D3.1.2b: Techniques for Resource Discovery

This report documents research and development work related to the Resource Discovery component of the SemaGrow Stack. This component maintains and serves metadata about the contents of each data source federated under the SemaGrow Stack. This metadata supports the SemaGrow Stack with making informed decisions regarding how to decompose a query into fragments and how to distribute these fragments among the federated data sources. By making such decisions the SemaGrow Stack not only optimizes query execution but also circumvents unavailable or unresponsive endpoints.
## Document History

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Author (Partner)</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Draft v0.8</td>
<td>13/12/2013</td>
<td>NCSR-D</td>
<td>First draft</td>
</tr>
<tr>
<td>Draft v0.9</td>
<td>20/12/2013</td>
<td>SWC</td>
<td>Internal review</td>
</tr>
<tr>
<td>Final v1.0</td>
<td>23/12/2013</td>
<td>NCSR-D</td>
<td>Delivered as D3.1.1</td>
</tr>
<tr>
<td>Draft v1.2</td>
<td>11/8/2014</td>
<td>NCSR-D</td>
<td>Added new material on Self-tuning histograms (new Sections 2.4 and 4.5), updated literature review (Section 2.5), various minor fixes.</td>
</tr>
<tr>
<td>Draft v1.3</td>
<td>13/10/2014</td>
<td>NCSR-D</td>
<td>The Sevod vocabulary for histogram serialization (Section 4.3)</td>
</tr>
<tr>
<td>Draft v1.4</td>
<td>30/10/2014</td>
<td>SWC</td>
<td>Internal review</td>
</tr>
<tr>
<td>Intermediate v1.5</td>
<td>21/10/2014</td>
<td>NCSR-D</td>
<td>Delivered as D3.1.2(i)</td>
</tr>
<tr>
<td>Draft v1.8</td>
<td>3/7/2015</td>
<td>NCSR-D</td>
<td>Minor edits and corrections, and updated planning (Section 3.4 and Section 4.5).</td>
</tr>
<tr>
<td>Draft v1.9</td>
<td>10/7/2015</td>
<td>SWC</td>
<td>Internal review</td>
</tr>
<tr>
<td>Final v2.0</td>
<td>31/7/2015</td>
<td>NCSR-D</td>
<td>Delivered as D3.1.2</td>
</tr>
<tr>
<td>Draft v2.1</td>
<td>18/1/2016</td>
<td>NCSR-D</td>
<td>Details of the non-methodological work has been added in Section 1.2.1.</td>
</tr>
<tr>
<td>Draft v2.2</td>
<td></td>
<td>SWC</td>
<td>Internal review</td>
</tr>
<tr>
<td>Final v3.0</td>
<td>4/3/2016</td>
<td>NCSR-D</td>
<td>Delivered as D3.1.2b</td>
</tr>
</tbody>
</table>
EXECUTIVE SUMMARY

This deliverable documents research and development work related to the Resource Discovery component, which maintains and serves metadata about the contents of each data source federated under the SemaGrow Stack. This metadata supports the SemaGrow Stack with making informed decisions regarding how to decompose a query into fragments and how to distribute these fragments among the federated data sources. By making such decisions the SemaGrow Stack not only optimizes query execution but also circumvents unavailable or unresponsive endpoints.

Although our work is largely based on previous research on distributed databases, our extension will specifically target the more open nature of the Linked Open Data cloud and the Semantic Web in general. For this, we need to devise methods for efficiently indexing the contents of a federation of data sources without assuming any centralized control over the way data is distributed among the members of the federation.

In this context, the work carried out in Task 3.1 and reported in this document pertains to:

- Developing, prototyping, and evaluating methods for using metadata regarding the schemas used in each data source (schema-level metadata), the data stored in each source (instance-level metadata), the known schema alignments, and measurements about system load in order to respond to a triple pattern with the list of data sources that might hold matching triples and the expected responsiveness if these sources.
- Developing, prototyping, and evaluating methods for extracting instance-level metadata from query feedback, that is, by analysing the results of the normal query workload without imposing overhead queries that are made by the system only for the purpose of extracting metadata about its data sources.
- Developing, prototyping, and evaluating methods for efficiently storing, maintaining, and serving such metadata.

This first report about work in Task 3.1 is structured as follows: after an introduction (including a description of the position of Resource Discovery in the SemaGrow Stack architecture) we proceed to review relevant literature. We then document the architecture and current state of development of the Resource Discovery modules, also presenting the research and development plan for subsequent reporting periods.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>INTRODUCTION</td>
<td>6</td>
</tr>
<tr>
<td>1.1</td>
<td>Purpose and Scope</td>
<td>6</td>
</tr>
<tr>
<td>1.2</td>
<td>Approach</td>
<td>6</td>
</tr>
<tr>
<td>1.3</td>
<td>Relation to other Work Packages and Deliverables</td>
<td>7</td>
</tr>
<tr>
<td>1.4</td>
<td>Big Data Aspects</td>
<td>10</td>
</tr>
<tr>
<td>2.</td>
<td>BACKGROUND</td>
<td>11</td>
</tr>
<tr>
<td>2.1</td>
<td>Resource Discovery</td>
<td>11</td>
</tr>
<tr>
<td>2.2</td>
<td>Source Description Schemas</td>
<td>12</td>
</tr>
<tr>
<td>2.3</td>
<td>Implementing POWDER</td>
<td>13</td>
</tr>
<tr>
<td>2.4</td>
<td>Self-Tuning Source Descriptions</td>
<td>14</td>
</tr>
<tr>
<td>2.5</td>
<td>Conclusions</td>
<td>16</td>
</tr>
<tr>
<td>3.</td>
<td>RESOURCE SELECTOR</td>
<td>18</td>
</tr>
<tr>
<td>3.1</td>
<td>Query Patterns</td>
<td>18</td>
</tr>
<tr>
<td>3.2</td>
<td>Available Information</td>
<td>18</td>
</tr>
<tr>
<td>3.3</td>
<td>Internal Architecture and Information Flow</td>
<td>20</td>
</tr>
<tr>
<td>3.4</td>
<td>Current status and future steps</td>
<td>21</td>
</tr>
<tr>
<td>4.</td>
<td>SOURCE METADATA</td>
<td>22</td>
</tr>
<tr>
<td>4.1</td>
<td>Internal Architecture and Information Flow</td>
<td>22</td>
</tr>
<tr>
<td>4.2</td>
<td>Instance-level Metadata Representation</td>
<td>23</td>
</tr>
<tr>
<td>4.3</td>
<td>The Instance Metadata Repository</td>
<td>26</td>
</tr>
<tr>
<td>4.4</td>
<td>Query Feedback Analysis</td>
<td>28</td>
</tr>
<tr>
<td>4.5</td>
<td>Current status and future steps</td>
<td>29</td>
</tr>
<tr>
<td>REFERENCES</td>
<td></td>
<td>31</td>
</tr>
<tr>
<td>APPENDIX A: RESOURCE DISCOVERY API</td>
<td></td>
<td>33</td>
</tr>
<tr>
<td>A1.</td>
<td>Interface EquivalentURI</td>
<td>33</td>
</tr>
<tr>
<td>A2.</td>
<td>Interface InstanceIndex</td>
<td>34</td>
</tr>
<tr>
<td>A3.</td>
<td>Interface Measurement</td>
<td>35</td>
</tr>
<tr>
<td>A4.</td>
<td>Interface PatternDiscovery</td>
<td>36</td>
</tr>
<tr>
<td>A5.</td>
<td>Interface QueryTransformation</td>
<td>36</td>
</tr>
<tr>
<td>A6.</td>
<td>Interface ResourceSelector</td>
<td>37</td>
</tr>
<tr>
<td>A7.</td>
<td>Interface SchemaIndex</td>
<td>38</td>
</tr>
<tr>
<td>A8.</td>
<td>Interface AnnotatedResource</td>
<td>39</td>
</tr>
<tr>
<td>APPENDIX B: STRHIST FOUNDATIONS</td>
<td></td>
<td>42</td>
</tr>
</tbody>
</table>
## LIST OF TERMS AND ABBREVIATIONS

<table>
<thead>
<tr>
<th>Term/Abbreviation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data source</td>
<td>In the context of this document, one of the SPARQL endpoints federated by the SemaGrow Stack. Besides their location, the SemaGrow Stack also retains metadata about its data sources' contents and schema.</td>
</tr>
<tr>
<td>Heterogeneous data sources</td>
<td>Data sources the data of which follow different schemas.</td>
</tr>
<tr>
<td>POWDER</td>
<td>The Protocol for Web Description Resources (POWDER) is the W3C Recommendation for expressing and serving RDF annotations over groupings of resources, where such groupings are defined via regular expressions over the literal values of their URIs.</td>
</tr>
<tr>
<td>Query pattern</td>
<td><em>Triple patterns</em> are sometime also called <em>query patterns</em> in this document, emphasising that they are the building blocks of queries.</td>
</tr>
<tr>
<td>Querying</td>
<td>Querying retrieves a solution sequence that corresponds to the ways in which the query’s pattern matches the data that is being queried.</td>
</tr>
<tr>
<td>RDF</td>
<td>The Resource Description Framework (RDF) is the W3C Recommendation for representing meta-information about resources in the World Wide Web.</td>
</tr>
<tr>
<td>SemaGrow Stack</td>
<td>The SemaGrow Stack integrates the SemaGrow components needed in order to offer a single SPARQL endpoint that federates a number of large-scale, heterogeneous data sources, also exposed as SPARQL endpoints.</td>
</tr>
<tr>
<td>SPARQL</td>
<td>SPARQL is a W3C Recommendation for expressing queries across diverse data sources, whether the data is stored natively as RDF or viewed as RDF via middleware. The results of SPARQL queries can be result sets (variable bindings) or RDF graphs.</td>
</tr>
<tr>
<td>SPARQL endpoint</td>
<td>A SPARQL endpoint is a machine-friendly (but also human-inspectable) service that enables clients to query an RDF knowledge base via the SPARQL language.</td>
</tr>
<tr>
<td>SPARQL endpoint federation</td>
<td>A SPARQL endpoint federation provides a single endpoint through which to query distributed remote SPARQL endpoints. Federations are queried in the SPARQL Federated Query Extension, a W3C Proposed Recommendation.</td>
</tr>
<tr>
<td>Triple pattern</td>
<td>SPARQL queries are made up of <em>triple patterns</em>, subject-predicate-object triples of either specific values or variables. Triple patterns are sometime also called <em>query patterns</em>, emphasising that they are the building blocks of queries.</td>
</tr>
<tr>
<td>Triple store</td>
<td>A triple store is a database that is optimized for the storage and retrieval of RDF data.</td>
</tr>
</tbody>
</table>
1. INTRODUCTION

1.1 Purpose and Scope

This report documents the Resource Discovery component of the SemaGrow Stack. Resource Discovery maintains metadata about data sources, so that an informed decision can be made regarding which data sources might hold triples matching a given triple pattern. This information is used by the Query Decomposition component in order to decompose the original query into fragments for which the sources that might hold relevant data are known. Furthermore, Resource Discovery takes into account known alignments between vocabularies, so that it also informs Query Decomposition about data sources that can only be queried by “translating” the original query into the schema of the data source.

In this context, the work carried out in Task 3.1 and reported in this document pertains to:

- Developing the Resource Selector, the module that uses metadata regarding the schemas used in each data source (schema-level metadata), the data stored in each source (instance-level metadata), the known alignments, and measurements about system load in order to respond to a triple pattern with the list of sources that might hold matching triples.
- Developing efficient methods for storing and indexing the metadata used by the Resource Selector.
- Measuring the efficiency of the Resource Selector and the accuracy of its predictions. This evaluates the benefit of using the Resource Selector by comparison to querying all sources in the federation.

This report is structured as follows: after this introduction (including a description of the interfaces between Resource Discovery and the rest of the SemaGrow Stack Components) we proceed to review the relevant literature in database indexing and in data source metadata formalisms and vocabularies, we then document the Resource Selector module (with some specifics of its Java implementation documented in Appendix A) and the metadata indexes, and conclude.

1.2 Approach

Work in Task 3.1 is organized in three phases:

- During the initial phase (M4-M12) we have established the internal architecture of the component, conforming to the overall system architecture (Deliverable D2.3.2), we have also developed a mock-up prototype that tests the architecture without implementing any advanced functionality. We have also reviewed the literature on data source annotation, and drafted a first version of the formalism that will be used internally to store data source metadata. Finally, we have carried out initial validation experiments regarding the development of efficient indexing methods for this formalism.
- During the main phase (M14-M24) we will elaborated and updated our review of the state-of-the-art in indexing and we carried out the bulk of the algorithmic design and prototype development work on extracting instance-level metadata.
- During the final phase (M25-M30) we will carry out experiments, work on the efficiency with which we index and serve instance-level metadata and the efficiency of the overall Resource Discovery prototype and make improvements where necessary. We will also polish and harden the implementation, preparing it for inclusion in the final Semagrow Stack prototype.

This work will be reported in two documents, Deliverable D3.1.1 (M12) reported on the results of the initial phase. Deliverable D3.1.2 (due M30) will update and expand Deliverable D3.1.1 with results obtained during the main and final phases. This intermediate version (D3.1.2i) reports on the results of the main phase and is prepared for the purposes of the 2nd year project review.

1.2.1 Overview of work reported in M25-M30

Work during M25-M30 is not methodological and mainly concerns improvements on the overall performance of the Resource Selector and integration with third-party systems. Therefore, most of the work is reflected in the prototypes. In particular, the following has been developed/improved during M25-M30

1. A mechanism that generates and executes ASK queries subsequent to a metadata source selection. The execution of such queries may reduce relevant data sources and improve the quality of the execution plans produced by the Query Decomposition component (D3.4.2)
2. A caching mechanism for both the metadata retrieval and ASK query results that improved the overall resource selection time.
3. Various minor improvements and stability fixes on the functionality of the Resource Selector component developed in previous months.

4. A configuration and instantiation module for plug in third-party source selection algorithms. This module can be used to enable the user to choose between various source selection algorithms without recompiling the SemaGrow stack codebase. Moreover, glue code has been developed for the integration of the HiBiSCus selection algorithm.

Work during M25-M30 also concerns preparing two publications disseminating Task 3.1 outcomes:


1.3 Relation to other Work Packages and Deliverables

The SemaGrow Stack integrates the components needed in order to offer a single SPARQL endpoint that federates a number of heterogeneous data sources, also exposed as SPARQL endpoints. The main difference between the SemaGrow Stack and most existing distributed querying solutions is that SemaGrow targets the federation of heterogeneous and independently provided data sources. In other words, SemaGrow aims to offer the most efficient distributed querying solution that can be achieved without controlling the way data is distributed among sources and, in general, without having the responsibility to centrally manage the data sources of the federation.

![Figure 1: Position of the Resource Discovery component in the overall SemaGrow Stack](image)

This section highlights the position of the Resource Discovery component in the overall SemaGrow Stack (Figure 1) and also explains the interactions with work in other tasks and the components delivered in them.

In summary, work in this task interacts with:
- WP2: Use Cases and Architecture and Task 5.1: Task Semantic Store Infrastructure, from where it receives requirements and the infrastructural work for the development of the SemaGrow Stack components.
- Task 3.2: Ontology Alignment, where the Query Pattern Discovery component is developed (see also Section 1.3.2).
• Task 3.4: Heterogeneous Distributed Semantic Querying, where the Query Decomposition and Federated Query Manager components are developed (see also Section 1.3.1).

• Task 5.2: Synergetic Semantic Annotation Environment, where the data source annotation tool is developed (see also Section 1.3.1). Work in this task sets the annotation schema and other requirements for the annotation tool.

• Task 5.4: Integration and Deployment, making components available for integration.

• WP6: Real-life Deployment and User Evaluation, receiving evaluation results.

1.3.1 Resource Discovery and Distributed Querying

Let us consider the following query: “find all situations where a potato crops yielded more than 5 while precipitation was less than 5”, expressed in SPARQL as:\(^1\)

\[
\]

The Query Decomposition component decides about the optimal way to break up queries based on what data sources hold triples that match the patterns in the query and, where more than one data sources have relevant data, parameters such as the number of results requested by the user or whether it is preferable to get fewer results very quickly. Query Decomposition gets these estimations from the Resource Selector.

More specifically, Query Decomposition sends to Resource Selector the triple patterns in the query and Resource Selector responds with information about which data sources should be contacted for each and an estimation of the number of matching triples and of the responsiveness of the data source (Table 1).

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Data Source</td>
</tr>
<tr>
<td></td>
<td><a href="http://potato-crops.org/endpoint">http://potato-crops.org/endpoint</a></td>
</tr>
<tr>
<td>VAR <a href="http://agro-sci.org/vocab#yield">http://agro-sci.org/vocab#yield</a> VAR</td>
<td><a href="http://all-crops.org/endpoint">http://all-crops.org/endpoint</a></td>
</tr>
<tr>
<td></td>
<td><a href="http://potato-crops.org/endpoint">http://potato-crops.org/endpoint</a></td>
</tr>
<tr>
<td>VAR <a href="http://agro-sci.org/vocab#located">http://agro-sci.org/vocab#located</a> VAR</td>
<td><a href="http://all-crops.org/endpoint">http://all-crops.org/endpoint</a></td>
</tr>
<tr>
<td></td>
<td><a href="http://potato-crops.org/endpoint">http://potato-crops.org/endpoint</a></td>
</tr>
<tr>
<td>VAR <a href="http://metdat.org/vocab#precipitation">http://metdat.org/vocab#precipitation</a> VAR</td>
<td><a href="http://weatherdata.org/endpoint">http://weatherdata.org/endpoint</a></td>
</tr>
<tr>
<td></td>
<td><a href="http://meteo-archive.org/endpoint">http://meteo-archive.org/endpoint</a></td>
</tr>
</tbody>
</table>

Resource Selector looks up with these responses in data source metadata, such as expected delay and used schemas (with estimated data volume). This metadata is used to, in our example, get the answers in Table 1 based on the predicate’s URI’s namespace; except for the answers for this pattern:

VAR <http://agro-sci.org/vocab#cropsOf> <http://plant.org/vegetable/potato>

which combines schema-level metadata with instance-level metadata based on the object’s URI. Specifically, this pattern also tells Resource Selector to look up <http://plant.org/vegetable/potato> in the instance-level metadata, which knows not only about the schema used, but also about specific instances in each data source. The instance-level index approximates the number of triples where this URI is mentioned in each data source.

Although in principle any repository infrastructure can be used to store such metadata, it is one of the research objectives of SemaGrow to experiment with taking advantage of naming convention regularities in order to represent these approximations; and to develop a triple store that is especially well suited to indexing naming regularities expressed as regular expressions. The schema and instance-level metadata vocabularies and the design of the relevant repositories are discussed in detail later in this document (Section 4).

\(^1\) We do not delve into units, such as tons of production per acre or mm of precipitation, or conversions between them.
This metadata is maintained semi-automatically: a core source description must be provided by the data provider when joining the federation, using authoring tools and tutorials produced by the project (WP5). Instance metadata and responsiveness statistics may also be provided, but are also automatically maintained based on measurements extracted from query results and received from the Federated Query Manager. The Source Metadata Access module receives and analyses measurements from the Federated Query Manager and also mediates access to the metadata repositories by Resource Selector and the manual data source annotation tool.

1.3.2 Resource Discovery and Ontology Alignment

In order to handle heterogeneous data sources, the Ontology Alignment Tool aligns the different schemas used by the various data sources. The vocabulary mappings established by this process are served to Resource Discovery by the Query Transformation component. Resource Selector tries all URIs in the patterns it receives against Query Transformation in order to check if there are known mappings. For each URI, Query Transformation responds with URIs in different schemas and a numerical evaluation of the semantic proximity, confidence in the mapping, or any other measure of preference towards a mapping.

Resource Selector uses its knowledge of the schemas used in each source, in order to decide whether there are further sources that could serve data matching the mapped triple pattern, besides the sources serving data matching the original triple pattern. Query Transformation responds on a URI-by-URI basis, so Resource Selector is responsible for filtering mapping combinations for the pattern as a whole, so that only mapped patterns that can be served by the same source are passed back to Query Decomposition. It should be noted that Resource Selector operates at the level of single patterns; a similar check about the feasibility of joining mapped patterns together based on their schema is performed by Query Decomposition, as this latter component operates at the level of the complete query.

In our example, ontology alignment would augment the output shown in Table 1 by providing the mappings shown in Table 2. As a result, the output for the first pattern in Table 1 would be as shown in Table 3.

<p>| Table 2: Example interaction between Resource Selector and Query Pattern Discovery |
|-----------------------------------|-----------------------------------|</p>
<table>
<thead>
<tr>
<th><strong>Input</strong></th>
<th><strong>Output</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>URI Mapping</td>
<td>Preference</td>
</tr>
<tr>
<td><a href="http://plant.org/id#potato">http://plant.org/id#potato</a></td>
<td><a href="http://www.freebase.com/m/05vtc">http://www.freebase.com/m/05vtc</a></td>
</tr>
<tr>
<td><a href="http://plant.org/id#potato">http://plant.org/id#potato</a></td>
<td><a href="http://agrodata.org/plantae/solanum">http://agrodata.org/plantae/solanum</a></td>
</tr>
<tr>
<td><a href="http://agro-sci.org/voc#cropsOf">http://agro-sci.org/voc#cropsOf</a></td>
<td><a href="http://agrodata.org/cultivation">http://agrodata.org/cultivation</a></td>
</tr>
</tbody>
</table>

<p>| Table 3: Example interaction between Query Decomposition and Resource Selector, including results obtained via mappings from Query Pattern Discovery |
|-----------------------------------|-----------------------------------|-----------------------------------|-----------------------------------|</p>
<table>
<thead>
<tr>
<th><strong>Mapped Pattern</strong></th>
<th><strong>Data Source</strong></th>
<th><strong>Estimated</strong></th>
<th><strong>Estimated</strong></th>
<th><strong>Semantic</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>VAR <a href="http://agro-sci.org/voc#cropsOf">http://agro-sci.org/voc#cropsOf</a></td>
<td><a href="http://plant.org/id#potato">http://plant.org/id#potato</a></td>
<td><a href="http://all-crops.org/endpoint">http://all-crops.org/endpoint</a></td>
<td>300</td>
<td>200</td>
</tr>
<tr>
<td>VAR <a href="http://agro-sci.org/voc#cropsOf">http://agro-sci.org/voc#cropsOf</a></td>
<td><a href="http://www.freebase.com/m/05vtc">http://www.freebase.com/m/05vtc</a></td>
<td><a href="http://all-crops.org/endpoint">http://all-crops.org/endpoint</a></td>
<td>400</td>
<td>50</td>
</tr>
<tr>
<td>VAR <a href="http://agrodata.org/cultivation">http://agrodata.org/cultivation</a></td>
<td><a href="http://agrodata.org/plantae/solanum">http://agrodata.org/plantae/solanum</a></td>
<td><a href="http://agrodata.org/endpoint">http://agrodata.org/endpoint</a></td>
<td>5000</td>
<td>30</td>
</tr>
</tbody>
</table>

To recapitulate and clarify the interaction between Resource Discovery and the rest of the Semagrow Stack, handling heterogeneity and distribution is broken down in the following three layers:

- At the level of individual URI resources, Query Pattern Discovery handles schema heterogeneity by providing mappings to (near) equivalent URI resources and Source Metadata Access handles distribution by providing...
references to data sources containing triples where URI resources are mentioned, regardless of their position in the triple.

- At the level of individual triple patterns, Resource Selector handles both heterogeneity and distribution by combining the information received from Query Pattern Discovery and Source Metadata Access into a list of sources where triples can be found that match the triple pattern (or a schema-mapped semantic equivalent).
- At the level of a complete query, the Query Decomposition and Federated Query Management components handles both heterogeneity and distribution by combining the information received from Resource Selector, the reactivity and completeness requirements set by the application posing the query, and the run-time responsiveness of the federated end-points at the query time.

1.4 Big Data Aspects

We see two distinct challenges in querying a federation of big data sources:

- Small results from big data: in this “needle in a haystack” situation we are joining properties from different big datasets in order to retrieve a result set that does not constitute big data by itself. The challenge here is to have an intelligent query execution planner that guides the executor engine along a query execution plan that never retrieves large quantities of results that do not contribute to the end-result because they do not join with subsequent query patterns. Such a query execution planner is the subject matter of Task 3.4, but it relies heavily on the availability of accurate instance-level metadata, which is the subject matter of this deliverable.
- Big results from big data: in this situation the result set constitutes big data, and no amount of query plan optimization can avoid this, since this is what has been requested by the client. Besides the challenges for the query execution engine (Task 3.4), handling this situation is also relevant to the ability of the histogram maintenance mechanism to efficiently handle query feedback that is big data.
2. BACKGROUND

2.1 Resource Discovery

Repository federation and distributed querying are key technologies for efficient and scalable large-scale semantic repositories, supported by the major triple store implementations and formalized in the SPARQL 1.1 specification (Prud’hommeaux and Buiil-Aranda, 2013). Notably, the new specification expects that either all members of a federation are involved in responding to a distributed query or that the query uses the SERVICE in order to specify which repositories to query about each triple pattern in the query.

The SemaGrow vision is that in the absence of such an explicit selection of data sources, intelligent federated querying will automatically:

- optimize query execution by only contacting sources that might hold relevant data; and also
- look for alternative sources of the same or semantically equivalent data, in order to circumvent unavailable or unresponsive endpoints.

Although our work is largely based on previous research on distributed databases, our extension will specifically target the more open nature of the Linked Open Data cloud and the Semantic Web in general. For this, we need to devise methods for lifting the requirement, common in many highly optimized distributed database systems, that:

- the same kind of information is stored across all nodes of the system using the same (or compatible) schema
- that there is central control over the way data is hashed over the distributed system, in order to optimize the distribution of triples in such a way that inter-node communication is minimized (Huang, Abadi, et al., 2011).

Although this level of control allows considerable optimizations, more dynamic solutions are needed in order to satisfy our requirement that data is not cloned and re-hashed into a centralized architecture. To achieve this, we need a way to formally and concisely describe the way data is distributed over the nodes, and to take this information into account when executing distributed queries.

Closer to SemaGrow, several systems maintain indexes of the (kinds of) information stored at each source. Schema-level indexes are light-weight indexes of the properties and types (classes) that occur at each source (Stuckenschmidt, Vdovjak, et al., 2004). Although efficient to maintain, such indexes are too coarse as they are missing instance-level information. Consider, for example, trying to select sources based on their containing triples using classes and properties from commonly used vocabularies, such as FOAF in general or AGROVOC in the SemaGrow domain. This would select practically all sources in the federation and thwart meaningful optimization.

Data summaries, on the other hand, combine schema and instance-level indexing. These stem from database concepts such as histograms, capturing statistics on selectivity and cardinality for the purpose of query optimization (Ioannidis, 2003). In order to index statistics about RDF triples, Q-Trees (Harth, Hose et al., 2010) combine the notion of histograms with that of R-Trees (Guttman, 1984), multi-dimensional trees typically used to index spatial data. The core idea is that a hash function maps URIs and values to numerical “coordinates” that are used to find the node(s) in the Q-Tree corresponding to a (possibly only partially instantiated) triple pattern. Each such node aggregates statistics about all triples within its “bounding box”. This statistics is data summaries for immediately responding to aggregate queries or parameters for query execution optimization; in SemaGrow this can be a ranked list of sources where data should be looked up.

The geometric metaphor employed by Q-Trees works well with heterogeneous data and is also robust to slightly outdated indexes (Hose, Klan et al., 2006). However:

- it does not allow for explicit declarations by repository maintainers regarding the kinds of data their repositories contain, but only relies on indexes incrementally built and maintained by successive queries;
- it fails to take into account the semantic similarity between resources and relies instead on a purely syntactic mapping from URIs and value strings to numerical coordinates. In fact, the method is known to be very sensitive to the hashing function, with no universally good function found (Harth, Hose et al., 2010); and
- it fails to intelligently break up buckets when capacity is exceeded, relying on simple area-minimizing strategies.

Research in SemaGrow is mainly concerned with addressing the first two points above. Addressing the first point will allow us to take advantage of natural groupings of data and of URI conventions which can be most succinctly represented by having the data providers describe the intentionality of such groupings and conventions, rather than trying to reconstruct it by extensional observations. Naturally, without excluding that the system refines and augments...
manually provided information by analysing queries and query results. Addressing the second point will allow us to handle schema heterogeneity and take advantage of known alignments between schemas.

2.2 Source Description Schemas

In this section we discuss formats and schemas relevant to source descriptions. It should be noted that this discussion is not meant to directly lead to the physical data model used internally in order to store metadata, but rather to serializations for interoperability and for human inspection and editing.

2.2.1 The VoID Vocabulary for linked datasets

VoID is a popular RDF Schema vocabulary for expressing metadata about RDF datasets. VoID was developed in the context of the W3C’s Semantic Web Interest Group and it intended as a bridge between the publishers and users of RDF data, with applications ranging from data discovery to cataloging and archiving of datasets (Alexander et al., 2011). VoID characterizes RDF datasets with metadata pertaining to:

- the title and description of the dataset, the source from where it is derived, its creator, publisher, date of creation, re-using the Dublin Core properties (DCMI, 2010) as well as data licensing
- the topic of the dataset
- technical details, such as the availability of SPARQL endpoints or the location and syntax of complete dataset downloads
- structural metadata, discussed in more detail immediately below
- linkset descriptions, specifying whether this dataset includes mappings to other datasets, and which.

Structural metadata are of particular interest to resource discovery, as they pertain to the naming conventions used for the individuals that the dataset describes as well as to the schemas used in the descriptions. More specifically:

- the void:uriSpace property states that the URIs of all the individuals described in a dataset start with the given prefix
- the void:uriRegexPattern property states that the URIs of all the individuals described in a dataset match the given regular expression in XSD syntax (Appendix G, Peterson et al., 2012)
- The void:vocabulary property states the identifier of (one of) the vocabulary (vocabularies) which the properties and classes used in the descriptions are drawn from

Furthermore, VoID defines properties for stating statistics about the size of the dataset, including the number of triples, the number of distinct subjects in these triples, etc. (Section 4.6, Alexander et al., 2011).

In order to account for datasets that comprise individuals from different namespaces or following different naming conventions or using different vocabularies, VoID foresees the void:subset property for structuring the overall dataset into its homogeneous parts; each part is a dataset in its own right and can be annotated using any of the properties presented above.

For the purposes of resource discovery, VoID is adequate for finding sources for grounding star patterns in the query, where the properties of a known ‘central’ individual need to be retrieved. It is less well-suited for finding sources for path patterns, since VoID makes no assertions regarding the individuals in the object position of triples.

2.2.2 Data Catalogues

The Data Catalogue Vocabulary (DCAT) is the W3C Recommendation for describing datasets in data catalogs (Maali and Erickson, 2014). DCAT aims to increase discoverability and to enable applications to consume metadata from multiple catalogs. Perfectly aligned with SemaGrow objectives, DCAT supports decentralized publishing of catalogs and facilitates federated dataset search across sites.

Having said that, DCAT operates at a level of abstraction that cannot cover SemaGrow’s specific requirements. More specifically, DCAT:

- re-uses the Dublin Core properties (DCMI, 2010) to provide authorship, creation date, licensing, and similar annotations for the catalogue itself as well as for the datasets it contains
- provides for topical characterizations that follow SKOS, the Simple Knowledge Organization Scheme (Miles and Bechhofer, 2009)
- provides for describing the datasets in the catalogue at the level of file format and media type, but not at the level of the knowledge representation technology or, even more so, the level of the specific schema used or the individuals the data is about
DCAT is being developed within the Government Linked Data (GLD) Working Group\(^2\) of the W3C, in tandem with more specific profiles (refinements). Closer to SemaGrow is ADMS, the Asset Description Metadata Schema (Archer and Shukair, 2013) is such a DCAT profile, originally developed by the European Commission’s ISA Program\(^3\) and subsequently adopted and further developed by the W3C GLD Working Group. ADMS defines the Semantic Asset class, the instances of which are the semantic resources (schemas, taxonomies, codelists, etc.) that afford interoperability.\(^4\) ADMS also defines the adms:supportedSchema property, used to link a data source with a Semantic Asset in order to denote that the latter is used to describe the datasets served by the former. With these extensions, the ADMS profile targets semantic interoperability more specifically than the generic DCAT model, and is adequate for the schema-level annotation of data sources.

### 2.2.3 The POWDER Protocol

The Protocol for Web Description Resources (POWDER) is a W3C Recommendation for expressing and publishing annotations of Web resources. POWDER takes advantage of natural groupings of URLs in order to annotate all the resources in a regular expression-delineated sub-space of the URI space. The use cases for POWDER are centred on machine-readable trustmarks, disambiguation of subject matter, declarations of accessibility compliance, mobile-friendliness, on-line safety, and so on. These all share the need to make statements about (parts of) Web sites, which is a more generally recurring need. POWDER allows making explicit the intention behind URI structure, which goes beyond exhaustively annotating all resources in a domain as it also represents the knowledge that not only currently known, but also unknown and future resources within a given URI space will have the properties implied by their position in the URI structure. For example, POWDER allows us to express the knowledge that any resource, currently existing or added in the future, under http://red.com has value “red” for the ex:colour property.

Compared to the ad-hoc formalisms and other standards fulfilling similar use cases such as robots.txt files, RFC 5785 (Nottingham and Hammer-Lahav, 2010), and the PICS protocol (Lassila, 1997), POWDER offers greater flexibility in defining URI spaces (regular expressions as opposed to URI prefixes) and has a rigorously defined semantics (Konstantopoulos and Archer, 2009) that affords it a well-defined position the Semantic Web architecture and interoperability with RDF applications.

### 2.3 Implementing POWDER

POWDER has seen various implementations, applied to different applications and domains, including publishing trustmarks conferred by third-party authorities to on-line medical content (Mayer, Karampiperis, et al., 2011) and repository compression (Konstantopoulos et al., 2011; 2012). All these implementations share the property of having very limited interaction with non-POWDER semantic data as POWDER results are combined with other data following a layered inference approach where POWDER statements contribute to semantic inference but not vice versa.

Although implementing POWDER as a distinct layer is sanctioned by the formal semantics of POWDER statements, this direct implementation of POWDER semantics forces a choice between efficiency and compression. On the one hand, forward-chaining approaches either generate all POWDER-inferred triples and push those to the semantic inference layer above or implement a combined RDFS/POWDER inference engine (Konstantopoulos, Archer, et al., 2012) that, again, generates all POWDER and RDFS-inferred triples.

Either way, any benefits POWDER can have to repository compression are lost since the POWDER-inferred triples are made explicit. On the other hand, backward-chaining approaches such as query rewriting achieve compression at the expense of efficiency. Query rewriting operates by transforming triple patterns involving POWDER-inferred predicates into equivalent FILTER clauses which apply the regular expression that the resource must match for the predicate to hold. As a result, and as previously noted by Konstantopoulos and Archer (2011), even the most restrictive POWDER statements can never be used to guard a query but can only be applied as tests over resources selected by previous triple patterns.

#### 2.3.1 Indexing in Databases

To fully realize the potential of POWDER we need to re-design the data structures underlying triple stores and databases in general. The typical design uses a tree structure to store triples (table rows in relational databases), coupled with hash tables that index the externally visible values against the internal node IDs used in the tree. These data structures support the following strategy when searching for a triple pattern (or a tuple):

\(^2\) [http://www.w3.org/2011/gld](http://www.w3.org/2011/gld)

\(^3\) [http://ec.europa.eu/isa/](http://ec.europa.eu/isa/)

1. The bound values in the triple are looked up in the hash table and replaced with internal node IDs
2. The node IDs in the tuple are looked up in the tree
3. The unbound variables are bound by the values retrieved from the tree

This strategy is very efficient when the variables in query patterns are bound to full URIs, but breaks down when retrieving POWDER annotations, as explained immediately below.

### 2.3.2 Retrieving resources by property

Let us assume that we are looking for all resources that have a given property. Current approaches maintain two or more trees for looking up the properties of a given resource (subject-predicate-object indexing) or the resources that have a given property (predicate-object-subject indexing) using the strategy outlined above. Such an index cannot, however, retrieve all resources that have a given POWDER-inferred property without testing every single URI in the knowledge base against the regular expression.

One possibility would be to maintain in the structure all (relevant to POWDER statements) regular expressions a node’s URI matches. Besides the large space usage (effectively annulling any compression benefits from POWDER) this would require prohibitive population costs for adding a POWDER statement to a large repository, since every single URI would have to be visited and checked against the new statement’s regular expression. It becomes obvious that to efficiently implement POWDER processing we need to also be able to efficiently retrieve all URIs that match a regular expression.

### 2.3.3 Retrieving properties of resources

Let us now assume that we are looking for the filler of a POWDER-assigned property when the subject is bound to a URI. We look up the URI in the hash table, but we find no explicit triples in the tree. A possible solution would be to have regular expressions themselves be the subject in the tree, and have the hash table return not only the internal ID of the URI, but also the internal IDs of all matching regular expressions. This approach, however, dramatically increases population costs: every single entry in the hash table needs to be revisited and updated when a new POWDER statement is added. Furthermore, it increases querying time by causing unnecessary regular expression tests.

### 2.4 Self-Tuning Source Descriptions

Significant research has been devoted to constructing and maintaining histograms, the statistics that query optimizations rely upon. In general, histograms are either static or self-tuning:

- Conventionally, static histograms are constructed by scanning the whole dataset and do not reflect subsequent changes in the data distribution, unless rebuilt (Muralikrishna and DeWitt, 1988; Ioannidis and Poosala, 1999)
- In more recent developments, self-tuning histograms leverage query feedback from queries executed by users to extract statistics and incrementally build and refine the histogram.

The latter have been successfully used in relational database management systems as a way to avoid the costly creation of static histograms of massive datasets. These techniques impose a low overhead, as they exploit query feedback collected during query execution. The resulting histogram is focused towards the current query workload, providing more accurate statistics for data regions that are being queried more frequently. Furthermore, they are able to adapt to changes in data distribution and thus are well-suited for datasets with frequently changing contents.

#### 2.4.1 Numerical data

The concept of using feedback from the query execution engine to estimate data distributions is introduced by Chen and Roussopoulos (1994) who propose representing the data distribution as a linear combination of model functions. They use curve fitting, and in particular recursive least square error minimization, to approximate the real distributions from query feedback.

Bruno and Chaudhuri (2002) propose using statistics on intermediate tables (SITs), that is, statistics on expressions corresponding to intermediate nodes of query plans. Their approach is to build histograms for cardinality estimation of selection queries. These one-dimensional histograms for an attribute consist of a set of buckets; each bucket stores the range that it represents and the number of tuples in this range. Adjacent buckets share their ranges’ edges and the ranges of all the buckets together cover the entire range of values of the histogram attribute. In order to estimate of cardinality of arbitrary selection-projection-join (SPJ) queries, histograms are propagated through the query plan. To avoid the propagation of errors through a sequence of operators, it is beneficial to have SITs that match intermediate sub-expressions of the query. Bruno and Chaudhuri also introduced a workload-driven technique to identify a small
subset of SITs that maximize the benefit to the query optimizer and a technique to determine the SIT sets and the auxiliary SITs that should be applied for a given input query. Consider, for instance an SPJ query of the form

\[
R \times S \text{ AND } S.a < 10
\]

where the histograms of tables \(R\) and \(S\) would be used to estimate the selectivity of \(R \times S\) ignoring \(S.a < 10\) and then the histogram of \(S.a\) would be propagated to estimate the selectivity of \(S.a < 10\) over the result of \(R \times S\). Following Bruno and Chaudhuri, we can use statistics that are built on the result of the query expression \(R \times S\) specifically on ranges of \(S.a\) values.

These ideas are relevant to the Learning Optimizer (LEO) framework (Stillger et al., 2001) used in DB2. LEO monitors query execution and accordingly adjusts the cardinality estimates and statistics used by the query optimizer. By comparing estimated and actual cardinalities, LEO gives positive or negative feedback to the statistics and the cardinality model used. Correlations can be also detected when estimates for individual predicates are known to be accurate but some combination of them is not. LEO does not modify statistics, but saves separately adjustment factors such that the product of the adjustment factor and the estimated selectivity derived from the DB2 statistics yields the correct selectivity. Stillger et al. (2001) demonstrated that LEO improves cardinality estimates by orders of magnitude, changing plans to improve performance by orders of magnitude, while adding less than 5% overhead to execution time when collecting query feedback.

**Self-tuning histograms** are progressively refined using data distributions inferred from feedback from the query execution engine about the actual selectivity of range selection operations: After each selection on attribute \(a\), the difference between the estimated and the actual result sizes is calculated and used to refine the frequencies stored in the buckets that cover the values of \(a\) returned by the query. Blame is shared between buckets in proportion to their current frequencies. A damping factor is used to adjust the sensitivity of the histogram to errors. At regular intervals, the histogram is restructured through merging and splitting buckets. Specifically, buckets with similar frequencies are merged and the reclaimed memory space used to split high frequency buckets to smaller and more accurate ones. STGrid (Aboulnaga and Chaudhuri, 1999) extends these ideas to *multidimensional self-tuning histograms* that use query workloads to refine a grid-based histogram structure. These self-tuning histograms are a low-cost alternative to traditional histograms with comparable accuracy. However, since the splitting (or merging) of each bucket entails the splitting (or merging) of several other buckets that could be far away from and unrelated to the original one, overall accuracy is degraded in order to satisfy the grid-partitioning constraint.

To alleviate the poor bucket layout problem of STGrid histograms, Bruno et al. (2001) introduce **STHoles histograms** that allow buckets to overlap, providing for superior structure in comparison with other bucketing schemes. The data structure of STHoles allows them to exploit feedback in a truly multidimensional way and is adopted by many subsequent algorithms (Srivistava et al., 2006; Roh et al., 2010; Khachatryan et al., 2012), including the one developed in SemaGrow. STHoles allow for inclusion relationships between buckets, resulting in a tree-structured histogram where each node represents a bucket. **Holes** are sub-regions of a bucket with different tuple density and are buckets themselves. To refine an STHoles histogram, query results are used to count how many tuples fall inside each bucket of the current histogram. Each partial intersection of query results and a bucket can be used to refine the histogram by drilling new holes, whenever the query results diverge from the prediction made through the bucket’s statistics.

In order to maintain a constant number of buckets, buckets with close tuple densities are merged to make space for new holes. A penalty function, which measures the difference in approximation accuracy between the old and the new histogram, is computed, so as to decide which buckets to merge. **Parent-child merges** are useful to eliminate buckets that become too similar to their parents; **sibling merges** are useful to extrapolate frequency distributions to yet unseen regions in the data domain and also to consolidate buckets with similar density that cover nearby regions.

**ISOMER** (Srivistava et al., 2006) is a more recent feedback-based algorithm for building and maintaining multidimensional histograms. ISOMER uses the histogram structure of STHoles and the information-theoretic principle of *maximum entropy* to refine the histogram based on *Query Feedback Records* (QFR). QFRs are \(<q,N(q)\>\) records that match queries against the size of the query result. Once ISOMER obtains a consistent set of QFRs, the algorithm computes the “simplest” (in terms of entropy) histogram that is consistent with all QFRs added so far. The result is a maximization problem under a system of constraints, solved with *iterative scaling*. Furthermore, to meet a space budget ISOMER discards QFRs merges buckets in a way similar to STHoles.

Except for storing cardinalities, another useful statistic for selectivity estimation of queries with equality or LIKE selection predicates is the number of distinct values. Kaushik and Suciu (2009) presented the first self-tuning histogram modelling cardinalities and distinct value counts, which was based on the same *entropy maximization* (EM) principle as ISOMER but with a different probability space. Due to the computational complexity of the resulting EM problem, they minimize instead the squared distance between the histogram’s estimates and the query feedback viewed as vectors. However, their method only constructs one-dimensional histograms on numerical or categorical data.
Markl et al. (2007) address the problem of combining complementary selectivity estimations from multiple sources (which themselves can be computed using ISOMER histograms) to obtain a consistent selectivity estimation using the idea of maximum entropy. Similar to the approach in ISOMER, this work exploits all available information and avoids biasing the optimizer towards plans for which the least information is known (Bruno et al., 2009).

Khachatryan et al. (2012) note that like traditional index structures such as R-Trees, STHoles fails in high-dimensional data spaces and is sensitive to the order of tree construction. As far as the latter is concerned, they argue that if the first few queries define a top-level bucket structure that is bad, the subsequent tuning is unlikely to correct it. They propose an initializing with subspace buckets which are derived from a subspace clustering algorithm. They use the MineClus cell clustering algorithm (Yiu M.L. and N. Mamoulis, 2003), which outputs a set of clusters with an assigned importance. Each cluster consists of tuples and has dimensions \(d_1, d_2, \ldots, d_k\). The corresponding bucket is the minimal rectangle containing these points-tuples and spans the entire length of every dimension not in \(d_1, d_2, \ldots, d_k\). They showed that the new initialization improves estimation quality and, in some situations, reduces the number of buckets.

### 2.4.2 String and Semantic Web Data

There has been relatively limited amount of work around string selectivity estimation in the field of relational databases. Chaudhuri et al. (2004) proposed to collect multiple candidate identifying substrings of a string using, for example, a Markov estimator and build a regression tree as a combination function of their estimated selectivities, in order to alleviate the selectivity underestimation problem of queries involving string predicates in previous methods, which used independence and Markov assumptions. Lim et al. (2005) introduced CXHist, which is a workload-aware histogram for selectivity estimation supporting a broad class of XML string-based queries. CXHist uses feature distributions to summarize queries and quantize their selectivities into buckets; it then uses a *Naïve Bayes* classifier to capture the mapping between queries and their selectivity.

Within the Semantic Web community itself, Ding et al. (2004) introduced *SWOOGLE*, a Semantic Web search engine that collects metadata, such as classes, class instances and properties for Web documents as well as relations between documents. Auer et al. (2012) introduced *LODStats*, where they compute statistics of RDF for large-scale RDF datasets using a statement-stream based approach and they provide definitions of 32 schema level statistical criteria. Closer to *SemaGrow*, *RDFStats* (Langegger and Woss, 2009), generates statistics for RDF sources. This statistics includes items such as instances per class and histograms per class, property and XML data type. Unlike the *SemaGrow* method, RDFStats histograms are statically re-generated from scratch by executing several and costly SPARQL queries.

For range estimations on strings, RDFStats mentions three possibilities:

- One bucket for each distinct string, resulting in large histograms
- Reducing strings to prefixes
- Using a hash function to reduce the number of distinct strings, although no appropriate general-purpose hash function has been identified.

As Harth et al. (2010) have also noted in relation to Q-Trees for indexing RDF triples (cf. Section 2.1), hashing URIs (or, rather, their string representation) fails to take into account the semantic similarity between resources and relies instead on a purely syntactic mapping from URIs and value strings to numerical coordinates; and no universally good function has been identified.

### 2.5 Conclusions

As stated above, one of our core technical goals is to develop a way to describe the way data is distributed over the nodes of a federation, so that it can be (a) efficiently served and taken into account when executing distributed queries; and (b) serialized in a way allowing for human input and inspection.

We have identified widely adopted approaches, such as VoID, that do not completely cover our requirements and POWDER that has the intuitive syntax, flexibility, and extendibility that allows it to cover our requirements, but is less widely adopted and lacks an efficient implementation of its full semantics.

Based on this, our approach is:

- To assume POWDER as the conceptual basis of our source metadata indexing, formalising the core concept of using regular expressions over URIs to succinctly represent instance-level metadata.
- To interoperate with popular existing schemas, most prominently VoID, by defining an refinement of VoID that exploits this concept to be able to express the full instance-level metadata used in the Semagrow Stack.

We have also reviewed methods for automatically building and maintaining such descriptions. Self-tuning histograms are perfectly aligned with the *SemaGrow* project’s objective of operating without placing any requirements at the
federated sources and can also exploit normal query workload as opposed to inducing histogram generation overheads. However, the relevant databases literature focuses on numerical and, to a lesser extent, categorical data and is not suitable for collecting statistics from RDF triple stores: URIs, which have a prominent position in RDF datasets, would have to be treated as categorical values and not be compressed into ranges. This contradicts one of SemaGrow’s core technical objectives, namely, to break the linear association between the size of the federated data sources and the size of the histograms that index their contents.
3. RESOURCE SELECTOR

This section details Resource Selector the module of the Resource Discovery that combines information received from Query Pattern Discovery and Source Metadata Access into a list of data sources that contain triples that match a triple pattern – or a schema-mapped semantically equivalent pattern.

3.1 Query Patterns

Triple patterns are the elementary query fragments that match triples. In SPARQL, as well as in all RDF graph query languages, triple patterns are made up of a subject, a predicate, and an object all of which can be either a variable or a specific value; the latter being either a URI resource or a concrete datatype value (Section 4.2, Harris and Seaborne, 2013). RDF graphs express both RDF data and their schema, so that URI resources denote both individuals and schema entities (properties and classes).

Table 4 lists all possible triple pattern types matching RDF data, where \( ?X, ?P, ?Y \) stand for variables; \textit{ind}, \textit{prop}, and \textit{class} stand for specific URIs denoting individuals, properties, and classes respectively; and \textit{val} stands for either URI or concrete values. We make distinction between the ubiquitous \textit{rdf:type} property that cannot be used to restrict the possible data source and other properties that can be used. Making this distinction, there are 12 possible types: the 8 combinations that we get by making 3 binary choices (variable vs. grounded value) and the 4 \textit{rdf:type} variations of the 4 combinations where the property is grounded. The right-hand of the table lists the indexing level at which the triple pattern can be exploited in order to predict which data sources to query: property and class URIs are indexed at the schema level, whereas the URIs of individuals are indexed at the instance level.

By observing Table 4 we see that there are four different pattern types where schema-level indexing does not offer any useful optimization, including patterns that appear in common queries, such exploratory queries (Type 100) and type-checking (Type 110t). Furthermore, Type 111, 111t, 011, and 110 triple patterns can also only be optimized by instance-level indexing since some classes and properties can also be extremely common within a domain. Although the importance of each index largely depends on the relative frequency of each triple pattern type in the queries of a given domain and application,\(^5\) it can be argued already at this level of abstraction that instance-level indexing generally offers significant optimization opportunities.

3.2 Available Information

Conceptually, the Resource Selector establishes a three-way association between triple patterns, data sources and related metadata (such as responsiveness and maintaining authority), and the multiplicity of triples in the data source that match the triple pattern (Figure 2). As discussed above, for many useful triple patterns we can use schema and instance-level indexes in order to restrict the range of data sources that need to be queried for a given triple pattern and the extent to which this can be achieved depends on the presence of characteristic, not universally used URIs.

\(^5\) To be measured for the SemaGrow use cases and reported in subsequent versions of this deliverable.
We can then exploit:

- the association between a URI and its schema and a schema-level index, an association between schemas and data sources
- an instance-level index, an association between a URI and the data sources it appears in

in order to calculate (or, at least, approximate) the association in Figure 2 without querying the data sources. Based on this, we refine the logical model as follows: Triple patterns are now broken down into their subject, predicate, object constituents. If any of these is a URI Resource, it is defined in a given schema or vocabulary, which can be identified from the URI’s namespace and looked up in a schema index in order to retrieve the data sources that use this schema. Furthermore, the URI can also be looked up in an instance index, in order to retrieve the data sources that contain triples that mention this URI.

Another feature of the Resource Selector is that it takes into account URI mappings, denoting the semantic equivalence or near equivalence between URI resources and produced by ontology alignment (cf. Deliverable D3.2). URI mappings are attributed with a numerical valuation of the semantic proximity between the mapped resources, used to rank alternative mappings in order of preference. The Resource Selector uses these mappings in order to expand each URI into a list of (near) equivalent URIs and then proceed to look up relevant data sources for all the members of the list.

Although closer to being viable than the association of all possible triple patterns, this logical model (shown in Figure 3) can only be approximated, since storing all URIs mentioned in all data sources of the federation would be even harder than centrally consolidating all these data sources. This approximation is discussed in Section 4 below; in this section we proceed to describe the operation of the Resource Selector assuming the existence of this data structure.
### 3.3 Internal Architecture and Information Flow

In this section we describe the Resource Selector's internal architecture and present the information flow triggered by executing a query.

#### 3.3.1 Description

Resource Selector is implemented in Java. Figure 4 shows the internal organization of the Resource Selector's implementation and the Java interfaces defined for its internal interaction as well as with its interaction with the Query Decomposition and Query Pattern Discovery components, as well as with its fellow Source Metadata module within the Resource Discovery component. The Javadocs of these interfaces are provided in Appendix A: Resource Discovery API.

The Resource Selector implementation is organized around a Controller that coordinates two distinct functionalities, Equivalent URI Retrieval and Source Selection.

Equivalent URI Retrieval interfaces with Query Pattern Discovery in order to expand a URI into a list of (URI, proximity) pairs, providing semantic (near) equivalents in different schemas and a numerical evaluation of the semantic proximity, confidence in the mapping, or any other measure of preference towards a mapping. This interaction is on a URI-by-URI basis, and Equivalent URI Retrieval is responsible for providing triple pattern mappings by combining possible equivalences and estimated the overall proximity of the mapped pattern to the original pattern. This is currently implemented as weak conjunction, assuming the minimum proximity among the proximities of the mappings of all mapped URIs in a pattern.

Source Selection interfaces with Source Metadata Concentrator in order to retrieve a list of annotated data sources for each URI in the expanded list of equivalent patterns. Since the interaction between Equivalent URI Retrieval and Query Pattern Discovery above is on a URI-by-URI basis, Source Selection is responsible for filtering mapping combinations for the pattern as a whole so that only mapped patterns that can be served by the same source are passed back to the Controller.
3.3.2 Information flow

Queries that are received by the SemaGrow SPARQL endpoint are forwarded to Query Decomposition, which sends isolated triple patterns to Resource Selector in order to receive a list of candidate data sources for each pattern and annotations pertaining to the source schema (semantic proximity), volume of matching triples, and usual availability and responsiveness. The entry point to Resource Selector is the Resource Selector Controller, which first uses Equivalent URI Retrieval in order to get a list of semantically (near) equivalent patterns and the numerical valuation of their semantic proximity.

As a second step, the Controller uses Source Selection for each of the (near) equivalent patterns and the original pattern in order to get a list of data sources that contain matching triples and metadata regarding these sources. The Controller sends the union of the selected sources for all (near) equivalent patterns back to Query Decomposition as a response to the triple pattern that the latter originally sent.

3.4 Current status and future steps

During the first reporting period, corresponding to the first phase of Task 3.1 (M4-M12) as specified in the approach to the overall Task (Section 1.2):

- We have established the internal architecture of the Resource Selector module and the internal interface with the Source Metadata module within the Resource Discovery component, as well as (in coordination with Tasks 3.2 and 3.4) the external interfaces to the Query Decomposition and Query Pattern Discovery components.
- We have assumed the current overall system architecture (Deliverable D2.3.2) as a starting point, but in the process of designing Resource Selector we have identified several points that need refinement and update. These will be reported in the intermediate overall system architecture (Deliverable D2.3.2i, M14) in coordination with Task 2.3.
- We have developed a mock-up prototype that tests the architecture without implementing any advanced functionality.

During the second reporting period and up to the second review corresponds to the main phase of Task 3.1 (M14-M24) as specified in the approach to the overall Task (Section 1.2). During that semester, the focus of Task 3.1 was on the underlying methods for extracting and representing source metadata. With respect to the resource selector module, we have developed a fully functional prototype that serves the source metadata extracted by the histogram maintenance method developed during this period (Chapter 4, below). This report was prepared at the end of that period and reviewed during the second review.

Work during the final phase (M25-M30) concerned stability and efficiency fixes and is not reflected on this report, but is only reflected in the prototypes produced. Work during this final phase also concerned preparing the following publication disseminating the work on serializing and representing histograms: A. Charalambidis, S. Konstantopoulos, and V. Karkaletsis, “Dataset Descriptions for Optimizing Federated Querying.” In WWW 2014 Companion Proceedings (poster presentation), Florence, May 2015.
4. SOURCE METADATA

In this section we describe the Source Metadata Access module and also present the current status and future plans for our research in extracting and representing instance-level metadata.

4.1 Internal Architecture and Information Flow

In this section we describe the internal architecture of the Source Metadata module and present the information flow triggered by executing a query.

4.1.1 Description

Metadata Access is implemented in Java. Figure 5 shows its internal organization.

There are three data access interfaces that abstract away from the specific repository or repositories used to store metadata. Schema Index Access handles the schema-level index of data sources containing triples that mention a given class or property. Instance Index Access handles the instance-level index of data sources containing triples that mention the given URI resource. Reactivity Index Access handles the reactivity annotations, such as expected availability, responsiveness, throughput, etc.

The Source Metadata Concentrator abstracts away from this three-tiered organization of metadata and provides an interface through which Resource Selector can receive all data sources mentioning a URI resource, and all reactivity annotations for these data sources.

Data source managers can use the Authoring Environment to manually provide schema and instance-level metadata. Although some minimal requirements will be set for the inclusion of a data source in the federation, it is envisaged that human annotators only provide high-level information, to be refined by the Measurements Analyser.

The Measurements Analyser receives from the Federated Query Manager measurements and statistics regarding:

- the number of instances that each federated data source provided in response to a given query
- the schemas that these instances follow
- the availability and responsiveness characteristics of each federated data source, as observed during receiving (or failing to receive) a response to a given query

From the first of these measurements, the Measurements Analyser extracts instance metadata regarding the number of triples matching a given pattern; from the second, it extracts schema metadata regarding the schemas used in each source; from the third, reactivity metadata.

4.1.2 Information flow

Queries that are received by the SemaGrow SPARQL endpoint are forwarded to Query Decomposition, which sends isolated triple patterns to Resource Selector in order to receive a list of candidate data sources for each pattern and annotations pertaining to the source schema (semantic proximity), volume of matching triples, and usual availability and responsiveness. The Resource Selector queries the Source Metadata Concentrator with a URI.

The Source Metadata Concentrator looks up his URI in the schema and instance index, using the Instance Index Access and Schema Index Access interfaces, and receives lists of data sources where this URI is mentioned.

As a second step, The Source Metadata Concentrator looks up these data sources, using the Reactivity Index Access interface, and receives reactivity annotations for these data sources. The list of annotated data sources is sent back to the Resource Selector.

After to Query Decomposition has decided about the optimal way to distribute the query, the Federated Query Manager plans and executes the distributed query. As results flow back from the distributed data sources to the Federated Query Manager, the latter observes this flow to extract measurements and forward them to the Measurements Analyser.
4.2 Instance-level Metadata Representation

Instance-level metadata supports the Resource Selector module with information regarding data sources that hold triples that mention a given URI resource. Figure 3 (Section 3.2 above) conceptualizes this information as the $U \times S \times N$ relation, where:

- $U$ is a URI resource
- $S$ is a data source
- $N$ is the number of triples in $S$ mentioning $U$

Naturally, such a relation is only an abstraction and cannot be completely stored and queried, since it is even larger than pulling together all the data in all of the sources of the federation. Schema-level indexing approximates this relation by only listing resources that are characteristic of a given schema – classes and properties – so that few and widely used URIs need to be indexed. In Section 2.2 we presented popular vocabularies for describing datasets. In this section we present the Sevod vocabulary, which extends VoID so that it can also cover Semagrow requirements on instance-level metadata. The top Sevod classes and their relationship to their VoID super-classes is shown in Figure 6.

Database histograms capture statistics on cardinality and selectivity for the purpose of query optimization. This statistics is organized as an inclusion hierarchy of buckets, where each bucket stores:

1. The name of an attribute
2. A range of values for this attribute
3. The cardinality of this attribute range, that is, the number of tuples where the given attribute has a value within the given range.

Histogram buckets are captured by void:Dataset instances, which can be hierarchically organized using the void:subset property. VoID, however, does not allow the distinction between dataset partitionings and overlapping datasets. To cover this requirement, Sevod introduces partitions as follows:
Definition: sevod:Partition denotes that a set of void:Dataset instances are a partition of another void:Dataset instance. This is done by using the property sevod:part to link the sevod:Partition instance with the instances that make up the partition and the property sevod:partitions to link it with the instance that is partitioned by them.

Definition: svd:partitions is the functional property that links a sevod:Partition instance with the void:Dataset for which it is a partition.

Definition: The sevod:part property links a sevod:Partition instance with each of the void:Dataset instances that make up the partition. All fillers of this property must also be fillers of the void:subset property of the void:Dataset instance that fills the sevod:Partition instance’s sevod:partitions property.

Figure 6 gives a characteristic example from the Trees4Future metadata.

A further characteristic of database histograms can VoID cannot capture is multi-dimensional buckets, corresponding in RDF to buckets that describe joins of triple patterns:

Definition: A sevod:Join instance connects two void:Dataset instances with an integer value that is (or estimates or approximates) the selectivity of the join of these datasets. The triple element that is joined is denoted by the specific subproperty of the sevod:joins property used to link the sevod:Join instance with the void:Dataset instances.

Definition: The sevod:joins property links a sevod:Join instance with the void:Dataset instances that are joined.

Definition: The sevod:joinSubject property links a sevod:Join instance j with a void:Dataset instance d iff
1. there is also a triple j svd:joins e, where e is a void:Dataset instance; and
2. j denotes the join of d on the subject of its triples with e on the element of its triples denoted by the specific subproperty of devod:joins used in the j svd:joins e triple above.

Definition: The sevod:joinPredicate property links a sevod:Join instance j with a void:Dataset instance d iff
1. there is also a triple j svd:joins e, where e is a void:Dataset instance; and
2. j denotes the join of d on the predicate of its triples with e on the element of its triples denoted by the specific subproperty of devod:joins used in the j svd:joins e triple above

Definition: The sevod:joinObject property links a sevod:Join instance j with a void:Dataset instance d iff
1. there is also a triple j svd:joins e, where e is a void:Dataset instance; and
2. j denotes the join of d on the object of its triples with e on the element of its triples denoted by the specific subproperty of sevod:joins used in the j svd:joins e triple above

Definition: The sevod:selectivity property links a sevod:Join instance with the instance of sevod:SelectivityValue that is (or estimates or approximates) the selectivity of the join denoted by the sevod:Join instance.
Figure 7: Example usage of sevod:Partition, taken from Trees4Future

Figure 8: Example usage of sevod:Join
Finally, bucket statistics do not need to be scalar values, but may also be more complex representations of constraints over or distributions of scalar values that are not precisely known. In order to cover this requirement, we define the range of the sevod:selectivity property to be a class instead of simple numerical fillers. In this manner, we encapsulate statistics under a class that can be extended to cover application-specific requirements. In order to ensure compatibility, we further require that instances of this sevod:SelectivityValue class must have an rdf:value property and that this property has as value an xsd:integer. Other, application specific, properties may be defined as needed for extensions of this class.

**Definition:** Instances of the sevod:SelectivityValue class denote an exact, estimated, or approximated measurement. sevod:SelectivityValue instances must have the rdf:value property with an filler that is an xsd:integer that is either the value itself (if the measurement is exact and certain) or a value that is appropriate to use by applications that do not take into account uncertainty or approximation parameters. Where appropriate, uncertainty or approximation parameters are given by application-specific properties.

### 4.3 The Instance Metadata Repository

#### 4.3.1 Cache-based implementation

As discussed in Section 2.3, inferring properties from POWDER statements can be straight-forwardly implemented by trying to match URIs against all regular expressions in POWDER statements. Looking for more efficient implementations, we have identified an opportunity in using the literal parts of a regular expression in order to quickly narrow down the set of URIs that can possibly match, and only testing those against the regular expression. Suppose, for example, that we maintain an index of all n-grams in the literal parts of regular expressions and all n-grams in the URIs of the knowledge base. When checking for URIs matching a given regular expression, we only need to check the URIs that contain all the n-grams that the regular expression contains, since all other URIs cannot possibly match.

For small values of $n$, this is more concise than maintaining the complete $URI \times regular\ expression$ relation (which is too large to maintain) and for well-chosen n-grams it can greatly improve performance. By well-chosen we mean:

- That the n-gram must appear in responses for frequent queries; but at the same time
- That the n-gram must appear in regular expressions that match relatively few instances, so that it is a meaningful restriction.

For example, the n-gram “tato1” is restrictive (the only matching resource is <http://all-crops.org/id/2011/potato1>), but will only be useful if <http://all-crops.org/id/2011/potato1> appears often in query responses. On the other hand, “http://” appears often, but appears in all URIs, so it offers no restriction. Since this balance is characteristic of caching, our first approach was also to cache useful n-gram matches.

In particular, we used the OpenRDF Sesame framework with PostgreSQL as the physical layer for our POWDER store. Sesame is the most widely-used framework for processing RDF data. It implements parsing, storage, reasoning and querying in SPARQL and offers a flexible Java API that is supported by all leading RDF storage solutions. PostgreSQL is the most stable and efficient open-source relational database management system and can also be used by Sesame as a storage back-end. Furthermore, PostgreSQL provides a plug-in mechanism for customized indexing.

---

Retrieve 1/100 of the repository
Retrieve 1/10,000 of the repository
Retrieve 1/1,000,000 of the repository

The red squares are the times to check the complete column against the regular expression. The blue triangles are the times when candidate tuples are retrieved from the 3-gram index and then checked.

**Figure 10: The time (in sec) to retrieve all matching values versus the size of the repository (in Mtriples).**

The Sesame front-end for PostgreSQL closely matches the typical architecture of triple stores:
- A hash table is used to map external values (URIs or concrete values) to internal identifiers.
- Tree structures are used to index triples of internal identifiers. It is typical to use at least two such indices, one for subject-predicate-object triples (efficiently retrieving all properties of a given resource) and one for predicate-object-subject triples (efficiently retrieving all resources that have a given property value).
- In order to retrieve triples that match a triple pattern, the binded elements of the pattern are first looked up in the hash table. The retrieved internal identifiers are then used to traverse the appropriate tree (depending on which elements of the triple pattern are binded) and find matching triples.
- The internal identifiers in these triples are reverse-looked up in the hash table in order to retrieve the external values (URIs or concrete values).

For the PostgreSQL-backed triple store, these data structures are relational tables and lookup is performed by executing SQL queries.

In order to try our caching approach, we have adapted the Sesame query executor so that it uses LIKE statements in the SQL it produces; these LIKE statements replace property value lookups into the appropriate regular expressions for POWDER-inferred properties. We have also implemented the reCache plug-in that overrides default table indexing for the table that maps URIs to internal identifiers, when responding to LIKE statements. reCache breaks down the literal parts of the regular expression into 3-grams and looks for matching candidates in a hidden SQL table it maintains. If the 3-gram is one that is in the cache, then it retrieves tuple pointers that can be used to directly access the triples table without any further searching. We then only test the URIs in the intersection of the triples pointed by all the literal 3-grams in the regular expression. If, on the other hand, the 3-gram is not in the cache, then reCache falls back to the default tree index (Figure 9).

### 4.3.2 Validation Experiments

Naturally, as with any caching scheme, the rules for deciding what to retain in the cache are essential for the scheme’s success. Before delving into this matter (which is planned for the second project year), we have run a simpler experiment for validating the approach. We have artificially constructed a dataset that has the following characteristic: there are three regular expressions such that 1/100, 1/10,000, and 1/1,000,000 of all the triples in the dataset mention URIs that match each expression, respectively. We have then queried this dataset with three different queries, so that each relies on the retrieval of exactly these triples.

The value of this exercise is to show us the maximum gain that we can ever hope to achieve, since the dataset and regular expressions are constructed so that queries are always answered from the cache and never fall back to the regular PostgreSQL index. It has also helped us map the conditions under which our approach optimizes the POWDER implementation. As can be clearly seen in Figure 10, for high-density queries that retrieve a large fraction of the data in the repository (1/100), it is preferable to rely on the default PostgreSQL index: the reduction in the number of regular expression checks is not great enough to justify the overheads of the algorithm described above. reCache and the
default index break even when retrieving around 1/10,000 of the data, and from then on reCache is increasingly more efficient than the default index.

Naturally, as repositories get larger we expect useful queries to retrieve a smaller fraction of the overall repository size, so that the expectation is that the system will behave more similarly to the 1/1,000,000 case on the right-hand side, giving us ample margin for developing a caching policy that is as close as possible to the blue line in the graph.

On the other hand, it should be noted that this exercise does not directly reflect a specific retrieval scenario: for the purposes of Resource Discovery, retrieval will be more concerned with the opposite direction where the properties of a given URI resource need to be retrieved, rather than all the URI resources that have a given property. The main value of the first validation experiment was to confirm the robustness of PostgreSQL plug-in index mechanism, and to gauge the points past which the optimization counter-balances the overheads.

4.4 Query Feedback Analysis

Query execution optimizers rely on histograms, data structures that approximate data distribution, in order to be able to apply their cost model and compare query execution plans. Histograms can be constructed by scanning the database tables and aggregating the values of the attributes in the table; and similarly maintained in the face of database updates.

This histogram lifecycle, however, cannot be efficiently applied by a system like SemaGrow that concentrates and serves to applications remote data sources over which it has no control: the Semagrow Stack cannot effect that the individual servers maintain and provide the needed statistics. Regardless of any optimization performed internally to each data source, the Semagrow Stack needs to independently maintain the statistics needed to identify data sources that contain relevant data and to optimize the distribution of the various sub-queries among them.

A related strain of work in relational databases is adaptive query processing where histograms are updated by observing and analysing the results of the queries that constitute the client-requested workload, as opposed to executing queries or otherwise imposing a database workload just in order to update the histograms. The majority of the relevant literature focuses on numerical attributes, exploiting the concept of an \( \text{term}[\text{interval}] \) as a description of a set of numerical values that is succinct and that has a length that can be used to estimate the cardinality of many different intervals that have roughly the same density (cf. Section 2.4).

In SemaGrow, we have developed the Self-Tuning RDF Histogram Builder (STRHist), a method that extends core adaptive query processing concepts to the domain of strings. Strings are currently typically treated as purely categorical symbols that can only be described by enumeration. This, however, disregards fact that there are several classes of strings that have an internal structure and that can be handled in a more sophisticated manner. Filesystem paths and URIs are prime examples where the structure of their string representation allows a mechanism as simple as prefixes to express intervals, i.e., sub-spaces of the overall string space that are interesting from the point of view of providing query optimization statistics. Although weaker than regular expressions, prefixes can be very efficiently applied and can capture interesting ranges in hierarchically structured string domains, such as those of filesystem pathnames and URIs. This is especially true for large-scale Semantic Web datasets where URIs are not manually constructed but are automatically generated following naming conventions.

4.4.1 Method

As SemaGrow operates under the premise that no modification will be required for the endpoints it federates, histograms need to be maintained without either access to data dumps or the ability to execute histogram maintenance queries that impose heavy overheads on the federated queries.

As is apparent from the discussion of the relevant literature (Section 2.4), Semantic Web data has not benefited to the extent that it could from developments in self-tuning histograms as no appropriate extension has been developed that can naturally handle URIs, the most prominent type of data in the Semantic Web.

In our treatment, string ranges are specified by prefixes. Prefixes can naturally express ranges of semantically related resources, given the natural tendency to group together relevant items in hierarchical structures such as pathnames and URIs. Although this mostly contributes to applying self-tuning histogram algorithms to Semantic Web data, we discuss how our structure is also amenable to regular expressions or other string patterns so that future extensions can address challenges from the wider data management community.

Appendix B formally defines multi-dimensional histograms over numerical, string, and categorical data. The core added value of this definition by comparison to the state of the art is adding the notion of descriptions in string dimensions, akin to intervals for numerical dimensions. This has considerable advantages for RDF stores and, more generally, in the
Semantic Web and Linked Open Data domain, where URIs have a prominent role and offer the opportunity to exploit the hierarchical structure of their string representation.

Specifically, we propose prefixes as the formalism for expressing string ranges, motivated by its applicability to URI structure. This is no loss of generality, since it is straightforward to use more expressive pattern formalisms (such as regular expressions) without altering the core method but at a considerable computational cost. The only requirement is that intersection and some notion of length can be defined. Length, in particular, can be used in the way STHoles uses it as an indication of a bucket's size relative to the size of its parent bucket, although that also depends on the specifics of the estimation functions and is not required. In the function we propose, the only requirement is that length equals 1 for fixed singleton values and more than 1 for intervals and prefixes. This is related to the fact that for range queries we return the statistics of the bucket that more tightly encloses the query, instead of returning an estimation based on the ratio of the volume occupied by the query to the volume of the overall bucket. In other words, we use length more as a metric of the size of description, rather than a metric of the bucket size (the number of tuples that fit this description). To compensate, we exactly measure in query results (rather than estimate) bucket size when shrinking buckets, compensating for the extra computational time by avoiding examining all combinations of buckets x dimensions.

Furthermore, we can afford a larger number of buckets in the same amount of memory (and thus more accurate estimations) than if strings where treated as categorical values.

### 4.4.2 Implementation

This method has been implemented in Java as the STRHist module of the Resource Discovery component. The Semagrow system's workflow is initiated by client application queries. These queries are analysed in the Query Decomposition component in order to build a query plan. The Query Decomposition component relies on cardinality statistics and other information (reliability, reactivity) about candidate sources, the federated endpoints that might hold data that matches the patterns in the queries. The query plans are executed by the Query Execution Engine that, besides joining results and serving them to the client application, also forwards to Resource Discovery measurements collected during query execution, including the query feedback needed by STRHist. STRHist analyses these query feedback logs in batches to maintain the histograms that are used by Resource Discovery in order to provide cardinality estimates to Query Decomposition.

The expected behaviour of the algorithm to these events would be to alter the existing histogram by adding a bucket containing statistics about the detected changes, in a way that it smoothens the error of the actual statistics against the estimated ones. This bucket addition could be thought of as drilling a hole into the bucket, which contains more accurate statistics about this area. This technique leads to the maintenance of more precise information about a region according to the data distribution. Considering how client applications access some areas more heavily than others, the algorithm should also drill holes in this case, so as to zoom into such critical regions and provide more accurate statistics. However, as someone could foresee, the consecutive drilling of holes in the histogram will inevitably lead to exhausting the available space, triggering the merging of similar buckets.

The whole histogram is represented as a tree of buckets, stored in an RDF store following the Sevod data model (cf. Section 4.2). The histogram is being continuously refined as queries are given as input. This input is extracted from query logs, persistently stored in local disk, and is pre-processed before feeding the algorithm. At the end of each batch of queries proceeding, the RDF store is updated.

The STRHist source code is included in the SemaGrow a public repository, http://github.com/semagrow

### 4.5 Current status and future steps

**During the first phase of Task 3.1 (M4-M12):**

- We have established the internal architecture of the Source Metadata module and the internal interface with the Resource Selector module within the Resource Discovery component. The external interfaces with the Federated Query Manager and the Authoring Environment are sketched but not fully specified at this stage, due to the early stage of development of the relevant tasks (Task 3.4 and 5.2).

- We have developed a mock-up prototype of the Source Metadata module that tests the architecture without implementing any advanced functionality.

**During the current reporting period (M14-M24), and as reported to be the plan at the end of the first period:**

- We have established the external interfaces with the Federated Query Manager in coordination with Task 3.4. The resulting refinements and updates to the overall system architecture are reported in Deliverable D2.3.3 in coordination with Task 2.3.
• We have established the interface with the Authoring Environment in coordination with Task 5.2 by developing the Sevod vocabulary, an RDF serialization for histograms. Work in this task involved developing the vocabulary; work in WP5 involved developing the software for serializing and deserializing histograms and the necessary extensions to the Authoring Environment for visualizing and editing histograms. The overall system architecture is not affected.

• We have elaborated and updated our review of the state-of-the-art.

• We have developed and prototyped STRHist, completing the bulk of the algorithmic design and prototype development work.

During the final phase (M25-M30) focused on the efficiency of storing and serving STRHist histograms to the query execution planner, polishing and hardening the implementation, preparing it for inclusion in the final Semagrow Stack. Work during this final phase also concerned preparing the following publications:


REFERENCES


Archer, P., K. Smith, and A. Perego (eds, 2009), Protocol for Web Description Resources (POWDER): Description Resources. W3C Recommendation, 1 September 2009, URL http://www.w3.org/TR/powder-dr


Appendix A: Resource Discovery API

This appendix documents the public Java interfaces that Resource Discovery exposes to the other components of the Semagrow Stack. Only the Method Summary is included here; the Method Detail section is available in the Javadocs distributed from the SemaGrow repository.

### A1. Interface EquivalentURI

public interface ` EquivalentURI`

For a URI this object holds its equivalent URI, the proximity of the equivalent to the original URI and the schema URI that the equivalent URI belongs to.

#### Method Summary

<table>
<thead>
<tr>
<th>Modifier and Type</th>
<th>Method and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>org.openrdf.model.URI</code></td>
<td><code>getEquivalent_URI()</code></td>
</tr>
<tr>
<td><code>Int</code></td>
<td><code>getProximity()</code></td>
</tr>
<tr>
<td><code>org.openrdf.model.URI</code></td>
<td><code>getSchema()</code></td>
</tr>
</tbody>
</table>

#### Method Detail

**getEquivalent_URI**

```java
org.openrdf.model.URI getEquivalent_URI()
```

**Returns:**
the equivalent URI

**getProximity**

```java
int getProximity()
```

**Returns:**
the proximity to the original URI

**getSchema**

```java
org.openrdf.model.URI getSchema()
```

**Returns:**
the identifier of the RDF schema of the equivalent URI
A2. Interface InstanceIndex

```java
public interface InstanceIndex
```

Instances of this class act as proxies for repositories holding instance-level metadata about the data sources of the federation.

### Method Summary

#### Methods

<table>
<thead>
<tr>
<th>Modifier and Type</th>
<th>Method and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>List&lt;AnnotatedResource&gt;</code></td>
<td><code>getEndpoints(org.openrdf.model.URI uri)</code></td>
</tr>
</tbody>
</table>

This method returns a `List` of `AnnotatedResource` instances, each of which indicate a data source and the volume of triples involving `uri` available at that data source.

#### Method Detail

**getEndpoints**

```java
List<AnnotatedResource> getEndpoints(org.openrdf.model.URI uri)
```

This method returns a `List` of `AnnotatedResource` instances, each of which indicate a data source and the volume of triples involving `uri` available at that data source.
A3. Interface Measurement

public interface Measurement

This interface specifies classes of load info and similar measurements regarding the current performance and availability of a data source.

Method Summary

Methods

<table>
<thead>
<tr>
<th>Modifier and Type</th>
<th>Method and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>long</td>
<td>getId()</td>
</tr>
<tr>
<td></td>
<td>This method returns the unique numerical identifier of this measurement instance.</td>
</tr>
<tr>
<td>String</td>
<td>getType()</td>
</tr>
<tr>
<td></td>
<td>This method returns a textual name or description of the kind of information held in this Measurement instance.</td>
</tr>
<tr>
<td>int</td>
<td>getValue()</td>
</tr>
<tr>
<td></td>
<td>This method returns the actual value of the measurement.</td>
</tr>
</tbody>
</table>

Method Detail

getId

long getId()

This method returns the unique numerical identifier of this measurement instance. This identifier is a timestamp, an increment, or, in general, any strictly monotone value series. This guarantees that a client only needs to retain the largest id received in order to be able to request all subsequent Measurement instances.

Returns:
the measurement id

g getType

String getType()

This method returns a textual name or description of the kind of information held in this Measurement instance.

Returns:
the type of the measurement

getValue

int getValue()

This method returns the actual value of the measurement.

Returns:
the value of the measurement
## A4. Interface PatternDiscovery

public interface **PatternDiscovery**

For a URI get equivalent URIs aligned with a certain confidence with the initial URI and belonging to a specific schema.

### Method Summary

<table>
<thead>
<tr>
<th>Modifier and Type</th>
<th>Method and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>List&lt;EquivalentURI&gt;</td>
<td>retrieveEquivalentPatterns()</td>
</tr>
</tbody>
</table>

### Method Detail

**retrieveEquivalentPatterns**

List<EquivalentURI> retrieveEquivalentPatterns()

**Returns:**

A List of EquivalentURIs.

## A5. Interface QueryTransformation

public interface **QueryTransformation**

### Method Summary

<table>
<thead>
<tr>
<th>Modifier and Type</th>
<th>Method and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>List&lt;EquivalentURI&gt;</td>
<td>retrieveEquivalentPatterns(org.openrdf.model.URI schema)</td>
</tr>
</tbody>
</table>

### Method Detail

**retrieveEquivalentPatterns**

List<EquivalentURI> retrieveEquivalentPatterns(org.openrdf.model.URI schema)

**Returns:**

A list of equivalent URIs aligned with a certain confidence with the initial URI and belonging to a specific schema.
A6. Interface ResourceSelector

public interface ResourceSelector

This component provides an annotated list of candidate data sources that possibly hold triples matching a given query pattern; including sources that follow a different (but aligned) schema than that of the query pattern. The sources are annotated with schema and instance-level metadata and predicted response volume from the data summaries endpoint; as well as run-time information about current source load. When a source following an aligned schema is used, the annotation also includes relevant meta-information, such as the semantic proximity of the query schema and the source schema.

Method Summary

Methods

<table>
<thead>
<tr>
<th>Modifier and Type</th>
<th>Method and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>List&lt;AnnotatedResource&gt;</td>
<td>getAnnotatedResources(org.openrdf.query.algebra.StatementPattern statementPattern, int measurement_id)</td>
</tr>
</tbody>
</table>

Public method that acts as entry point to the ResourceSelector module.

Method Detail

getSelectedResources

List<AnnotatedResource>

getSelectedResources(org.openrdf.query.algebra.StatementPattern statementPattern, int measurement_id)

Public method that acts as entry point to the ResourceSelector module.

Parameters:

statementPattern - the StatementPattern (query pattern) to examine.

measurement_id - used to determine how many load info measurements should be returned for each candidate source endpoint. Each measurement has an incremental id so this method returns the load info of an endpoint where id >= measurement_id

Returns:

A list of AnnotatedResource objects. Empty list if no resources found. Null in case of exceptions.
### A7. Interface SchemaIndex

public interface **SchemaIndex**

Instances of this class act as proxies for repositories holding schema-level metadata about the data sources of the federation.

#### Method Summary

<table>
<thead>
<tr>
<th>Modifier and Type</th>
<th>Method and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>getEndpoints</strong> (org.openrdf.model.URI uri)</td>
</tr>
</tbody>
</table>

This method returns a *List* of AnnotatedResource instances, each of which indicate a data source and the volume of triples involving uri available at that data source.

#### Method Detail

**getEndpoints**

```
List<AnnotatedResource> getEndpoints(org.openrdf.model.URI uri)
```

This method returns a *List* of AnnotatedResource instances, each of which indicate a data source and the volume of triples involving uri available at that data source.

**Returns:**
A8. Interface AnnotatedResource

public interface AnnotatedResource

This class is used to represent a ResourceSelector result. It contains a combination of equivalent URIs that make up a query pattern, their proximity to the original URIs, the source endpoint that holds triples for this pattern, the predicted volume of the results, and the responsiveness of the endpoint.

Method Summary

<table>
<thead>
<tr>
<th>Modifier and Type</th>
<th>Method and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>org.openrdf.model.URI</td>
<td>getEndpoint()</td>
</tr>
<tr>
<td>List&lt;Measurement&gt;</td>
<td>getLoadInfo()</td>
</tr>
<tr>
<td>org.openrdf.model.URI</td>
<td>getObject()</td>
</tr>
<tr>
<td>int</td>
<td>getObjectProximity()</td>
</tr>
<tr>
<td>org.openrdf.model.URI</td>
<td>getPredicate()</td>
</tr>
<tr>
<td>int</td>
<td>getPredicateProximity()</td>
</tr>
<tr>
<td>org.openrdf.model.URI</td>
<td>getSubject()</td>
</tr>
<tr>
<td>int</td>
<td>getSubjectProximity()</td>
</tr>
<tr>
<td>int</td>
<td>getVar()</td>
</tr>
<tr>
<td>int</td>
<td>getVol()</td>
</tr>
<tr>
<td>void</td>
<td>setLoadInfo(List&lt;Measurement&gt; loadInfo)</td>
</tr>
<tr>
<td>void</td>
<td>setObject(org.openrdf.model.URI object)</td>
</tr>
<tr>
<td>void</td>
<td>setObjectProximity(int objectProximity)</td>
</tr>
<tr>
<td>void</td>
<td>setPredicate(org.openrdf.model.URI predicate)</td>
</tr>
<tr>
<td>void</td>
<td>setPredicateProximity(int predicateProximity)</td>
</tr>
<tr>
<td>void</td>
<td>setSubject(org.openrdf.model.URI subject)</td>
</tr>
<tr>
<td>void</td>
<td>setSubjectProximity(int subjectProximity)</td>
</tr>
</tbody>
</table>

Method Detail

getEndpoint

org.openrdf.model.URI getEndpoint()

Returns:
the source endpoint that holds triples for this pattern

getVol
int getVol() 
Returns: 
the volume of the triples

getVar

int getVar() 
Returns: 
the variance of the volume of the triples

getSubject

org.openrdf.model.URI getSubject() 
Returns: 
the equivalent subject of the candidate query pattern (if any, else null).

setSubject

Void setSubject(org.openrdf.model.URI subject) 
Parameters: 
subject - the equivalent subject of the candidate query pattern (if any, else null) to set

getSubjectProximity

int getSubjectProximity() 
Returns: 
the proximity of the equivalent subject to the original URI (if any, else -1).

setSubjectProximity

void setSubjectProximity(int subjectProximity) 
Parameters: 
subjectProximity - the proximity of the equivalent subject to the original URI (if any, else -1) to set

getPredicate

org.openrdf.model.URI getPredicate() 
Returns: 
the equivalent predicate of the candidate query pattern (if any, else null).

setPredicate

void setPredicate(org.openrdf.model.URI predicate) 
Parameters: 
predicate - the equivalent predicate of the candidate query pattern (if any, else null) to set

getPredicateProximity
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int getPredicateProximity()</code></td>
<td>Returns: the proximity of the equivalent predicate to the original URI (if any, else -1).</td>
</tr>
<tr>
<td><code>void setPredicateProximity(int predicateProximity)</code></td>
<td>Parameters: <code>predicateProximity</code> - the proximity of the equivalent predicate to the original URI (if any, else -1)</td>
</tr>
<tr>
<td><code>org.openrdf.model.URI getObject()</code></td>
<td>Returns: the equivalent object of the candidate query pattern (if any, else null).</td>
</tr>
<tr>
<td><code>void setObject(org.openrdf.model.URI object)</code></td>
<td>Parameters: <code>object</code> - the equivalent object of the candidate query pattern (if any, else null) to set</td>
</tr>
<tr>
<td><code>int getObjectProximity()</code></td>
<td>Returns: the proximity of the equivalent object to the original URI (if any, else -1).</td>
</tr>
<tr>
<td><code>void setObjectProximity(int objectProximity)</code></td>
<td>Parameters: <code>objectProximity</code> - the proximity of the equivalent object to the original URI (if any, else -1) to set</td>
</tr>
<tr>
<td><code>List&lt;Measurement&gt; getLoadInfo()</code></td>
<td>Returns: the load info of the source</td>
</tr>
<tr>
<td><code>void setLoadInfo(List&lt;Measurement&gt; loadInfo)</code></td>
<td>Parameters: <code>loadInfo</code> - the load info of the source to set</td>
</tr>
</tbody>
</table>
Appendix B: STRHist Foundations

This appendix introduces some basic definitions and then proceeds to establish a new histogram structure that extends the structure of the STHoles algorithm with the ability to cover strings, as well as the main algorithms that construct and refine our proposed histogram structure.
1. PRELIMINARIES

Let $D$ be a dimension, any subset of $D$ be a range in $D$, and $\mathcal{P}(D)$ the set of all possible ranges in $D$. A range can be defined either implicitly by constraints over the values of $D$ or explicitly by enumeration. Note that $D \in \mathcal{P}(D)$, meaning that a range does not need to impose a restriction but can also include the whole dimension.

Let $H$ be histogram of $n$ dimensions $D_1, \ldots, D_n$. Let $\forall H$ be the set of all possible $n$-dimensional vectors $(r_1, \ldots, r_n)$, where $\forall i \in [1, n]: r_i \in \mathcal{P}(D_i)$

A histogram is represented as an inclusion hierarchy of buckets; we shall use $B_H$ to denote the set of buckets of a histogram $H$.

**Definition 1.** Each bucket $b \in B_H$ is an entity of histogram $H$ such that:

* $b$ is associated with a box$(b) \in V(H)$, the vector that specifies the set of tuples that the bucket describes.
* $b$ is associated with a size$(b)$ which indicates the number of tuples that match box$(b)$
* $b$ is associated with $n$ dvc$(b, D_i), i \in [1, n]$ which indicates the number of distinct values appearing in each dimension $D_i$ of the tuples that match box$(b)$

We define the density of a bucket $b$ to be the quantity

$$\text{density}(b) = \frac{\text{size}(b)}{\prod_{i \in \mathcal{D}} \text{dvc}(b, D_i)}$$

**Definition 2.** Every histogram implicitly includes a bucket $b_\top$ such that box$(b_\top) \equiv (D_1, \ldots, D_n)$ that is, the bucket that imposes no restrictions in any of the dimensions of $H$ and includes all tuples. We call this the top bucket $b_\top$.

The implication of Definition 2 is that the overall size of the dataset and the number of distinct values in each dimension should be known (or at least approximated) regardless of what query feedback has been received. In our implementation we assume the root bucket (the top-most bucket of the hierarchy) as an approximation of the top.

Let $Q_H$ be the set of all possible queries over the tables covered by $H$. Regardless of how they are syntactically expressed, we perceive $Q_H$ as the set of all possible restrictions over the dimensions of $H$; thus:

**Definition 3.** Each query $q \in Q_H$ is an entity of histogram $H$ such that:

* $q$ is associated with a box$(q) \in V(H)$, the vector that specifies the restrictions expressed by the query
* $q$ is associated with a size$(q)$ which indicates the number of tuples that are returned by executing $q$

In our approach, we assume string prefixes as the description language for implicitly defining string ranges. The motivation is discussed in the Introduction (Section ??). We use $\text{Pref}(r)$ to denote the set of prefixes that specify a string range.

We define intersection between ranges and between boxes as follows:

**Definition 4.** Given two ranges of the same dimension, $r_1, r_2 \in \mathcal{P}(D)$, their range intersection $r_1 \cap r_2$ is defined as follows:

1. If $r_1, r_2$ are string ranges defined by sets of prefixes, then $r_1 \cap r_2 = \{p | \text{Pref}(r_1) \cap \text{Pref}(r_2) \land \exists p \in \text{Pref}(r_1) \land p \text{ is the longest (more specific) of the two)}$
2. If one of the ranges is a string range defined by sets of prefixes (say $r_1$ without loss of generality) and the other is an explicit set of strings (say $r_2$), then $r_1 \cap r_2 = \{v | \exists v \in \mathcal{P}(H) \land \exists p \in r_1 : p \text{ is a prefix of } v}$
3. In any other case, $r_1 \cap r_2 = r_1 \cap r_2$

**Definition 5.** Given two boxes $v_1, v_2 \in V(H)$ from the $n$-dimensional histogram $H$, let $v_1 = (r_{1,1}, \ldots, r_{1,n})$ and $v_2 = (r_{2,1}, \ldots, r_{2,n})$. We define box intersection:

$$v_1 \cap v_2 = (r_{1,1} \cap r_{2,1}, \ldots, r_{1,n} \cap r_{2,n})$$

**Definition 6.** Given two boxes $v_1, v_2 \in V(H)$ from the $n$-dimensional histogram $H$, let $v_1 = (r_{1,1}, \ldots, r_{1,n})$ and $v_2 = (r_{2,1}, \ldots, r_{2,n})$. We say that $v_1$ encloses $v_2$ iff $\forall i \in [1, n]$ at least one of the following holds:

1. $r_{2,i} \subseteq r_{1,i} \subseteq D_i$, that is, none of the ranges is the complete dimension and $r_{2,i}$ is contained within $r_{1,i}$
2. $r_{2,i} = D_i$ and $r_{1,i} \subseteq D_i$, that is, if one of the ranges is the complete dimension then it is enclosed by the one that is not
3. $r_{2,i} = r_{1,i} = D_i$, that is, both ranges are the complete dimension

It should be noted that we have defined an unrestricted dimension as being enclosed by (rather than enclosing) a restriction. The rational behind this will be revisited in conjunction with bucket merging (Section ??).

**Definition 7.** Given two boxes $v_1, v_2 \in V(H)$ from histogram $H$, $v_1$ tightly encloses $v_2$ iff

1. $v_1$ encloses $v_2$ and
2. There is no $u \in V(H)$ such that $v_1 \equiv u \equiv v_2$

**Definition 8.** Given a query $q \in Q_H$, we associate with $q$ the best fit, the set of buckets $b(q) \subseteq B_H$ such that

$$\forall b \in b(q) : \text{box}(b) \text{ tightly encloses box}(q)$$

**Lemma 1.** For every query there is always a non-empty best fit.

**Proof.** Since there is always at least one bucket that encloses any box$(q)$, the top bucket $b_\top$ (Definition 2). If there is no other bucket that encloses box$(q)$, then $b_\top$ tightly encloses box$(q)$ (Definition 7) and thus $b(q) = \{b_\top\}$, which is non-empty. If there are other buckets that enclose box$(q)$, then there is also at least one that tightly encloses box$(q)$, so $b(q)$ is non-empty. □
2. CARDINALITY ESTIMATION

Being able to predict the size of querying results is important input for query execution optimizers, but the specifics of how this optimization is performed is outside the scope of this paper. We will here proceed to define metrics over the values associated with the buckets of \(H\) in order to predict \(\text{size}(q), q \in \mathcal{Q}_H\), the number of results returned by \(q\). More specifically, we will define an extension of self-tuning histograms that handles string values in a more sophisticated manner than explicit ranges.

In the literature, numerical intervals are used to succinctly define ranges and efficiently decide if a query is enclosed by a bucket or not. The numerical difference between the interval’s starting and ending value is sometimes used to define range length in a manner than explicit ranges. In the literature, numerical intervals are used to succinctly define ranges and efficiently decide if a query is enclosed by a bucket or not. The numerical difference between the interval’s starting and ending value is sometimes used to define range length in a manner that is more sophisticated than explicit ranges.

Given this, we define range length as follows:

**Definition 9.** Given a histogram dimension \(D\) and a range \(r \in \mathcal{P}(D)\) we define the function length : \(\mathcal{P}(D) \rightarrow \mathbb{R}\) as follows:

1. Unrestricted ranges that span the whole dimension have length 0.
2. If \(r\) is a numerical range defined by interval \([x, y]\), then length\((r) = y - x + 1\). The addition of the unit term guarantees that the length cannot be zero, even if the numerical range is an exact number.
3. If \(r\) is a string range defined by a set of prefixes \(\text{Pref}(r)\), then length\((r) = |1 + \text{Pref}(r)|\), the number of prefixes defining the range.
4. In any other case, including explicitly defined numerical and string ranges, then length\((r) = |r|\), the number of distinct values in the range.

What should be noted in Definition 9 is that the only situation in which the length of a range can be 1 is when it is an individual value. This is important for Definition 10 immediately below:

**Definition 10.** Given a histogram \(H\), we define the function \(\text{est}_H : \mathcal{V}(H) \rightarrow \mathbb{R}\) as follows:

\[
\text{est}_H(q) = \sum_{b \in \mathcal{H}(q)} \frac{\text{size}(b)}{\prod_{i, \text{length}(r_i) = 1} \text{dvc}(b, D_i)}
\]

We propose this function as an estimator of the number of tuples that lie inside \(q\), given a histogram.

3. HISTOGRAM CONSTRUCTION AND REFINEMENT

The construction of the histogram follows the same high level steps as the STHoles algorithm. In particular, we start with an empty histogram. For each query \(q\) in the workload, we identify candidate buckets \(b\) that intersect with \(q\).
Algorithm 3 Shrink a bucket that is enclosed by the intersection of b and q and does not partially intersect any other bucket.

function SHRINK_BUCKET(b, q)
    c ← box(q) \ box(b)
    P ← \{b_i ∈ children(b) | c ∩ box(b_i) \= ⊆ ∩ box(b_i) \= c\}
    while P \= ∅
        Get first bucket b_i ∈ P and dimension j such that shrinking c along j by excluding b_i results in the smallest reduction of c.
        Shrink c along j
        P ← \{b_i ∈ children(b) | c ∩ box(b_i) \= ⊆ ∩ box(b_i) \= c\}
    for all attributes i do
        Count from the result the number of tuples in c, T_c
    return (c, T_c, d_c(i))

Bin combinations at each step. Furthermore, in STHoles the number of tuples in this shrunk subregion is estimated assuming uniformity; instead, we measure exactly the number of tuples and distinct values per dimension.

4. BUCKET MERGING

In order to maintain a limit on the number of buckets and memory usage, buckets need to be merged in order to make it possible for new holes to be drilled. We follow the overall procedure of STHoles, looking for the pair of buckets that can be merged with minimal impact on the cardinality estimations; we diverge from STHoles with respect to how we compute the box, size, and dvc associated with the merged bucket as well as with respect to the penalty measure that guides the merging process towards merges that have the smallest impact.

More specifically, we allow parent-child and sibling merges. Let b_p, b_i be two buckets in the n-dimensional histogram H and let H’ be the histogram after the merge and b_m the bucket in H’ that replaces b_1 and b_2. In the parent-child case, one of the two buckets, let that be b_1, tightly encloses the other. In this case, we merge b_2 into b_1, so that box(b_m) ⊆ box(b_1).

Our penalty measure should be proportionate to the absolute differences between the cardinality estimates made from the the two original buckets and the cardinality estimates made from the resulting bucket. Let Q_1, Q_2 be two sets of queries that would have b_1 and b_2 resp. be involved in their cardinality estimation. That is:

∀k ∈ \{1, 2\}∀q ∈ Q_k : b_k ∈ bh(q)

Furthermore, Q_k contain all and only the n queries so that for each dimension D_i, i = 1, . . . , n of H there is exactly one query that:

• has a specific value for the dimension D_i and this value is inside the range of box(b_k) for D_i. It is not important what this particular value is, since the estimation is the same for all values inside the bucket’s box’s range.

• is completely unspecified for all other dimensions.

We define the penalty to be the sum over all q ∈ Q_k of the estimation error differences between H and H’. As seen in Definition 10, only the terms that involve b_k are significant for calculating the difference, and thus:

\[\text{penalty}_H(b_1, b_2) = \sum_{q \in Q_1} |\text{est}_H(q) - \text{est}_H'(q)| + \sum_{q \in Q_2} |\text{est}_H(q) - \text{est}_H'(q)|\]

\[= \sum_{i ∈ [1, n]} \left[\frac{\text{size}(b_1)}{\text{dvc}(b_1, i)} - \frac{\text{size}(b_m)}{\text{dvc}(b_m, i)}\right] + \sum_{i ∈ [1, n]} \left[\frac{\text{size}(b_2)}{\text{dvc}(b_2, i)} - \frac{\text{size}(b_m)}{\text{dvc}(b_m, i)}\right]\]

\[= \sum_{i ∈ [1, n]} \left|\frac{\text{size}(b_1)}{\text{dvc}(b_1, i)} - \frac{\text{size}(b_2)}{\text{dvc}(b_2, i)}\right|\]

In sibling merges, let b_p be the bucket that tightly encloses both b_1 and b_2. The merged bucket encloses siblings b_1 and b_2 and is also extended to enclose any further siblings b_i that partially overlap with either b_1 or b_2. The size of b_m is estimated by adding; the distinct values count of b_m is estimated by the maximum distinct values count among the merged buckets:

1. box(b_p) tightly encloses box(b_m)

2. box(b_m) tightly encloses both buckets b_1, b_2

3. box(b_m) tightly encloses the boxes of all children of b_p that either of b_1, b_2 partially intersects with. That is, box(b_m) encloses box(b_c) for all b_c such that:

• b_p tightly encloses b_c, and

• box(b_1) partially overlaps box(b_c) or box(b_2) partially overlaps box(b_c)

4. size(b_m) = \sum_{k = 1, 2, i \in \ldots} size(b_k)

5. dvc(b_m) = \max_{k = 1, 2, i \in \ldots} dvc(b_k)

In Point 3 above, it should be stressed that the partially intersecting buckets b_i are not merged into b_m, but that the latter is expanded so that it can assume b_m as its children. This is because in some algorithms (including STHoles), box(b_m) can become larger than box(b_1) ∪ box(b_2) in order to have a ‘rectangular’ description with a single interval in each dimension. As a result, it might cut across other buckets; box(b_m) should then be extended so as to subsume those as children. In order to avoid, however, dropping informative restrictions, STHoles only extends box(b_m) along dimensions where the boxes of b_c do have a restriction. In order to capture this, we have defined the encloses relation (Definition 6) in a way that makes unrestricted dimensions enclosed by (rather than enclosing) restrictions. In analogy to the parent-child case, we define the penalty to be the sum over all q ∈ Q_k of the estimation error differences between
\[ H \text{ and } H': \]
\[
\text{penalty}_H(b_1, b_2) = \sum_{q \in Q_1} |est_H(q) - est_{H'}(q)| + \sum_{q \in Q_2} |est_H(q) - est_{H'}(q)| = \\
\sum_{i \in [1, n]} \left| \frac{\text{size}(b_1)}{\text{dvc}(b_1, i)} - \frac{\text{size}(b_m)}{\text{dvc}(b_m, i)} \right| + \\
\sum_{i \in [1, n]} \left| \frac{\text{size}(b_2)}{\text{dvc}(b_2, i)} - \frac{\text{size}(b_m)}{\text{dvc}(b_m, i)} \right| = \\
\sum_{i \in [1, n]} \left| \frac{\text{size}(b_1)}{\text{dvc}(b_1, i)} - \sum_{k=1, 2, c_1, \ldots} \frac{\text{size}(b_k)}{\max_{k=1, 2, c_1, \ldots} \text{dvc}(b_k, i)} \right| + \\
\sum_{i \in [1, n]} \left| \frac{\text{size}(b_2)}{\text{dvc}(b_2, i)} - \sum_{k=1, 2, c_1, \ldots} \frac{\text{size}(b_k)}{\max_{k=1, 2, c_1, \ldots} \text{dvc}(b_k, i)} \right| \]