Deliverable 6.2.2
Second Version of the Faceted Browsing Benchmark

<table>
<thead>
<tr>
<th>Dissemination Level</th>
<th>Public</th>
</tr>
</thead>
<tbody>
<tr>
<td>Due Date of Deliverable</td>
<td>Month 30, 31/05/2018</td>
</tr>
<tr>
<td>Actual Submission Date</td>
<td>Month 35, 31/10/2018</td>
</tr>
<tr>
<td>Work Package</td>
<td>WP6 - Benchmarks IV: Visualization and Services</td>
</tr>
<tr>
<td>Task</td>
<td>T6.2</td>
</tr>
<tr>
<td>Type</td>
<td>Other</td>
</tr>
<tr>
<td>Approval Status</td>
<td>Approved</td>
</tr>
<tr>
<td>Version</td>
<td>1.0</td>
</tr>
<tr>
<td>Number of Pages</td>
<td>28</td>
</tr>
</tbody>
</table>

**Abstract:** This report describes the achievements of the second version of the Faceted Browsing Benchmark. The concepts, design, implementation and evaluation results of the benchmark are detailed.

The information in this document reflects only the author’s views and the European Commission is not liable for any use that may be made of the information contained therein. The information in this document is provided "as is" without guarantee or warranty of any kind, express or implied, including but not limited to the fitness of the information for a particular purpose. The user thereof uses the information at his/her sole risk and liability.

This project has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No 688227.
History

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Reason</th>
<th>Revised by</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>05/03/2018</td>
<td>First draft</td>
<td>Claus Stadler</td>
</tr>
<tr>
<td>0.2</td>
<td>10/04/2018</td>
<td>Initial API design</td>
<td>Claus Stadler</td>
</tr>
<tr>
<td>0.3</td>
<td>16/05/2018</td>
<td>Initial conceptualization</td>
<td>Claus Stadler</td>
</tr>
<tr>
<td>0.4</td>
<td>26/06/2018</td>
<td>Revision of the design</td>
<td>Claus Stadler</td>
</tr>
<tr>
<td>0.5</td>
<td>18/07/2018</td>
<td>Improved link with prior deliverable</td>
<td>Claus Stadler</td>
</tr>
<tr>
<td>0.6</td>
<td>13/08/2018</td>
<td>Revision of checkpoint generalization</td>
<td>Claus Stadler</td>
</tr>
<tr>
<td>0.7a</td>
<td>24/08/2018</td>
<td>Addition of further components</td>
<td>Claus Stadler</td>
</tr>
<tr>
<td>0.7b</td>
<td>27/08/2018</td>
<td>Description of LIVED data generator</td>
<td>Pavel Smirnov</td>
</tr>
<tr>
<td>0.8</td>
<td>24/09/2018</td>
<td>Peer review comments</td>
<td>Irini Fundulaki</td>
</tr>
<tr>
<td>0.9</td>
<td>25/10/2018</td>
<td>Revision according to feedback</td>
<td>Claus Stadler</td>
</tr>
<tr>
<td>1.0</td>
<td>27/10/2018</td>
<td>Approval</td>
<td>Claus Stadler and Jens Lehmann</td>
</tr>
</tbody>
</table>

Author List

<table>
<thead>
<tr>
<th>Organization</th>
<th>Name</th>
<th>Contact Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>InfAI</td>
<td>Claus Stadler</td>
<td><a href="mailto:cstadler@informatik.uni-leipzig.de">cstadler@informatik.uni-leipzig.de</a></td>
</tr>
<tr>
<td>AGT</td>
<td>Pavel Smirnov</td>
<td><a href="mailto:psmirnov@agtinternational.com">psmirnov@agtinternational.com</a></td>
</tr>
<tr>
<td>FORTH</td>
<td>Irini Fundulaki</td>
<td><a href="mailto:fundul@ics.forth.gr">fundul@ics.forth.gr</a></td>
</tr>
<tr>
<td>IAIS</td>
<td>Jens Lehmann</td>
<td><a href="mailto:Jens.Lehmann@iais.fraunhofer.de">Jens.Lehmann@iais.fraunhofer.de</a></td>
</tr>
</tbody>
</table>

Page 1
### Abbreviations and Acronyms

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDF</td>
<td>Resource Description Framework</td>
</tr>
<tr>
<td>SPARQL</td>
<td>SPARQL Protocol and RDF Query Language</td>
</tr>
<tr>
<td>JSON</td>
<td>JavaScript Object Notation</td>
</tr>
<tr>
<td>KG</td>
<td>Knowledge Graph</td>
</tr>
<tr>
<td>SUT</td>
<td>System Under Test</td>
</tr>
<tr>
<td>PMF</td>
<td>Probability mass function</td>
</tr>
<tr>
<td>CP</td>
<td>Chokepoint</td>
</tr>
<tr>
<td>KPI</td>
<td>Key Performance Indicator</td>
</tr>
</tbody>
</table>
Executive Summary

This document describes the second and final version of the Faceted Browsing Benchmark. It covers the workload conducted from M19 until M35. This benchmark facilitates performance and correctness comparison of SPARQL-compliant database systems by means of simulated faceted browsing sessions over streaming data. The target audience for the main sections of this deliverable are Semantic Web developers and researchers (due to the technical nature of the content), whereas the evaluation results may guide decision makers who need to make informed choices among existing database systems.

The final version of the benchmark builds upon and generalizes the concepts of the first one in order to offer a novel flexibly and extensible benchmark generation framework. This version dynamically adapts to virtually arbitrary datasets (regardless of their domain), while supporting spatial and thematic dimensions. This contrasts the prior version, which is based on query templates whose parameters are obtained using domain specific queries.
Contents

1 Introduction .................................................. 6

2 Preliminaries .................................................. 7
   2.1 RDF and SPARQL ........................................... 8
   2.2 Random Weighted Selection ................................ 9
   2.3 Caching of Generated Artifacts .......................... 9

3 The Faceted Browsing Benchmark Generation System .......... 9
   3.1 System Architecture ....................................... 9
   3.2 The Faceted Browsing System ............................. 11
      3.2.1 The Faceted Browsing Fluent API .................... 12
      3.2.2 SPARQL Query Generation ............................ 12
   3.3 Review of previously identified checkpoints ............... 13
   3.4 Generalization of Checkpoints ............................ 16
   3.5 Processing Hierarchical Data ............................. 17
   3.6 Dataset Analysis ........................................... 18

4 The LIVED Data Stream Generator ................................ 20
   4.1 Persisting Data Streams .................................... 23

5 Hobbit Integration .............................................. 23
   5.1 Benchmark Execution ....................................... 23
   5.2 Published Resources ....................................... 24
   5.3 Evaluation ................................................ 25

6 Related Work .................................................. 25

7 Conclusions and Future Work .................................. 26
List of Figures

1. Example of refinement of keyword search results using faceted browsing on the Amazon online shop ........................................... 6
2. Benchmark Generator vs Generated Benchmark ........................................... 7
3. Architecture of the Faceted Browsing Benchmark Generator v2 ........................... 10
4. The most relevant classes and methods of the Faceted Browsing System. They enable traversal of the data as well as retrieval of faceted browsing related information, such as available values or facet value counts. ........................................... 11
5. Process of generating a SPARQL query from a faceted query .............................. 13
6. SPARQL queries for facet values .................................................................. 14
7. SPARQL queries for computing facet counts and facet value counts based on facet value relation .................................................. 14
8. Screenshot of locations of residential buildings exported from OpenStreetMap ........ 21
9. Hierarchy of observations grouped by sensing devices ...................................... 22
10. The LIVED data model ............................................................................. 22
11. Architecture of the Hobbit Platform ........................................................... 24
Figure 1: Example of refinement of keyword search results using faceted browsing on the Amazon online shop

1 Introduction

Faceted browsing is ubiquitous on the Web today. Most if not all major online shops and media (video, images and music) platforms provide at least some degree of 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 generally products by relevant features, as shown in Figure 1. Conceptually, in many cases the underlying data can be seen as knowledge graphs (KG), i.e. entities and values are represented as nodes, and labeled edges are used to specify their semantic relation. In cases where data is modeled using RDF (and ontologies), it already forms a KG. In this deliverable, we present our SPARQL-based faceted browsing benchmark generation framework, designed to assess triple store performance in regard to the faceted browsing paradigm.

In more technical terms, faceted browsing stands for a session-based and state-dependent interactive method for query formulation over a multi-dimensional information space. Usually, a user first defines an initial set of items, such as by choosing a category or providing a keyword. Afterwards, the user is presented with a selection of properties and corresponding values which can be used to filter these items. A main characteristic of faceted browsing is, that a user navigates between sets of items; either by restricting such a set or by traversing to a related one. Thereby, one of the most important advantage of faceted browsing is the prevention of the user navigating into empty results. This is accomplished by only presenting the user with filter options based on the remaining items’ properties. For cognitive reasons, end user applications tend to keep the filter options simple, i.e. the set of properties and values one can filter by is reduced to the most relevant ones and nesting of properties is avoided. However, Linked Data datasets are usually more sophisticated. Yet, those simpler datasets can be derived from complex data using extract-transform-load or data virtualization techniques (views). Hence, without loss of generality, our faceted browsing benchmark makes no assumptions about the data and treats all available properties - including nested ones - as candidate facets.

In this work, we first present the HOBBIT Faceted Browsing Benchmark Generator v2 which is
used to create benchmarks under certain configurations for the evaluation, as sketched in Figure 2. The benchmark generator extends over prior version in the following aspects:

- Novel SPARQL-based faceted browsing engine and API implemented in Java
- Benchmark task generation now works in domain-independent and data-driven fashion
- Chokepoint definitions (predefined types of faceted browsing interactions) of the prior version have been generalized
- Support for streaming data

The remainder of this document is structured as follows: We start with the preliminaries fundamental to the understanding the benchmark framework’s design in section 2. The design and implementation of the faceted browsing benchmark generation system and its components are presented in section 3.

Afterwards, in section 4 we describe the Long Device Level Energy Data LIVED data generator that is part of this benchmark framework. The integration of the system into the Hobbit platform, as well as pointers to our published artifacts are described in section 5. We conclude this document and give pointers to future work in section 7.

2 Preliminaries

The engine is data-driven, which means that facets and facet values correspond directly to the available properties and property values, respectively, in an RDF dataset.
2.1 RDF and SPARQL

The Resource Description Framework (RDF) is a W3C standard for data interchange\(^1\). The most fundamental introduced entity is the RDF graph which conceptually is a set of triples. A triple can be seen as a labelled “link” between two nodes.

Formally, let there be sets of IRIs \(I\), blank nodes \(B\) and literals \(L\). Further, let the set of RDF terms \(T := I \cup B \cup L\). An RDF Graph \(G\) is defined as \(G \subset (I \cup B) \times I \times T\).

Although RDF graphs are often depicted as conventional labeled graphs, they are formally defined of ternary relations which in turn correspond to directed labeled 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, in some cases commonly used graph algorithms may not be applicable to given RDF data.

SPARQL (a recursive acronym for SPARQL Protocol and RDF Query Language) is a W3C standard that devises protocols and languages for querying and updating RDF\(^2\). As our benchmark builds on SPARQL, we require systems-under-test to support it.

Paths

With SPARQL 1.1, property paths expressions were introduced to the query language. Since then it is possible to query a dataset for all pairs of resources connected via sequences of RDF predicates that match a given path expression. For instance, \(?s \text{ rdfs:subClassOf* } ?super\) will yield any pair of resources reachable by zero-or-more forward traversals along the rdfs:subClassOf predicate. For our work, we introduce the notion of simple paths as (possible empty) sequences of steps which are (predicate, direction) pairs. Direction is either forwards or backwards.

Partitioned Sparql Construct Queries

```
PartitionedQuery
  .from("CONSTRUCT WHERE { ?s a :Person ; rdfs:label ?l }")
  .partitionBy("s")
  .executeOn(sparqlEndpoint)
  .forEach(partition -> partition.get("s").getProperty(RDFS.label))
```

Listing 1: Pseudo code for a partitioned query and property access

\(CONSTRUCT\) is a standard SPARQL query form which allows for retrieval of RDF graphs. However, many use cases do not demand for a single monolithic RDF graph, but rather individual partitions that form logical objects and that can be processed independently. For example, a query about employee and departments may look like \(CONSTRUCT WHERE ?employee :employedAt ?department\). From the query alone, one cannot deduce whether it is department-, employee- or triple-centric. However, if we denoted e.g. \(?department\) as a partition variable, we can easily group triples based on this condition for further processing. A partitioned construct query is thus simply a pair comprised of a construct query and a set of partition variables. Note, that partitioning could also be applied to SELECT queries. However, in the case of \(CONSTRUCT\), access of retrieved data becomes object-like and thus more convenient: Similar to as one would access properties of a JSON object, one can – starting from

\(^1\)https://www.w3.org/RDF/
\(^2\)https://www.w3.org/TR/sparql11-query/
a binding of one of the partition variables – access its retrieved predicates, as sketched in Listing 1. In our work, we use this mechanism for obtaining facet values and counts.

2.2 Random Weighted Selection

In our system, pseudo random numbers to drive several aspects of the benchmark generation, such as which value(s) to pick when generating constraints. Pseudo random numbers are typically equally distributed within a given range, such as the (closed) unit-interval \([0, 1]\). In order to create different distributions, we employ the following method in our system:

A *probability mass function* (PMF) is a function that gives the probability that a discrete random variable is exactly equal to some value.\(^3\) All probabilities of a PMF must be non-negative and sum up to 1. Consequently, each item can be assigned a unique range within the unit interval.

In our case, we typically use *weight functions* that associate items, such as facet values, with (numeric) weights. Normalizing the weight function by the total weight gives the PMF. Hence, we can perform random weighted selection by first (randomly) choosing a number in the unit-interval, and then taking the item whose range overlaps with it.

2.3 Caching of Generated Artifacts

All SPARQL-related benchmark systems in Hobbit allow for configuration of the data and task generation. As in general, such generation processes can be very time-consuming, caching of the generated artifacts can significantly speed up repeated benchmark runs with “similar” parameters against different systems under test. In general, configuration parameters of generators can be classified as identifying and non-identifying. Examples for the former are random seed, number of iterations, whereas examples for the latter are maximum allowed RAM usage and timeouts. Clearly, a deterministic generation procedure yields upon successful termination the same output under a given set of identifying parameters - regardless of that of the non-identifying ones. As a consequence, generated artifacts, such as datasets and tasks, can be effectively cached by associating them with the appropriate identifying parameters and performing appropriate lookups to prevent repeated generation.

3 The Faceted Browsing Benchmark Generation System

In this section we present the design and implementation of our new benchmark generation framework. To avoid confusion, please note, that the faceted browsing benchmark generation system is built upon a faceted browsing system (non-benchmark related).

3.1 System Architecture

Figure 3 depicts the components of our faceted browsing benchmark generation system.

- 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 browsing interactions.

\(^3\)https://en.wikipedia.org/wiki/Probability_mass_function
Figure 3: Architecture of the Faceted Browsing Benchmark Generator v2

- The Dataset Analyzer’s purpose is to obtain metadata about a dataset. Most relevant to us is querying for all used predicates and their ranges, and the property-join summary (see subsection 3.6). This metadata is used by the path finders.

- The Exploratory Path Finder tries to generate - starting from a set of resources - simple paths that meet given specifications.

- The Schema-based Target-constrained Path Finder is used to find 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 first uses the property join summary to generate candidate paths. The system under test (SUT) SPARQL endpoint is then queried for whether there are actually resources connected by the candidate paths.

- The Query Optimizer is used to post-process generated SPARQL queries. For instance, whenever possible, variables in triple patterns are substituted with constants and filter expressions are simplified.

- The Middleware Layer is the place where certain virtual data transformations highly relevant to faceted browsing can be accomplished using query rewriting. Although this layer is not used in the benchmark generator, we created prototype implementations which are summarized here for completeness:

  - Property Visibility: In many cases, it is undesired to have all predicates of a dataset to show up as facets in a faceted browser. This component can hide predicates by means of injecting appropriate FILTER statements into SPARQL queries based on given whitelists or blacklists.

  - Virtual Predicate Rewrite: Sometimes predicates desired to appear in a faceted browser are not directly present in a dataset. For example, browsing people by age may feel more natural than by birth date. Obviously, in a dataset, the latter predicate is preferred as
Figure 4: The most relevant classes and methods of the Faceted Browsing System. They enable traversal of the data as well as retrieval of faceted browsing related information, such as available values or facet value counts.

it refers to static information, whereas the former one changes every year. This rewrite component allows one to define custom virtual predicates using SPARQL queries.

3.2 The Faceted Browsing System

The most fundamental component of our benchmarking system is unsurprisingly the faceted browsing system. It provides the means necessary to interact with an RDF dataset according to the faceted browsing paradigm. The entry point to the system is the FacetedQuery. It is comprised of the following constituents:

- A **base SPARQL concept** that denotes the initial set of resources. Typically it denotes all of an RDF graph's subjects, but it could also be set to e.g. the instances of a certain class or the set of predicates.

- A set of (facet) **constraints** - We represent constraints as a SPARQL expression with references to entities that represent traversals of the data. The type of the traversal determines whether multiple constraints are to be combined using conjunctions or disjunctions.

- The **focus** denotes the set of resources that serve as the base for facet value counts computations. Note, that despite constraints being SPARQL expressions, they are not injected into final SPARQL query directly: One the one hand, every reference needs to be resolved to its final variable and graph pattern. On the other one, additional transformations may be required, such as converting a spatial constraint involving a bounding box into the appropriate vocabulary and/or SPARQL dialect, such as WGS84, GeoSPARQL or Virtuoso's flavor of it.

The focus of a FacetedQuery is of type **FacetNode**, whose related interfaces are depicted in Figure 4 and are described as follows:
• A FacetNode instance can be seen as a specific variable in a SPARQL query and thus intentionally represents a set of resources. Using the .fwd() or .bwd() methods, one obtains a FacetDirNode entity that represents the set of facet and facet values reachable either in forward or backward direction. Note, that in principle a FacetNode could provide the functionality to access the union of both traversals, however we leave that for future work.

• 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 new 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 FacetedQuery comprises a list of constraints that are expressions. The ConstraintFacade allows creation of equality, inequality, range and spatial expressions, where one of the arguments corresponds to the respective Facet(Multi)Node.

3.2.1 The Faceted Browsing Fluent API

The FacetedQuery API is a fluent API and features interfaces for data traversals, constraint management and data retrieval. However, generation of the final SPARQL queries involves further steps which are sketched in Figure 5. In fact, our implementation of the API uses a backing RDF model to capture the state of faceted queries. Adding or removing constraints or changing the focus of a faceted query actually modifies an RDF model, which acts as the session state. This is consistent with our used definition of faceted browsing: ... a session-based and state-dependent interactive method for query formulation ... The advantage of this approach is, that by tracking the changes to the RDF model, we can easily support reverting the session state. This accounts for the use case, where users want to undo their recent interactions in order to go back to a prior state. The Query Generation Driver is a component that generates partitioned SPARQL queries from the state of the RDF model. We named the API for interaction with partitioned SPARQL queries DataQuery. The API features methods for sampling, shuffling, slicing (i.e. limit/offset), and filtering partitions, as well as declaration of additional properties to be retrieved.

3.2.2 SPARQL Query Generation

In this subsection, we show how the most important faceted browsing information needs translate to SPARQL queries.

Facet Values

In the simplest case, a query for facets and facet values is simply the query shown in Figure 6. Query generation becomes more complex when paths are involved, such as performing faceted browsing with
a focus on actors, while obtaining indirectly facets of directors. The aspect that makes SPARQL query generation from facet constraints even more complex is, that when we want to obtain a FacetNode’s available values, we need to apply all constraints - except for those that directly affect the FacetNode: Consider this example: A “year” facet has the values 2016, 2017 and 2018. If we added the constraint `facetNode.fwd(:year).constraints.eq(2018)`, then we have to distinguish these cases:

- Obtain the set of resources that satisfy *all* given constraints. In this case, essentially a single group graph pattern that covers all constraints needs to be generated.

- Obtain the set of *available facet values* of the year facet. In this case, our previously added constraint for the year 2018 needs to be ignored, because we also expect 2016 and 2017 as results. If we asked for *all* facets (or facet values) that can be reached in either forward or backward direction, a union has to be generated that correctly yields the facets and facet values under the given constraints.

**Facet Counts and Facet Value Counts**

Facet counts denote the number of a facet’s available distinct values. Typically, these counts give a user an impression about the relevancy of a facet without having to inspect individual facet values. The *facet value count* indicates for every value of a predicate how many unique focus resources there are. In both cases, the corresponding queries involve application of a group-by operation on facet value relation as shown in Figure 7.

### 3.3 Review of previously identified checkpoints

In this section we review the checkpoints identified for the first version of the benchmark. A chokepoint is essentially a named faceted browsing interation followed by a subsequent information need (such as facet counts or available values), whose performance and correctness is subject to evaluation. The obtained results define the values of the key performance indicators of the benchmark.
### Listing 2: Forward facets

```sparql
SELECT * {
?focus ?facet ?facetValue
}
```

### Listing 3: Backward facets

```sparql
SELECT * {
?facetValue ?facet ?focus
}
```

### Listing 4: Union of both directions

```sparql
SELECT ?focus ?isFwd ?facet
?facetValue {
?focus ?facet ?facetValue
BIND(?isFwd = true)
}
UNION
?facetValue ?facet ?focus
BIND(?isFwd = false)
}
```

### Listing 5: Example where the ?focus variable is in a different triple pattern than ?facet and ?facetValue.

```sparql
SELECT ?focus ?facet ?facetValue {
?x
a :Movie ;
:actor ?focus ;
}
```

### Listing 6: Facet counts

```sparql
SELECT ?facet
(COUNT(DISTINCT ?facetValue)
AS ?facetCount)
{ # Facet values relation supplying
?focus ?facet ?facetValue
}
GROUP BY ?facet
```

### Listing 7: Facet value counts

```sparql
SELECT ?predicate ?value
(COUNT(DISTINCT ?focus)
AS ?facetValueCount)
{ # Facet values relation supplying
?focus ?facet ?facetValue
}
GROUP BY ?focus
```

Figure 6: SPARQL queries for facet values

Figure 7: SPARQL queries for computing facet counts and facet value counts based on facet value relation
• **CP1**: Property value based transition  
  (Find all instances which, additional to satisfying all restrictions defined by the state within the browsing scenario, have a certain property value)

• **CP2**: Property path based transition  
  (Find all instances which additionally realize this property path with any property value)

• **CP3**: Property path value based transition  
  (Find all instances which additionally have a certain value at the end of a property path) This is CP1 with a property path instead of a property.

• **CP4**: Property class value based transition  
  (Find all instances which additionally have a property value lying in a certain class)

• **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 behind a directly related property)

• **CP7**: Change of numerical data related via a property path of length strictly greater than one edge  
  (Similar to 7, but now the numerical data is indirectly related to the instances via a property path)

• **CP8**: Restrictions of numerical data where multiple dimensions are involved  
  (Choke points 7 and 8 under the assumption that bounds have been chosen for more than one dimension of numerical data, here, we count latitude and longitude numerical values together as one dimension)

• **CP9**: Unbounded intervals involved in numerical data  
  (Choke points 7, 8, 9 when intervals are unbounded and only an upper or lower bound is chosen)

• **CP10**: Undoing former restrictions to previous state  
  (Go back to instances of a previous step)

• **CP11**: Entity-type switch changing the solution space  
  (Change of the solution space while keeping the current filter selections)

• **CP12**: Complicated property paths or circles  
  (Choke points 3 and 4 with advanced property paths involved)

• **CP13**: Inverse direction of an edge involved in property path based transition  
  (Property path value and property value based transitions where the property path involves traversing edges in the inverse direction)

• **CP14**: Numerical restriction over a property path involving the inverse direction of an edge  
  (Additional numerical data restrictions at the end of a property path where the property path involves traversing edges in the inverse direction)
3.4 Generalization of Chokepoints

As mentioned, most checkpoints are related to generating paths and imposing restrictions on the (non-empty set of) reached values. Paths across predicates can be generated in two manners:

- **starting point / path** driven: Generate a path candidate using traversal from the set of resources that match the current facet selection and then check which constraints can be applied on that path.

- **value / endpoint** driven: Assemble a set of candidate target properties based on certain value-based characteristics, then determine which of them can be reached from the current facet selection.

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.
- Number of required reverse traversals (#bw): Some CPs require paths to contain reverse traversals in order for them to qualify as candidates.
- Primary PMF (P-PMF): A PMF controlling the likelihood for forward and backwards traversals. The primary PMF uses “draw-with-replacement semantics”, i.e. whenever a (direction, weight) pair is chosen, that entry is removed.
- Fallback PMF (F-PMF): A static PMF (i.e. without draw-with-replacement) that is consulted once the primary PMF is consumed. This is useful to allowing paths lengths greater than there are entries in the primary PMF.
- Final predicate: The (type of) predicate that must appear in the last step of a path.
- Values: Constraint types over the values.

Table 1 summarizes the path and constraint characteristics of the checkpoints.

Most checkpoints are related to finding paths in the data and imposing restrictions on the reached values. However, the following ones are noteworthy as they do not fall into this schema:

- **CP4** In order for a value to lie in a certain class, that value has must have a corresponding rdf:type statement. Hence, we first search for a path ending in rdf:type. Then we move up to the parent FacetNode, pick a candidate value among those having rdf:type properties, and constrain that FacetNode to the chosen value.
- **CP10** is about reverting the faceted browsing session state.
- **CP11** is about navigating from one set of resources (such as movies) to a related one (such as actors). Application of this checkpoint changes the focus of a faceted query. In our design of CP11, we pick both a property and a type. The values of the chosen property are then constrained to that type. Combinations leading to empty result sets are rejected.
- **CP12’s** focus is on non-simple property path expressions as well as cycle detection. This CP is not considered in version 2 of the benchmark.
### 3.5 Processing Hierarchical Data

Hierarchical data is common in RDF/OWL, such as in the form of class hierarchies. Unsurprisingly, for this reason CP5 was devised (substitute a class for one of its sub-classes). Generally, in order to establish a hierarchy in the case of faceted browsing, we require an additional binary relation that denotes a parent-child relationship between facet values, e.g. `rdfs:subClassOf`.

In the remaining we present important SPARQL queries to work with hierarchical data. Note, that `rdfs:subClassOf` acts as a placeholder that can be substituted for any other property path expression.

#### Cycle-aware Detection of Top Level Elements

Top-level elements (roots) of a hierarchy are starting points for its exploration. The naive approach for constructing a hierarchy from a binary relation, such as denoted by `rdfs:subClassOf` is, is to use all elements without parents as top-level elements. However, as all elements in a cycle have parents, this approach may yield hierarchies that do not cover all entries of the underlying relation. The query shown in Listing 8 detects all top-level elements including those that are members cycles. Formally, roots are all nodes for which all parents occur as descendents: On the one hand, a node without parents trivially satisfies this condition. On the other hand, a node has to be a root if it is the ancestor of all its parents.

```sparql
SELECT DISTINCT ?root {
  [] rdfs:subClassOf ?root
  FILTER(NOT EXISTS {
    ?parent rdfs:subClassOf ?root
  })
}
```
Cycle Detection

The query in Listing 9 is used to detect resources that are members of one or more cycles. It can be used to ensure that e.g. a chosen sub class is a strict one.

```
SELECT DISTINCT ?start {
    ?start rdfs:subClassOf+ ?end
    FILTER(?start = ?end)
}
```

Listing 9: Query for entities that are members of one or more cycles

Cluster cycles

While the previous query simply detects whether resources are part of a cycle, Listing 10 extends this query to yield all cycles (in the form of generated ids) and their members. It is mainly used for logging / reporting purposes.

```
# Derive a cycle ID from the ordered list of cycle members,
# and pair this ID with every respective member
SELECT (GROUP_CONCAT(STR(?related)) AS ?cycle) ?member {
    # Associate each member of a cycle with
    # the ordered list of all further cycle members
    SELECT ?member ?related {
        # Find all resources part of a cycle
        SELECT DISTINCT ?member {
            ?member rdfs:subClassOf+ ?end
            FILTER(?member = ?end)
        }
        # ORDER BY ?member ?related }
    } GROUP BY ?member
}
```

Listing 10: Query for listing all cycles with their members

3.6 Dataset Analysis

In order to implement the CPs, we need a model which holds information about the predicates and their ranges in the benchmark dataset. For example, if we seek paths ending in a numeric predicate,
we need to know which of the predicates are numeric. For this purpose, we create metamodel – data summaries – in RDF as follows: Initially the metamodel contains static triples about which types are numeric (Listing 11). Using the query of Listing 12, we then retrieve all of a benchmark dataset’s predicates and add the appropriate “predicate a rdf:Property” statements to the metamodel. For every literal in the dataset we append the literal’s datatype to the appropriate property’s ranges (Listing 13). Finally, we can query the model for numeric predicates (Listing 14). Furthermore, in order to drive path finding, we enrich the metamodel which information joinsWith relations between all pairs of predicates whose joins are non-empty (Listing 15). Given a set of resources for which we want to find out whether there exist (forward-only) paths to numeric or spatial predicates, we can initially retrieve all of these resources’ outgoing predicates and then apply standard graph search over the join summary in order to find paths that connect the initial predicates with the target ones.

---

Listing 11: Custom RDF metamodel to denote numeric XSD types

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

# NOTE in the XML Schema, 'numeric' is a boolean attribute of XML types, 
# cf. https://www.w3.org/TR/xmlschema11-2/#rf-numeric 
# here, we modeled "numeric" as a class
xsd:int rdfs:subClassOf xsd:numeric .
xsd:integer rdfs:subClassOf xsd:numeric .
xsd:long rdfs:subClassOf xsd:numeric .
xsd:decimal rdfs:subClassOf xsd:numeric .
xsd:float rdfs:subClassOf xsd:numeric .
xsd:double rdfs:subClassOf xsd:numeric .
```

Listing 12: Query to retrieve all predicates used in a dataset

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
CONSTRUCT { ?p a rdf:Property} 
WHERE { SELECT DISTINCT ?p { ?s ?p ?o } }
```

Listing 13: Query to derive an RDF metamodel of ranges for all predicates used with literal values

```
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdf: <http://www.w3.org/2000/01/rdf-schema#>

FILTER(isLiteral(?o)) . BIND(datatype(?o) AS ?r) } }
```

Listing 14: Query on the combined metamodels to determine all numeric properties

```
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdf: <http://www.w3.org/2000/01/rdf-schema#>

```

Listing 15: Query to enrich the metamodel with joinsWith relations

```
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdf: <http://www.w3.org/2000/01/rdf-schema#>

CONSTRUCT { ?x :joinsWith ?y } { 
```
5 # Exclude RDF container membership properties
6 FILTER(!STRSTARTS(STR(?x), 'http://www.w3.org/1999/02/22-rdf-syntax-ns#_'))
7 FILTER(!STRSTARTS(STR(?y), 'http://www.w3.org/1999/02/22-rdf-syntax-ns#_'))
8
9 # Further filter conditions may be added as the use case demands
10}

Listing 15: Construction of the property join summary

4 The LIVED Data Stream Generator

In this section we detail the data generator we created for the faceted browsing benchmark. The data generator simulates a smart home use case where buildings are equipped with devices that host sensors that fire events of observed values. Hence, the generator is comprised of two parts: A dataset of building locations, and the sensor event generation.

The dataset of building locations is built out of real-world data by using a Germany excerpt of OpenStreetMap, retained only those items tagged as “residential buildings”, and converted it to RDF using LinkedGeoData. This way, an initial 134 million triples were obtained for 912 thousand residential buildings. As on this base data, the time required for analysis and simple faceted queries was already in the magnitude of minutes (and thus unsuitable for interactive browsing), we subsequently stripped the data down to the building locations only using the SPARQL query in Listing 16 and our LinkedGeoData toolkit (an outcome of the GeoKnow FP7 project). The final dataset contains all 912K residential buildings, with 2.7 million triples related to stating the type and locations, of which 4000 had labels.

A depiction of the dataset using our Facete2 system is shown in Figure 8. The sensor events generator is built out of LIVED (Long Device Level Energy Data) dataset and uses the LIVED data model to emulate streams of sensors’ observations. The data generator incorporates two types of sensing devices (Pikerton ZBS-110 and Multisensor Mbs 121) and emulates a cumulative load generated by a number of households equipped by specified number of sensing devices. The hierarchy of observations grouped by a device is presented at Figure 9. The data generator is intended for benchmarking performance of analytical applications as well as data processing infrastructures, so it not contains any mimicking model and observation values are generated randomly within possible ranges of each particular sensor. The data generator supports RDF and JSON as output formats and RabbitMQ and REST as destination protocols. The structure of LIVED data model (static and streamed parts) is presented at Figure 10.

The parameters of the dataset generator and their defaults are as follows:

- **SPARQL_ENDPOINT_URL** The SPARQL endpoint to query for location of residential buildings (houses).
- **HOUSES_COUNT** The number of house to fetch from the given SPARQL endpoint for which sensor events will be generated. Default: 10000

---

4 https://github.com/GeoKnow/LinkedGeoData
5 https://github.com/GeoKnow/Facete2
6 http://hobbit.agtinternational.com/
**Figure 8:** Screenshot of locations of residential buildings exported from OpenStreetMap

- **DEVICES_PER_HOUSEHOLD_MIN** The minimum number of devices to generate for a house. Default: 1
- **DEVICES_PER_HOUSEHOLD_MAX** The maximum number of devices to generate for a house. Default: 10
- **SENSORS_PER_DEVICE** The number of sensors per device. Default: 10
- **DATA_SENDING_PERIOD_MS** The increment for timestamps of sensor events. Default: 1000ms
- **ITERATIONS_LIMIT** Number of iterations. Each iteration, all sensors will fire.
- **OUTPUT_FORMAT** The output format, either JSON or RDF. Default: RDF

```
PREFIX lgdo: <http://linkedgeodata.org/ontology/>
PREFIX ogc: <http://www.opengis.net/ont/geosparql#>
PREFIX geom: <http://geovocab.org/geometry#>

CONSTRUCT WHERE {
  ?s a lgdo:BuildingResidential ; geom:geometry ?g .
  ?g ogc:asWKT ?w .
}
```

Listing 16: SPARQL query to extract only the locations of buildings
Figure 9: Hierarchy of observations grouped by sensing devices

Figure 10: The LIVED data model
4.1 Persisting Data Streams

While the LIVED data stream generator is capable of generating fresh streams of data in respect to its configuration, using persisted streams in a benchmark has the advantages of guaranteed determinism and time reduction. In accordance with suggested approaches by the RDF stream processing serialization working group\(^7\), for streams that emit RDF graphs, we can for every event associate its triples with a freshly allocated graph name. This effectively yields an RDF Dataset (with only a single named graph), for which Trig is one of the standard serialization formats. Trig exhibits the useful feature, that concatenation of two Trig files yields again a valid Trig file (in contrast to e.g. XML or JSON). Thus, with Trig, incoming streaming data can be persisted to a file by simply appending to it.

```trig
:event1 {
  :House64437_device2_SmartMeter_Current.Observation_0
    a lived:CurrentObservation ;
    lived:observationType "IRMS" ;
    sosa:featureOfInterest lgd:way64437 ;
    ssn:observationResult :House64437_device2_SmartMeter_Current_Observation_0_SensorOutput ;
    ssn:observedBy :House64437_device2_SmartMeter .
}
:event2 {
  # ...
}
```

Listing 17: Observation event in TRIG format

Note, that Trig conceptually represents an RDF dataset, which in turn does not define an order of its named graphs or contained triples. For this reason, in order to work with such kind of data, we had to create a couple of Jena extensions: First, an order-preserving dataset implementation, that yields graphs, subjects, predicates and objects in the order of their first occurrence upon insert of quads (such as upon de-serialization from a Trig file). Conversely, distinct events must not share graph names, otherwise they will become conflated. Fortunately, Jena’s Trig serialization system also preserves the order, allowing for roundtrips. Second, a shallow wrapper around Jena’s Trig reader that emits such order-preserving datasets for consecutive quads with the same graph name.

5 Hobbit Integration

In this section, we describe how the previously introduced components are combined to form a Hobbit benchmark. For reference, the architecture of the Hobbit platform is depicted in Figure 11. Detailed explanations can be found in the deliverables D2.1 and D2.2.

5.1 Benchmark Execution

In the prior section we have described the scenario generator and data generator. What needs yet to be explained is, how these components work together in an actual benchmark execution. The process

\(^7\)https://www.w3.org/community/rsp/wiki/RSP_Serialization_Group
Figure 11: Architecture of the Hobbit Platform

is quite straightforward: First, based on the given benchmark configuration, the task generator (TG) assembles an initial RDF dataset, which is the union of the spatial dataset about houses and an initial amount of sensor events. The TG then bulk loads this data into its reference store and the system under test. Afterwards, the TG generates the tasks and reference results. Thereby, it alternately consults the scenario generator for the next checkpoint’s SPARQL query and the data generator for the next sensor events. The data of the sensor events is wrapped as a SPARQL update request and is treated as simply another SPARQL request task. Once sufficient tasks and reference results have been generated, the TG sends the tasks out to the system under test, as well as the reference results to the evaluation store (ES). When all tasks have been processed, the evaluation module processes the information in the ES and aggregates for each checkpoint its correctness and performance value.

5.2 Published Resources

The resources of the Faceted Browsing Benchmark are publicly available on GitHub\(^8\). We have consolidated the resources for both versions of the benchmark into the following uniform structure:

- \textit{faceted-browsing-benchmark-parent} - The root of the Java project
  - \textit{faceted-browsing-benchmark-vX-parent} - The root of the first and second versions of the benchmark (X = \{1, 2\})
    * \textit{faceted-browsing-benchmark-vX-core} - The core components of the benchmark, such as data and task generation.
    * \textit{faceted-browsing-benchmark-vX-hobbit} - Hobbit integration of the core
    * \textit{faceted-browsing-benchmark-vX-hobbit-docker-parent} - Docker containers for deployment on a Hobbit platform
      - \textit{benchmark-controller}
      - \textit{task-generator}

\(^8\)https://github.com/hobbit-project/faceted-browsing-benchmark
- data-generator
- evaluation-module

- auxiliary-docker-resources - Additional docker containers that can be used as resources in benchmarks
  
  - podigg-leaf-via-web-server - Podigg data generator image used in the first version of the benchmark
  
  - linkedgeodata-20180719-germany-building - Preloaded database with buildings used in the second version

The project is configured to deploy all docker containers to a release repository\(^9\) from where the Hobbit platform can retrieve them.

### 5.3 Evaluation

Unfortunately due to issues with the path finding component which resulted in the generated queries not to match the checkpoint definitions the results at the time of submission were still skewed. Please refer to the following URL for the latest results:

https://github.com/hobbit-project/faceted-browsing-benchmark/tree/master/results

### 6 Related Work

A comprehensive survey on faceted search on RDF/S datasets has recently been performed by [13]. There, the authors analyze 30 systems in regard to a formal state-based model that captures the essential functionality of faceted exploration. On that basis, different types of transitions are identified, relating to classes, properties, property paths, and complex transition markers (used e.g. to handle blank nodes and support disjunctions). Further, the authors present a terminology alignment of common fundamental concepts in faceted search that in prior literature occurred under varying names.

Faceted exploration overlaps with OLAP (OnLine Analytical Processing) as both domains deal with multi-dimensional data, as each facet can be seen as a dimension. However, the main distinguishing element is, that OLAP cubes have both a fixed schema and information demand (such as sales per month per department) [3]. Under this perspective, it is unsurprising that faceted search systems have also been employed as a means to select resources of interest carrying statistical information (such as using the DataCube vocabulary) for subsequent visualization [8] [10].

A lot of research have been done on overcoming specific challenges in regard to the information overload [14] [4], a phenomenon which occurs if the information available exceeds the user’s ability to process it. Hence, in faceted exploration scenarios this happens when any of the involved sets, namely those of facets, facet values or result items, becomes too large. For example, datasets, such as DBpedia, may contain thousands of predicates, triggering the need for automatically determining an order of relevancy (possibly in regard to the current state). But also a single relevant facet may have several thousand of values (e.g. authors), hence clustering approaches, such as referred to in [11] can be applied to obtain reduction.

\(^{9}\)https://git.project-hobbit.eu/cstadler/faceted-browsing-benchmark-releases
Several benchmarks for assessing the SPARQL performance of triple stores have been devised in the past, such as LUBM [7], SP2 [12], BSBM [5], and WatDiv [2]. Back in 2008, a lack for repeatable benchmarks for faceted browsing engines has been noted [9]. Since then, most progress in overcoming this lack has been made in the geo-spatial domain, where simulated faceted browsing scenarios have been applied to evaluate the performance of queries having varying numbers of restrictions along the spatial and thematic dimensions. These scenarios consist in general of simple facet selections and iterative panning and zooming of the map [6]. It is noteworthy, that [1] considers tile-based precomputed facet counts. However, to the best of our knowledge, our work introduces the first system dedicated to a detailed list of specific choke points of faceted browsing on RDF data.

7 Conclusions and Future Work

In this deliverable, we presented the second version of the SPARQL-based faceted browsing benchmark framework. The highlights of our work are (1) a system for creating faceted queries, (2) a scenario generator that simulates checkpoint-motivated interactions taking place in a faceted browsing session, (3) streaming data generation, (4) the benchmark generator formed by the combination of scenario and data generation, and (5) integration into the Hobbit platform.

The framework is designed for flexibility and extensibility. For example, one can add custom checkpoints and configure the likeliness for them to be picked during scenario generation. Examples of future work include extending the scenario generator to also create conjunctive constraints. Furthermore, we aim at releasing the faceted browsing system as a standalone library, as it may simplify certain Linked Data application development tasks. For instance, one use case is to realize a faceted browser for DCAT-based dataset catalogues which also supports bulk operations (e.g. downloading) over all items matching a faceted query. From the scientific side, we plan to analyze how well SPARQL query caching systems perform in regard to faceted browsing workloads and investigate the potential for improvements.

References


10Latest version: https://github.com/GeoKnow/GeoBenchLab/tree/master/FacetBench


