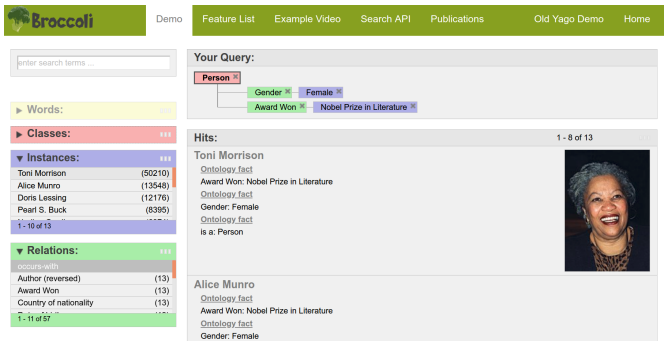
Systems that are not directly faceted search applications but make use of some of that functionality are (visual) query builders. Again, one can in general distinguish between formal languages which may have a visual notation,

[2] https://github.com/GeoKnow/GeoBenchLab/tree/master/FacetBench
[3] https://lucene.apache.org/
[4] http://qald.aksw.org/index.php?x=challenge&q=6

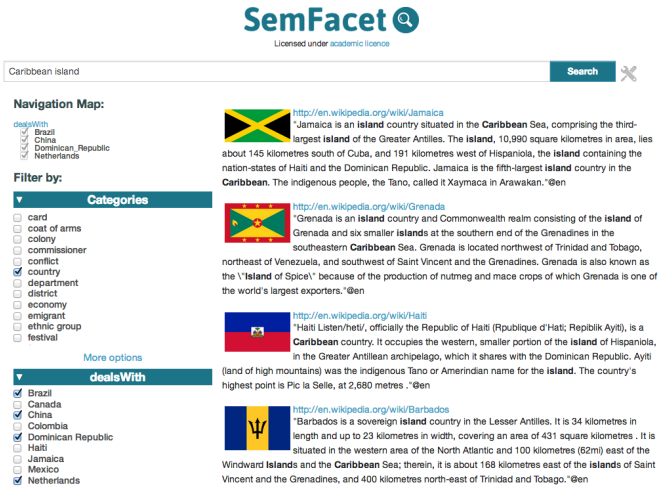
Table 1: Feature comparison of selected Semantic Faceted Search systems. Assessments are based on their user interfaces and project websites. "Advertised faceted search" API refers to whether a project's web pages or at a minimum the source code's test cases indicate the existence of a programmatic or web API that may enable reuse of SFS functionality in custom Semantic Web applications.

Feature	Broccoli	Grafa	OptiqueVQS	Sparklis	SemFacet	*Facete*
SPARQL interoperability	✗	✗	✓	✓	✗	✓
Tree-shaped queries	✓	✗	✓	✓	✓	✓
Facet counts	✓	✓	✓	✓	✗	✓
Facet value counts	✓	✗	✗	✓	✗	✓
Aggregation functions	✗	✗	✓	✓	✓	✗
Concept disjunction	✗	✗	✗	✓	✗	✗
Negative existential quantification	✗	✗	✗	✓	✗	✗
Source code available	✗	✓	✓	✗	✓	✓
Advertised faceted search API	Web[5]	(Lucene)	programmatic[6]	✗	✗	programmatic[7]

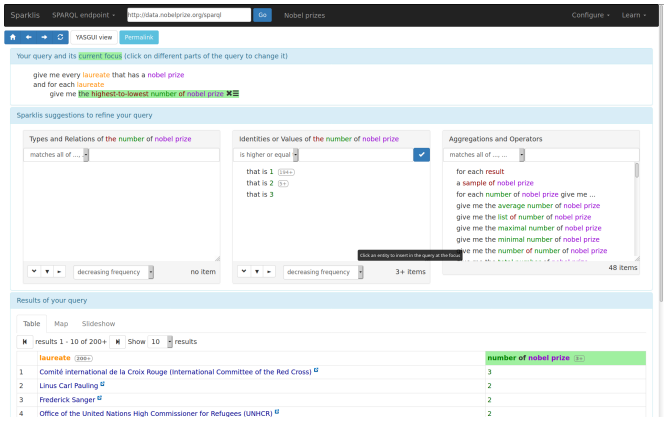
[5] Documentation at `http://broccoli.informatik.uni-freiburg.de/api-desc/` (retrieved 2020-09-29)
[6] Test cases of `https://gitlab.com/ernesto.jimenez.ruiz/OptiqueVQS` (retrieved 2020-09-29)
[7] Landing page of `https://github.com/hobbit-project/facete3` (retrieved 2020-09-29)



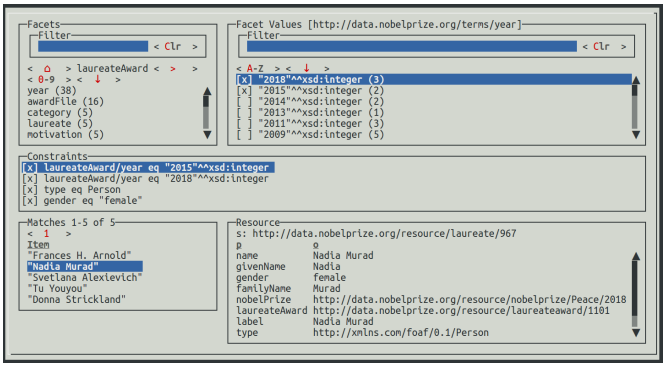
(a) Broccoli: Fast non-SPARQL-based SFS for tree-shaped queries



(b) SemFacet: SPARQL-based SFS for tree-shaped queries with aggregation and ranking support



(c) Sparklis: Allows for construction of complex expressions based on a controlled natural language while featuring a user interface suitable for non-expert users.



(d) Facete: SPARQL-based SFS for tree-shaped queries. Built using the technology presented in this paper.

Figure 1: Depiction of different SFS tools

and query builder systems, that may offer user interfaces with varying degrees of sophistication. Examples of query builder systems are OptiqueVQS [24], QueryVOWL [25] and Virtuoso's iSPARQL[8].

Table 1 summarises features of selected SFS systems, including *Facete*. Tree-shaped queries (arising from traversal along properties) and facet (value) counts are relevant features supported by most SFS systems and therefore in the primary scope of our benchmark generation framework.

Ongoing research is the efficient combination of query containment and query caching approaches with the goal of improving SPARQL query performance. For SFS, this would allow for the injection of (pre-computed) indexes as well as their on-demand creation [26, 27].

3. Preliminaries: RDF and SPARQL

The Resource Description Framework (RDF) is a W3C standard for data interchange[9]. The most fundamental notions are as follows: Let there be pairwise disjoint sets of IRIs I, blank nodes B and literals L. Further, let the set of *RDF terms* be $T := I \cup B \cup L$. The set of *concrete* RDF terms is denoted by $IL := I \cup L$. An *RDF graph* G is defined as $G \subseteq (I \cup B) \times I \times T$, whereas the elements of this set are called *RDF triples*.

Consequently, a triple t is a three-tuple whose components in order are referred to as *subject*, *predicate* and *object*, abbreviated as s, p and o, respectively. Likewise, the set of subjects, predicates and objects of an RDF graph G are defined as the sets $\{X|(s,p,o) \in G\}$ with $X \in \{s,p,o\}$.

The following excerpt from the RDF specification[10] clarifies the purpose of these concepts: "Any IRI or literal denotes something in the world [...]. These things are called *resources*"; the term is synonymous with *entity*. Blank nodes do not identify specific resources. Asserting an RDF triple says that some relationship, indicated by the predicate, holds between the resources denoted by the subject and object. This statement corresponding to an RDF triple is known as an *RDF statement*. Statements involving blank nodes say that something with the given relationships exists, without explicitly naming it.

Furthermore, the RDF specification also defines a basic vocabulary together with basic entailment rules. Most prominently, a triple with a predicate of `rdf:type` states that the entity referred to in the subject position is an instance of the one referred to in the object position.

Although RDF graphs are often depicted as conventional labelled graphs, they are formally defined as ternary relations which in turn correspond to directed labelled pseudo graphs. Pseudo graphs allow for multiple edges to exist between a given pair of nodes, as well as for the same node to act as the start and end of an edge. Hence, algorithms for graphs based on the conventional (V, E) model with V a set of vertices, E a set of edges with $E \subseteq V \times V$, are in general not applicable to arbitrary RDF graphs.

SPARQL (a recursive acronym for *SPARQL Protocol and RDF Query Language*) is a W3C standard[11] that devises a protocol and language for querying and updating RDF. Our benchmark generator will produce SPARQL queries that correspond to faceted search and browsing operations on an RDF graph. The generated benchmarks can be executed on any system implementing SPARQL. SPARQL is defined in terms of operations on *RDF datasets*, which are comprised of a default graph and a set of named graphs. However, for the purpose of this work we define RDF dataset to be synonymous to RDF graph. The following is based on [28], which succinctly captures essential notions of SPARQL formally. Note that the succinctness comes at the cost of certain deviations from the W3C specification. A minor difference is that the semantics of SPARQL are defined in terms of bags (a.k.a. multisets) in the W3C specification whereas [28] uses sets. The nature of this deviation is akin to that of SQL and relational algebra. A significant part of our work is concerned with the syntactic construction of SPARQL queries that answer faceted search information needs. For this purpose, we consider the set semantics as an appropriate choice to convey our ideas, however any practical implementation will use bag semantics. Recent studies specifically about SPARQL under bag semantics have been conducted in [29] and [30].

Assume a set of variables V that is distinct from T. A member of the set $(I \cup B \cup V) \times (I \cup V) \times (T \cup V)$ is called a *triple pattern* and it is also the basic form of a *graph pattern*. A special case is ϵ which stands for an empty graph pattern. A SPARQL *condition* is constructed using elements of the set $V \cup I \cup L$, logical connectives ($\neg$, $\wedge$, $\vee$), inequality symbols ($\neq$, $<$, $\leq$, $\geq$, $>$), the equality symbol ($=$), and unary predicates such as `bound` and `isBlank`.

We define $vars(e)$ as the set of all variables occurring in a condition e.

Let R and S be graph patterns and e a SPARQL condition. The syntax of SPARQL graph pattern expression is inductively defined as

$$\epsilon \mid R \mid R \texttt{ AND } S \mid R \texttt{ UNION } S \mid R \texttt{ FILTER } e$$

where each expression is again a graph pattern.

We also use $vars(gp)$ to refer to the set of variables occurring in the graph pattern gp; more precisely its triple patterns and conditions. For this work, we only consider SPARQL evaluation over RDF graphs. The semantics of a graph pattern expression is then defined in terms of an evaluation function $[[\cdot]]_G : GP \rightarrow \Omega$ that yields w.r.t. an RDF graph G for a given graph pattern a set of *solution bindings* Ω. A set of solution bindings is also referred to as (a query's) *result set*.

A solution binding $\mu : V \rightarrow T$ is a partial function from variables V to RDF terms T. We use $\mathrm{dom}(\mu)$ to

[8] `http://vos.openlinksw.com/owiki/wiki/VOS/VirtFCTFeatureQueries`
[9] `https://www.w3.org/RDF/`
[10] `https://www.w3.org/TR/rdf11-concepts/#resources-and-statements`
[11] `https://www.w3.org/TR/2013/REC-sparql11-overview-20130321/`

denote the domain of μ, i.e. the subset of V where μ is defined. Furthermore, for a triple pattern t let $\mu[t]$ yield the triple where the variables of t have been appropriately assigned. Two solution bindings μ_1 and μ_2 are *compatible*, denoted by compatible(μ_1, μ_2), iff for any common variable $v \in \text{dom}(\mu_1) \cap \text{dom}(\mu_2)$ it holds that $\mu_1(v) = \mu_2(v)$. The evaluation functions for the most common operations used in this work are defined as follows:

- $[[\epsilon]]_G = \{\mu_0\}$ where $\text{dom}(\mu_0) = \emptyset$
- $[[t]]_G = \{\mu \mid \text{dom}(\mu) = \text{vars}(t) \wedge \mu[t] \in G\}$
- $[[R \texttt{ AND } S]]_G = [[R]]_G \bowtie [[S]]_G$
- $[[R \texttt{ UNION } S]]_G = [[R]]_G \cup [[S]]_G$
- $[[R \texttt{ FILTER } e]]_G = \{\mu \mid \mu \in [[R]]_G \wedge \mu \models e\}$
  i.e. μ is a model for the condition e

The semantics of epsilon is that of a single solution binding μ_0 whose domain is the empty set.

Thereby, the operators over sets of solution bindings Ω_R and Ω_S are defined as:

- $\Omega_R \bowtie \Omega_S := \{\mu = (\mu_R \cup \mu_S) \mid \mu_R \in \Omega_R \wedge \mu_S \in \Omega_S \wedge \mathit{compatible}(\mu_R, \mu_S)\}$
- $\Omega_R \cup \Omega_S := \{\mu \mid \mu \in \Omega_R \vee \mu \in \Omega_S\}$

Furthermore, given a variable $v \in V$ and set of variables or terms $x_{1..n} \in (T \cup V)$, we write

$$v \texttt{ NOT IN } (x_1 \dots x_n)$$

as a short form of the condition $(v \neq x_1) \wedge \dots \wedge (v \neq x_n)$.

A graph pattern is the main building block for a *SPARQL query*, however it is not a SPARQL query itself. The SPARQL query whose evaluation result is exactly that of its graph pattern is:

```
SELECT * WHERE graph-pattern
```

The construction of appropriate graph patterns for realising faceted search on RDF graphs is a major part of our work. Although as a final step our approach requires grouping and aggregation over our constructed graph patterns, additional formalisation of these aspects of SPARQL would not contribute to further clarity. We therefore assume the reader to be familiar with the `DISTINCT` keyword and basic analytic queries such as:

```
SELECT ?p (COUNT(?o) AS ?c)
WHERE graph-pattern
GROUP BY ?p
```

In the following, we describe a model for representing faceted search queries and a procedure for the construction of appropriate SPARQL graph patterns from them.

4. Semantic Faceted Search Query Generation Model

In this section we first introduce fundamental definitions, especially that of a *faceted query* and a *facet query configuration*. On this basis, we define the *information needs* a faceted search system has to satisfy, namely *matching values*, *facet value counts*, *facet counts*.

The purpose is to present human and/or machine agents with statistics about available information items and their relations under a given set of constraints. This serves as a guide for data exploration because it provides a-priori insights about the effects of adding additional constraints or traversing along the predicates of an RDF graph. We propose a model for faceted search that covers the first four features of Table 1 and thus allows for SPARQL-driven computation of facet counts and facet value counts for sets of resources that are reachable from a starting set via traversal of paths along the predicates of an RDF graph. From the systems we surveyed, only Sparklis and our work exhibit all four of these features. Our realisation of a SPARQL-driven SFS system is based on the syntactic transformations of faceted queries and facet query configurations to corresponding SPARQL queries. The following descriptions are the conceptual basis for task generation within our schema-agnostic SFS benchmark generator. This means that the benchmark generator described in Section 5 only has to alter the state of faceted queries and facet query configurations and the translations described in this section yield the appropriate SPARQL queries.

Our approach treats every predicate of an RDF graph as a *facet*. Consequently, facets are IRIs. In our model, the directions *fwd* and *bwd* are used to indicate whether a triple's object or subject, respectively, corresponds to a facet's value. Furthermore, we require RDF graphs to be blank node free: This can be achieved by replacing blank nodes with IRIs, a process called *Skolemisation*. Skolemisation is an application of RDF graph transformations, which – in practice – can be accomplished using data materialisation (extract-transform-load) and/or data virtualisation (query rewriting) techniques. As a consequence, conceptually, our approach can operate on an RDF graph G' that results from such a transformation of an RDF graph G. Renaming facets, removing less relevant ones or computing new ones can be accomplished using RDF graph transformations as well. Hence, without loss of generality, on the level of SFS query generation we do not have to take these aspects into account.

4.1. Definitions

Definition 1 (relation pattern)
A SPARQL relation pattern is a pair $srp = (gp, \bar{v})$ *comprised of a graph pattern* gp *and a non-empty sequence of variables* $\bar{v} := \langle v_1 \dots v_i \dots v_n \rangle$ *with* $v_i \in V$*. A variable may have multiple occurrences in the sequence. We name the set of variables* $\{v \in \bar{v}\}$ distinguished variables*. We write*

pattern$(srp) := gp$. *Note that we do not require* v_i *to be in* $vars(gp)$.

The reason for naming this a relation pattern is, that given the SPARQL evaluation result $\Omega = [[gp]]_G$ *w.r.t. an RDF graph* G*, the following relation can be constructed from the sequence of variables, where* T *is the set of RDF terms and nil a symbol indicating the absence of a value:*

$$\{(\mu(v_1), \ldots, \mu(v_n)) \mid \mu \in \Omega\} \subseteq (T \cup \{nil\})^n$$

We use the notation SRP^n *to refer to the set of SPARQL relation patterns whose sequence of variables has size* n*. For this work, we need the sets* SRP^1 *and* SRP^3 *which we refer to as unary and ternary relation patterns, respectively. The corresponding SPARQL query* q_{srp^n} *is formed by a syntactic transformation to* `SELECT DISTINCT` v_1 `...` v_n `WHERE` `gp`*. Note that it is not strictly necessary for the sequence of variables that are added to the projection of the SPARQL SELECT query to be unique. Repeated mentions of the same variable in the projection is semantically equivalent to a single mention due to the nature of bindings as partial functions from variables to RDF terms.*

Definition 2 (concept pattern)
A SPARQL concept pattern *is a special case of a SPARQL relation pattern whose sequence of distinguished variables has size 1. The set of SPARQL concept patterns is thus the set* SRP^1*. All notions of SPARQL relation patterns apply, however additionally, given a concept pattern* $sc = (gp, \langle v \rangle)$*, we introduce* $dvar(sc) := v$ *to conveniently refer to the single distinguished variable directly.*

Definition 3 (path and step)
A facet path*, in short* path*, is a possibly empty sequence of* steps*. A* step *is an element of* $I \times DIR \times A$*, with* I *the set of IRIs,* $DIR := \{fwd, bwd\}$ *the set of directions forwards or backwards, and* A *a set of aliases. A step corresponds to a traversal along the set of those triples in the RDF graph whose predicate matches that of the step. For a step* $s = (predicate, direction, alias)$*, we write* $dir(s)$*,* $predicate(s)$ *and* $alias(s)$ *to obtain its predicate, direction and alias, respectively.*

Note that steps are variable-free. Our approach includes translation of paths to graph patterns where variable names are allocated dynamically as to appropriately connect the triple patterns contributed by each step. The alias influences the variable allocation.

Let P *and* S *be the sets of paths and steps, respectively. We define two functions for obtaining a path's parent path and for extracting a path's last step as follows:*

parent $: P \to P \cup \{nil\}$ *with* $parent(\langle s_1 ... s_n \rangle) := \langle s_1 \ldots s_{n-1} \rangle$*,* $parent(\langle s \rangle) := \langle\rangle$ *and* $parent(\langle\rangle) := nil$*.*

lastStep $: P \to S \cup \{nil\}$ *with* $lastStep(\langle s_1 ... s_n \rangle) := s_n$ *and* $lastStep(\langle\rangle) := nil$*.*

Definition 4 (path variable mapping)
The path variable mapping $\varphi : P \to V$ *is an* injective *function that maps paths to SPARQL variables. Consequently, it must exhibit the property that for two given paths* p_x *and* p_y *it holds that* $\varphi(p_x) = \varphi(p_y) \to p_x = p_y$*. In practical terms,* φ *yields for any path a variable that is unique to that path, which in turn establishes a one-on-one relationship between paths and variables.*

Definition 5 (constraint)
A constraint c *is a tuple* $(\psi|e)$*, with* e *a SPARQL condition and* $\psi : vars(e) \to P$ *a mapping of* e*'s variables to paths. Furthermore, let* $vars(c)$ *and* $paths(c)$ *be the sets of all variables and paths mentioned in* c*, respectively.*

For a concrete constraint with a mapping, we write $c = (?variable_1 \to path_1, ?variable_2 \to path_2 \mid condition)$*.*

Definition 6 (constraint variable normalisation)
Let $c = (\psi, e)$ *be a constraint and* φ *be a path variable mapping. We define*

$$\psi_\varphi := \{\varphi(p) \to p | v \in \mathrm{dom}(\psi) \wedge p = \psi(v)\}$$

and

$$e_\varphi = e_{[\{v \mapsto \varphi(p) | v \in vars(e) \wedge p = \psi(v)\}]}$$

The constraint $c_\varphi := (\psi_\varphi, e_\varphi)$ *has thus all variable names tied to the naming scheme of* φ*. This definition is used for the subsequent graph pattern construction.*

Definition 7 (faceted query)
A faceted (search) query fq *is a tuple* (b, C, f) *with* b *a SPARQL concept pattern,* C *a set of constraints, and* f *a path, called focus path. The set of faceted queries is named* FQ*. We refer to* b *as the* base concept pattern *which intensionally describes an initial set of values which are RDF terms. The actual set of values over an RDF graph* G *– also called* b*'s extension w.r.t.* G *– is obtained by evaluating* b*'s corresponding SPARQL query. Each item in the (possibly empty) set of constraints* C *imposes further restrictions on the initial set of values. The focus path transitions from the remaining set of items to a related one. Hence, a faceted query serves as the base for generating the SPARQL query that upon evaluation over an RDF graph* G *yields the set of values reachable via the focus path* f *starting from those resources in* G *that match both the base concept pattern* b *and the constraints in* C*. The most common base concept is* $((?s?p?o), \langle ?s \rangle)$ *which intensionally describes all of an RDF graph's subjects.*

Definition 8 (facet query configuration)
Given a faceted query fq*, a path* p *and a direction* dir*. A facet query configuration* fqc *is a tuple* (fq, p, dir)*. The configuration serves as the base for generating the SPARQL query that yields the facets and facet values in direction* dir *at path* p *w.r.t. the items that match the faceted query* fq*. In this context we refer to* p *as the* facet source path*. The set of faceted query configurations is named* FQC*.*

4.2. Faceted Search Information Needs

The key task of an SFS system is to derive appropriate *relations* from faceted queries and facet query configurations that satisfy the *information needs* of a user w.r.t.

```
  (?root rdf:type ?x)
  FILTER (?x = Character)
AND
  ((?ca1 character ?root) AND
   (?ca1 genre ?ca2))
  FILTER (?ca2 = Scifi)
AND
  ((?ca1 character ?root) AND
   (?ca1 genre ?cb2))
  FILTER (?cb2 = Fantasy)
```

Listing 1: Graph pattern that corresponds to the set of constraints in Figure 2.

$$\begin{aligned}&\texttt{Character}(\textbf{root}) \wedge \\ &\quad \exists ca1\ (\texttt{character}(ca1, \textbf{root}) \wedge \\ &\qquad \exists ca2\ (\texttt{genre}(ca1, ca2) \wedge \texttt{Scifi}(ca2)) \\ &\qquad \exists cb2\ (\texttt{genre}(ca1, cb2) \wedge \texttt{Fantasy}(cb2))\end{aligned}$$

Example 1: First-order logic formula representing the prior graph pattern for matching the values of the faceted query

an RDF graph. We refer to these information needs as *matching values*, *facet value counts* and *facet counts*. Given a faceted query, the set of matching values can be expressed in terms of an appropriately constructed *SPARQL concept pattern*. Facet value counts and facet counts are expressed in terms of a *SPARQL query* based on a facet query configuration. Note that counting requires grouping and aggregation, which is supported by SPARQL queries but not by graph patterns. Yet, facet value counts and facet counts can be derived from a common intermediary graph pattern which we refer to as the *focus-facet-value* graph pattern. This is a SPARQL relation pattern with exactly three variables that, in order, correspond to focus, facet and facet value.

We introduce the letter τ with several subscripts to denote procedures for the construction of a graph pattern from their input. All constructions are w.r.t. a path variable mapping φ. Specifically, we introduce the transformations τ_p^φ for paths, τ_C^φ for sets of constraints and τ_b^φ for concept patterns.

Matching Values

The construction $matchingValues : FQ \to SRP^1$ takes a faceted query as input and yields a corresponding SPARQL concept pattern that intensionally describes the set of matching values. In the following, we introduce the concepts and definitions required to formally define this construction at the end of Section 4.7.

In order to define the construction of the SPARQL query for facet value counts and facet counts, we use an intermediary SPARQL relation pattern: The *focus-facet-value* relation pattern is a ternary relation pattern that relates each resource in the set of focus resources to applicable (facet, facet value) pairs. Let $ffv : FQC \to SRP^3$ be a procedure that yields for a facet query configuration a ternary relation pattern which we call the focus-facet-value (ffv) relation pattern. We refer to the corresponding graph pattern as `ffv-gp`. Without loss of generality, assume that this relation pattern's variables are named focus, facet and value.

Definition 9 (facet value counts query)
The facet value counts for a given facet query configuration fqc are intensionally described using the following SPARQL query $q_{fvc}(fqc)$*:*

```
SELECT ?facet ?value (COUNT(DISTINCT ?focus)
                      AS ?facetValueCount)
WHERE ffv-gp
```

Definition 10 (facet counts query)
The facet counts for a given facet query configuration fqc are intensionally described using the following SPARQL query $q_{fc}(fqc)$*:*

```
SELECT ?facet
     (COUNT(DISTINCT ?value) AS ?facetCount)
WHERE ffv-gp
```

Note that in the simplest case, the SPARQL relation pattern $(\langle ?s, ?p, ?o\rangle, (?s, ?p, ?o))$ qualifies as a focus-facet-value relation pattern, however, as we show shortly, the construction w.r.t. constraints becomes significantly more complex.

In the remainder of this section, we present graph pattern constructions from paths, constraints and SPARQL concept patterns in order to ultimately enable translation of faceted queries and facet query configuration to the SPARQL relation patterns that answer the information needs.

4.3. Considerations for Constraints

Before we describe the actual graph pattern construction, we first present some typical forms of constraints in order to realise restrictions on facet paths. The basic form of a constraint is to impose an equality restriction on a path, such as $(?x \to \langle(\texttt{rdf:type}, fwd, \_)\rangle \mid ?x = Movie)$.

In order to express an existential restriction on a path, we need a condition that evaluates to true for any value reachable by the path. For this purpose we can use the SPARQL condition `BOUND`$(?x)$. Likewise, an (in)equality restriction on an arbitrary path with an arbitrary constant has the form $(?x \to \langle s_0 \dots s_n\rangle \mid ?x \oplus \text{const})$, with $\oplus$ a placeholder for $=, \neq, <, \leq, \geq, >$. Range constraints can

$$
\begin{array}{llllll}
c_1 & (?x & \rightarrow & \langle\,(\texttt{rdf:type}, \mathit{fwd}, x)\rangle & \mid & ?x = \text{Character}) \\
c_2 & (?y & \rightarrow & \langle\,(\texttt{character}, \mathit{bwd}, \text{a1}) \;,\; (\texttt{genre}, \mathit{fwd}, \text{a2})\rangle & \mid & ?y = \text{Scifi}) \\
c_3 & (?y & \rightarrow & \langle\,(\texttt{character}, \mathit{bwd}, \text{a1}) \;,\; (\texttt{genre}, \mathit{fwd}, \text{b2})\rangle & \mid & ?y = \text{Fantasy})
\end{array}
$$

constraint id | constraint variable | step 1 | step 2 | SPARQL condition

Figure 2: A set of example constraints

be built from a logical conjunction of two inequalities, such as $?x \geq \mathit{min} \wedge ?x < \mathit{max}$ with $\mathit{min},\ \mathit{max} \in L$. In practice, the most common case is that a constraint only affects a single path. However, our model allows for constraints involving multiple paths for improved expressivity: As an example, the set of resources for which the birth date is later than the death date can be expressed as a constraint using
$(?x \rightarrow \langle(\texttt{birthDate}, \mathit{fwd}, \_)\rangle\,,$
$?y \rightarrow \langle(\texttt{deathDate}, \mathit{fwd}, \_)\rangle \mid ?x > ?y).$

Steps and Aliases

Consider the RDF graph in Figure 3 and the constraints in Figure 2, which describe characters of movies whose genres include *both* sci-fi and fantasy. The constraints c_2 and c_3 in Figure 2 have the first step in common, i.e. the same predicate, direction and alias. However, the second step differs in alias. It is the use of different aliases that allows for the specification of the set of movies that are both sci-fi *and* fantasy. By assigning unique variables to every distinct path using a path variable mapping φ (Def. 6), the graph pattern shown in Listing 1 can be derived. Thereby, all paths of all constraints are considered to start from the empty path's variable $\varphi(\langle\rangle)$, in this example named $?\mathit{root}$. Note that a SPARQL query that projects only the root variable of the graph pattern (roughly) corresponds to the first-order logic formula shown in Example 1. First-order logic formulas are often used in literature to capture theoretical aspects of faceted search, however there are certain differences to SPARQL (such as open vs closed world assumption) for which a detailed discussion is out of scope. In this work we focus on realising information needs using SPARQL.

A set of constraints thus corresponds to a graph pattern – or more precisely a SPARQL concept pattern – that intensionally describes a set of resources. In the following we first formalise the transformation of constraints to SPARQL concept pattern, and subsequently extend the procedure to faceted queries and facet query configurations.

4.4. Translating Paths to Graph Patterns

The main issue that needs to be tackled by the graph pattern construction from paths is to correctly handle the variable naming. For this purpose we use the path variable mapping φ.

All paths are assumed to start from a common *root* variable obtained from $\varphi(\langle\rangle)$. Each step in a path is assigned a corresponding target variable and yields a triple pattern that must be correctly connected to the target variable of the previous step (or *root*). For any path p' that is either p itself or a transitive parent of p, the corresponding variable of the resulting graph pattern can be obtained simply via $\varphi(p')$.

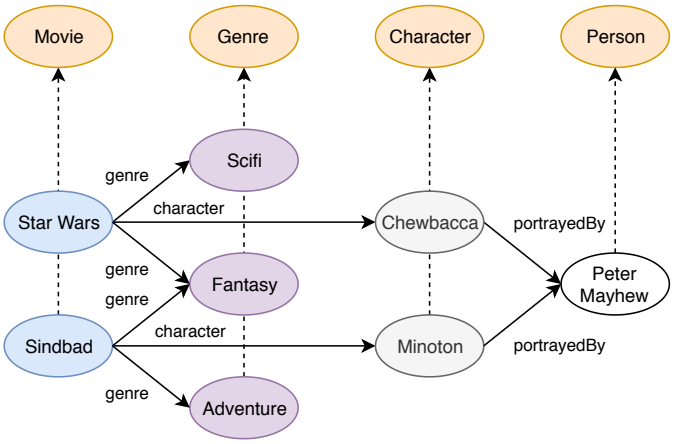


Figure 3: Example RDF graph. Dashed lines denote `rdf:type` relations.

Definition 11 (path to graph pattern translation) *In order to obtain a graph pattern $gp \in GP$ from a path $p \in P$ we introduce the recursive translation procedure $\tau_p^\varphi : P \rightarrow GP$ w.r.t. a path variable mapping φ.*

The translation of paths to SPARQL graph patterns τ_p^φ w.r.t. a given path variable mapping φ is recursively defined as follows:

1. $\tau_p^\varphi(\langle\rangle) := \epsilon$
2. $\tau_p^\varphi(\langle s_1 \ldots s_n\rangle) := \mathit{parent}\ \texttt{AND}\ \mathit{pattern}$
   where
   $$\mathit{parent} = \tau_p^\varphi(\langle s_1 \ldots s_{n-1}\rangle)$$
   and
   $$\mathit{pattern} = \begin{cases} (\mathit{source} \quad \mathit{predicate}(s_n) \quad \mathit{target}), \\ \qquad \text{if } \mathit{dir}(s_n) = \texttt{fwd} \\ (\mathit{target} \quad \mathit{predicate}(s_n) \quad \mathit{source}), \\ \qquad \text{if } \mathit{dir}(s_n) = \texttt{bwd} \end{cases}$$
   with
   $$\mathit{target} = \varphi(\langle s_1 \ldots s_n\rangle)$$
   and
   $$\mathit{source} = \varphi(\langle s_1 \ldots s_{n-1}\rangle)$$

4.5. Translating Constraints to Graph Patterns

Recall that a constraint c is a tuple $(\psi|e)$, with e a SPARQL condition and $\psi : vars(e) \to P$ an association of e's variables to paths. Hence, a constraint is essentially a SPARQL condition involving zero or more variables that are mapped to paths. Given a path variable mapping φ and the path to graph pattern translation τ_p^φ, we can construct the translation $\tau_C^\varphi : C \to GP$ of a single constraint $c = (\psi|e) \in C$ as follows:

Given a set of graph patterns $\{gp_1, \ldots, gp_n\}$, we write

$$\texttt{AND}\,\{gp_1, \ldots, gp_n\} := gp_1 \;\texttt{AND}\; \ldots \;\texttt{AND}\; gp_n \quad .$$

(Note that $\texttt{AND}\,\emptyset := \epsilon$.) Then

$$\tau_C^\varphi(c) := \texttt{AND}\,\left\{\tau_p^\varphi(p) | p \in paths(c)\right\} \;\texttt{FILTER}\; e_\varphi$$

This means that w.r.t. φ, the resulting graph pattern is an AND conjunction from every path's corresponding graph pattern, with the condition appended as a SPARQL FILTER graph pattern.

For a set of constraints C we can now define

$$\tau_C^\varphi(C) := \texttt{AND}\,\{\tau_C^\varphi(c) | c \in C\}$$

Note that although the definition of τ_C^φ is succinct, it may lead to redundancy in the constructed graph patterns, such as shown in Listing 1, where the triple pattern (?ca$_1$ character ?root) exists twice. This redundancy can be eliminated in two ways: Semantics-preserving query optimisation techniques can be employed to post-process the resulting graph pattern expressions. Alternatively, observe that in our case, the construction is based on a conjunction of graph patterns connected with AND and FILTER. Hence, it would be possible to alter the transformation procedure such that triple patterns and conditions are first collected in separate *sets* and to only afterwards connect them with AND. This would yield a graph pattern that is semantically equivalent to the one produced by τ_C^φ without the redundancy.

4.6. Multiple Conditions affecting the same Set of Paths

If a set of constraints C contains multiple members which apply a condition to the same set of paths, there is the following consideration:

Assume two constraints

$$c_1 : (?x \to \langle(\texttt{rdf:type}, fwd, a)\rangle \,|?x = Character)$$
$$c_2 : (?y \to \langle(\texttt{rdf:type}, fwd, a)\rangle \,|?y = Movie).$$

We further assume that under a given φ the variables ?x and ?y – which refer to the same path – are aligned to the variable ?v. Converting the constraint's path to a graph pattern using τ_p^φ and appending the conditions as a SPARQL FILTER leads to the following graph pattern:

```
  ?root rdf:type ?v
  FILTER(?v = Character)
AND
  ?root rdf:type ?v
  FILTER(?v = Movie)
```

which is equivalent to

```
?root rdf:type ?v
FILTER(?v = Character ∧ ?v = Movie)
```

However, the pattern's evaluation over any RDF graph yields an empty set of solution bindings because the condition $?v = Movie \wedge ?v = Character$ is not satisfiable. It is preferable that multiple constraints affecting the same set of paths result in a disjunction of their conditions. In our example this would lead to

```
?root rdf:type ?v
FILTER(?v = Movie ∨ ?v = Character)
```

In order to capture this formally, we introduce the notion of *disjunction reduction* of a set of constraints, which is relevant for Section 4.8. We use the term reduction because if disjunctions of conditions are created then the resulting set of constraints has fewer elements than the original one.

Definition 12 (disjunction reduction)
The disjunction reduction $\cdot^\vee$ of set of constraints C, denoted by $C^\vee$, is obtained by deriving a single new constraint for every subset of C that makes use of the same mapping of variables to paths. The SPARQL condition of such a derived constraint becomes the logical disjunction of the conditions in that subset:

$$C^\vee := \left\{\left(\psi, \bigvee\{e | (\psi_o, e) \in C \wedge \psi_o = \psi\}\right) \mid (\psi, \_) \in C\right\}$$

Accordingly, $C_\varphi^\vee$ denotes a set of constraints whose variables were *first* aligned w.r.t. φ and that was *subsequently* reduced using $\cdot^\vee$. For convenience, we "push" disjunction reduction into the graph pattern construction τ_C^φ to yield $\tau_C^{\varphi\vee}$:

$$\tau_C^{\varphi\vee}(C) := \tau_C^\varphi(C^\vee)$$

The disjunction reduction is thus a post-processing of sets of constraints based on the aforementioned practical considerations. The translation of a set of constraints to a conjunction of graph patterns using τ_C^φ remains unchanged.

4.7. Translating Faceted Queries to SPARQL Concept Patterns

We have almost all notions in place to construct the graph pattern for a faceted query $fq = (b, C, f)$. The only part missing is the proper adjustment of variables in the base SPARQL concept pattern b w.r.t. the variable mapping φ. Without loss of generality, we assume that the set of variables occurring in b's graph pattern is disjoint with the set of variables that paths may be mapped to. Formally, we make the assumption $vars(pattern(b)) \cap img(\varphi) = \emptyset$. In practice, this can be accomplished by appropriate variable renaming or using a reserved prefix for path variable names. Under this assumption, we can combine b's graph pattern with the one obtained from the constraints once

b's distinguished variable is mapped to that of the empty path, and the corresponding transformation τ^{φ}_{scp} becomes:

$$\tau^{\varphi}_{scp}(b) := pattern(b_{[dvar(b)\mapsto\varphi(\langle\rangle)]})$$

We can now define τ^{φ}_{fq} which yields a faceted query's graph pattern:

$$\tau^{\varphi}_{fq}((b, C, f)) := \tau^{\varphi}_{scp}(b) \texttt{ AND } \tau^{\varphi^{\vee}}_{C}(C) \texttt{ AND } \tau^{\varphi}_{p}(f).$$

Finally:

Definition 13 (matching value)
The set of matching values of a faceted query is intensionally captured by the SPARQL concept pattern: matchingValues : $FQ \to SRP^1$ *defined as*

$$matchingValues((b, C, f)) := (\tau^{\varphi}_{fq}((b, C, f)), \langle\varphi(f)\rangle)$$

4.8. Translating Facet Query Configurations to Graph Patterns

So far we have shown how to construct the SPARQL concept pattern that yields a faceted query's matching values. The construction is essentially based on a conjunction of the graph patterns that are obtained by appropriate transformations of a faceted query's constituents. However, the construction of the focus-facet-value relation pattern, from which facet counts and facet value counts can be directly derived, needs additional considerations.

Recall that a facet query configuration *fqc* is a tuple (fq, p, dir). The configuration serves as the base for generating the SPARQL query that yields the facets and facet values in direction *dir* at path p w.r.t. the items that match the faceted query fq. The goal is now to define *ffv* : $FQC \to SRP^3$ which for a facet query configuration yields the of the ternary focus-facet-value (ffv) SPARQL relation pattern.

Figure 4 shows two example configurations, each based on a single constraint and a different parametrisation for focus path and facet path. For each configuration, the graph patterns and their results for evaluation on the example graph in Figure 3 are given. The base concept is left out for simplicity – eventually it is integrated via `AND` analogous to the construction for the matching values.

Example (a) in Figure 4 asks for the immediate outgoing facets of movies. The count for each facet value refers to the number of matching resources in the subject position because of the empty focus path. Note that in this example, two graph patterns are required for capturing all facets, facet values and counts: One specifically for the `rdf:type` facet and one for all other facets.

In Example (b) in Figure 4, the constraint demands matching values to be of type movie. The facet path demands a forward traversal along the *character* predicate. The set of values reachable via this traversal are exactly characters which subsequently have *type* and *portrayed by* facets. The only values of these facets are *Character* and *Peter Mayhew*, respectively. For either facet value, there are the same 2 corresponding movies, which are related to the same 3 different genres. Consequently, because the focus path is set to $\langle(\texttt{genre}, fwd, \_)\rangle$, the facet value count for either facet value is 3.

As can be seen from Example (a) in Figure 4, a facet's contribution to ffv's overall graph pattern depends on the parametrisation. In the following, we define this construction.

Given a facet query configuration *fqc* = (fq, *facetPath*, *facetDir*) with $fq = (b, C, focusPath)$, where b is the base concept pattern and C a set of constraints. The corresponding facets and facet values are computed from the set of RDF triples reachable via the path *facetPath* in the direction *facetDir*. In order to obtain the set of available values for a facet, all constraints affecting it need to be excluded.

For this purpose, we introduce the helper function *affectingConstraints*. For any constraint c having a path that is reachable from the path *facetPath* in direction *facetDir* via a single step s, a relation between the facet $i = predicate(s)$ and c is established:

$$\begin{gathered} affectingConstraints : (P, DIR, \mathcal{P}(C)) \to \mathcal{P}(I \times C) \text{ with} \\ affectingConstraints(facetPath, facetDir, C) := \\ \{(\ predicate(lastStep(path)), c\ ) \\ \mid\ c \in C \wedge path \in paths(c) \wedge parent(path) = facetPath \\ \wedge\ dir(lastStep(path)) = facetDir\} \end{gathered}$$

From *affectingConstraints*'s resulting relation $AfC \subseteq I \times C$ we can obtain the set of facets affected by constraints:

$$\begin{gathered} affectedFacets : \mathcal{P}(I \times C) \to \mathcal{P}(I) \text{ with} \\ affectedFacets(AfC) := \{i \mid (i, \_) \in AfC\} \end{gathered}$$

From AfC we can also derive the function *adjustedConstraints*, which for every affected facet i yields the set of all constraints *except* those affecting i:

$$\begin{gathered} adjustedConstraints : I \times \mathcal{P}(I \times C) \to \mathcal{P}(C), \\ adjustedConstraints(i, AfC) := \{c \mid (j, c) \in AfC \wedge i \neq j\} \end{gathered}$$

As an example, in the facet query configuration (a) of Figure 4, the facet path is the empty path $\langle\rangle$ and the set of constraints contains one item c which is

$$c : (?x \to \langle(\texttt{rdf:type}, fwd, \_)\rangle | ?x = \text{Movie})$$

The focus-facet-value relation pattern should be constructed for the facets in forward-direction relative to the empty facet path. It turns out that the facet path is a parent of the only constraint, and that the direction of the constraint path's last step matches that of the requested facet's direction. Hence, the relation of affecting constraints contains a single tuple:

$$AfC_c = \{(\texttt{rdf:type}, c)\}$$

Consequently, the adjusted constraints for a facet i w.r.t. AfC_c are:

$$adjustedConstraints(i, AfC_c) = \begin{cases} \emptyset & \text{if } i = \texttt{rdf:type} \\ \{c\} & \text{otherwise} \end{cases}$$

Now that we can obtain for any facet its adjusted constraints, we can generate the graph pattern contributions for the focus-facet-value relation:

Given a facet query configuration $fqc = (fq, facetPath, facetDir)$ with $fq = (b, C, focusPath)$, where b is a concept pattern, C a set of constraints, *facetPath* a path and *facetDir* a direction. Let φ be a path variable mapping. Further, let $AfC = affectingConstraints(facetPath, facetDir, C)$ be the relation of (facet-)affecting constraints and $F = affectedFacets(AfC)$ be the set of affected facets.

There is one contributed graph pattern for each facet $i \in F$, and in addition the *residual* graph pattern. The *facet triple pattern* is common to all contributions and defined as:

$$pattern = \begin{cases} (\varphi(facetPath) \quad ?p \quad ?v) & \text{if } facetDir = \texttt{fwd} \\ (?v \quad ?p \quad \varphi(facetPath)) & \text{if } facetDir = \texttt{bwd} \end{cases}$$

The variable $?p$ corresponds to the facets and $?v$ to the facet values. Without loss of generality, we assume that $?p$ and $?v$ are variables that are not mentioned in any other graph pattern of the composition described in the following. Analogous to Section 4.7 this can generally be accomplished by variable renaming.

For every $i \in F$ the pattern contribution is

$$\begin{aligned} facetGP(fqc, i) := {} & \tau^{\varphi}_{scp}(b) \ \texttt{AND} \\ & \tau^{\varphi\vee}_{C}(adjustedConstraints(i, AfC)) \ \texttt{AND} \\ & \tau^{\varphi}_{p}(focusPath) \ \texttt{AND} \\ & \tau^{\varphi}_{p}(facetPath) \ \texttt{AND} \ pattern \\ & \texttt{FILTER}(?p = i) \end{aligned}$$

Note, that $\tau^{\varphi\vee}_{C}$ means that disjunction reduction is performed on the set of constraints passed as arguments. The *residual graph pattern* is created from all the facets connected to the facet path that are *not* affected by constraints. If $F = \emptyset$, then the residual pattern is ϵ, otherwise the residual pattern includes all facets which have not received special handling:

$$\begin{aligned} residualGP(fqc) := {} & \tau^{\varphi}_{scp}(b) \ \texttt{AND} \\ & \tau^{\varphi\vee}_{C}(C) \ \texttt{AND} \\ & \tau^{\varphi}_{p}(focusPath) \ \texttt{AND} \\ & \tau^{\varphi}_{p}(facetPath) \ \texttt{AND} \ pattern \\ & \texttt{FILTER}(?p \ \texttt{NOT IN} \ F) \end{aligned}$$

The graph pattern construction for the focus-facet-value construction under a facet query configuration *fqc* with $i \in F$ can now be formally captured as:

$$\tau^{\varphi}_{fqc}(fqc) := \texttt{UNION}\,\{facetGP(fqc, i)\}\,\texttt{UNION}\,residualGP(fqc)$$

This construction leads to the graph patterns shown in Figure 4. Finally, we define the construction of the ternary focus-facet-value relation pattern as

$$\begin{aligned} ffv &: FQC \to SRP^3 \\ ffv(fqc) &:= (\tau^{\varphi}_{fqc}(fqc), \langle \varphi(focusPath), ?p, ?v \rangle) \end{aligned}$$

where *focusPath* stands for *fqc*'s focus path.

From this ternary relation pattern, we can derive the SPARQL queries for the facet value counts and facet values as defined in Definition 9 and Definition 10.

4.9. Creating Constraints from Facet Values

We now have all the tooling in place to generate the SPARQL query which yields the actual facet values over an RDF graph G from a facet query configuration *fqc*. The final aspect that has to be shown is how the iterative process of a faceted search session can be realised. In essence, the values of such a result set need to be combined with the information in *fqc* in order to create new constraints which leads to a new facet query configuration fqc'.

For a given facet query configuration $fqc = (fq, facetPath, facetDir)$, we can obtain the SPARQL query $\mathtt{q}_{\texttt{fvc}}(fqc)$ for the facet value counts according to Definition 9. Its evaluation $\Omega_{fvc} := [[\mathtt{q}_{\texttt{fvc}}(fqc)]]_G$ yields a set of solution bindings Ω_{fvc} with the variables *facet*, *value* and *count*. Based on Ω_{fvc} we define a partial function that yields for each facet the set of RDF terms that act as the available facet values:

$$\begin{aligned} afv &: I \to \mathcal{P}(T) \\ afv(i) &:= \{\mu(value) \mid \mu \in \Omega_{fvc} \wedge \mu(facet) = i\} \end{aligned}$$

Every facet $i \in dom(afv)$ can be used to extend the *facetPath* by an additional step based on the *facetDir*:

$$facetPath' = facetPath \oplus \langle (i, facetDir, \_) \rangle$$

where $\oplus$ stands for sequence concatenation.

For a facet i we can now easily create new constraints based on the available values $afv(i)$ at $facetPath'$. For example, from two arbitrary values $v_1, v_2 \in afv(f)$ we can construct constraints such as

$$\begin{aligned} c_{equals} &: (\{?x \to facetPath'\}, ?x = v_1) \\ c_{range} &: (\{?x \to facetPath'\}, ?x \geq v_1 \wedge ?x < v_2) \end{aligned}$$

As long as it is ensured that conditions derived from the RDF terms in $afv(i)$ match a non-empty subset of $afv(i)$, the matching values of a new facet query configuration fqc' that includes that constraint will not become empty. Because of the disjunction reduction we can create an arbitrary number of equals constraints using $facetPath'$ and $afv(f)$ and it will "naturally" result in a disjunction of the conditions instead of causing an empty set of matching values.

#	Facet Query Configuration				
	Faceted Query				
	Base Concept	**Constraints**	**Focus Path**	**Facet Source Path**	**Direction**
(a)	$((?s?p?o), \langle ?s \rangle)$	$(?x \to \langle(\texttt{rdf:type}, \textit{fwd}, \_)\rangle \mid ?x = \text{Movie})$	$\langle\rangle$	$\langle\rangle$	fwd
(b)	$((?s?p?o), \langle ?s \rangle)$	$(?x \to \langle(\texttt{rdf:type}, \textit{fwd}, \_)\rangle \mid ?x = \text{Movie})$	$\langle(\text{genre}, \textit{fwd}, \_)\rangle$	$\langle(\text{character}, \textit{fwd}, \_)\rangle$	fwd

#	contribution for facet	**Graph Patterns** P (`SELECT ?p ?v (COUNT(DISTINCT(?f AS ?c))) WHERE` P)	**Facet Value Counts** ?p :?v_1?c_1, …, ?v_n?c_n
(a)	`rdf:type`	(base concept AND) (?root ?p ?v) `FILTER` (?p = `rdf:type`)	• type: Genre 3, Movie 2, Character 2, Person 1
		UNION	
	(residual)	(base concept AND) (?root `rdf:type` Movie) AND ((?root ?p ?v) `FILTER` (?p `NOT IN` (`rdf:type`)))	• genre: Scifi 1, Fantasy 2, Adventure 1 • character: Chewbacca: 1, Minoton: 1
(b)	(residual)	(base concept AND) (?root genre ?f) AND ((?root character ?c) AND (?c ?p ?v))	• type: Character 3 • portrayed by: Peter Mayhew: 3

Figure 4: This figure exemplifies facet value count computation from facet query configurations. The first table shows two independent examples of facet query configurations (a) and (b). Thereby the columns base concept (pattern), constraints and focus path correspond to the components of a faceted query. The second table shows the corresponding generated graph patterns with their respective evaluation results on the example data. In general the base concept pattern is made part of every contribution by appropriately renaming its variables. Both configurations use the same constraint but they differ in the focus and facet source path. Recall that the facet source path determines "from where" to obtain facet values and *dir* whether in forward or backward direction. The focus path determines what to count. Configuration (a) demonstrates, that even though the `rdf:type` facet is restricted to *Movie*, its values are not affected by that constraint. In general, when constructing the graph patterns for the facet values reachable from the facet source path in the given direction, then for every of these facets that carries constraints, these constraints need to be excluded in the graph pattern contribution. In this example, the graph pattern contributed by the `rdf:type` facet is based on an exclusion of constraints on `rdf:type`. The practical consequence is that this enables a user to select additional values of this facet in order to create a disjunction of conditions, such as "movies or characters". However, all other – residual – facet value counts are computed w.r.t. all constraints, in this case the restriction to movies. The complete focus-facet-value relation pattern for each configuration is obtained from the *union* of the contributions. In configuration (b) there does not exist a constraint on any facet reachable from the facet path via character in forward direction, hence there is no need for exclusions and only the residual contribution remains.

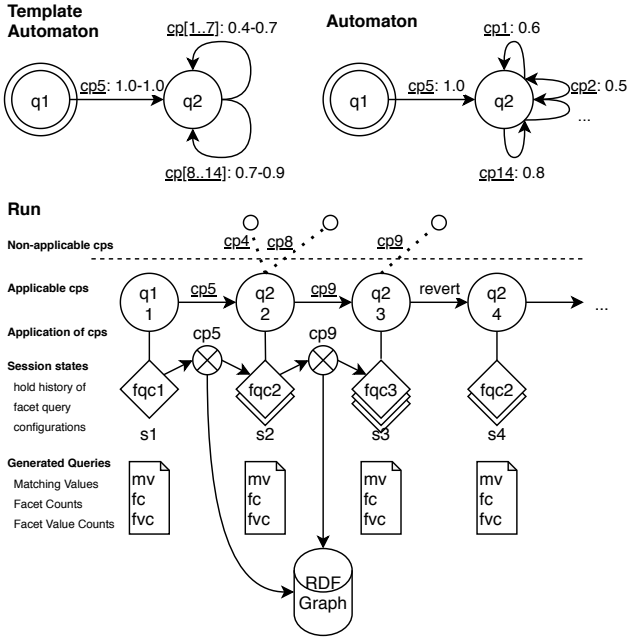


Figure 5: Automata involved in the benchmark generation. A concrete probabilistic automaton is instantiated from the template automaton by randomly picking a concrete value for every range and subsequently normalising them. The first step for the generation of a benchmark scenario is to obtain a run from the automaton. The first state in the run is mapped to an initial stack of facet query configurations. Stacks for subsequent states in the run are computed inductively: In the special case of a *revert* transition the new stack is a copy of original one with the top element removed. Otherwise, the new stack is a copy of the original one with a new element pushed. This new element is the result of a chokepoint function applied to the top of the original stack and the given RDF graph. Error conditions are popping an empty stack and a chokepoint function that yields *nil*. If an error occurs the new stack is the empty stack and the action of any transition will yield again an empty stack. A run is viable if the last state's corresponding stack is non-empty. For a viable run SPARQL queries for the matching value, facet counts and facet value counts are created for every stack's top facet query configuration.

5. The Faceted Search Benchmark Generation Framework

In this section we present our benchmark generation system. In a nutshell, the goal of the benchmark generator is to yield sequences of SPARQL queries that are the result of simulated sessions of interactions with a faceted search system. Because of the SPARQL-driven nature of our approach, the resulting queries correspond to specifications of essential faceted search information needs. In consequence, our approach is representative for SPARQL-driven systems that capture the relations of matching values, facet counts and facet value counts intensionally as SPARQL queries. These essential information needs are not exhaustive and different use cases may require the definition of new ones.

We use the following terminology to describe the simulation: A *scenario* is single simulated session. Within that session an agent performs a sequence of interactions. After each interaction and before the first one the faceted search systems needs to present the agent with answers to a set of information needs. Computing the whole set of answers is referred to as a *task*. For SPARQL-driven faceted search a task comprises a sequence of workloads that are SPARQL queries. Hence, the output of our benchmark generator is a sequence of SPARQL queries that relate to scenarios, tasks and workloads.

The simulation corresponds to a random exploration of an RDF graph under a given a set of possible interaction types which we also refer to as *chokepoints*. The approach to random exploration is divided into a macro and micro level: On the macro level the choice for an interaction type is made. The desired action is driven by a probabilistic automaton. For example, the choice to add a constraint that restricts a path to a numeric value. On the micro level, a concrete path and value w.r.t. an RDF graph are chosen. A possible outcome is that that the macro level decision cannot be implemented on the micro level due to the available data and state of the faceted search session. In such a case, the macro level choice is rejected.

Further conceptual grounding of our work is based on a recent survey of SFS systems conducted by [31], that identified four basic types of actions, namely class-based browsing, property-based browsing, property path-based browsing, and entity type switch. This served as the basis for our previous work [1], where selected common variations of these transition types were chosen to form a *conceptual* faceted search benchmark framework. In total, 14 variations were derived and labelled as *chokepoints* CP1 to CP14. The term *chokepoint* in this context originates from [32] where it is defined as *those technological challenges underlying a benchmark, whose resolution will significantly improve the performance of a product.* Thus, objectively capturing relevant aspects of faceted search is needed in order to identify potential bottle necks.

In the remainder of this section we first describe the probabilistic model used in the macro level to simulate scenarios. Afterwards we describe the different chokepoints that operate on the micro level. Finally, we explain relevant aspects of the benchmark generator implementation and the underlying *Facete* framework.

5.1. Faceted Search Benchmark Generation using Probabilistic Automatons to simulate User Behaviour

In this section we detail the conceptual model that forms the backbone to faceted search benchmark generation. We first formalise the notion of chokepoint. Then we describe the use of a probabilistic automaton to generate so called *runs* which are translated into corresponding faceted query configurations which are finally translated to SPARQL queries using the model introduced in Section 4.

Definition 14 (chokepoint)
Let FQC be the set of faceted search query configurations and $\mathcal{G}$ be the set of all RDF graphs. A chokepoint is a function $cp : FQC \times \mathcal{G} \to FQC \cup \{nil\}$. We refer to a chokepoint as not applicable *w.r.t. given arguments if it yields nil for them.*

The automaton is used as a generator for sequences comprised of states and transitions called *runs*.

The following definitions are adapted from [33][12].

Definition 15 (probability distribution) *Given a finite set S, a probability distribution over S is a function $f : S \to [0, 1]$ such that $\sum_{s \in S} f(s) = 1$. We denote by $\mathcal{D}(S)$ the set of all probability distributions over S.*

Definition 16 (probabilistic automaton) *A probabilistic automaton is a tuple $\mathcal{A} = (Q, \rho_I, \Sigma, \delta)$, where*

- *Q is a finite set of states;*
- *$\rho_I \in \mathcal{D}(Q)$ is the initial probability distribution;*
- *Σ is a finite alphabet;*
- *$\delta : Q \times \Sigma \to \mathcal{D}(Q)$ is the transition function*

The language of an automaton $\mathcal{A}$ is the set of all sequences that can be generated from the alphabet and is denoted by $L(\mathcal{A})$ with $L(\mathcal{A}) \subseteq \Sigma^*$. The elements of $L(\mathcal{A})$ are also referred to as *words*.

A run of $\mathcal{A}$ over a finite (resp. infinite) word $w = \sigma_1\sigma_2 \ldots$ is a finite (resp. infinite) sequence $\bar{r} = q_0\sigma_1 q_1\sigma_2 \ldots$ of states and letters such that (i) $\rho_I(q_0) > 0$, and (ii) $\rho(q_i, \sigma_{i+1})(q_{i+1}) > 0$ for all $0 \leq i \leq |w|$. We define the length of a run, denoted by $|\bar{r}|$, to be equal to the number of transitions and thus equal to $|w|$. We use the notations $\bar{r}_i^q$ and $\bar{r}_j^\sigma$ to refer to the state and letter at the i-th and j-th position, respectively, in the run $\bar{r}$ with $0 \leq i \leq |\bar{r}|$ and $0 \leq j < |\bar{r}|$, respectively.

The probability for a run $\bar{r}$ to be generated by automaton $\mathcal{A}$ is given by $\mathbb{P}^{\mathcal{A}}(r) := \rho_I(q_0) \cdot \prod_{i=1}^{|\bar{r}|} \rho(q_{i-1}, \sigma_i)(q_i)$.

Our model for the state of a faceted search session is a that of a sequence of facet query configurations. In order to realise operations that modify a faceted search session – including a revert operation – we introduce conventional stack operations *top*, *pop* and *push* on sequences. In the following we use the terms sequence and stack interchangeably.

- $top(\langle fqc_0 \ldots fqc_n \rangle) := fqc_n$ returns the top most (last) item with the exception $top(\langle\rangle) := nil$
- $pop(\langle fqc_0 \ldots fqc_n \rangle) := \langle fqc_0 \ldots fqc_{n-1} \rangle$ returns a sequence with the top item removed with the exception $pop(\langle\rangle) := \langle\rangle$ such that *pop* on an empty sequence yields again an empty sequence
- $push(\langle fqc_0 \ldots fqc_n \rangle, fqc_x) := \langle fqc_0 \ldots fqc_n fqc_x \rangle$ returns a new stack with an element fqc_x as the new top.

[12] We leave out the weight function in our description because we do not need it.

Given an initial stack $s_0 = push(\langle\rangle, fqc_0)$ and a run $\bar{r}$, a benchmark is generated as follows: Let $l = |\bar{r}|$. First, inductively compute all s_i for $1 \leq i \leq l$ by applying the chokepoint function referred to by $\bar{r}_i^\sigma$, denoted by cp_{σ_i} to the prior fqc – or revert to a previous state – as follows:

$$s_{i+1} = \begin{cases} \langle\rangle & \text{if } s_i \text{ is empty,} \\ pop(s_i) & \text{if } \bar{r}_i^\sigma = \text{revert} \\ push(s_i, cp_{\sigma_i}(top(s_i), G)) & \text{else.} \end{cases}$$

A run qualifies as *viable* if $top(s_l) \neq nil$. If a run is viable, generate for every facet query configuration at $top(s_i)$ the queries for the facet counts, facet value counts and matching values as described in Section 4. This yields a sequence of related queries which we refer to as *scenario*.

The *initial state* fqc_0 is denoted by a facet query configuration whose base concept is the set of all subjects, the set of constraints is empty, the focus and facet paths are empty paths as well, and the direction is *fwd*.

As a final extension, our framework supports the specification of a template automaton to derive differently weighted probabilistic automata. The template automaton uses for each state transition a numeric range instead of a specific probability. When instantiating an actual probabilistic automaton from the template automaton, for each transition a value is chosen at random within the range and subsequently normalised in order to obtain the probability. The whole process is depicted in Figure 5.

The essence of our benchmark generator is the selection and application of chokepoint functions from a pool of available ones in order to produce different facet query configurations from which the benchmark tasks are created.

In the remainder of this section, we first summarise the chokepoints mentioned in [1] and subsequently present a model that captures their generation. Finally, we present our implementation.

5.2. Chokepoint Library

In this section we present the concrete set of chokepoints that our system supports.

CP1 Property value based transition
Find all instances which, additionally to satisfying all restrictions defined by the state within the browsing scenario, have a certain property value

CP2 Facet path based transition
Find all instances which additionally realise this facet path with any property value

CP3 Facet path value based transition
Find all instances which additionally have a certain value at the end of a facet path. N.b. This is CP1 with a facet path instead of an immediate property

CP4 Property class value based transition
Find all instances which additionally have a property value lying in a certain class

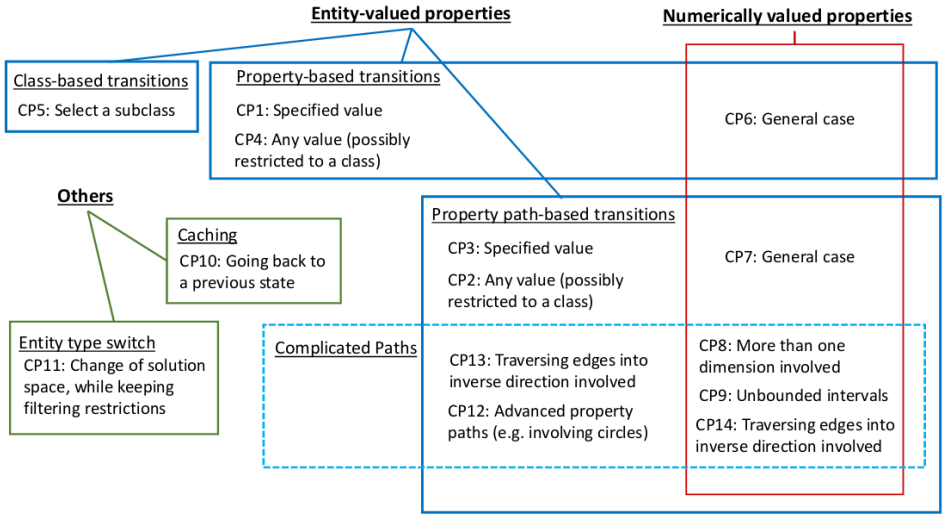


Figure 6: Overview of chokepoint transition types

CP5 Transition of a selected property value class to one of its subclasses
For a selected class that a property value should belong to, select a subclass

CP6 Change of bounds of directly related numerical data
Find all instances that additionally have numerical data lying within a certain interval for a directly related property

CP7 Change of numerical data related via a facet path of length strictly greater than one edge
Similar to CP6, but now the numerical data is indirectly related to the instances via a facet path

CP8 Restrictions of numerical data where multiple dimensions are involved
This is a combination of transition types CP6 and CP7 where bounds are chosen for more than one dimension of numerical data. For example, adding range restrictions to latitude and longitude predicates simultaneously in order to realise a bounding box constraint on a related spatial entity falls into this category

CP9 Unbounded intervals involved in numerical data
A variation of CP8 where intervals can be unbounded and/or only an upper or lower bound is chosen

CP10 Undoing former restrictions to previous state
Revert the faceted browsing session to an earlier state

CP11 Entity-type switch changing the solution space
Change of the solution space while keeping the current filter selections. As an example based on Figure 3, consider a constraint that restricts matching values to those that are of type *movie*. Application of CP11 may set the focus to items reachable via the *character* predicate. Hence, the new solution space would be the characters of things that are movies.

CP12 Advanced facet paths or circles
A combination of transition types CP3 and CP4 with "advanced" facet paths involved. Advanced hereby means "rarely used" patterns, such as following the same predicate in forward and backward direction. "Circle" in this context means that the set of resources reached via a (non-zero length) path overlaps with the set of resources from where the path started. An example is the traversal from an initial set of movies to their genres and back to the movies again. This can be used to find related sets of items based on an initial sample.

CP13 Inverse direction of an edge involved in facet path based transition
Transitions where the facet path involves traversing edges in the inverse direction

CP14 Numerical inequality restriction over a property path involving the inverse direction of an edge
Additional numerical data restrictions at the end of a facet path where the path involves traversing edges in the inverse direction. As an example, filtering the set of characters by the release year of the respective movies requires an inverse step along the *character* predicate and a numerical constraint on the release year predicate.

5.3. A Model to capture Chokepoint Characteristics

In this section a model that captures the previously introduced chokepoints is devised in order to make them applicable for benchmark generation. It can be observed that most chokepoints are related to generating paths and imposing restrictions on the (non-empty set of) reached

Table 2: Characteristics of chokepoints w.r.t. restrictions on path lengths, directions and predicates involved in steps, and the type of the SPARQL conditions to generate for the values reached by these paths. *F* indicates a 100% likeliness to choose a forward traversal, whereas *F, B* refers to 50%/50% chance for either forward or backward traversal. The shown parameters were used in our chokepoint definitions, but our framework also allows for defining new characteristics. CP5, CP8 and CP10 are listed separately as they do not exactly fit the schema. Note, that paths generated for CP11 (entity type switch) can occur in either direction, for CP13 they must include a backward traversal (in any step), and for CP14 they always start with a backward traversal.

Chokepoints	Path Length		Min #bwd	Direction Choices		Restriction(s) on ...	
	Min	Desired	traversals	Consumable	Fallback	Final Pred.	Values
CP1	1	1			F		(exists)
CP2	1	3			F		(exists)
CP3	1	3			F		equals
CP4	2	2			F	`rdf:type`	
CP6	1	1			F	(numeric)	closed range
CP7	1	3			F	(numeric)	closed range
CP9	1	3			F	(numeric)	open range
CP12	1	3			F, B	`rdf:type`	
CP13	1	2	1		F, B		equals
CP14	2	2		B	F, B	(numeric)	equals
CP5	reuse path of existing equals-constraint					`rdf:type`	pick subclass
CP8	apply multiple CPs that involve numeric constraints						
CP11	change focusPath					`rdf:type`	
CP10	return to previous state						

values. Our model to cover the path generation aspects of length and direction changes uses the following parameters:

- Minimum length: The minimum length for a path to qualify as a candidate
- Desired length: The desired length of a path. Our path finding algorithm will try to generate candidate paths of that length, if the dataset and active facet constraints allow for it.
- Minimum number of required backward traversals: CPs may require paths to contain reverse traversals in order for them to qualify as candidates.
- Consumable direction pool: A pool of (direction, weight) pairs from which can be drawn using "draw-with-replacement" semantics, i.e. whenever a pair is chosen to generate a step in a path, that entry is removed. In this work, we used a weight of 1 for all directions.
- Fallback direction pool: Static (direction, weight) values (i.e. without draw-with-replacement) that are resorted to once the other pool is consumed. This allows for paths having lengths greater than the size of the consumable pool.
- Restrictions on final predicate: The set of predicates that may appear in the last step of a path.
- Restrictions on values: Constraint types over the values, with the following meanings: `(exists)` imposes no further restriction on the values reached by a path other than that this set is non-empty. `Equals` generates an equals-constraint from a random pick of the available values. Likewise, open range, closed range creates inequality expressions. "Pick subclass" considers a datasets' class hierarchy to replace an equals constraint on a class with one of an arbitrary sub-class.

Table 2 summarises the path and constraint characteristics of the chokepoints.

5.4. Framework Architecture

In this section we present the faceted search benchmark generation framework which assembles several components. These include the query generation based on Section 4 and the chokepoints described in Section 5.2. Figure 7 depicts its components grouped by the phases of the benchmark process for which they are relevant, namely preparation, generation and execution. In the following, we briefly explain the components.

- The *Scenario Generator* drives the benchmark generation and uses the APIs and services provided by the other components. A scenario is a sequence of faceted search transitions.
- The *Dataset Analyser*'s purpose is to obtain metadata about a dataset. Most relevant to us are the used predicates and their ranges, and the schema-graph, i.e. which instance types are connected by which properties. This metadata is used by the pathfinding sub-system.
- The *Path Finder* finds simple paths that end in a given set of predicates, such as a numeric one or a pair that represents longitude and latitude. This component uses the schema-graph to generate candidate paths, which are subsequently validated using a reference triple store. As this path finder only operates on the schema graph, it is independent of the number of instance data. However, the required time for the

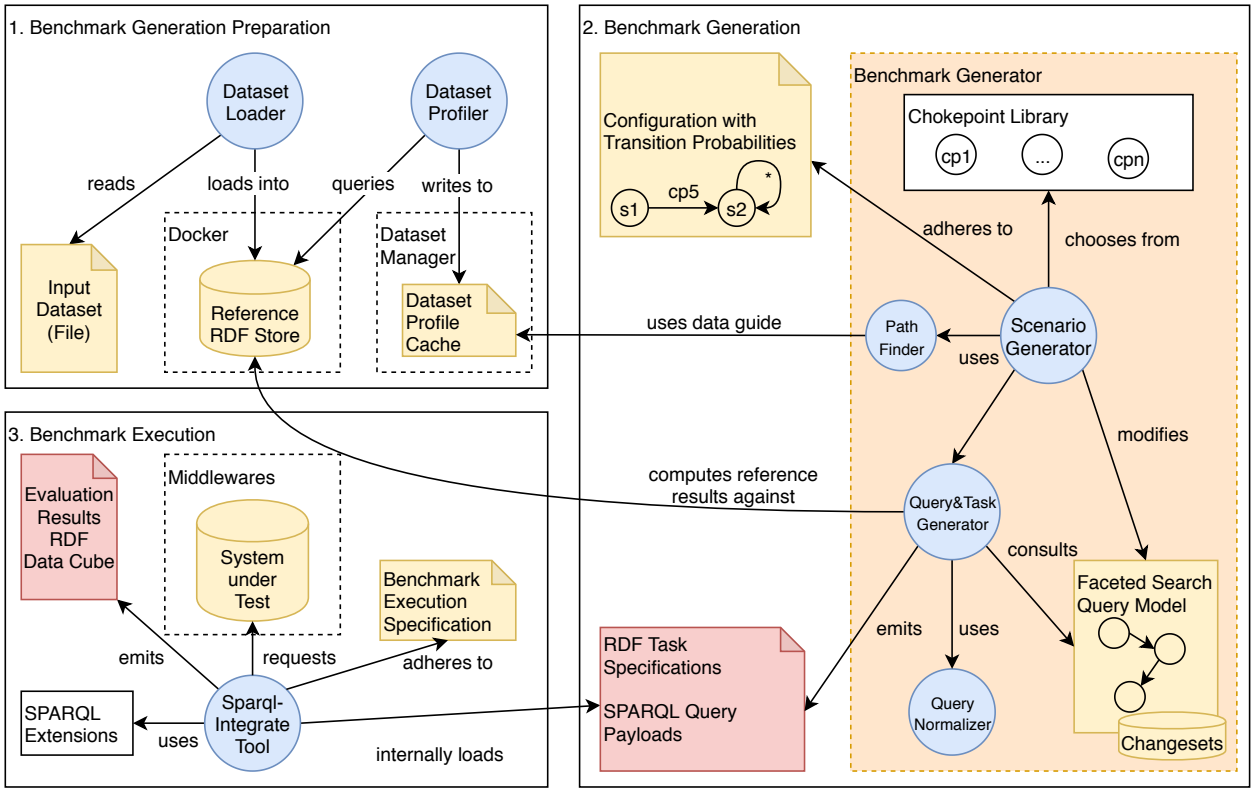


Figure 7: Architecture of the Schema-agnostic SPARQL-driven Faceted Search Benchmark Generator

dataset analysis is directly proportional to the size of the instance data.

- The *Query Normaliser* is used to post-process generated SPARQL queries in order to make them more natural as if they were written by a human. For instance, whenever possible, variables in triple patterns are substituted with constants and filter expressions are simplified. The main purpose is to improve the comparability and quality of our evaluation results – we want to know how well systems can execute faceted search queries in contrast to how well they can handle artefacts of our query generation. Query normalisation may be necessary in order for the generated queries to execute successfully on some triple stores due to issues with them[13].

- The *Middleware Layer* is the conceptual place where any virtual data transformations relevant to faceted search can be transparently added – i.e. independently of the SFS system – under the assumption of SPARQL interoperability. Query rewriting systems, such as OWL reasoners and ontology-based data access systems, e.g., Ontop[14], may go here. Furthermore, many RDF stores provide features that can be used for virtual Skolemisation of blank nodes using query rewriting or result set transformations. For example, our *Facete* framework ships with an embedded middleware that enables faceted search over blank nodes by employing query rewriting techniques that virtually expose them as IRIs. Additional query rewriting techniques can be used for creating logical RDF views over physical RDF graphs, such as by means of introducing new predicates or hiding existing ones. As an example, browsing people by *age* is often more convenient than by *birth date*. In a dataset, typically the latter predicate is preferred as it refers to static information, whereas the former one changes every year. Middlewares allow for decoupling of functionality from SFSs and such that other SPARQL-interoperable tools may benefit from transformed data and/or advanced features as well. For the evaluation of this work, we pass our benchmark queries directly through to the systems under test.

5.5. A Java domain-specific Language for Faceted Search

In this section we present our Java domain-specific language (DSL) our benchmark generator is built upon. The purpose of the DSL is to ease carrying out faceted search based on the model and query generation approaches described in Section 4 and facilitate decoupling of faceted search functionality from applications such as graphical user interfaces. In object oriented programming DSLs are used to allow for succinct expression of actions w.r.t. a

[13] Example of an issue that was mitigated by improving our query normaliser:
`https://github.com/openlink/virtuoso-opensource/issues/822`

[14] `https://ontop-vkg.org/`

certain application domain. The starting point of our DSL is an instance of the interface *FacetedQuery* which corresponds to Definition 7 and thus comprises a base SPARQL concept pattern that specifies the initial set of items, a set of constraints, and a focus path. In addition, a FacetedQuery holds a connection to a SPARQL endpoint which allows for execution of queries – most prominently those that correspond the faceted search information needs. In our DSL, traversals along facet paths and steps (Definition 3) are performed via method calls on a *FacetNode* interface. A FacetNode thereby corresponds to a variable in a graph pattern and a performing a step corresponds to the creation of a triple pattern. Consequently, traversal along paths corresponds to the construction of a graph pattern.

A FacetedQuery offers two FacetNodes as starting points for path traversals: *.root()* and *.focus()*. The former corresponds to the base concept pattern's variable, whereas the latter corresponds to the variable reachable via a given (possibly empty) path starting from *.root*().

Note that Definition 3 defines steps as a three-tuple comprised of an IRI, a direction (fwd or bwd) and an alias. Our DSL splits traversal of a step into individual method calls for each of its components such as `.fwd().via(iri).alias("a")`. These calls yield additional instances which allow executing requests for faceted search information needs. For example, retrieval of all facet values counts in forward direction is accomplished using `.fwd().facetCounts()`. Constraints can be created and subsequently activated using e.g. `.constraints().eq(10).activate()`. FacetNode and the further involved interfaces are depicted in Figure 8 and detailed in the following.

- A *FacetNode* instance can be seen as a specific variable in a graph pattern and thus intensionally represents a set of resources. Using the *.fwd()* or *.bwd()* methods, one obtains a *FacetDirNode* entity that represents the set of facets and facet values reachable either in forward or backward *direction*.

- *FacetDirNode* An intermediate entity for access to all facets in either forward or backward direction, backed by the *immediate* triples of the resources represented by the FacetNode. The methods *availableValues* yields *all* values from which constraints can be created, whereas *remainingValues* yields all values that do not yet satisfy any existing constraint. The latter method is used in our benchmark in order to create non-subsumed constraints. The FacetDirNode interface allows for forward and backward traversals via RDF predicates.

- *FacetMultiNode* An entity representing all SPARQL variables reached via the same origin and predicate. Conjunctive constraints are managed here, and each addition of a constraint results in a new FacetNode – and thus underlying SPARQL variable – to be allocated. The *primary* FacetNode – i.e. the default FacetNode for disjunctive constraints – is reached via *.one()*.

- *ConstraintFacade* The constraint facade provides a convenient way to list and append constraints that affect a given Facet(Multi)Node. Recall that a Faceted Query comprises a list of constraints that are expressions. The constraint facade allows creation of equality, inequality, range and spatial expressions, where one of the arguments corresponds to the respective Facet(Multi)Node.

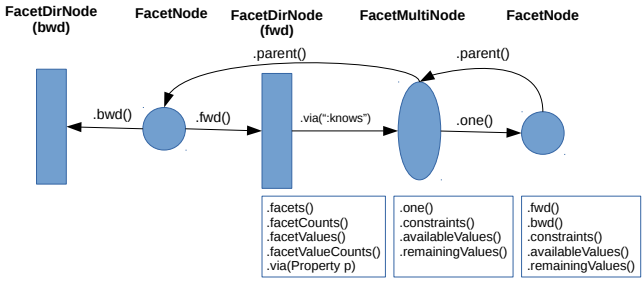


Figure 8: The most relevant interfaces and their methods of our faceted search DSL. They enable traversal of the data as well as retrieval of faceted-search-related information, such as available values or facet value counts.

All retrieval methods, such as *.facetValueCounts()* and *.facetCounts()*, yield an instance of *DataQuery* which wraps the SPARQL query that corresponds to an information need. *DataQuery* is an interface which provides method to apply subsequent SPARQL query transformations and modify query execution parameters. For example, slicing, filtering or the addition of extra joins is supported as well as configuring query execution timeouts.

An example of the DSL together with the composed query is shown in Figure 10.

5.6. An RDF Model for Faceted Search Queries

An RDF model is the representation of a domain model as an RDF graph. In Section 4 we devised a formal model for faceted search query generation from paths and constraints. In our implementation, we represent this domain model in RDF as shown in Figure 9. The advantages of this approach are:

- *Change Tracking and Undo*: Changes to a faceted search query can be tracked and reverted simply by book-keeping of additions and removals of triples. This generic mechanism is used in the benchmark generator for undoing effects of faceted search transitions.

- *Persistence*: The state of a faceted search query can be saved and loaded using standard RDF syntaxes –

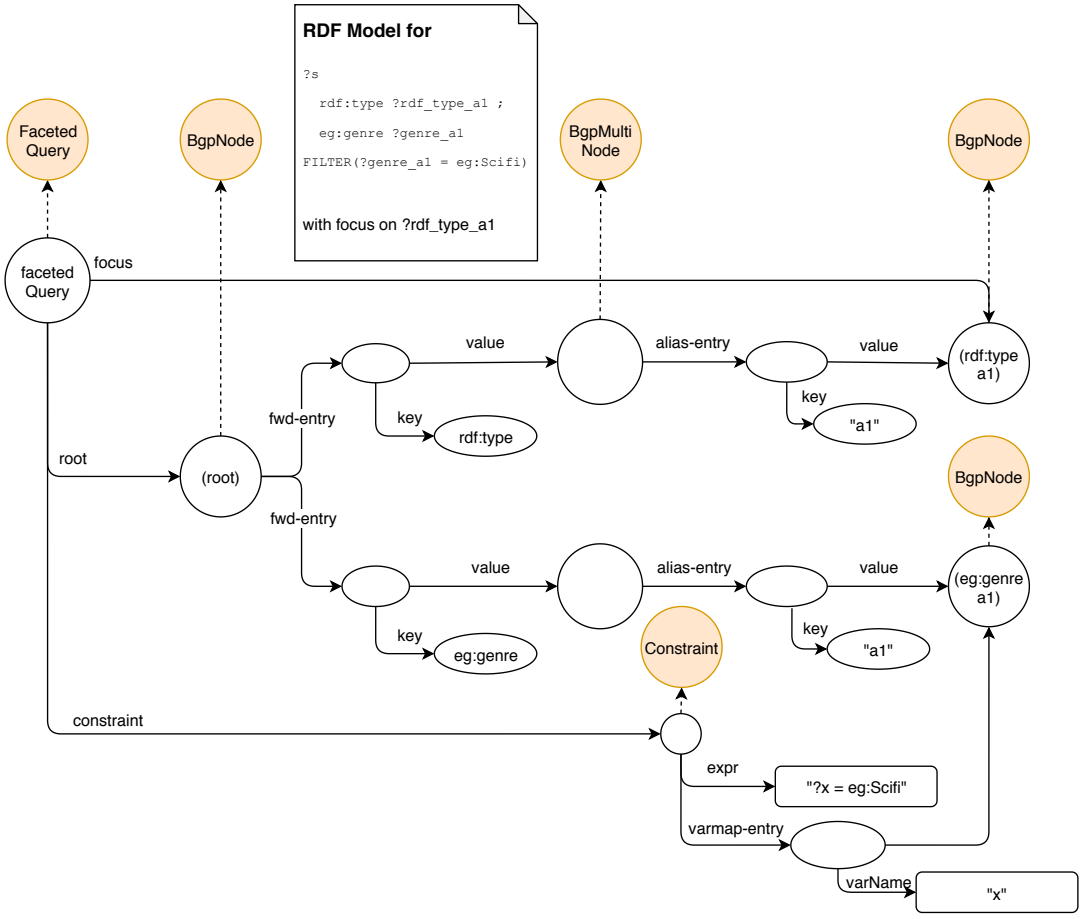


Figure 9: Faceted Query RDF Model

```
FacetValueCount fc =
  // -- Faceted Search API
  fq.focus()
     .fwd(RDF.type).one()
     .constraints()
       .eqIri(``eg:Movie'').activate()
     .end()
     .parent() // back to focus
  .fwd().facetValueCounts()
  // --- DataQuery API
  .randomOrder().limit(1)
  .exec()
  // --- RxJava API
  .firstElement()
  .timeout(10, TimeUnit.SECONDS)
  .blockingGet();
```

Listing 2: Java domain-specific language for faceted search query construction

```
SELECT ?p ?o ?c {
  { SELECT ?p ?o (COUNT(DISTINCT ?s) AS ?c) {
    # Facets and facet values without rdf:type
    ?s a eg:Movie ;
      ?p ?o
    FILTER(?p != rdf:type)
  } GROUP BY ?p ?o HAVING (!bound(?o) || !isBlank(?o)) }
 UNION
  { SELECT ?p ?o (COUNT(DISTINCT ?s) AS ?c) {
    # Values of the rdf:type facet
    ?s a ?o
    BIND(rdf:type AS ?p)
  } GROUP BY ?p ?o HAVING (!bound(?o) || !isBlank(?o)) }
} ORDER BY ASC(rand()) LIMIT 2
```

Listing 3: Facet value counts for movies

Figure 10: Juxtaposition of our DSL and the generated SPARQL query for a faceted search query for movies

possibly in conjunction with RDF stores. This makes our system a native Linked Data citizen.

- *Single point-of-truth*: Our API implementation is backed solely by this RDF graph, therefore changes to it are always reflected when requesting current facet/-value counts.

5.7. Benchmark Generator Configuration and Output

The benchmark generator is configured with an RDF document describing the number of faceted search scenarios to generate as well as how many interaction/transition types to apply within a scenario. The application of transitions is controlled using a probabilistic automaton (see Section 5.1). This is aimed at making it possible to configure the benchmark generator to simulate natural behaviour more closely, albeit for this work, models for real-world user interaction patterns are out of scope. For execution of benchmarks, we provide a benchmark runner implemented as a SPARQL `CONSTRUCT` query emitting data cube observations based on the SPARQL extensions provided by the Sparql-Integrate tool which is part of our RDF Processing Toolkit[15]. With this approach, benchmark queries are forwarded to the system-under-test by exploiting SPARQL's `SERVICE` clause. Details about the configuration and invocation can be found on the project website.

6. Evaluation

In this section we present our empiric validation of the SPARQL-driven faceted search benchmark generation. We perform two related studies: First, we evaluate the performance of several triple stores w.r.t. the most basic query that computes facet counts. In this experiment we scale the amount of data that participates in the aggregation by using different limits for that query. The results provide insights about how much data a triple store can process in what time on certain hardware in order to compute facet counts and facet value counts. This yields an estimate for the size of result sets at which interactive performance is no longer possible. Afterwards, we use our benchmark generator from Section 5 to generate benchmarks for two different datasets. Recall that the benchmark generator uses an internal reference triple store on which faceted search sessions are simulated. These simulations yield sequences of SPARQL queries which form the benchmark. These datasets are then loaded into triple stores that act as systems under test. The systems under test are then evaluated for their performance in executing the sequences of SPARQL queries that correspond to the faceted search sessions. The SPARQL result sets of the reference triple store are part of the generated benchmark and we verified that all systems under test yield the correct responses. For every chokepoint function of the benchmark generator we verified the correctness of the implementation using at least one unit test that involves comparing generated queries against expected ones on testing data.

6.1. Triple Stores

The following triple stores are used in our evaluation: *Virtuoso*[16] is one of the most popular triples store as it powers the SPARQL endpoint of DBpedia[17]. BlazeGraph[18] is well known because powers the SPARQL endpoint of Wikidata[19]. Ontotext's GraphDB[20] (formerly called OWLIM) is a triple store that gained recognition in the Semantic Web community for its reasoning features and by powering early Linked Data services at the British Broadcasting Corporation (BBC). *Eclipse rdf4j*[21] is a major Semantic Web framework for Java. It ships with in-memory and disk-based SPARQL-engines. For these systems there existed working docker images which we could readily use for our evaluation purposes.

6.2. Datasets

For benchmark data, we used two datasets; one generated dataset using the *Public Transport RDF Dataset Generator* PoDiGG [34]. The generated data has a small schema with main entities being stops, routes (sequences of stops) and connections (instances of routes at certain times). We generated a dataset of size approximately 4 million triples. The second dataset was taken from the German National Library (DNB) the Catalogue of Serials[22] (ZDB[23]). It contains the real world data from this catalogue, comprising about 40 million triples.

Using our framework, we then generated faceted browsing benchmarks using a Virtuoso 7.2.5[24] as the reference triple store (via the tenforce/virtuoso docker image[25]). We also conducted experiments with DBpedia data, however, its schema graph has too many possible combinations of paths. As such, we defer to future work before we can select a sensible subset of DBpedia to evaluate the pathfinding components on.

6.3. Performance of Basic Facet Count Computation

As a first measure of dataset behaviour on these two datasets, a systematic analysis of aggregation over increasing RDF data sizes from 1M to 4M resp. 1M to 50M triples was executed, using the query

```
SELECT ?p (COUNT(DISTINCT ?o) AS ?c) { { SELECT *
      { ?s ?p ?o } LIMIT X } } GROUP BY ?p
```

[15] https://github.com/SmartDataAnalytics/RdfProcessingToolkit

[16] https://virtuoso.openlinksw.com/

[17] https://dbpedia.org/sparql

[18] https://blazegraph.com/

[19] https://query.wikidata.org/

[20] https://www.ontotext.com/products/graphdb/

[21] https://rdf4j.org/

[22] https://github.com/hobbit-project/facete3-fsbg-results/releases (archived from https://data.dnb.de/opendata/zdb_lds_20190305.nt.gz)

[23] https://zdb-katalog.de/

[24] https://virtuoso.openlinksw.com/

[25] https://github.com/tenforce/docker-virtuoso

Table 3: Run-time (in seconds) of computing number of subjects per property on different subsets of the data.

System	Limit								
	PoDiGG				DNB ZDB				
	1M	2M	3M	4M	1M	4M	6M	12M	34M
Blazegraph	3	5	16	17	3	12	33	129	1244
GraphDB	1	2	3	4	1	5	7	11	34
rdf4j	7	15	22	27	18	41	55	112	315
Virtuoso	1	2	4	4	1	5	8	12	44

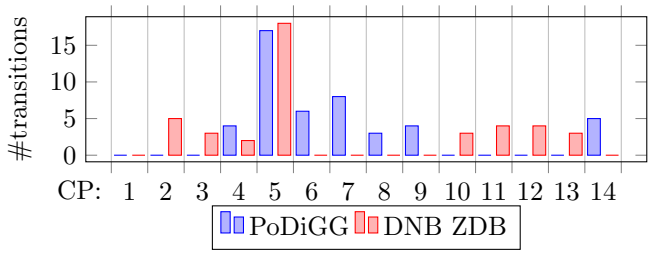


Figure 11: Dataset characteristics : number of times a transition was applied across all generated scenarios

The resulting execution times are shown in Table 3. User experience findings suggest, that response times below 0.1 seconds are perceived as instant reactions, times below 1 second may be noticed as slight delays but are typically non-interruptive, whereas around 10 seconds is the limit for keeping a user's attention [35]. Our findings show, that contemporary triple stores under the load of real world faceted search queries reach or exceed this threshold already for moderate data sizes.

6.4. Experimental setup

The generation of the benchmarks as well as the subsequent execution were run on a Intel E5-2620 2.1GHz system with 16 cores and 120GB or RAM total, and SATA hard drives (not SSD). We used the benchmark generator on each dataset and stored the resulting benchmark. The benchmark generator configuration file used was the `config-all.ttl`, which enables the generation of all transition types described in Section 5.2 to equal chances, and requires that the first transition is always CP5. For each dataset, twelve scenarios were generated with each simulating a number of user interaction steps, each step corresponding to one transition type.

For the execution of the benchmarks, each triple store was limited to 10GB of JVM memory. Only for GraphDB we requested the free version which cannot be used in docker (personalised link download required), the other stores were instantiated from docker-compose files. We then loaded the source datasets using the respective system-provided user interface for data load/import. Finally, the generated benchmarks were executed three times, and the last results were stored. To run the benchmarks, we used Sparql-Integrate as introduced in Section 5.7. This tool sends the SPARQL queries contained in the benchmark to the configured endpoint and records the correctness of the result as well as the required time.

All docker containers, the detailed configuration, generated benchmarks, SPARQL query for benchmark execution, and complete results of the executed benchmarks, are provided in the additional materials on the website.

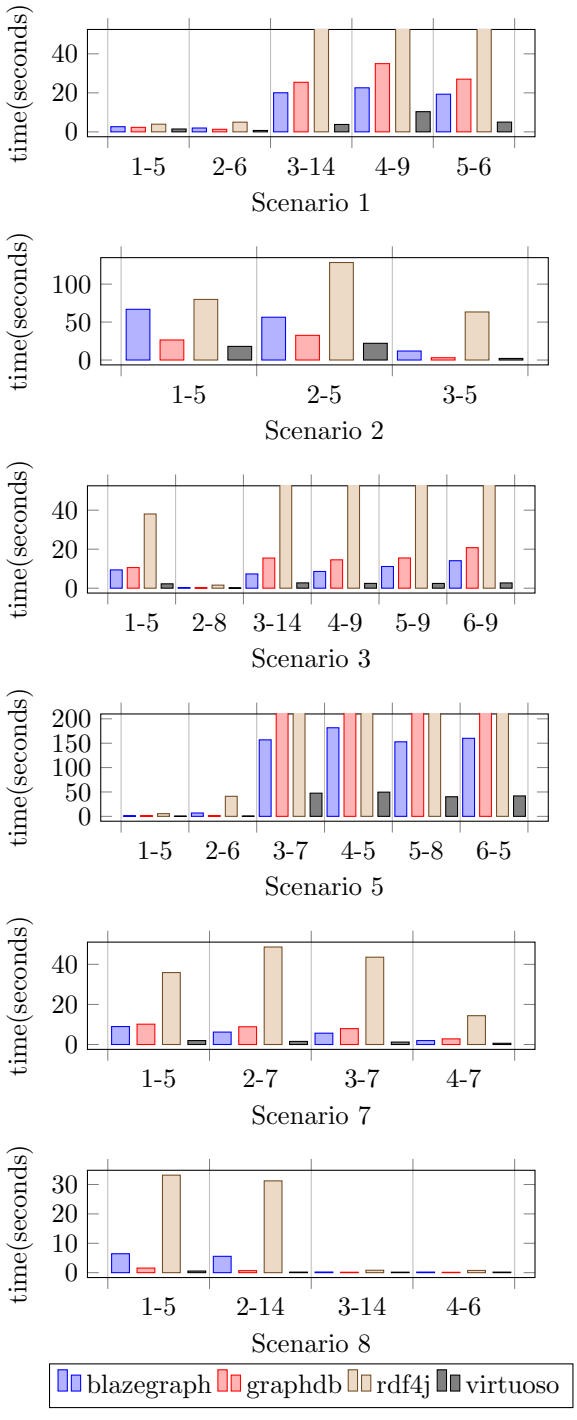
6.5. Results

First, we take a look at the generated benchmarks themselves. It can be observed nicely that the benchmark generator will produce benchmarks targeted to the given dataset. In Figure 11, it can be seen that the transition types were chosen according to applicability on the dataset. For example, on the DNB ZDB dataset, no numeric facets were available. Thus, the transition types CP6–9 have not been chosen. Consequently, the PoDiGG dataset did not provide any inverse property path relations. Since each generated scenario had to start with CP5, this transition type was also chosen more often than the others.

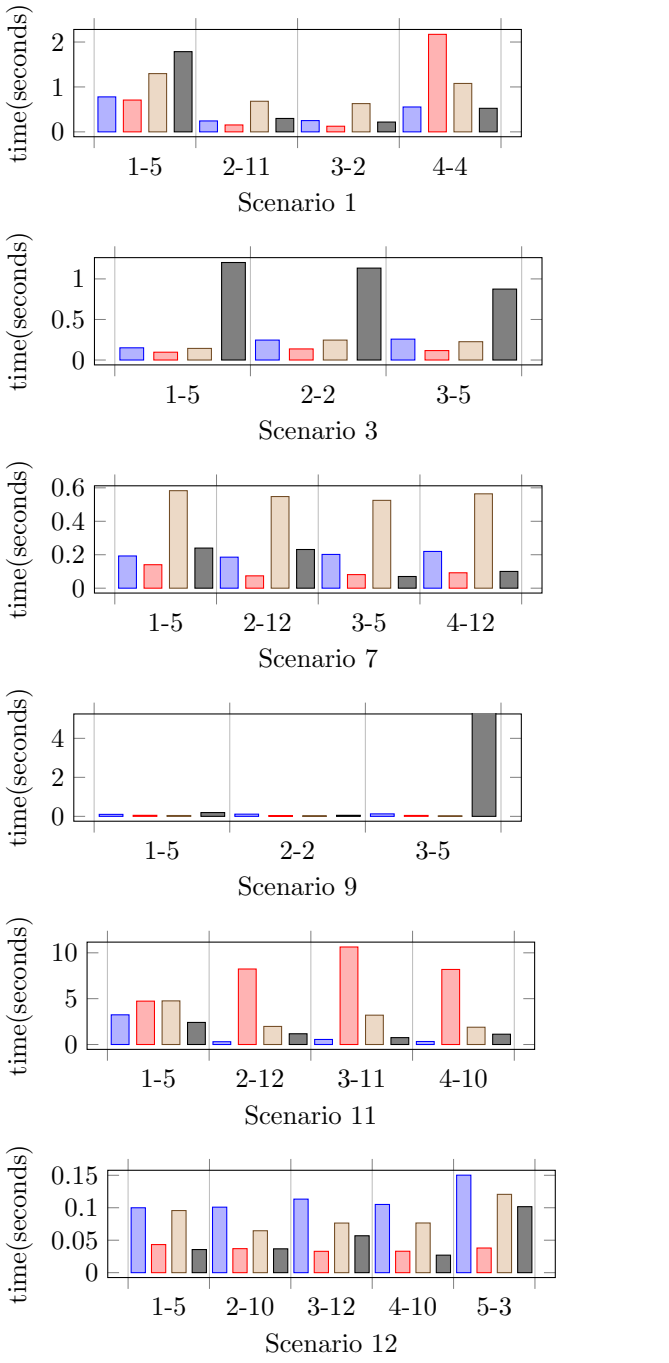
Furthermore, if we now consider the execution times of the benchmarks on the various triple stores in Figure 12, for the artificial PoDiGG dataset, we can see that Virtuoso is generally the fastest and rdf4j the slowest to answer. For these generated benchmarks, one notable observation is that the result set size after each faceted search step was still exceeding 900 resources. The response times are also often several seconds and thus not suited for interactive faceted browsing according to the definitions in Section 6.2. However, performance was better on the DNB ZDB dataset. There, we can observe a different behaviour of triple stores. In general, Virtuoso performed worse compared to the PoDiGG dataset. The execution times were much faster on this dataset, and thus suitable for real time browsing in most cases.

In Figure 12 (b), Scenario 9, we can observe that Virtuoso performed much slower on the last step. We evaluated the query in depth (see Figure 13). Between step 2 and 3, only the node variable `?v_4` was added to the query. Sometimes simple additions can cause a massive slow-down in the query planner of a triple store.

The evaluation shows that among the selected triple stores, Virtuoso usually outperforms other systems, especially when they already require several seconds to execute a query. As a consequence, this store allows for significantly improving the time it takes to explore data on-the-fly. However, there are cases where the generated



(a) Timed results on generated PoDiGG benchmark



(b) Timed results on benchmark generated on DNB ZDB data set

Figure 12: Performance of different triple stores on generated benchmark scenarios. The x-axis labels show the ordinal number of the transition in the scenario followed by the id of the choke point that was applied. The measured times are the total times spent on tasks in a scenario and therefore are the sum of the execution times of the queries for facet counts, facet value counts and matching items.

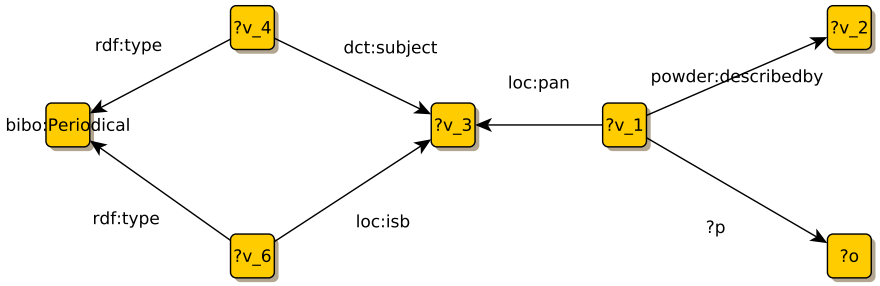


Figure 13: Example query graph of DNB ZDB Scenario 9, step 3, each edge representing a triple pattern in the query.

benchmarks reveal that generic data exploration can lead to performance-sensitive queries (e.g. Scenario 5) where even the fastest systems can no longer provide interactive performance. On the one hand, these insights can be particularly useful when offering faceted search over RDF graphs to users. For example, a-priori knowledge of such corner cases may be used in (future) semantic faceted search systems in order to improve usability by means of pre-configuration that either prevents such transitions at all or uses sampling methods at the expense of abandoning the calculation of exact result items and/or counts. Also, such cases may be prioritised for caching and conversely, the effects of caching systems can be evaluated using the generated benchmarks. On the other hand, the benchmark generation may reveal anomalies or even issues with datasets such as incorrect or sub-par relations that cause performance problems. This work enables practical studies w.r.t. the strengths, weaknesses and limits of SPARQL-driven faceted search.

7. Conclusions and Future Work

In this work, we presented a schema-agnostic faceted search benchmark generation framework for triple stores. In accordance with the Semantic Web vision where autonomous agents are able to explore the Web of Data in order to solve tasks on someone's behalf efficient exploratory search mechanisms are needed. Faceted search is a form of exploratory search that enables systematic exploration with a-priori insights about the available data under a set of constraints. As a consequence, the class of SPARQL queries that realise the faceted search paradigm on RDF graphs is of particular relevance.

Performance of SPARQL-driven faceted search depends mainly on the triple stores' capabilities of aggregating and counting data. Some benchmarked systems reached timeouts on query loads on relatively small dataset sizes. This suggests, that even today, live faceted search on SPARQL endpoints is associated with a high performance cost. As our evaluation of a simple aggregation on a large volume of data shows, interactive performance on such data is only achievable by means of indexing. Indexing can be performed on different levels: While it is possible to cache entirely on the SPARQL level, it may be advantageous to index on the level of an intermediate language for SPARQL-driven faceted search. The reason is, that very basic domain specific operations, such as "yield all facet values" under a given facet query configuration, may result in relatively complex SPARQL queries, as shown in Figure 10. Our benchmark generation framework thus provides fundamental functionality for these kinds of future research: On the one hand, creation of such an index (via SPARQL) requires assembling exactly the queries our faceted search engine generates. On the other hand, our implementation provides a high level abstraction for faceted search queries which may be suitable for use in conjunction with advanced indexing strategies. In any case, our benchmark generator can be used to automatically evaluate faceted search over arbitrary RDF datasets in order to detect potential performance issues.

We see the following directions for future work: First, comparison of our generated benchmarks with existing SPARQL-driven benchmarks in order to provide a bigger picture such as by means of assessing the similarities and differences of benchmarks w.r.t. used SPARQL language features. Second, evaluation of to what extent existing SFS systems can be integrated into our framework. And third, extension of our benchmark generation framework to cover features of advanced SFS systems, such as unions and negated existential restrictions on paths.

Acknowledgements

This work was supported by grants from the EU H2020 Programme for the projects HOBBIT (GA no. 688227) and QROWD (GA no. 732194) and the Federal Ministry of Transport and Digital Infrastructure (BMVI) for the LIMBO project (GA no. 19F2029G).

References

[1] H. Petzka, C. Stadler, G. Katsimpras, B. Haarmann, J. Lehmann, Benchmarking faceted browsing capabilities of triplestores, in: Proceedings of the 13th International Conference on Semantic Systems, ACM, 2017, pp. 128–135.

[2] Y. Guo, Z. Pan, J. Heflin, Lubm: A benchmark for owl knowledge base systems, Web Semantics: Science, Services and Agents on the World Wide Web 3 (2-3) (2005) 158–182.

[3] M. Schmidt, T. Hornung, G. Lausen, C. Pinkel, Spˆ 2bench: a sparql performance benchmark, in: 2009 IEEE 25th International Conference on Data Engineering, IEEE, 2009, pp. 222–233.

[4] C. Bizer, A. Schultz, The berlin sparql benchmark, International Journal on Semantic Web and Information Systems (IJSWIS) 5 (2) (2009) 1–24.

[5] G. Aluç, O. Hartig, M. T. Özsu, K. Daudjee, Diversified stress testing of rdf data management systems, in: International Semantic Web Conference, Springer, 2014, pp. 197–212.

[6] G. Garbis, K. Kyzirakos, M. Koubarakis, Geographica: A benchmark for geospatial rdf stores (long version), in: International Semantic Web Conference, Springer, 2013, pp. 343–359.

[7] D. Tunkelang, Faceted search, Synthesis lectures on information concepts, retrieval, and services 1 (1) (2009) 1–80.

[8] H. Bast, F. Bäurle, B. Buchhold, E. Haussmann, Broccoli: Semantic full-text search at your fingertips, arXiv preprint arXiv:1207.2615 (2012).

[9] J. Moreno-Vega, A. Hogan, Grafa: Scalable faceted browsing for rdf graphs, in: International Semantic Web Conference, Springer, 2018, pp. 301–317.

[10] M. Hildebrand, J. Van Ossenbruggen, L. Hardman, /facet: A browser for heterogeneous semantic web repositories, in: International Semantic Web Conference, Springer, 2006, pp. 272–285.

[11] P. Heim, J. Ziegler, S. Lohmann, gfacet: A browser for the web of data, in: Proceedings of the International Workshop on Interacting with Multimedia Content in the Social Semantic Web (IMC-SSW08), Vol. 417, Citeseer, 2008, pp. 49–58.

[12] E. Oren, R. Delbru, S. Decker, Extending faceted navigation for rdf data, in: International semantic web conference, Springer, 2006, pp. 559–572.

[13] G. Cheng, W. Ge, Y. Qu, Falcons: searching and browsing entities on the semantic web, in: Proceedings of the 17th international conference on World Wide Web, ACM, 2008, pp. 1101–1102.

[14] J. Davies, R. Weeks, Quizrdf: Search technology for the semantic web, in: 37th Annual Hawaii International Conference on System Sciences, 2004. Proceedings of the, IEEE, 2004, pp. 8–pp.

[15] R. Hahn, C. Bizer, C. Sahnwaldt, C. Herta, S. Robinson, M. Bürgle, H. Düwiger, U. Scheel, Faceted wikipedia search, in: International Conference on Business Information Systems, Springer, 2010, pp. 1–11.

[16] J. Waitelonis, H. Sack, Towards exploratory video search using linked data, Multimedia Tools and Applications 59 (2) (2012) 645–672.

[17] J. Moreno-Vega, A. Hogan, Grafa: Scalable faceted browsing for rdf graphs, in: International Semantic Web Conference, Springer, 2018, pp. 301–317.

[18] L. Wenige, J. Ruhland, Similarity-based knowledge graph queries for recommendation retrieval, Semantic Web 10 (6) (2019) 1–31.

[19] S. Ferré, A. Hermann, Reconciling faceted search and query languages for the semantic web, in: Int. J. Metadata, Semantics and Ontologies, 2012, pp. 37–54.

[20] S. Ferré, Squall: A controlled natural language for querying and updating rdf graphs, in: International Workshop on Controlled Natural Language, Springer, 2012, pp. 11–25.

[21] S. Ferré, Sparklis: an expressive query builder for sparql endpoints with guidance in natural language, Semantic Web 8 (3) (2017) 405–418.

[22] M. Arenas, B. C. Grau, E. Kharlamov, Š. Marciuška, D. Zheleznyakov, Faceted search over rdf-based knowledge graphs, Journal of Web Semantics 37 (2016) 55–74.

[23] E. Sherkhonov, B. C. Grau, E. Kharlamov, E. V. Kostylev, Semantic faceted search with aggregation and recursion, in: International Semantic Web Conference, Springer, 2017, pp. 594–610.

[24] A. Soylu, E. Kharlamov, D. Zheleznyakov, E. Jimenez-Ruiz, M. Giese, M. G. Skjæveland, D. Hovland, R. Schlatte, S. Brandt, H. Lie, et al., Optiquevqs: a visual query system over ontologies for industry, Semantic Web 9 (5) (2018) 627–660.

[25] F. Haag, S. Lohmann, S. Siek, T. Ertl, QueryVOWL: A visual query notation for linked data, in: Proceedings of ESWC 2015 Satellite Events, Vol. 9341 of LNCS, Springer, 2015, pp. 387–402.

[26] T. Mailis, Y. Kotidis, V. Nikolopoulos, E. Kharlamov, I. Horrocks, Y. Ioannidis, An efficient index for rdf query containment, in: Proceedings of the 2019 International Conference on Management of Data, ACM, 2019, pp. 1499–1516.

[27] C. Stadler, M. Saleem, A.-C. N. Ngomo, J. Lehmann, Efficiently pinpointing sparql query containments, in: International Conference on Web Engineering, Springer, 2018, pp. 210–224.

[28] J. Pérez, M. Arenas, C. Gutierrez, Semantics and complexity of sparql, ACM Transactions on Database Systems (TODS) 34 (3) (2009) 1–45.

[29] R. Angles, C. Gutierrez, The multiset semantics of sparql patterns, in: International semantic web conference, Springer, 2016, pp. 20–36.

[30] M. Kaminski, E. V. Kostylev, B. C. Grau, Query nesting, assignment, and aggregation in sparql 1.1, ACM Transactions on Database Systems (TODS) 42 (3) (2017) 1–46.

[31] Y. Tzitzikas, N. Manolis, P. Papadakos, Faceted exploration of rdf/s datasets: a survey, Journal of Intelligent Information Systems 48 (2) (2017) 329–364.

[32] P. Boncz, T. Neumann, O. Erling, Tpc-h analyzed: Hidden messages and lessons learned from an influential benchmark, in: Technology Conference on Performance Evaluation and Benchmarking, Springer, 2013, pp. 61–76.

[33] K. Chatterjee, L. Doyen, T. A. Henzinger, Probabilistic weighted automata, in: International Conference on Concurrency Theory, Springer, 2009, pp. 244–258.

[34] R. Taelman, R. Verborgh, T. De Nies, E. Mannens, Podigg: a public transport rdf dataset generator, in: Proceedings of the 26th International Conference on World Wide Web Companion, International World Wide Web Conferences Steering Committee, 2017, pp. 843–844.

[35] J. Nielsen, Usability engineering, Elsevier, 1994.