UiO : University of Oslo

Vidar Norstein Klungre

Adaptive Query Extension Suggestions for Ontology-Based Visual Query Systems

Thesis submitted for the degree of Philosophiae Doctor

Department of Informatics

Faculty of Mathematics and Natural Sciences

SIRIUS - Centre for Scalable Data Access

2020

© Vidar Norstein Klungre, 2020

Series of dissertations submitted to the
Faculty of Mathematics and Natural Sciences, University of Oslo
No. 2307

ISSN 1501-7710

All rights reserved. No part of this publication may be
reproduced or transmitted, in any form or by any means, without permission.

Cover: Hanne Baadsgaard Utigard.
Print production: Reprosentralen, University of Oslo.

To Johanne, Sverre, Ingve, and Frede

Abstract

Ontology-based visual query systems enable users to construct ad-hoc queries in a way that is intuitive, and which allows them to receive feedback directly from the system interface. In this query construction setting, *dead-ends* are the undesirable query extensions leading to queries without any answers. Dead-end detection is then the problem of detecting and possibly disabling such extensions. The standard approach used to detect dead-ends, which requires querying over the data, is often too inefficient. This can be solved by using efficient indices, but current solutions only work when the user queries are limited to one single class and a fixed number of properties. We consider systems where the user is allowed to make more complex queries with two or more connected classes, but it is impossible to ensure both perfect and efficient dead-end detection in this setting because it would require an index of infinite size.

This thesis introduces an index-based framework that can be used to efficiently *approximate* dead-end detection in systems that support ad-hoc, complex queries. In order to use this framework, it is necessary to provide a configuration structure where the classes and properties to support are given. This configuration determines both which parts of the data to include in the index, and how precise the dead-end detection approximation will be.

Finding the configuration that leads to the highest possible precision while keeping the cost (index size) at an acceptable level is a non-trivial combinatorial optimization problem. The search space of possible configurations is too extensive for exhaustive search, and finding the true cost and precision of a configuration is time-consuming. We propose a solution to this problem where efficient cost and precision estimates are used to guide the search for the optimal configuration. Our evaluation of this search, which uses an extensive benchmark based on Wikidata, shows that it is able to efficiently compute non-trivial configurations with both high precision and an acceptable cost.

Acknowledgements

This work would not have been possible without the help and support of many people. I would like to thank all those who have helped and inspired me during my study.

First and foremost, I would like to thank my two supervisors Martin Giese and Ahmet Soylu for their continuous support and invaluable guidance throughout the whole thesis project. Martin deserves a big thanks for helping me to keep my motivation up for over four years. His deep insight, extensive experience, and exceptional ability to find structure where I only see chaos has been essential. I simply could not wish for a better main supervisor. A big thank you goes to Ahmet for his patience and support, for guiding me with his expertise in visual query systems, and for pushing me towards the goal by encouraging me to publish my work.

I would like to extend my gratitude to all my friends and colleagues at ASR/SIRIUS/LogID for their support of my growth as a researcher, and for making such a welcoming and cooperative work environment. I would like to thank all my co-authors for the opportunity to collaborate with them. In particular, I would like to thank Ernesto Jiménez-Ruiz and Evgeny Kharlamov, for our collaboration on ontology projection, ranking, and dead-end detection in the context of ontology-based visual query systems. Furthermore, I would like to thank Leif Harald Karlsen, Daniel Lupp, Andreas Nakkerud, Lars Tveito, Sigurd Kittilsen, Martin Skjæveland, Dag Hovland, Tom Christoffersen, and Basil Ell for all the exciting and helpful discussions we have had related to my work and other interesting topics. A special thank you goes to Arild Waaler for the incredible work he has done to build up the logic and semantic data research community at UiO, with connections to strong academic and industrial partners from all over the world.

Finally, I would like to thank my family and friends for their never-ending support in all aspects of life. A special personal thank you goes to my lovely wife, Johanne, and my three adorable children, Sverre, Ingve, and Frede, for their incredible patience during this period. I promise that I will never do this again!

The work presented in this thesis was funded by SIRIUS – Centre for Scalable Data Access (Research Council of Norway, project no.: 237889).

Contents

Abstract **iii**

Acknowledgements **v**

Contents **vii**

List of Figures **xi**

List of Tables **xiii**

1 Introduction **1**
1.1 Contributions 3
1.2 Research Papers 3
1.3 Thesis Structure 6

2 Preliminaries **9**
2.1 Semantic Technologies 9
2.1.1 RDF Data Model 9
2.1.2 OWL 12
2.1.3 SPARQL 13
2.2 Mathematical Definitions 16
2.2.1 Functions 17
2.2.2 Labeled, Directed Graphs 17

3 Dead-End Detection **21**
3.1 Data Access Systems 21
3.1.1 Aspects of Data Access Systems 21
3.2 Existing Systems 26
3.2.1 Faceted Search Systems 26
3.2.2 RDF-Based Systems 28
3.2.3 Information Retrieval Systems 34
3.3 Dead-End Detection 37

4 Ontology-Based Visual Query Systems **43**
4.1 Resource Graphs 44
4.2 Main Structures 50
4.2.1 The Navigation Graph 52
4.2.2 The Dataset 53
4.2.3 Queries 54
4.3 Query Answers 59

5 Query Extensions **63**
5.1 User Actions 63
5.2 Legal Extensions 68
5.3 Productive Extensions 70
5.4 Value Functions 73
5.4.1 Simple Value Functions 73
5.4.2 Precision and Recall 74
5.4.3 Advanced Value Functions 78
5.4.4 Comparison of Value Functions 80

6 The Index-Based Extension Framework **83**
6.1 The Configuration-based Value Function: $S_a^{\mathcal{Z}}$ 83
6.1.1 Configuration Queries 84
6.1.2 The Configuration-based Value Function: $S_a^{\mathcal{Z}}$ . . 87
6.1.3 Experiment 1: Precision of $S_a^{\mathcal{Z}}$ 93
6.2 The Extension Index 99
6.2.1 Index Construction 101
6.2.2 Index Efficiency 109
6.2.3 Index Cost 110
6.2.4 Bucketing 111
6.3 Optimal Configuration Queries 112
6.3.1 Experiment 2: Pareto Optimal Configuration Queries 114
6.4 Configuration Sets 116
6.4.1 Special Configuration Sets 119
6.4.2 The Configuration Generation Problem 122

7 The Wikidata Benchmark **127**
7.1 WD Navigation Graph 129
7.2 WD Dataset 129
7.3 WD Query Log 132
7.3.1 Query Transformation Process 132
7.3.2 Transformed Queries 134

8 Configuration Generation **139**
8.1 Cost and Precision Estimation 139
8.1.1 Basic Counts 142
8.1.2 Edge Target Distributions 144
8.1.3 Cardinality Estimation: ans 146
8.1.4 Cardinality Estimation: ans_P 148
8.1.5 Cardinality Estimation: ans_O 149
8.1.6 Cardinality Estimation: ans_E 151
8.2 Search Methods 153
8.3 Evaluation 157
8.3.1 Evaluation based on $\mathcal{L}_B$ 158
8.3.2 Evaluation based on $\mathcal{L}_A$ 167

9 Conclusion and Future Work **171**
9.1 Conclusion 171
9.2 Future work 173

Bibliography **179**

Index **185**

List of Figures

2.1 Visual presentation of The Semantic Web stack by Wikipedia user Marobi1, licensed under CC0. 9
2.2 Graph representation of a small RDF dataset. The two blue circles represent resources, while the yellow rectangle represents a literal. 11
2.3 Two labeled, directed graphs: G and G'. 19

3.1 Screenshot of the faceted search interface used by PriceSpy, where the facet section is on the left side, and the result section is on the right side. 27
3.2 Screenshot of RDF Surveyor. 29
3.3 Screenshot of OptiqueVQS. 31
3.4 Google search engine results for the query "information retrieval search engine", ranked by relevance. 35
3.5 The QA feature of Google Search answering a factual question. . 36
3.6 Google auto-completing a user's question. 40

4.1 The resource graph defined in Example 4.1.4. 46
4.2 A visual representation of a dataset. The types of each instance and data value is given below its name. 55
4.3 The two rooted queries $\mathcal{Q}_1$ and $\mathcal{Q}_2$. 57

5.1 Query $\mathcal{Q}_9$ and how focusing on each of its three object variables results in three different rooted queries. 66
5.2 The query $\mathcal{Q}_1$, and the resulting query $\mathcal{Q}_2$ after extension. . . . 68
5.3 The query $\mathcal{Q}_3$. 74

6.1 Two queries, $\mathcal{Q}_4$ and $\mathcal{Q}_5$, to the left, and two configuration queries, $\mathcal{Z}_1$, and $\mathcal{Z}_2$ to the right. $\mathcal{Q}_4$ is not covered by $\mathcal{Z}_1$, but its subquery $\mathcal{Q}_5$ is covered by $\mathcal{Z}_1$. $\mathcal{Z}_2$ is the subquery of $\mathcal{Z}_1$ that corresponds to $\mathcal{Q}_5$. 85
6.2 The partial query $\mathcal{Q}_3$, its extended version $\mathcal{Q}_4$, the configuration query $\mathcal{Z}_1$, and query $\mathcal{Q}_5$, which is the result after pruning $\mathcal{Q}_4$ with respect to $\mathcal{Z}_1$. 90
6.3 The two queries $\mathcal{Q}_7$ and $\mathcal{Q}_8$ are the only two maximal subqueries in $\mathsf{prune}(\mathcal{Q}_6, \mathcal{Z}_2)$. 91
6.4 Precisions for Query 2.6. 96
6.5 Precisions for Query 2.8. 97
6.6 Precisions for Query 3.5. 97
6.7 Average precision of all queries of size 6. 98

6.8 The configuration query $\mathcal{Z}_2$. 101
6.9 The query $\mathcal{Q}_2$. 103
6.10 Configuration queries $\mathcal{Z}_1$ and $\mathcal{Z}_2$. 106
6.11 SPARQL queries with nested optionals used to calculate $\mathsf{ans}_O(\mathcal{Z}_1, \mathcal{D})$ and $\mathsf{ans}_O(\mathcal{Z}_2, \mathcal{D})$. 106
6.12 Chart showing all considered configuration queries when using Query 6.2 to calculate precision. The connected configurations are Pareto optimal. 115
6.13 The same results as in Figure 6.12, just with a normalized y-axis. 116
6.14 Pareto optimal configuration queries for all 29 queries with a normalized cost. The red curve shows the median, and the blue curve is the upper quartile. 117
6.15 The six configuration queries in $\mathcal{W}_r$ when using the navigation graph from Example 4.2.3. 120
6.16 The two configuration queries in $\mathcal{W}_l$ when using the navigation graph from Example 4.2.3. 121

8.1 The navigation graph from Example 4.2.3 with basic counts included. 144
8.2 The precision and cost of the six reference configuration sets $\mathcal{W}_d$, $\mathcal{W}_r^d$, $\mathcal{W}_r$, $\mathcal{W}_l^d$, $\mathcal{W}_l$, and $\mathcal{W}_m$ over query log $\mathcal{L}_B$. $\mathcal{W}_d$ is not visible because the y-axis uses a logarithmic scale. 159
8.3 Configuration sets generated by the Greedy Query Weight Method (yellow) and the Random Method (red) over query log $\mathcal{L}_B$. 161
8.4 Configuration sets generated by the Greedy Query Weight Method (yellow), the Greedy Precision Method (teal), and the six reference configuration sets over query log $\mathcal{L}_B$. 162
8.5 Configuration sets generated by the Greedy Query Weight Method (yellow), the Greedy Precision Method (teal), the Exploratory Method (orange), and the six reference configuration sets over query log $\mathcal{L}_B$. 163
8.6 The sequence of configurations generated by the Exploratory Method over the query log $\mathcal{L}_B$, with a maximum cost of $M = 1.0 \times 10^7$ (green crosses). 164
8.7 Precision and cost of every generated configuration when evaluated with respect to query log $\mathcal{L}_A$. 168
8.8 Precision and cost of every generated configuration with precision above 0.95 when evaluated with respect to query log $\mathcal{L}_A$. 169

List of Tables

3.1 The 17 different aspects of data access systems we consider. . . . 22

4.1 Summary of the sets of components needed to model $\mathcal{N}$, $\mathcal{D}$, and $\mathcal{Q}$. 51
4.2 Answers returned by $\mathsf{ans}(\mathcal{Q}_1, \mathcal{D})$. 60
4.3 Answers returned by $\mathsf{ans}(\mathcal{Q}_2, \mathcal{D})$. 60

5.1 Summary of the five value functions S_d, S_r, S_l, S_o, and S_e. . . . 82

6.1 All the 50 functions in $\mathcal{I}_{\mathcal{Z}_2} = \mathsf{ans}_S(\mathcal{Z}_2, \mathcal{D})$. 102
6.2 Index $\mathcal{I}'_{\mathcal{Z}_2} = \mathsf{ans}_O(\mathcal{Z}_2, \mathcal{D})$, which only includes the maximal functions from $\mathcal{I}_{\mathcal{Z}_2} = \mathsf{ans}_S(\mathcal{Z}_2, \mathcal{D})$. 104
6.3 Step 1. 105
6.4 Step 2. 105
6.5 Step 3. 105
6.6 Step 4. 105
6.7 Every function remaining from $\mathsf{ans}_O(\mathcal{Z}_2, \mathcal{D})$ after introducing existential object variables, and before removing subfunctions one more time. 108
6.8 Every function in $\mathsf{ans}_E(\mathcal{Z}_2, \mathcal{D})$. 108
6.9 Summary of the six special configuration sets we have considered. 122

7.1 The number of incoming and outgoing properties of each class and datatype in the WD navigation graph. In: Incoming properties. Out: Outgoing properties. DP: Outgoing data properties. OP: Outgoing object properties. 130
7.2 The number of instances in the WD dataset typed to each of the 15 classes. 131
7.3 All types of triples that occur more than 100000 times in the WD dataset sorted by frequency. 132
7.4 Weight of the queries grouped by size after the transformation process. 135
7.5 The effect of removing queries with small weight from each of the two query logs. 136
7.6 Overview of queries in $\mathcal{L}_A$ grouped by size. 136
7.7 Overview of queries in $\mathcal{L}_B$ grouped by size. 136
7.8 Popularity of each class in the two query logs $\mathcal{L}_A$ and $\mathcal{L}_B$. . . . 137

8.1 Summary of the six special configuration sets we have considered. 158

8.2 The precision and cost of the six reference configuration sets $\mathcal{W}_d$, $\mathcal{W}_r^d$, $\mathcal{W}_r$, $\mathcal{W}_l^d$, $\mathcal{W}_l$, and $\mathcal{W}_m$ with respect to query log $\mathcal{L}_B$. 159
8.3 The two types of index tables each of the 80 object property configuration queries in $\mathcal{W}_r$ can correspond to. 160
8.4 Overview of the 15 classes and why most of our methods only construct configuration queries for four of them when calculating precision based on $\mathcal{L}_B$. 165
8.5 The precision and cost of the six reference configuration sets $\mathcal{W}_d$, $\mathcal{W}_r$, $\mathcal{W}_r^d$, $\mathcal{W}_l$, $\mathcal{W}_l^d$, and $\mathcal{W}_m$ with respect to query log $\mathcal{L}_A$. 168

Chapter 1

Introduction

In a world where companies tend to rely more frequently on data and data-driven decisions, the need for employees with the ability to find and process data has increased. The task of extracting useful sets of data requires technical skills and knowledge about databases, in order to access the necessary data, but also sufficient knowledge about the domain and how to use and interpret data related to it. This combination of skills is quite rare among single employees, which means that often two persons, one database expert and one domain expert, are needed. This scenario is not ideal. In addition to the cost of involving an additional employee in the process, it can also be problematic for the domain expert to precisely express their information need to the database expert in a language both of them understand.

It is possible to make systems that allow the domain expert to formulate ad-hoc queries directly over the data using the domain vocabulary they already know, but this requires that the system has access to a domain model and a way to connect the data to this model. Due to the emergence of the Semantic Web [5] and ontology-based data access technologies [30, 37, 44], we have seen an increased interest in *ontology-based visual query systems* [51, 2] in the recent years. These are systems that have access to a given domain ontology, and which use visual elements to support query construction over this domain. Such a system could, for example, show a visual representation of the partially constructed query, and present the domain implicitly by providing lists of valid modifications to the constructed query. From this, the user can select an extension to make a new version of the query. This extension process is repeated until the user is satisfied, and after that, they can run the final query over the dataset.

In rare cases, it may be interesting for a user to know that a query returns no answers, but we are going to assume that the user is searching for a non-empty subset of the data. In this setting, we want to make a system that is able to effectively detect and prevent the user from selecting extensions that lead to queries with no answers. Such extensions are called *dead-end extensions*.

This feature of detecting dead-ends is actually quite common in the search paradigm used by most e-commerce websites, known as *faceted search*. In these systems, the search query is modified by adding or removing filters on the available properties/facets, which means that each such filter is a potential dead-end. For example, if a user is searching for cars, and they apply a filter indicating that they only want cars produced in the last five years, then an additional filter on the lowest price range will be detected as a dead-end unless a cheap and new car actually exists in the data.

In general, dead-ends can be found by simply executing the queries generated by each of the possible extensions, and flag those where the corresponding result

is empty. But, unless the database is optimized for these kinds of queries, this process could take minutes, which is not acceptable because it interrupts the user's workflow. State of the art faceted search systems solve this problem by using index-based search engines like Lucene/SOLR or Elastic Search. However, these engines only support queries over one single class, which means that they cannot be used in our more general case where the query may consist of multiple connected classes. In fact, since we allow queries with arbitrarily many connected classes, an index to support every such query would need to be infinitely large.

Since it is impossible to guarantee both efficient and complete dead-end detection using a finite amount of memory, we are instead going to consider approximations that detect most dead-ends, but not necessarily all of them. This can be done by only considering certain essential subsets of the data, which then only requires a finite index.

In this thesis, we present a flexible framework based on this idea, where a configuration structure is used to determine which parts of the data to include in the index. In general, a small configuration leads to a cheap index in terms of memory consumption, but low precision on dead-end detection. An extensive configuration, on the other hand, will generally lead to a more costly index, but also higher precision. This trade-off between precision and memory usage is not straightforward, because certain combinations of classes, properties, or patterns, can have multiplying effects on the index size. Hence, finding suitable configurations is a non-trivial task. To solve this problem, we also provide methods to calculate optimal configurations based on the underlying data, query logs, and a given maximum index size.

1.1 Contributions

The scientific contributions of this thesis are listed below:

C1: We have developed an index-based framework that allows ontology-based visual query systems to detect dead-end extensions of queries that combine multiple classes.

C2: We have made a benchmark based on Wikidata.[1] This benchmark consists of a dataset, an ontology, and a query log with queries that our VQS model supports, rewritten from original arbitrary Wikidata queries.

C3: We have defined a way to configure the framework in C1, and we have developed methods to generate configurations that lead to systems with high precision and a cheap corresponding index. Using the benchmark described in C2, we have evaluated all these configuration generation methods.

The core contribution, which is the index-based framework (C1), has gone through several iterations during the thesis project period, and most of our early experimental work was related to the improvement of this framework. The experiments gave promising results but highlighted the importance of setting up the system based on the relevant dataset and predictions of which queries users are going to make. Hence, we started to work on methods to generate configurations based on the dataset and query logs (C3). In order to evaluate the configuration generation methods experimentally, we needed a large query log over a dataset with a corresponding ontology, and this led us to make the Wikidata benchmark (C2).

1.2 Research Papers

In this section, we list eight relevant papers published by the author, numbered from P1 to P8. All of them are peer-reviewed except for P3. Most of these papers are related to contribution C1. Our work done on C2 and C3 was recently finished and has not been published yet.

P1: Ontology-based Visual Querying with OptiqueVQS: Statoil and Siemens Cases [53]

A peer-reviewed demo paper presented at *The 2nd Norwegian Big Data Symposium* (NOBIDS 2016) and published in *Proceedings of the 2nd Norwegian Big Data Symposium (NOBIDS 2016)*

Authors: Ahmet Soylu, Martin Giese, Ernesto Jiménez-Ruiz, Evgeny Kharlamov, Rudolf Schlatte, Christian Neuenstadt, Özgür L. Özçep, Hallstein Lie, Vidar N.

[1] https://www.wikidata.org/

Klungre, Sebastian Brandt, and Ian Horrocks

Summary: This paper presents *OptiqueVQS*, an ontology-based visual query system developed as a part of the EU project *Optique*[2] [16] where six European universities, four industrial partners, and over 40 researchers, including the author participated. The paper gives a short description of the role OptiqueVQS had in the two biggest industrial cases we did in Optique Project in collaboration with Statoil (now Equinor) and Siemens. The author of this thesis contributed by participating at the OptiqueVQS user evaluation at Siemens. He also presented the paper and a demo of OptiqueVQS at NOBIDS 2016.

P2: KeywDB: A System for Keyword-Driven Ontology-to-RDB Mapping Construction [64]

A peer-reviewed demo paper presented at *The 15th International Semantic Web Conference* (ISWC 2016) and published in *Proceedings of the ISWC 2016 Posters & Demonstrations Track*.

Authors: Dmitriy Zheleznyakov, Evgeny Kharlamov, Vidar N. Klungre, Martin G. Skjæveland, Dag Hovland, Martin Giese, Ian Horrocks, and Arild Waaler

Summary: This paper presents KeywDB, a useful tool in the ontology-based data access (OBDA) setting defined by Optique Project, which allows domain experts to add or suggest missing classes and their corresponding mapping to relational databases while formulating queries. KeywDB was discontinued after Optique Project ended. The author of this thesis had an active role in KeywDB project. He implemented large parts of KeywDB, and did all of the evaluation. He took part in the writing process, and he presented the results at ISWC2016.

P3: A Faceted Search Index for Graph Queries [24]

Research report no. 469 of the Dept. of Informatics, UiO, 2017

Author: Vidar N. Klungre

Summary: This paper presents the first definitions of the navigation graph (a simplification of the ontology), the dataset, and typed, tree-shaped queries, which are the three most important components of the VQS framework we use. Furthermore, the paper introduces the *configuration query*, and how this structure is used to both define an index over the data, and how it prunes queries before executing them over this index. The paper is quite elaborate and was originally written to support the short demo paper published at ISWC 2017 (P4), which had a hard page limit.

[2] http://www.optique-project.eu/

P4: A Faceted Search Index for OptiqueVQS [25]

A peer-reviewed demo paper presented at *The 16th International Semantic Web Conference* (ISWC 2017) and published in *Proceedings of the ISWC 2017 Posters & Demonstrations and Industry Tracks*.

Authors: Vidar N. Klungre and Martin Giese

Summary: This paper presents our implementation of the index-based system in OptiqueVQS. It was published at ISWC 2017, and the system was presented at a combined poster and demo session at the same conference, where we got valuable feedback from researchers in the Semantic Web community. The paper and the corresponding implementation shows that our dead-end detection system can be used in real VQSs like OptiqueVQS.

P5: Approximating Faceted Search for Graph Queries [26]

A peer-reviewed workshop paper presented at *The 12th International Workshop on Scalable Semantic Web Knowledge Base Systems* (SSWS 2018) and published in *Proceedings of the 12th International Workshop on Scalable Semantic Web Knowledge Base Systems*.

Authors: Vidar N. Klungre and Martin Giese

Summary: This paper presents measures to evaluate a given configuration with respect to precision, by comparing it to a fully adaptive system, i.e., a system that perfectly detects dead-ends. It presents results from an early experiment where our method was evaluated using various queries and configuration queries over the NPD Factpages RDF dataset. This experiment highlights the trade-off between precision and configuration size (and hence index size), and it shows that relatively small configurations often can provide very high precision. Furthermore, it shows how crucial object variables in the configuration query can be in certain situations.

P6: Evaluating a Faceted Search Index for Graph Data [27]

A peer-reviewed conference paper presented at *The 18th International Conference on Ontologies, DataBases, and Applications of Semantics* (ODBASE 2018) and published in *On the Move to Meaningful Internet Systems. OTM 2018 Conferences*.

Authors: Vidar N. Klungre and Martin Giese

Summary: This paper introduces a new and better cost measure on configuration queries, defined by the number of cells in its corresponding index table. Based on this, and the already established precision measure over one query, we explored

the full space of configurations using the same experimental setup as in P3. We presented all configurations in a cost/precision-diagram and highlighted Pareto-optimal configurations. By doing this for all the queries, we again confirmed that a small index can lead to high precision in the best-case scenario.

P7: On Enhancing Visual Query Building over KGs Using Query Logs [28]

A peer-reviewed conference paper presented at *The 8th Joint International Semantic Technology Conference* (JIST 2018) and published in *Semantic Technology: Proceedings 8th Joint International Conference, JIST 2018, Awaji, Japan, November 26–28, 2018, Part of the Lecture Notes in Computer Science book series (LNCS, volume 11341)*

Authors: Vidar N. Klungre, Ahmet Soylu, Martin Giese, Arild Waaler, and Evgeny Kharlamov

Summary: This paper presents the idea of utilizing query logs, i.e., previously executed queries to enhance VQSs. In particular, it presents some ideas on how to use such query logs to rank and detect dead-ends among query extensions.

P8: Query Extension Suggestions for Visual Query Systems Through Ontology Projection and Indexing [29]

A peer-reviewed journal paper published in *New Generation Computing* in 2019.

Authors: Vidar N. Klungre, Ahmet Soylu, Ernesto Jimenez-Ruiz, Evgeny Kharlamov, and Martin Giese

Summary: This extensive paper covers not only our framework for index-based dead-end detection but also the process of projecting OWL 2 ontologies into navigation graphs, which is required by our model of VQSs. In addition to unifying all theory and experimental results from the previous papers, and existing theory on ontology projection, the paper also includes new results from experiments measuring the performance of the ontology projection algorithm.

1.3 Thesis Structure

The structure of this thesis is as follows: in Chapter 2, we present preliminary theory that will be useful when we later present the main work of the thesis. Then, in Chapter 3, we describe and present some examples of data access systems, and how dead-end detection is done in these systems today. In Chapter 4, we define the core parts of the VQS model we use, including the navigation graph, the dataset, and the queries it supports. In Chapter 5, we describe how to extend a query in the VQS, and how the VQS can suggest extensions to the user. Then, in Chapter 6, we present the index-based extension framework, which is the most

central part of the thesis. In Chapter 7, we present the Wikidata benchmark, and in Chapter 8 we present a set of configuration generation methods, which we also evaluate over the Wikidata benchmark. Finally, in Chapter 9, we conclude and present future work.

Chapter 2

Preliminaries

In this chapter, we present preliminary theory that we build on in this thesis. The chapter contains two sections: Section 2.1 gives a brief introduction to semantic technologies, including a description of the RDF data model, SPARQL, and OWL, while Section 2.2 covers some standard mathematical definitions we are going to make use of.

2.1 Semantic Technologies

To better understand the work in this thesis, the reader should know the basics of semantic technologies. In particular, we require some knowledge about *OWL* and *SPARQL*, the two most popular languages used to describe ontologies and queries respectively, in addition to *RDF*, which is the framework that both of them are based on. RDF, SPARQL, and OWL are all central parts of the *Semantic Web stack* (Figure 2.1), which is a collection of technologies recommended by the *World Wide Web Consortium* (W3C). They support features that are a central part of the vision of the Semantic Web, like effortless semantic integration of multiple heterogeneous data sources, and reasoning over data. Semantic technologies have also gained attention in the industry, where they are being explored as a core technology of knowledge management systems [22, 23]. In this section, we cover the essentials of these three standards[1].

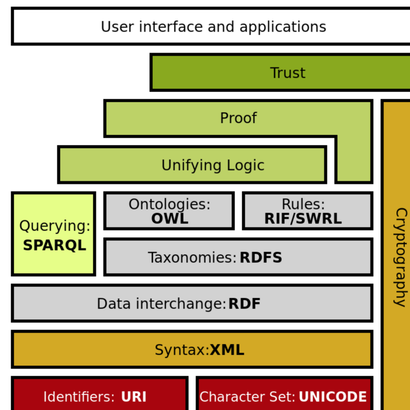


Figure 2.1: Visual presentation of The Semantic Web stack by Wikipedia user Marobi1, licensed under CC0.

2.1.1 RDF Data Model

The Resource Description Framework (RDF) is a conceptual data model, initially made to annotate web-accessible resources, but which has later been developed

[1] If the reader needs more resources on this topic, we suggest either to read a textbook about semantic technologies (e.g. [18]) or to visit each technology's official specification page online.
RDF: https://www.w3.org/TR/rdf11-concepts/
SPARQL: https://www.w3.org/TR/sparql11-overview/
OWL: https://www.w3.org/TR/owl2-overview/

into a general-purpose language for describing structured information. It is the most fundamental semantic layer of the Semantic Web stack, which both SPARQL and OWL are based on.

RDF can be used to express basic statements about entities and the relations between them. Each such statement follows a strict triple pattern, made up of three parts in the following order: a *subject*, a *predicate*, and an *object*. A statement is interpreted as follows: The subject is the entity we want to state something about, while the predicate and object, answer how and what the subject is related to respectively. For example, the two statements "The USA borders Canada.", and "The population of the USA is 330 mill.", can be represented as triples like this:

subject	**predicate**	**object**
USA	borders	Canada
USA	population	330 mill

Because every RDF statement follows this triple pattern, they are also called *RDF triples* or just *triples*.

In general, each subject, predicate, and object must either be a *resource*, a *literal*, or a *blank node*, but in our work, we do not consider blank nodes, so we simply ignore them. A literal is a data value with a corresponding datatype, so in the two triples above, only the object of the second triple is a literal. This literal consists of the number 330 mill., and its corresponding type, which is integer, and it can be expressed formally as given below, where an XSD datatype is used to specify the datatype:

```
"330000000"^^xsd:integer
```

Everything that is not a literal, i.e., all entities and relations, is considered to be resources in RDF. These resources must have a global identifier, which allows them to be recognized and reused in multiple triples to form more complex structures. To achieve this, RDF requires that each resource is associated with a Uniform Resource Identifier (URI). For example, we could assign URIs to the resources in the two statements above in the following way:

Resource	**URI**
USA	`http://www.example.com/USA`
Canada	`http://www.example.com/Canada`
borders	`http://www.example.com/borders`
population	`http://www.example.com/population`

URIs are usually long and complicated, and often many of them share a common prefix. For that reason, it is common to replace these prefixes with some shorter alias. For example, the prefix `http://example.com/` is a part of each URI above, so we can simplify the URIs by replacing this prefix with an alias `ex`.

```
@prefix ex: <http://example.com/> .
```

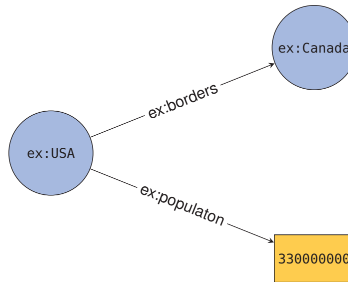


Figure 2.2: Graph representation of a small RDF dataset. The two blue circles represent resources, while the yellow rectangle represents a literal.

Now we can present our two triples in a more compressed, formal, and machine-readable format:

Two RDF triples.

```
1  @prefix ex: <http://example.com/> .
2
3  ex:USA   ex:borders      ex:Canada.
4  ex:USA   ex:population   "330000000"^^xsd:integer.
```

The way we have defined the prefix and how we present triples, correspond to a way of serializing RDF that is called *Turtle*.[2] There are several different serializations for RDF, but Turtle has the advantage of being relatively compact, and easy to read both for machines and humans.

A collection of RDF triples is called an *RDF dataset*. The RDF dataset above consists of two RDF triples, and can be visualized as a labeled, directed graph, where each vertex corresponds to a subject or object, while each edge corresponds to a predicate (see Figure 2.2). If two vertices in the graph have the same URI, then they are merged into one vertex, like `ex:USA` in our example.

This graph representation of an RDF dataset gives origin to the term *RDF graph*, which is used as a synonym for an RDF dataset. In general, it is not always possible to translate an RDF dataset into an RDF graph as we did above. For example, if a resource occurs in the subject position in one triple, and the predicate position in another one, then it is not clear whether it should be an edge or a vertex in the RDF graph. However, almost all the RDF graphs we consider in the thesis can be visualized like this, and we will also use the same kind of visualization for other kinds of graph structures throughout the thesis.

Vocabularies It is up to the publishers of RDF datasets to decide which URIs to give the included resources, but at the same time, it is advantageous when different publishers use identical URIs when they refer to the same resource, because this simplifies the data integration process if the datasets are going to

[2] https://www.w3.org/TR/turtle/

be combined later. RDF encourages the creation and use of shared *vocabularies*, which are documents that contain a collection of URIs, combined with a description of the resource each URI intends to represent in a human-readable format. Vocabularies are usually limited to a specific domain, and the defined resources are commonly associated with URIs that all share the same prefix. For example, FOAF is a popular vocabulary used to describe persons and their corresponding data and relations. It contains resources with URIs that all start with the prefix:

```
@prefix foaf: http://xmlns.com/foaf/0.1/
```

One such resource is `foaf:Person`, which is defined as the class of all persons.

Another vocabulary worth mentioning is the *RDF vocabulary*,[3], which is a part of the RDF standard. It is defined by the following prefix:

```
@prefix rdf: http://www.w3.org/1999/02/22-rdf-syntax-ns#
```

This vocabulary describes a set of general resources related to the RDF model, which are useful in many different contexts. One of the most important resources it defines is `rdf:type`, which is used to assert class membership. For example, we can state that `ex:Alice` is a person with the following triple:

```
ex:Alice rdf:type foaf:Person.
```

The resource `rdf:type` makes a clear connection between instances and their corresponding class, and it will have a central role in our work.

2.1.2 OWL

Knowledge management systems should ideally use the same vocabulary as the user, because this makes the interaction between the system and the user effortless. Specifically, in the context of query construction systems, which is what we consider in this thesis, this allows the user to formulate their information need directly to the system. For example, if the user wants information about actors with first name Daniel, then the system must provide classes and properties that allows the user to express this need, for example, with an actor class and a first name property.

Additionally, in order for the system to behave intelligently, it must use the classes and properties in the way that the user expects. I.e., the system must incorporate general relationships and rules about them, and use these rules to determine how to present information to the user. For example, the system should know that an actor is a person and that every person has exactly one first name. Hence, the system should allow users to construct queries asking for the first name of actors.

Both of the features presented above can be achieved by using what is known as an ontology. To use the words of Guarino et al. [17]: “An ontology is a

[3] https://www.w3.org/1999/02/22-rdf-syntax-ns

formal, explicit specification of a shared conceptualization.". I.e., an ontology fixes a shared vocabulary (classes and properties), and the general relationships between them. When this ontology is about a specific domain, it is known as a *domain ontology*.

There are different ways of formalizing ontologies, but W3C recommends the Web Ontology Language (OWL) (see the Semantic Web stack in Figure 2.1), and in the context of RDF, OWL is the undisputed standard ontology language. Technically, OWL is a language that can be used to express a variety of general rules over a domain, called axioms. For example, one such axiom could state that all actors are also persons, and another axiom could state that every person has exactly one first name. An OWL ontology is then simply a set of such axioms.

OWL ontologies can be processed by reasoners like Pellet [48] and Hermit [46], and this allows systems to infer logical consequences from the ontology and possibly data related to the ontology. For example, if the dataset states that A is an actor, then a reasoner should be able to infer that A must also be a person, and hence, that they must have a first name. OWL ontologies are supported by the open-source ontology editor Protégé [39], which also has an extensive plugin library.

2.1.3 SPARQL

The *SPARQL Protocol And RDF Query Language* (SPARQL) is the W3C recommended RDF query language. It is used to extract data from RDF datasets, similar to how SQL is used to query over the relational model. Readers familiar with SQL should notice many similarities between the two query languages, however, since they operate over two fundamentally different data models, they also have significant differences.

SPARQL supports four types of queries: `SELECT`, `CONSTRUCT`, `ASK` and `DESCRIBE`, but only `SELECT` is relevant to our work in this thesis, so we simply ignore the three other types. The simplest form of `SELECT` query consists of only one *basic graph pattern*, which is an RDF graph, i.e., a set of RDF triples, where some of the resources may be replaced by variables. When such a query is executed over an RDF dataset, it returns all possible ways to replace the variables in the graph pattern with entities such that the pattern perfectly matches a subgraph of the data.

Let us consider an example. The following SPARQL query, $\mathcal{Q}_1$, asks for the population of all countries that share a border with the USA:

$\mathcal{Q}_1$: Population of countries that borders the USA.

```
1  PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
2  PREFIX ex:  <http://example.com/> .
3
4  SELECT ?country ?population
5  WHERE {
6     ?country   rdf:type        ex:Country.
7     ?country   ex:borders      ex:USA.
8     ?country   ex:population   ?population.
9  }
```

This query consists of three major parts, denoted by the upper-case keywords `PREFIX`, `SELECT`, and `WHERE`. The first part, covered by the two lines starting with the `PREFIX` keyword, are just prefix declarations, similar to those we presented for RDF in Section 2.1.1. The second part of the query is line 4, starting with the keyword `SELECT`. The part after this keyword specifies which format the query results should have. In this example, this is just a list of the two variables to return: `?country` and `?population`. Every word in SPARQL starting with a question mark is a variable. The remaining part, starting with the keyword `WHERE` on line 5, defines the actual query pattern we want to match in the dataset. In this example, the clause only defines one basic graph pattern consisting of three triple patterns. The first of these declares a variable `?country` of type `ex:Country`, while the two others relate this variable to the resource `ex:USA` and another variable `?population` via the two predicates `ex:borders` and `ex:population` respectively.

Now consider the example RDF dataset below, which contains facts about four countries.

RDF dataset: four countries.

```
1   @prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
2   @prefix ex:  <http://example.com/> .
3
4   ex:Mexico   rdf:type        ex:Country.
5   ex:USA      rdf:type        ex:Country.
6   ex:Canada   rdf:type        ex:Country.
7   ex:Cuba     rdf:type        ex:Country.
8   ex:USA      ex:borders      ex:Canada.
9   ex:USA      ex:borders      ex:Mexico.
10  ex:Mexico   ex:borders      ex:USA.
11  ex:Canada   ex:borders      ex:USA.
12  ex:Canada   ex:population   "37000000"^^xsd:integer.
13  ex:USA      ex:population   "330000000"^^xsd:integer.
14  ex:Mexico   ex:population   "129000000"^^xsd:integer.
```

If query $\mathcal{Q}_1$ is executed over this dataset, we get two possible assignments of the variables `?country` and `?population`:

?country	?population
ex:Mexico	129000000
ex:Canada	37000000

SPARQL supports a broad set of filters. For example, it is possible to filter only on countries with a population less than 100 million like this:

$\mathcal{Q}_2$: Countries that border the USA with a population less than 100 million.

```
1  PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
2  PREFIX ex: <http://example.com/> .
3
4  SELECT ?country ?population
5  WHERE {
6     ?country   rdf:type        ex:Country.
7     ?country   ex:borders      ex:USA.
8     ?country   ex:population   ?population.
9     FILTER (?population < 100000000).
10 }
```

This filter excludes Mexico from the list of results, leaving Canada as the only remaining country.

?country	?population
ex:Canada	37000000

A general SPARQL filter has the form `FILTER (exp)`, where `exp` is a boolean expression that has to be true for a potential solution to be accepted. SPARQL supports an extensive set of operators and functions, which can be used to construct such boolean expressions, including `>`, `<`, `=`, `!=`, `>=`, `<=`, `&&`, `||`, `+`, `-`, `*`, `/`, `bound`, and `regex`. A full overview can be found in the SPARQL specification.[4]

Another feature of SPARQL that is central to the work of this thesis is the `OPTIONAL` keyword, which makes it possible to define parts of a query to be optional, similar to how `LEFT JOIN` works in SQL. For example, if we want a list of all the countries and their corresponding population, given that it exists in the dataset, then we have to enclose the triple about the population with an `OPTIONAL` clause like this:

$\mathcal{Q}_3$: All countries and their population, if it exists.

```
1  PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
2  PREFIX ex:  <http://example.com/> .
3
4  SELECT ?country ?population
5  WHERE {
6     ?country rdf:type ex:Country.
7     OPTIONAL { ?country ex:population ?population. }
8  }
```

The resulting tuples of this query are given in the table below. It lists all the four countries from our dataset, but only the population for three of them, since the population of Cuba is missing in the data. If the `OPTIONAL` clause around the population triple in $\mathcal{Q}_3$ was not included, Cuba would not have been included in the results at all.

[4] https://www.w3.org/TR/sparql11-query/

?country	?population
ex:Mexico	129000000
ex:Canada	37000000
ex:Cuba	
ex:USA	330000000

The third feature of SPARQL we are going to cover is `UNIONS`s, which makes it possible to combine the results of two smaller query patterns. For example, query $\mathcal{Q}_4$ below requests all countries that either has a population of more than 100 mill. people, or shares a border with the USA.

$\mathcal{Q}_4$: Union of two graph patterns.

```
1  PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
2  PREFIX ex: <http://example.com/> .
3
4  SELECT ?country
5  WHERE {
6     {
7        ?country rdf:type      ex:Country.
8        ?country ex:population ?population.
9        FILTER (?population > 100000000).
10    }
11    UNION
12    {
13       ?country rdf:type      ex:Country.
14       ?country ex:borders    ex:USA.
15    }
16 }
```

$\mathcal{Q}_4$ gives the following results:

?country
ex:Mexico
ex:Canada
ex:USA

In addition to what we have already described, it is worth mentioning that SPARQL features aggregation functions like `COUNT`, `SUM`, `AVG`, `MIN`, and `MAX`, and it allows the user to group these aggregates by a particular set of variables with the `GROUP BY` keyword. It also supports `DISTINCT` and `REDUCE`, which are keywords that can be used to remove duplicates in sets of tuples.

2.2 Mathematical Definitions

In this short section, we review the definitions of a few standard mathematical concepts that will be used throughout this thesis. The first part presents three definitions related to functions in general, while the second part covers some useful basic concepts related to labeled, directed graphs.

2.2.1 Functions

Given a function defined on a domain X, we may be interested in the function restricted to only a subset of X. This is called *function restriction*.

Definition 2.2.1 (Function Restriction). Let $f: X \to Y$ be a function and let X_s be a subset of X. The restriction of f to X_s, denoted $f{\restriction}_{X_s} : X_s \to Y$ is defined as $f{\restriction}_{X_s}(x) = f(x)$ for all $x \in X_s$.

$\dashv$

Another mathematical definition we will use in this thesis is the function over a set defined below.

Definition 2.2.2 (Function of Set). Let $f: X \to Y$ be a function and let X_s be a subset of X. Then $f(X_s)$ is the f-image of X_s onto Y, i.e.,

$$f(X_s) = \{f(x) \mid x \in X_s\}$$

$\dashv$

A result of this is that we can use the statement $f(X_s) \subseteq Y_s$ to express that each value in X_s maps to something in Y_s, i.e.,

$$f(X_s) \subseteq Y_s \Leftrightarrow \{f(x) \mid x \in X\} \subseteq Y_s \Leftrightarrow f(x) \in Y_s \; \forall x \in X_s$$

Definition 2.2.3 (Function Composition). Let $f: X \to Y$ and $g: Y \to Z$ be two functions. The *function composition* of g and f, denoted $g \circ f$, is the function from X to Z defined by $(g \circ f)(x) = g(f(x))$ for all $x \in X$. $\dashv$

When defining functions, it will sometimes be convenient to just list the set of mappings from each element in the domain to its corresponding element in the codomain. For example, if $X = \{x_1, x_2, x_3\}$, $Y = \{y_1, y_2, y_3\}$, and $f: X \to Y$ is defined by $f(x_1) = y_1$, $f(x_2) = y_1$, $f(x_3) = y_3$, then the following shorthand notation may instead be used:

$$f = \{x_1 \mapsto y_1, x_2 \mapsto y_1, x_3 \mapsto y_2\}$$

2.2.2 Labeled, Directed Graphs

In Chapter 4 we will define a special kind of graph, called a *resource graph*, to model RDF datasets, SPARQL queries, and the navigation graph. Resource graphs will share many properties with standard labeled, directed graphs, so in this short section, we will just cover some basic notions related to them.

A standard labeled, directed graph is formally defined as a pair $G = (V, E)$, where V and E are sets called *vertices* and *edges* respectively. An edge $e \in E$ in this graph is defined as a triple (v_s, l, v_t) where $v_s \in V$ is the starting point of the directed edge, called the *source vertex* of e, $v_t \in V$ is the endpoint of the edge, called the *target vertex* of e, and l is the *label* associated with the edge.

Given two graphs $G = (V, E)$ and $G' = (V', E')$, a function $f: G \to G'$ is called a *graph homomorphism* if it maps the endpoints of each edge in G to endpoints of an edge in G' with the same direction and label.

Definition 2.2.4 (Labeled Directed Graph Homomorphism). Let $G = (V, E)$ and $G' = (V', E')$ be two labeled, directed graphs. A function $f \colon V \to V'$ is called a *homomorphism* from G to G' if

$$(v_s, l, v_t) \in E \implies (f(v_s), l, f(v_t)) \in E' \tag{2.1}$$

⊣

If a function f from G to G' is bijective, its inverse f^{-1} is well-defined, and if both f and f^{-1} are homomorphisms, then f is an *isomorphism*, and G and G' are said to be *isomorphic*.

Definition 2.2.5 (Labeled Directed Graph Isomorphism). Let $G = (V, E)$ and $G' = (V', E')$ be two labeled, directed graphs. A bijective function $f \colon V \to V'$ is called an *isomorphism* between G and G' if both f and f^{-1} are homomorphisms. If f is an isomorphism, then G and G' are said to be *isomorphic*. ⊣

Intuitively, both homomorphisms and isomorphisms are functions that preserve structure between the two graphs in some way. An isomorphism preserves the full structure between the graphs, i.e., the two graphs become completely equal if the vertices are renamed by the function f. Homomorphisms, on the other hand, are less restrictive and can map between non-isomorphic graphs.

Example 2.2.6. Consider the two graphs G and G' in Figure 2.3, and the three functions f_1, f_2, f_3 from G to G' defined below.

$$\begin{aligned} f_1 &= \{a \mapsto A, b \mapsto A, c \mapsto A, d \mapsto A\} \\ f_2 &= \{a \mapsto A, b \mapsto B, c \mapsto A, d \mapsto B\} \\ f_3 &= \{a \mapsto A, b \mapsto B, c \mapsto C, d \mapsto D\} \end{aligned}$$

Function f_1, which maps every vertex in G to the vertex A in G', is not a homomorphism. We can prove this by considering the edge $e = (a, p, b)$ in G: the two endpoints of e (a and b) are both mapped to A, but there is no edge from A to A with label p. Function f_2 is a homomorphism because each edge of G corresponds to the edge (A, p, B) in G'. Notice that the two vertices C and D are completely ignored by this function. Hence f_2 would have been a homomorphism even if C and D were removed from G'. Function f_3 is an isomorphism: (a, p, b), (b, p, c), and (c, p, d) corresponds to (A, p, B), (B, p, C), and (C, p, D) respectively. We also see that we get G' if we replace each variable in G with its corresponding upper-case letter.

♦

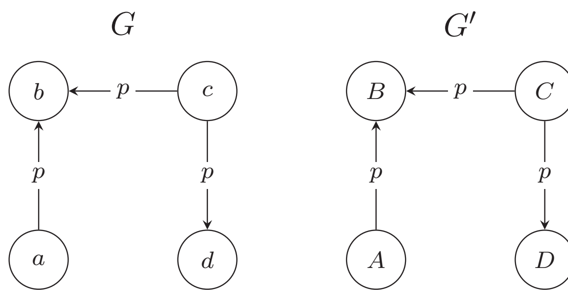


Figure 2.3: Two labeled, directed graphs: G and G'.

Chapter 3

Dead-End Detection

In this chapter we describe existing data access systems, and how they approach the problem of detecting dead-ends, i.e., how they prevent the user from constructing queries that lead to no answers. We start by introducing data access systems in general, including some of the most relevant aspects of such systems in Section 3.1. Then, in Section 3.2, we present some examples of existing systems. Finally, in Section 3.3 we take a more detailed look at how dead-end detection is done in data access systems today, and in particular how it is done in the systems from Section 3.2.

3.1 Data Access Systems

When we use the term *data access system*, we refer to any system that allows a user to access and interact with data. This definition is very broad, and it includes *ontology-based visual query systems*, which is the kind of system we target in this thesis. The term also includes other *RDF-based systems*, *faceted search systems*, *information retrieval systems*, and many other types of systems. By considering all of these different kinds of systems, and not only VQSs, we give context to our work. Furthermore, much of the existing work related to what we try to do in this thesis project is not tied directly to VQSs, but other similar systems. For example, dead-end detection is very established in faceted search systems, and it is important that we understand how it is done in those systems before we start to describe our approach for VQSs, which covers less restrictive queries.

3.1.1 Aspects of Data Access Systems

Before we consider any actual systems or types of systems, we need to describe and discuss the most relevant *aspects* of such systems. We have also included aspects related to the systems' UI, their intended user group, and the dataset they are supposed to work with, since they are closely related to the system itself. Defining these aspects now makes it easier to understand and compare the different concrete systems when we present them, and it enables us to better understand how our work relates to existing systems and approaches. Data access systems have been studied many times before, and all the aspects we present below have been described in earlier papers (see [31, 11, 52, 49]). The purpose of defining these aspects here is not to compete with existing analyzes, but rather to give the selection of aspects that we believe are relevant to our work. For an overview of the 17 different aspects we consider, see Table 3.1, where each of them is presented and described briefly.

Aspect	Description
User Intention	What the user intends to do.
User Skills	How skilled the user is.
Usability	How usable the system is in general.
Query Expressivity	How expressive the queries are.
Query Repetitiveness	How much queries tend to repeat.
Provided Actions	Which actions the system provides to the user.
Interactivity	How interactive the system is.
Query Presentation	How the query is presented.
Results Presentation	How results are presented.
Domain Presentation	How the domain is presented.
Data Structure	How the data is structured.
Data Quality	How complete and granular the data is.
Data Volume	How large the dataset is.
Data Velocity	How frequently the dataset is updated.
Portability	How easy it is to port the system to other datasets.
Efficiency	How efficient the system is.
Adaptivity	How well the system adapts to the context.

Table 3.1: The 17 different aspects of data access systems we consider.

User Intention We start by considering the aspect of whether the user is looking for a specific subset of the data, or if they just want to explore the dataset. Our work will focus on the scenario where the user wants to extract a specific subset of the data, i.e., they have an exact *information need*. In this setting, the user needs a system that allows them to transform this information need into some kind of query, which can then be executed over the data to give answers, which will then be presented back to the user. This is in contrast to the scenario where the user wants to understand the dataset or domain as a whole. If this is the case, then they would benefit much more from a system that is made for this, like an *exploration system* [61], or a *data browser* [15].

User Skills Users who know how to use databases, query languages, and who have technical skills in general, are called *database experts* or *IT experts*. They have the skills needed to fetch data directly from databases, but unfortunately, they often lack the domain knowledge necessary to make value out of the data they extract. This is in contrast to *domain experts*, who have the needed domain knowledge but lack the technical skills. They need access to the data but are not able to access it without help. One solution to this problem is to let the database expert map data from the database to some conceptual model of the domain used by the system. After that is done, the system can be used by multiple domain experts to formulate queries over the domain they already know. Some authors also use the two terms *lay users* and *casual users* [11]. They are more or less considered to be synonyms, and they refer to users who in general lack technical skills, i.e., those who are not IT experts. Casual users could also refer

to those who will only use the system once or a few times. For those users, it is important with an intuitive system, which does not require any upfront training. The opposite of casual users is those who work with a system for hours, days, or years, like employees in a company. These users will eventually learn how to to use the system, even if it is less intuitive initially [11].

Usability A common metric used to measure the quality of a system is *usability*, which combines four more concrete quality metrics: effectiveness (i.e., accuracy and completeness), efficiency (i.e., time/effort required), learnability (i.e., time and effort required to learn the system), and user satisfaction [54]. In general, one should aim for systems with as high usability as possible. This can be done to some degree by using general design principles, but in order to achieve the best possible result, it is very important to also consider the needs of the target user group. A system can score high on usability for one user group, but low for another. The usability of systems can also be both quantified and compared by using the standardized SUS scale, [7] for example.

Query Expressiveness The next thing we consider is the system's *expressivity*, which is a measure of how much can be represented in the query language supported by the system. If the expressivity of the system is lower than the expressivity of a user's information need, the user has to simplify their need before they can formulate it as a query, which means that the system will be less useful, or not useful at all to them. In general, higher expressivity is better, but this often comes at the expense of the usability [57, 52]. The challenge, for most systems, is therefore to find the right balance between expressivity and usability for the target user group. It should allow all, or most of their information needs to be formulated, while still providing the highest possible level of usability.

Query Repetitiveness If we consider all the information needs of all the users over time, it may be possible to discover certain patterns. If some parts are repeated frequently, or even specific queries occur often, for example, it may be worthwhile to consider a template-based query system, where users can just fill in the small parts that change from one query to the next. Or, if the queries that are constructed follow a strict, predictable pattern, it should be possible to automate the process entirely. In our work, we are focusing on the more challenging scenario, where users want answers to *ad-hoc queries*.

Provided Actions and Interactivity The UI of the system should allow the user to interact easily with the query they are formulating. In practice, this means to provide actions that allow the user to add, delete, or edit any part of the query, while at the same time only allowing queries covered by the expressiveness of the system. Furthermore, as in any interactive UI, each user action should be followed up with an appropriate response.

Query Presentation What the UI actually displays to the user varies between systems, but it is natural to present the status of the partial query in some way. One way of doing this is to present the textual representation of the query, expressed in the relevant query language, but, since many users are not familiar with this language, it is often better to provide some kind of visual representation [11]. Many systems use a graph-based representation of the query, but it is also possible to have a form-based UI where the query is both formulated and presented by the selections in the form. If the query includes filters, each filter must be displayed with the variable they are applied to.

Results Presentation Some systems will also display the results of the partial query in the UI, and update this after every change done to the constructed query. This is useful, at least if the user wants to explore the results before they continue to modify the query. However, it can also be time-consuming, since it requires the query to be executed over the data, but this depends on the size and complexity of the query and the underlying dataset. If the amount of data is small, and the query is simple, then this may only take a few milliseconds, which is not a problem at all. However, if the query is complex, and it needs to be executed on a large dataset, this may take minutes, which is too much time, at least if the user needs the results before they can continue to work. The alternative is to only show results when the user specifically asks for them, for example, when the user believes they are done with the query.

Domain Presentation Finally, if the system has a model of the domain, like an ontology, this could be displayed to the user. This is probably more relevant for domain exploration systems than query construction systems. It is also possible to display the actual database schema in cases where a domain model does not exist, and the user has to construct queries directly over the schema.

So far we have considered the user, their information needs, the queries that can be made, and the UI of the system. Now we will consider some aspects related to the underlying data.

Data Structure We start by considering the format of the data, i.e., how it is organized. It is, for example, of interest to know which data model the data uses, if any, and if it follows constraints defined by a database schema or a similar higher-level structure. This very much determines how structured the data is, and hence how easy it is to extract useful subsets from it. For example, consider data in a relational database. It must adhere to the relational model, where a predefined schema determines its structure and the constraints it must adhere to. Other kinds of databases, which fall under terms like NoSQL, graph databases, or key-value stores, also have enough structure to allow precise querying, but there is not necessarily a predefined schema over the data stored in these databases. In contrast to this, we have everything that is considered unstructured data, which covers collections of text documents in natural language, images, videos, and

audio files. These formats are hard to access and query over directly, and usually, they have to be processed into more structured formats, either by humans or statistical methods, before the data can be used.

The aspects related to the structure of the data determine if the system is a *data retrieval system* or an *information retrieval system*. In data retrieval systems, the underlying data is very structured, and this allows the user to formulate precise queries, which can also be precisely answered. In the information retrieval setting, on the other hand, the underlying data is not structured in a way that enables the system to give precise answers to queries. In this setting, it is impossible to know precisely which objects should be returned, and often statistical methods are used to calculate a ranking of each object before the top-k most relevant of them are returned. A Web search engine is a typical example of an information retrieval system, which attempts to rank web documents of different sizes, types, and complexities based on a simple query containing just a set of keywords.

Data Quality Next, we consider the *data quality* aspect, which is a measure of how detailed, correct, and complete the data is. A typical example of a dataset with high quality is a dataset covering a specific domain, controlled by a company, where data is collected and updated by sensors and computer programs. The opposite of this, a dataset with low quality, could be a dataset where humans are responsible for updating and adding data, and where the domain is very broad, and where it is hard to control who is editing what. An example of this is Wikipedia, since it is a huge collaborative project, which aims to cover all areas of human knowledge.

Data Volume The next aspect related to data, is its *volume*, i.e., how many facts, or bytes are stored in the dataset. Larger datasets are in general harder to both store and access, because there is more data to manage and search over. Some datasets are even so large that they cannot be stored on a single computer, which means that they have to be stored in some kind of distributed database. Smaller datasets, on the other hand, could possibly be stored in in-memory databases or other storage solutions that boost the efficiency of processing by storing the data closer to the processing unit.

Data Velocity The third aspect related directly to the data is its *velocity*, which indicates how frequently the data is updated, or how fast new data is added to the dataset. The velocity of the data, and how crucial up-to-date data is for the users, affects how successful indexing is going to be. It is, for example, pointless to construct an index if it becomes outdated before it is even is set up. Since our work is based on indices of the data, we require data with relatively low velocity.

The aspects described so far have had a natural relationship to either the user, the query, the UI, or the dataset, but the three remaining aspects are more general and relate to the system as a whole.

Portability The system's degree of *portability* refers to its ability to work on arbitrary domains without the need for manual tuning or configuration in advance. For example, if a system made for the oil and gas domain contains source code or UI widgets that are specifically related to drilling, then these parts of the system need to be rewritten before it can be used on another unrelated domain. It is also worth considering whether the system requires some kind of upfront processing, data analysis, or indexing each time it encounters a new dataset. This does not usually require manual work, but it will cause some extra setup time before the system can be used.

Efficiency The next aspect to consider is the system's *efficiency*, i.e., how fast it finishes important tasks like fetching results, making suggestions, and preprocessing data. In general, faster results are better, but there is no need to improve the efficiency if the system is already fast enough, at least if such an improvement lowers the system's quality on other aspects. For example, if an interactive system provides new action suggestions to the user in 50ms, it is already so fast that the user cannot even notice the delay, so improving it to something like 10ms should not have high priority.

Adaptivity Last, but not least, we consider the system's *adaptivity*, which refers to the system's ability to adapt based on the context, or previous experience. For example, a system that always presents the same static list of filters, is not very adaptive, while a system that detects dead-ends, and removes filters based on this, scores high on adaptivity. Another example of high adaptivity is when a system ranks possible actions by how frequently they have been used in previously constructed queries.

3.2 Existing Systems

In this section, we are going to present and discuss three categories of data access systems: faceted search systems, RDF-based systems, and information retrieval systems. For each of these three categories, we will consider some concrete systems, and highlight interesting aspects of them.

3.2.1 Faceted Search Systems

Faceted Search [59] is a search paradigm that is commonly used by e-commerce websites like eBay,[1] PriceSpy,[2] and Amazon.[3] It allows users to search for objects of a given class by applying filters to independent properties of the class, called *facets*, in any order preferred by the user. This is often also combined with free-text search.

[1] https://www.ebay.com/
[2] https://www.pricespy.co.uk/
[3] https://www.amazon.com/

Figure 3.1: Screenshot of the faceted search interface used by PriceSpy, where the facet section is on the left side, and the result section is on the right side.

After the user has selected a particular class, e.g., a product category from the given taxonomy, they are presented with the faceted search interface, which is typically divided into two sections: a *facet section*, and a *result section*. The facet section lists each of the facets related to the relevant class, together with elements that allow the user to add, remove, or change filters to each of them. Meanwhile, the result section displays a list with all the objects satisfying the set of active filters. This result list is updated after each filter change. For example, consider the interface of PriceSpy in Figure 3.1, where the user is searching over the class of mobile phones. The facet section to the left shows three facets: Price, Brand, and Operating system, and two active filters: price lower than £100, and operating system equal to Android. The result section to the right displays the 281 mobile phones that satisfy both of these filters.

The set of filters chosen by the user defines a query, which can be formulated in the query language of the underlying database. Each time the user changes any of the filters, this query is updated in the background before it is executed over the dataset. This leads to an updated result list, which is then presented in the result section. Since the user can see the active filters in the filter section, and the results of the query in the result section, there is no need to present the query explicitly in the UI.

Queries generated by faceted search systems are quite simple since they only ask for objects belonging to one particular class. It is, for example, not possible to ask for mobile phones that fit a particular kind of phone cover, since

this requires a relation between two different classes: phone and cover. More precisely: queries generated by faceted search systems must contain exactly one variable typed to the selected class, and this variable can only be connected to at most one variable for each facet. In the context of SPARQL, these queries are called *star-shaped queries*, while in the context of SQL, they are queries without any joins. These queries can be answered efficiently by using state-of-the-art search engines, and this scales well to millions of objects. For example, eBay allows its users to search in eBay's collection of 44 million books using faceted search. Two well-known search engines, which both support faceted search, are SOLR[4] and Elastic search.[5]

In addition to the standard faceted filtering we described above, it is also common to include adaptive features that help the user to select useful filters. One such feature is to add a number to each possible filter, indicating how many objects will be returned if the user activates the filter. This can be seen in Figure 3.1, under the brand facet, where for example Google is listed with the number 2 behind it. This tells the user that if they filter on Google phones, only two phones will remain. Another related feature is the detection of dead-ends, which in this context are filters that lead to queries without answers. Dead-ends are usually disabled or removed entirely, in order to prevent the user from selecting them.

The dead-end detection framework we present in this thesis achieves the same thing as dead-end detection in faceted search, in the sense that they both prevent the user from making queries without answers. But, while faceted search is limited to queries containing only one class, i.e., star-shaped queries, we will instead address the problem of detecting dead-ends when the queries combine variables typed to multiple classes. More precisely, we are going to detect dead-end extensions of arbitrary large typed, tree-shaped SPARQL queries.

3.2.2 RDF-Based Systems

When we refer to RDF-based systems, we mean data access systems that help the user to access data from an RDF dataset. A special kind of RDF-based systems is what we call *ontology-based systems*, which are systems that use an ontology actively. In this section we take a closer look at five RDF-based systems: RDF Surveyor [61, 60],[6] PepeSearch [62],[7] OptiqueVQS [51, 54],[8] SemFacet [1, 3, 4, 2],[9] and Rhizomer [8, 9].[10]

Among these five systems, OptiqueVQS plays a special role, because it defines the VQS model we have based our work on [55]. Therefore, the description of OptiqueVQS is much more detailed than the description of the other four

[4] https://lucene.apache.org/solr/
[5] https://www.elastic.co/
[6] http://tools.sirius-labs.no/rdfsurveyor/
[7] https://github.com/guiveg/pepesearch
[8] https://sws.ifi.uio.no/project/optique-vqs/
[9] https://www.cs.ox.ac.uk/isg/tools/SemFacet/
[10] http://rhizomik.net/html/rhizomer/

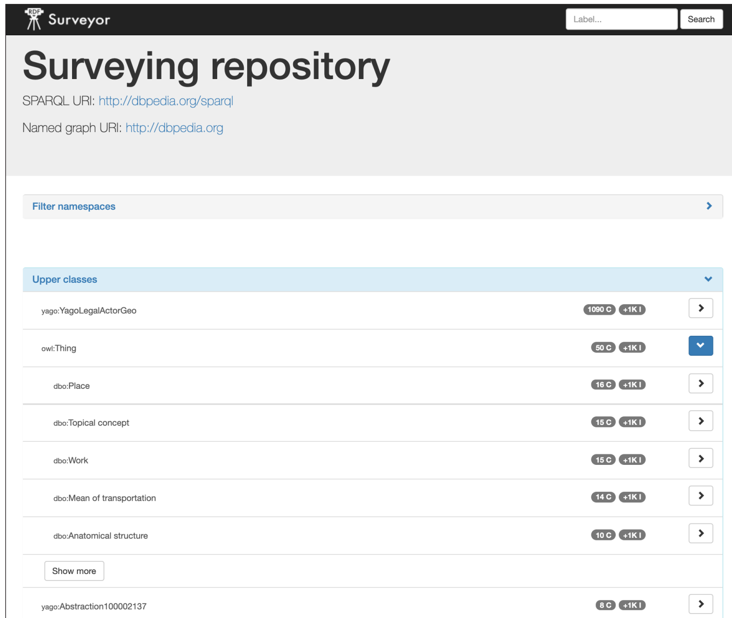


Figure 3.2: Screenshot of RDF Surveyor.

systems. That said, both SemFacet and Rhizomer use similar models, while both RDF Surveyor and PepeSearch use simpler models, so our system will be applicable also for the four other systems we present.

The five systems we have decided to present, are interesting because they have some kind of search functionality, which helps the user to find specific subsets of the data. The alternative to this is RDF-based systems that are more focused on the task of exploring or browsing the data. They are often called *linked data browsers*, or just *RDF-based exploration systems*. Some examples of such systems are Tabulator [6], Marbles,[11] Lodview,[12] and Phuzzy.link [43], just to mention a few.

RDF Surveyor RDF Surveyor [61, 60][13] is a lightweight, browser-based *linked data exploration system*. It requires no manual setup, just a link to the SPARQL endpoint of the dataset to explore. After this link has been provided, RDF Surveyor constructs a taxonomy, i.e., a hierarchy, of the classes in the dataset based on class membership and subclass axioms in the dataset. This taxonomy is then presented to the user, such that they can select a particular class to browse. After selection, the system returns a list of instances belonging to that class. Instances or classes can also be found by using the provided search bar

[11] http://mes.github.io/marbles/
[12] https://www.lodview.it/
[13] http://tools.sirius-labs.no/rdfsurveyor/

on the top, but except for that, there are no filtering options. Figure 3.2 shows the interface of RDF Surveyor, where the taxonomy of DBpedia [32][14] classes is presented as an indented tree, ready to be selected by the user.

RDF Surveyor is very portable since it can be used directly from the web browser after the URL to the endpoint has been specified. It is targeted towards lay users and has a clean, simplistic interface. The system poses queries directly to the data source it is connected to after each user action. The most challenging task is to generate the class hierarchy, but this is only done once at the beginning of the session and takes less than 5 seconds with large RDF-graphs like DBpedia. Other queries posed to the data source are relatively simple, and the system responds relatively fast (within about a second), even when working over large datasets.

PepeSearch PepeSearch [62][15] is a form-based search interface over RDF data, designed for casual users. It uses a SPARQL endpoint analyzer[16] to extract the set of relevant classes and properties. A PepeSearch session starts by letting the user select one of these classes to focus on. A form is then generated based on not only this class and its properties, but also all classes related to the focus class, and their properties. This form can be filled out by the user, and after submitting it, the system generates a corresponding SPARQL query which returns answers when it is executed over the dataset.

The set of queries that can be made by PepeSearch, i.e., its expressivity, is especially interesting. It is very common to support queries with one class and its properties, but PepeSearch extends this idea by also including the properties of every class connected to this focus class. While it is clever to associate another class' property to the focus class, it is also important to highlight when such a property belongs to another class, to not confuse the user. PepeSearch achieves this by having different sections in the form, one for each of the connected classes. The expressivity of PepeSearch is still lower than the expressivity we are targeting with our system, which is arbitrarily large, typed, and tree-shaped queries.

The PepeSearch paper [62] also includes an interesting related work section, where several related systems are presented and compared. This includes SPARQL editors, keyword-based search interfaces, question-answering systems, graph-based query editors, and form-based query editors.

OptiqueVQS OptiqueVQS [51, 54] is an ontology-based visual query system, which was developed as a part of Optique Project [16], an EU project that lasted from 2012 to 2016.[17] The goal of Optique Project was to develop methods and software to support ontology-based data access (OBDA). One of the main products was the *Optique Platform*: an end-to-end OBDA system.

[14] https://wiki.dbpedia.org/
[15] https://github.com/guiveg/pepesearch
[16] https://github.com/simenheg/sparql-endpoint-analyzer
[17] http://optique-project.eu/

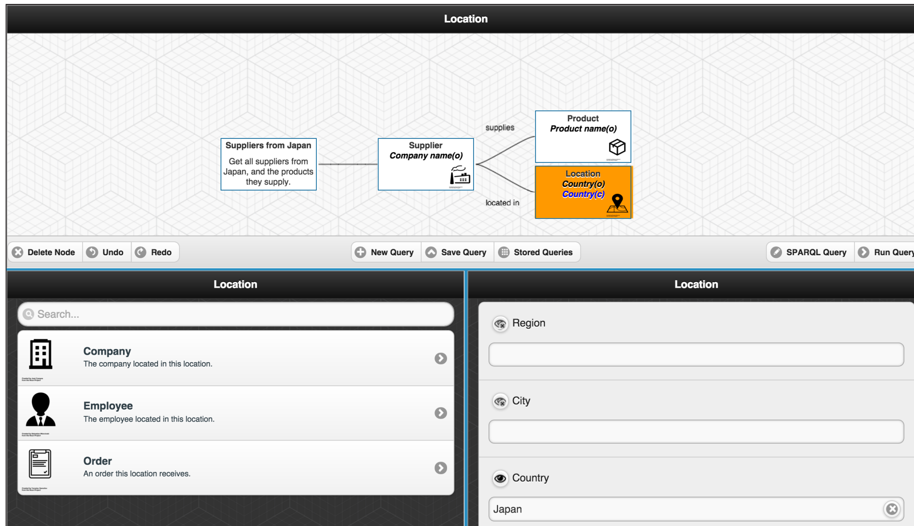


Figure 3.3: Screenshot of OptiqueVQS.

After setting up this platform, end-users will be able to construct SPARQL queries in OptiqueVQS, guided by an input ontology. Then, by using R2RML mappings and Ontop [10], this SPARQL query will be rewritten into an SQL query, which can then be distributed efficiently over the set of data sources using Exareme.[18] The results from this process will then be piped back to OptiqueVQS, where they are presented to the end-user. Recently, OptiqueVQS was separated from the Optique platform, to make a standalone application that can be used to construct SPARQL queries over any input ontology, before sending it to a given SPARQL endpoint.[19]

A typical user of OptiqueVQS is an employee in a company with a complex information need, who needs access to company data, but who struggles to do so because they lack IT and database skills. Without OptiqueVQS, this employee would need to contact a database expert each time they need access to data, which is obviously not ideal. OptiqueVQS helps the employee to access data without assistance from a database expert in two ways. First, it provides a visual system with an intuitive UI, which allows the user to construct queries without having to know the query language. Second, it uses the vocabulary of the domain ontology, which the user already knows. In order to make the interface as simple as possible, and not overwhelm the user, OptiqueVQS does not support full SPARQL. It supports tree-shaped queries where each variable is typed to exactly one class or datatype, and where filters can be applied to the data properties.

[18] http://madgik.github.io/exareme/dfl.html
[19] https://sws.ifi.uio.no/project/optique-vqs/

The interface of OptiqueVQS, which is displayed in Figure 3.3, contains one large section at the top, where the current state of the query is presented as a graph. Below this, to the left and right, are two more sections, listing possible object property extensions, and data property extensions respectively. If we ignore the leftmost vertex of the query, which only displays the title of the query, the graph-based visualization in Figure 3.3 displays three vertices, all with the shape of a rectangular box. Each box represents a variable of the query that is typed to a given class, and the name of their corresponding classes, *Supplier*, *Product*, and *Location*, are displayed inside the boxes. The three boxes are connected by edges representing object properties, and this tree-shaped graph makes the core of the query. In our example from Figure 3.3, there is one variable typed to *Supplier*, which is connected to both a *Product* via the *supplies* object property, and a *Location* via the *locatedIn* object property. In addition, there is a filter on the data property that links the supplier to the name of its country, and this filter states that the country name should be Japan. Below is the SPARQL query that corresponds to this query. The three variables $?c1$, $?c2$, and $?c3$ correspond to the supplier, the product, and the location respectively.

SPARQL query: Suppliers from Japan, and their products.

```
1  SELECT * WHERE {
2     ?c1 rdf:type ex:Supplier.
3     ?c2 rdf:type ex:Product.
4     ?c3 rdf:type ex:Location.
5
6     ?c1 ex:supplies ?c2.
7     ?c1 ex:located_in ?c3.
8
9     ?c1 ex:company_name ?a1.
10    ?c2 ex:product_name ?a2.
11    ?c1 ex:country ?a3.
12
13    FILTER(?a3 = "Japan").
14 }
```

By selecting one of the boxes, i.e., a variable in the query, it will turn orange, and the variable will become the *focus variable*. The two sections below will then update to show information and actions related to this focus variable. The left side will display a list of possible object property extensions, i.e., pairs consisting of one object property, and the target class of this property. The user can select any of these pairs to extend the query with a new variable connected to the focus variable. The selected pair will then specify the type of the new variable, and which object property is used to connect the focus variable to this new variable. For example, in Figure 3.3, the variable of type *Location* is in focus, and the system suggests a list of three possible object property extensions in the bottom left section. The first extension suggested in this list is a new variable of type *Company*, which will be connected to the *Location* variable by a *locatedin* property. Meanwhile, the bottom right section displays the set of all datatype properties related to the class in focus. By selecting any of these, a new variable

will be added, but this variable will not be displayed as a box, like those who are assigned to a class. Instead, the data property will appear inside the box of the variable it is attached to. For example, the variable of type *Location* is connected to a variable via the data property *country*, which relates a location to the name of the country it lies inside. The bottom right section also contains elements that allow the user to specify filters.

Which object properties and data properties to display in the two bottom sections, should be determined by the ontology that is given to the system. But an ontology itself does not state how queries should be constructed, so OptiqueVQS uses an algorithm to project the ontology into a *navigation graph*, which is a graph structure that clearly states how the classes and properties in the ontology are allowed to be combined. For example, if the ontology contains the classes *Supplier* and *Location*, and an object property *locatedIn*, then the projection algorithm may, based on, for example, range and domain axioms, conclude that suppliers should be allowed to be connected to locations in queries. The navigation graph will then get an edge with label *locatedIn* from the vertex of *Supplier* to the vertex of *Location*. In general, there are many ways to project an ontology into a navigation graph, which can be used directly by a VQS. But, since the ontology contains classes and properties that the user is familiar with, these should be reused in the navigation graph, and they should be set up in alignment with the axioms in the ontology. The concrete projection algorithm used by OptiqueVQS has been described in earlier work [50, 29].

OptiqueVQS strikes a good balance between usability and expressivity for the type of users it is targeting, which are persons with complex information needs, like engineers in a company, for example. The high usability has been confirmed by user studies done with employees from Statoil and Siemens [54, 52]. These users tend to construct fairly complex queries, with up to 10 different variables, and various filters, and these queries may take minutes, or even hours to run, because the database may need to join data from multiple large sources. This means that there is not enough time to retrieve answers from the dataset during the construction phase, and therefore, OptiqueVQS does not present results after every query change, but only when the query is finished and the user decides to execute it over the data.

OptiqueVQS does not support dead-end detection on the possible query extensions, but this was requested by employees at Statoil and Siemens. In particular, the user studies showed that dead-end detection based on combinations of filters on one single variable, like in faceted search, was not enough. Users explicitly asked for the removal of suggestions that contradicted with filters made on other variables in the query. This was one of the main factors that motivated us to start on the work of this thesis.

The work of this thesis is based on a VQS model that is almost identical to the one used by OptiqueVQS, which we define in Chapter 4 and Chapter 5. We also assume that the VQS targets domain experts, and that it takes too long time to run queries over the underlying dataset for interactive use.

SemFacet SemFacet [1, 3, 4, 2] is a semantic facet-based search system. Like OptiqueVQS, it generates a navigation graph based on an input OWL2 ontology, which is then used to make a facet-based UI. This UI allows the user to construct conjunctive SPARQL queries with filters on the different facets, which can be executed over the corresponding dataset. In contrast to OptiqueVQS, which requires exactly one type per variable, SemFacet can be used to construct queries where the variables are not necessarily typed. SemFacet presents the constructed query as a text-based indented tree. This, together with the results, are always visible to the user, and updated as the user changes the query. This is inspired by faceted search, and in contrast to OptiqueVQS, which only displays the query, but not the results before they are explicitly requested. This approach relies on an underlying database that can consistently return answers to queries fast enough to make the system usable.

Rhizomer Rhizomer [8, 9] is a web application that facilitates publishing and exploration of Semantic Web data. The exploration component allows the user to construct queries with variables belonging to different connected classes, just like OptiqueVQS and SemFacet does. In fact, the functionality of Rhizomer is very similar to the one of SemFacet: it can be used to combine multiple classes, it presents facets and possible filter actions, it updates results after each change to the query, and it allows the user to select which class to focus on.

All of the RDF-based systems presented above need to somehow determine which queries the users should be allowed to make, and they do this by considering either the dataset, the ontology, or both of them. Here we assume that the dataset only contains facts about instances and their properties (also known as the ABox), while the ontology only describes the conceptualization of the relevant domain (also known as the TBox). For example, OptiqueVQS relies only on the given ontology, while PepeSearch will work fine even when just the dataset is available. An ontology declares explicitly all classes and properties that are available in the relevant domain, and it may also contain useful relationships between these classes and properties, like subclass relationships, for example. Ontology-based systems will use the same classes and properties for all datasets, and they will more closely match the actual domain. Data-driven RDF-based systems will also be able to find classes and properties, but only those that are actually used in the given dataset. A dataset alone does not explicitly provide any high-level relationships between the classes, like subclass relationships, for example, but some systems may still try to generate possible class hierarchies from the data. Both ontology-based systems and more data-driven systems have advantages and disadvantages, and both approaches have use-cases.

3.2.3 Information Retrieval Systems

Information retrieval systems are systems that attempt to find relevant information in unstructured data. They differ from systems that construct formal queries over structured databases, in the sense that it is not absolutely

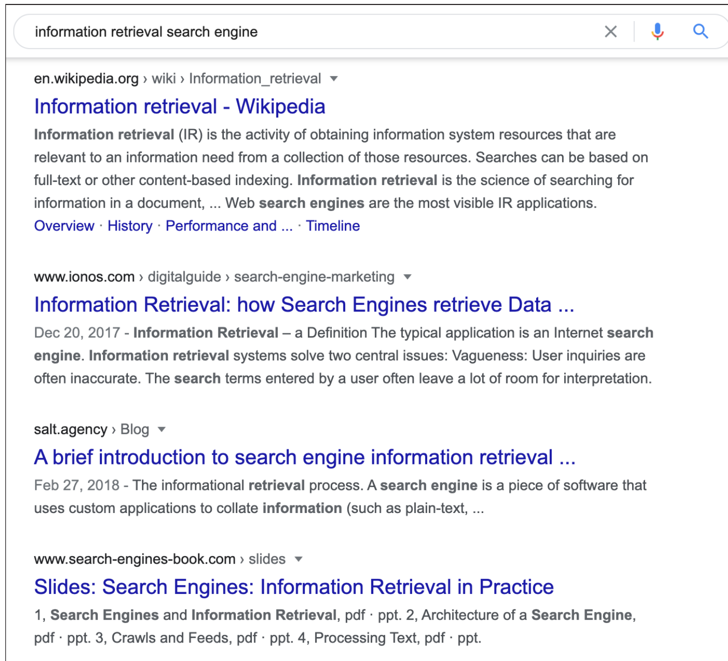


Figure 3.4: Google search engine results for the query “information retrieval search engine”, ranked by relevance.

clear anymore when a particular document, or part of a document is a match of a query, i.e., information retrieval systems work over a *probabilistic setting*. We will consider two types of information retrieval systems: *search engines* and *question answering systems*.

Search Engines In general, the term *search engine* refers to any information retrieval system that helps users to find relevant information in unstructured data, but here we will specifically consider systems that search in collections of text documents, and where queries are represented by a sequence of keywords. A well-known example of such a system is Google Search,[20] which is a *Web search engine*, i.e., a search engine that finds information in documents published on the World Wide Web. Figure 3.4 shows the interface of Google Search. It contains a search bar at the top, where the query, i.e., the sequence of keywords can be entered, and a list of resulting documents sorted by relevance below.

In the simple case where the user is just looking for one, or maybe a few documents of high relevance, and a query to express the need is simple, search engines are very convenient: they do not require the user to know any formal query language, nor do they limit the vocabulary of the user. Furthermore,

[20] https://www.google.com/

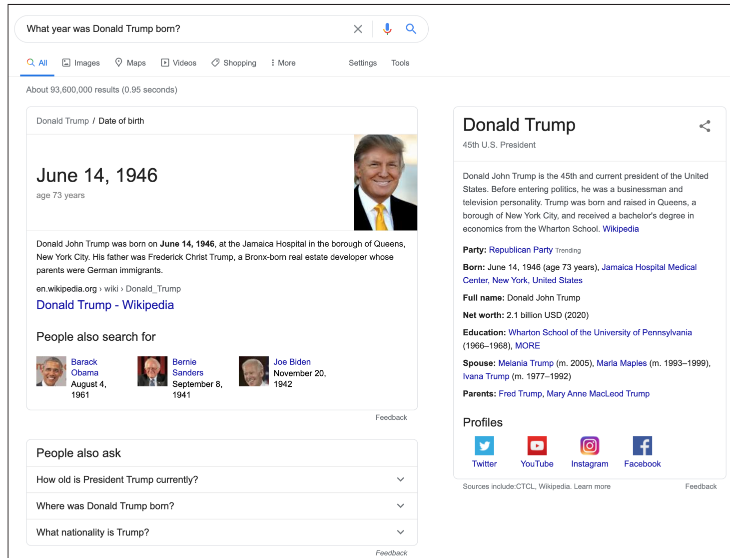


Figure 3.5: The QA feature of Google Search answering a factual question.

by presenting a ranked list of multiple documents, which the user will have to manually pick from in the end, the system will have a good chance of finding a document the user is satisfied with. On the other hand, if the user has a more complex information need, which requires more structure than the available documents have, then search engines will not be very useful. For example, if the user only wants the documents mentioning children of persons who have been the head of state of a country outside Europe, then a search engine with a keyword-based query language will not be of any help. This is the kind of trade-off between expressivity and usability we mentioned in Section 3.1.

Question Answering Systems *Question Answering Systems* (QA systems) are systems that attempt to give concrete answers to questions presented by users in natural language. Some well-known examples of such systems are IBMs Watson, which is famous for being the first computer program to beat humans in the game of Jeopardy!, and Wolfram Alpha,[21] which excels at finding answers based on concrete facts. QA is also a key feature of popular voice assistants like Siri, Cortana, and Google Assistant. [19]. It is also worth mentioning PowerAqua [34], which is a QA system that uses semantic data to answer questions. Google Search, which is primarily a search engine, is also able to answer simple factual questions (see Figure 3.5).

There are two main approaches to QA, depending on which kind of data source is available to the system. If the system has access to a knowledge base,

[21] https://www.wolframalpha.com/

like Wikidata[22] or YAGO,[23] for example, then the only challenging part is to find a formal query that matches the question. This requires techniques from both natural language processing (NLP) and knowledge representation (KR). On the other hand, if the system only has access to a collection of natural language documents, where the answer has to be found, it must instead use a combination of techniques from information retrieval (IR) and NLP, similar to the techniques used by search engines. If the system has access to both types of sources, it is also possible to make hybrid systems. They will usually perform better than systems that only consider one type of source [63].

QA systems are easy to use and require no training because the users can simply pose their questions in natural language. However, natural language is not as precise as formal query languages, and even if the question is well-formulated, the system may still not be able to answer it precisely every time, because this requires reasoning and full understanding of both natural language and the relevant domain. Question answering has for a long time been, and is still, one of the most challenging tasks in the NLP research field.

3.3 Dead-End Detection

In everyday language, the term dead-end refers to paths that do not lead any further, and hence cannot lead to the goal. Following a dead-end is never productive, because the only way out is back the way one came from. A dead-end in the context of computer systems has a similar meaning: it refers to a state or an action that leads to a state, which is not the goal state, and which only leads back to the state one just came from. In the data access setting, where the user is searching for information in a dataset, we can be even more concrete: a dead-end is an action that leads to a situation where no results are returned to the user. This happens when the query that produces results becomes too restrictive.

For example, in faceted search systems, the results are produced by the query defined by the active filters. The user starts a session without any filters, which usually returns too many objects, and a natural path towards relevant results involves the addition of one or more filters. Filters restrict the query and reduce the number of results, and if a filter limits the query down to no results, it is by our definition considered to be a dead-end. When this happens, the user must remove at least one of the added filters before results will be presented again.

Dead-end detection is then the task of finding every dead-end action, i.e., every action that leads to a query without answers. Such actions are not productive, so it is common to either label them, to warn the user, or to disable them entirely, such that the user cannot even select them.

The results returned by dead-end queries are empty, and hence useless in themselves. Without dead-end detection, users cannot know for sure whether a query is going to return results or not, and they may construct queries that

[22] https://www.wikidata.org/
[23] http://yago.r2.enst.fr/

are only revealed to be dead-end queries after they have been made. Dead-end detection is useful in these cases because the system can then tell the user about dead-end queries right before the user creates them, and this completely eliminates the reason to make the query – there is no need for the user to construct a query without results when they already know that it will return no results. This is similar to how dead-end signs tell drivers about dead-end roads in real life. With lack of a better word to describe actions and queries that are not dead-ends, we are going to use the term *productive*. Conversely, we may use the term *unproductive* when we refer to queries or actions that are in fact dead-ends.

Perfect Dead-End Detection The most straightforward way for a system to check if an action is a dead-end is to construct its resulting query in the background, and run it over the dataset to see if it returns any results. This requires one query for each possible action provided in the given situation.

Often, detection can be done more efficiently by using the answers of one single query to detect dead-ends among many actions. For example, in faceted search, it is possible to extend the current query with a single variable corresponding to a given facet, and if this variable is without filters, the resulting query will return the set of values that should be presented as possible filters on this facet. This trick may reduce the overall time it takes to compute dead-ends, but it does not eliminate the need to run at least one query over the underlying database.

Since the system's future actions depend on the answers of these extended queries, it is important that the underlying database is able to return answers fast enough. A user may accept to wait for a few seconds, but ideally, the system should be able to detect dead-ends in less than 200ms, which is close to what humans consider to be instant [36].

The time required to answer these extended queries depends on the size and structure of the underlying database, and the complexity of the extended query. But the extended query is just a small modification of the current, active query, which is bounded by the complexity of the query language of the systems. So in general, the problem of detecting dead-ends becomes harder as the expressivity of the supported query language increases. If the database is small, and/or the expressivity of the system is simple enough, it should be unproblematic to get answers within an acceptable time frame. But if this is not the case, then it will be necessary to turn to other solutions, like an index that is optimized for answering these kinds of extended queries. The size of such an index will again depend on the size of the underlying dataset and the expressivity of the system: an index over a large database and a system with high expressivity will lead to a large index. In particular, if the expressivity of the queries is not bounded in size, then the corresponding index will, in general, have to be infinitely large to cover all possible queries.

Dead-End Detection in Existing Systems Now that we have established what dead-end detection is, we will re-visit the different systems we presented in Section 3.1,

to see if they feature dead-end detection, and how they are able to do it.

Faceted Search Systems often include dead-end detection on the available filters, and this can be implemented efficiently over large datasets by using state-of-the-art search engines. As we have already established, these solutions ensure efficient dead-end detection over millions of objects, but they are limited to one class and its corresponding properties/facets, i.e., star-shaped queries.

RDF Surveyor allows the user to select which class to browse, and then it shows all instances of this class that exist in the dataset. The system does not provide any filtering opportunities beyond that, so dead-end detection in this context would be to prevent the user from selecting classes without instances. RDF Surveyor ranks the available classes such that classes with the most instances are presented first, but it does not label or remove empty classes, i.e., it does not feature any form of dead-end detection.

PepeSearch does not include any dead-end detection features, and allows the user to fill out the form in a way that returns no answers. But given the limitation on the queries it allows: a single class, its connected classes, and all their properties, it would be possible to implement dead-end detection efficiently by using a finite index. The key insight here is to consider each property of a connected class to be a facet of the main class. By doing this, we have translated the PepeSearch query into a star-shaped query similar to those supported by faceted search systems. This means that PepeSearch can use SOLR, or another search engine that supports faceted search, to implement efficient dead-end detection over all the included properties.

The three remaining RDF-based systems described in Section 3.1 all have different approaches to dead-end detection. **OptiqueVQS** suggests a list of values for each property, but this list is not adaptive, it just presents the static list of every value in the dataset that corresponds to the given property. This way of suggesting values is very easy to implement and does not require much effort, but it does not prevent dead-ends, of course. Both SemFacet and Rhizomer support dead-end detection over multiple multiple classes, which is challenging because the queries can be arbitrarily large. **SemFacet** is still able to achieve reasonable results performance, but it does so by using RDFOx [40] as the underlying database. RDFox is an in-memory, scalable RDF store, which boosts the performance of the whole system, including dead-end detection. However, this solution will struggle in the scenario we consider, were complex queries with up to 10 variables are posed over very large datasets. **Rhizomer** on the other hand, only provides data value suggestions for a given property after it has been specifically requested by the user. This prevents the system from spending time on dead-end detection right after each query change, but it still does not make the process efficient, and Rhizomer also struggles as the size of the queries and data increases.

Dead-end detection is a less common feature in **search engines** and **QA systems**, and there are several reasons for this. The first one is that the query construction process is not very iterative: often the query is formulated by the user without any breaks, hence, there is no time for the system to tell the user about dead-ends. This is especially the case for QA-systems with

Figure 3.6: Google auto-completing a user's question.

voiced-based interfaces. If we consider QA-systems and search engines with text-based interfaces, and assume that the user may take breaks during the query construction process, then there is room for the system to provide feedback to the keywords and letters written by the user so far. Google Search already does this with its drop-down list of suggestions on how to complete the query (see Figure 3.6), and a similar kind of list could be presented also for QA systems. But, since these systems work over a probabilistic setting, it is not clear to neither the system nor the user what a dead-end really is. The system has a way of calculating the relevance of items, and it is possible to define dead-ends based on this. For example, the system can define a dead-end query to be one that does not return any items with relevance higher than 0.5. This approach is still problematic, since the relevance score set by the system likely differs from the relevance perceived by the user, and since it is hard to set a relevance threshold that fits all users. And, even if dead-ends are removed, the vocabulary of the domain these systems cover is often so big that the number of remaining suggestions exceeds what it is sensible to present to the user. In practice, this is solved by only showing the top-k most relevant query extension suggestions, but this is quite different from the type of dead-end detection we are trying to achieve in this thesis.

Approximating Dead-end Detection As already indicated, we will be working with a model that is very similar to the one that OptiqueVQS uses. I.e., we will try

to detect dead-end extensions of arbitrary large tree-shaped queries where every variable is typed exactly once, and where filters are allowed on data properties.

Since there is no way to ensure both correct and efficient dead-end detection with a finite index over this type of query, we are going to consider a way of approximating it instead. The idea is to define an index that stores precomputed answers of queries that use the most central classes and properties of the ontology or navigation graph. If the query the system needs to execute in order to detect dead-ends is covered by this index, then it can retrieve answers efficiently directly from the index, and use the answers to find and present all dead-ends. But, if the query is not covered by the index, then the system needs to prune it until the remaining part can be answered by the index. When the system has to prune the query, it may fail to detect some dead-ends, but, by selecting what to index in a clever way, the approximation will give very good results.

In the following chapters we will present the formal VQS model we use, and the type of queries we target, in addition to the framework that allows us to calculate most dead-ends in the way we just described.

Chapter 4

Ontology-Based Visual Query Systems

In the previous chapter, we discussed a variety of different data access systems and how dead-end detection is done in each of them. In particular, we presented the VQS model we will be using in this thesis project, which is based on OptiqueVQS, and then we established that it is impossible to achieve perfect, efficient, and space-efficient dead-end detection, since this model allows arbitrary large queries of multiple related classes.

In this chapter, we present this VQS model formally. This requires formal descriptions of the three most central parts of the VQS, which are:

- Tree-shaped queries $\mathcal{Q}_1, \mathcal{Q}_2, \ldots, \mathcal{Q}_k$, all with one type per variable, and filters only on data properties.
- A navigation graph $\mathcal{N}$, which dictates how classes and properties can be combined in these queries.
- A dataset $\mathcal{D}$, which the queries will be executed over.

We assume that both $\mathcal{N}$, and $\mathcal{D}$ are static, and that the same versions of $\mathcal{N}$ and $\mathcal{D}$ will be used over multiple *sessions*, where a session is defined to be a single user's process of constructing a query, and then executing it over the dataset $\mathcal{D}$. In fact, the index we present in Chapter 6 needs to be rebuilt every time either $\mathcal{N}$ or $\mathcal{D}$ changes, which means that our approach is going to be less useful unless they are both fixed over a long period of time. We will mostly be focusing on one single session, and within this session, it is safe to consider both $\mathcal{N}$ and $\mathcal{D}$ to be fixed.

A single session starts with an empty query, and the user can modify it by selecting between available actions (see Section 5.1). Each action results in a new version of the query, and this kind of transition can be made multiple times until the user is finally satisfied with it. At this point, they should run the query over the dataset $\mathcal{D}$, and wait for the results to be returned by the system.

In other words, a single session receives a fixed navigation graph $\mathcal{N}$, and a fixed dataset $\mathcal{D}$, and produces a sequence of queries $\mathcal{Q}_0, \mathcal{Q}_1, \ldots, \mathcal{Q}_k$, where $\mathcal{Q}_0$ is the empty query, and $\mathcal{Q}_k$ is the final query. Within this session, we are going to focus mostly on the period of time right after a transition from $\mathcal{Q}_{i-1}$ to $\mathcal{Q}_i$ $(0 < i \leq k)$, when the system needs to detect dead-ends among the possible actions. In this context, we will only consider the current version of the query, $\mathcal{Q}_i$, hence, we will just ignore the subscript index and refer to it as the *partial query* $\mathcal{Q}$.

4.1 Resource Graphs

In Section 2.1.1 we saw how RDF data can be represented as a graph consisting of two types of vertices: instances and literals. This same partitioning can also be made in the queries supported by our system, where each variable is associated with either a class or a datatype, and it will exist in the navigation graph we use, where the same classes and datatypes define two types of vertices. In other words, if we define an appropriate graph structure where the vertices are split into two parts, we can use it to define the core parts of $\mathcal{N}$, $\mathcal{D}$, and $\mathcal{Q}$.

In this section, we describe *resource graphs*, which is the kind of graph structure we are going to use. As already indicated, these graphs allow two types of vertices, which we call *object vertices* and *data vertices*. Resource graphs also have labeled, directed edges. Each resource graph must be defined over a universe, which is a triple $(\Gamma_o, \Gamma_d, \Gamma_l)$ that declares all available object vertices Γ_o, data vertices Γ_d, and edge labels Γ_l respectively.

Definition 4.1.1 (Universe). A *universe* is a triple $(\Gamma_o, \Gamma_d, \Gamma_l)$, where Γ_o, Γ_d, and Γ_l are pairwise disjoint sets, and where each element $l \in \Gamma_l$ has a unique inverse, denoted $l^{-1} \in \Gamma_l$, which has l itself as inverse, i.e., $(l^{-1})^{-1} = l$. $\dashv$

The requirement that every label must have an inverse, allows us to define *inverse edges* later (see Definition 4.1.8).

Definition 4.1.2 (Resource Graph). A *Resource Graph* G over a universe $(\Gamma_o, \Gamma_d, \Gamma_l)$, is a triple $G = (V_o, V_d, E)$, where $V_o \subseteq \Gamma_o$ is the set of *object vertices*, $V_d \subseteq \Gamma_d$ is the set of *data vertices*, and $E \subseteq V_o \times \Gamma_l \times (V_o \cup V_d)$ is the set of *edges* included in the graph. If both V_o and V_d are finite sets, then G is a *finite* resource graph. $\dashv$

Given a resource graph $G = (V_o, V_d, E)$, we will frequently need to access its three elements V_o, V_d, and E, in addition to the full set of vertices $V_o \cup V_d$. Below, we define four functions that simplifies the process of accessing each of them.

Definition 4.1.3. Given a resource graph $G = (V_o, V_d, E)$, we define the following four functions to return all vertices, all object vertices, all data vertices, and all edges respectively:

$$
\begin{aligned}
&\text{All vertices:} && \bar{V}(G) = V_o \cup V_d \\
&\text{Object vertices:} && \bar{V_o}(G) = V_o \\
&\text{Data vertices:} && \bar{V_d}(G) = V_d \\
&\text{Edges:} && \bar{E}(G) = E
\end{aligned}
$$

$\dashv$

Example 4.1.4. Before a concrete resource graph can be constructed, we need to first declare a universe, i.e., sets of available object vertices, data vertices, and labels:

$$\Gamma_o = \{Person, Country\}$$

$$\begin{aligned}\Gamma_d &= \{Integer, String\}\\ \Gamma_l &= \{visited, visitedBy, knows, borders, age, age^{-1},\\ &\quad population, population^{-1}, name, name^{-1}\}\end{aligned}$$

where

$$\begin{aligned}visited^{-1} &= visitedBy\\ knows^{-1} &= knows\\ borders^{-1} &= borders\end{aligned}$$

Now, let $G = (V_o, V_d, E)$ be the resource graph over $(\Gamma_o, \Gamma_d, \Gamma_l)$ where

$$\begin{aligned}V_o &= \Gamma_o = \{Person, Country\}\\ V_d &= \Gamma_d = \{Integer, String\}\\ E &= \{(Person, visited, Country),\\ &\quad (Country, visitedBy, Person),\\ &\quad (Person, knows, Person),\\ &\quad (Country, borders, Country),\\ &\quad (Person, age, Integer),\\ &\quad (Person, name, String),\\ &\quad (Country, population, Integer),\\ &\quad (Country, name, String)\}\end{aligned}$$

This resource graph contains two object vertices given by $\bar{V}_o(G) = V_o = \{Person, Country\}$, two data vertices given by $\bar{V}_d(G) = V_d = \{Integer, String\}$, and eight edges given by $\bar{E}(G) = E$. In total it has four vertices, given by $\bar{V}(G) = V_o \cup V_d = \{Person, Country, Integer, String\}$. Figure 4.1 shows a visual representation of G, where blue circles are object vertices from V_o, yellow rectangles are data vertices from V_d, and the labeled, directed edges are the edges in E. ♦

If we consider the resource graph from Example 4.1.4, we see that each edge starts in an object vertex. This is a property of resource graphs that follows directly from Definition 4.1.2. Even though the universe defined in Example 4.1.4 is quite minimalistic, there are still some unused labels of Γ_l. For example, the label *age* is used in the resource graph, while age^{-1} is not. Definition 4.1.2 only states that used components have to exist in the universe, so our example is still valid. It is also worth noting that the inverse of each property is included in Γ_l, which is a requirement (see Definition 4.1.1).

When visualizing resource graphs and similar structures in this thesis, we will use the same color codes as the graph in Figure 4.1, where object vertices are represented by blue circles, and data vertices are represented by yellow rectangles.

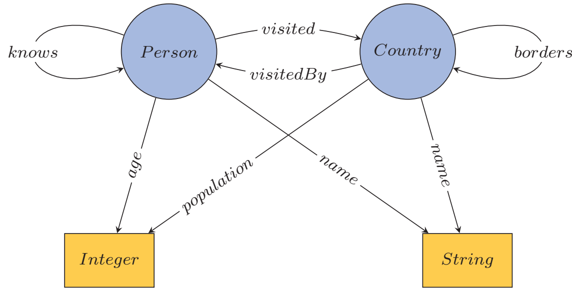


Figure 4.1: The resource graph defined in Example 4.1.4.

Resource graphs are not really graphs, according to the standard mathematical definition. However, we can transform any resource graph into a labeled, directed graph by merging V_o and V_d into one set of vertices, which gives us the graph $(V_o \cup V_d, E)$. The reader should keep this possible transformation in mind, and be aware of the fact that many of the following definitions about resource graphs are just modified versions of corresponding definitions related to standard, labeled, directed graphs. Many of the properties of labeled, directed graphs are therefore also going to apply to resource graphs.

Edges So far the edges we have considered have been limited to connections going out from object vertices in a resource graph. But we need to consider edges in a more general sense, so now, we extend the scope of edges to cover any connection between two vertices from $\Gamma_o \cup \Gamma_d$ with a label from Γ_l.

Definition 4.1.5 (Edges in Universe)**.** The set of all edges in the universe $(\Gamma_o, \Gamma_d, \Gamma_l)$ is given by

$$(\Gamma_o \cup \Gamma_d) \times \Gamma_l \times (\Gamma_o \cup \Gamma_d)$$

⊣

Each of these edges is modeled like edges in labeled, directed graphs, where the first, second, and third element refers to the *source*, *label*, and *target* of the edge respectively. Given an edge, we need a quick way to access each of these three parts, and below we introduce three functions that help us with this.

Definition 4.1.6 (Source, Label, and Target of Edge)**.** Let $(\Gamma_o, \Gamma_d, \Gamma_l)$ be a universe, and let $e = (v, l, v')$ be an edge in this universe. The three elements of this edge: v, l, and v' are called the *source*, *label*, and *target* of e respectively. We can

access each of them directly from e by using their corresponding function below:

$$\begin{aligned} \text{Source}: &\quad \mathsf{src}(e) = v \\ \text{Label}: &\quad \mathsf{lab}(e) = l \\ \text{Target}: &\quad \mathsf{tar}(e) = v' \end{aligned}$$

⊣

Two edges are considered equal if their corresponding triples are identical, i.e., if they have the same source, label, and target. This allows us to construct graphs with interesting arrangements like e.g. multiple edges with the same label but different source or target, like the two edges labeled by *name* in the graph presented in Figure 4.1, or multiple edges between the same pair of vertices, where the labels differ.

Similar to how the vertices are split into object vertices and data vertices, we can also split the edges into two parts, *object edges* and *data edges*, based on the type of their targets.

Definition 4.1.7 (Object Edges and Data Edges)**.** Let $(\Gamma_o, \Gamma_d, \Gamma_l)$ be a universe, and let e be an edge in this universe. If $\mathsf{tar}(e) \in \Gamma_o$, then e is called an *object edge*, and if $\mathsf{tar}(e) \in \Gamma_d$, then e is called a *data edge*. Given a resource graph G over $(\Gamma_o, \Gamma_d, \Gamma_l)$, we define $\bar{E}_o(G)$ and $\bar{E}_d(G)$ to be the object edges and data edges of G respectively. ⊣

Given an edge, we define its inverse to be the edge with the inverse label, pointing in the opposite direction. Such an edge can always be constructed since the universe requires that each label in Γ_l actually has an inverse (see Definition 4.1.1).

Definition 4.1.8 (Edge Inverse)**.** Let $(\Gamma_o, \Gamma_d, \Gamma_l)$ be a universe, and let $e = (v_s, l, v_t) \in (\Gamma_o \cup \Gamma_d) \times \Gamma_l \times (\Gamma_o \cup \Gamma_d)$ be an edge in this universe. The inverse of e, denoted e^{-1}, is defined as $e^{-1} = (v_t, l^{-1}, v_s)$. ⊣

Assume that we have a resource graph G over $(\Gamma_o, \Gamma_d, \Gamma_l)$. It is possible to construct the inverse e^{-1} of every edge $e \in \bar{E}(G)$, but this inverse will not necessarily be a part of G itself. If e is a data edge, for example, then its inverse will start in a data vertex, which means that it cannot be included in any resource graph.

Example 4.1.9. Let us re-visit the resource graph from Example 4.1.4. Three of the edges in this graph are:

$$\begin{aligned} e_1 &= (Person, visited, Country) \\ e_2 &= (Country, visitedBy, Person) \\ e_3 &= (Country, name, String) \end{aligned}$$

The source, label, and target of e_1 is given by

$$\begin{aligned}\mathsf{src}(e_1) &= Person\\ \mathsf{lab}(e_1) &= visited\\ \mathsf{tar}(e_1) &= Country\end{aligned}$$

Since *visited* is the inverse of *visitedBy*, i.e., $visited^{-1} = visitedBy$, then we get

$$\begin{aligned}e_1^{-1} &= (Person, visited, Country)^{-1}\\ &= (Country, visited^{-1}, Person)\\ &= (Country, visitedBy, Person)\\ &= e_2\end{aligned}$$

I.e., e_1 and e_2 are inverses of each other. By using the definition of edge inverses, we can construct the inverse of e_3:

$$e_3^{-1} = (String, name^{-1}, Country)$$

This is a proper edge over the given universe, but it can never be included in G, since its source is a data vertex. ♦

Although an edge e is not the same as its inverse e^{-1}, they will be considered equivalent in certain settings, like when they are used in queries. For example, querying for all persons that have visited a country, should give the same results as querying for all countries visited by a person. It is just the same query flipped.

Subgraphs A *subgraph* of a resource graph is just another resource graph where only a subset of the vertices and edges from the original graph is included.

Definition 4.1.10 (Subgraph). Let $G = (V_o, V_d, E)$ be a resource graph. A resource graph $G' = (V'_o, V'_d, E')$ is a subgraph of G if $V'_o \subseteq V_o$, $V'_d \subseteq V_d$, and $E' \subseteq E$. $\dashv$

Homomorphisms and Isomorphisms In Section 2.2.2, we formally defined homomorphisms between two labeled, directed graphs. Now we will give a similar homomorphism definition for resource graphs.

Definition 4.1.11 (Resource Graph Homomorphism). Let G and G' be two resource graphs over $(\Gamma_o, \Gamma_d, \Gamma_l)$ and $(\Gamma'_o, \Gamma'_d, \Gamma_l)$ respectively. A function $f\colon \bar{V}(G) \to \bar{V}(G')$ is called a *homomorphism* from G to G' if all the following properties are satisfied:

$$f(v) \in \bar{V}_o(G') \text{ for all } v \in \bar{V}_o(G) \tag{4.1}$$

$$f(v) \in \bar{V}_d(G') \text{ for all } v \in \bar{V}_d(G) \tag{4.2}$$

$$(v, l, v') \in \bar{E}(G) \implies e \in \bar{E}(G') \text{ or } e^{-1} \in \bar{E}(G') \text{ where } e = (f(v), l, f(v')) \tag{4.3}$$

$\dashv$

Equations 4.1 and 4.2 in the definition above states that f has to map data vertices in G to data vertices G', and object vertices in G to object vertices in G'. Equation 4.3 formulates the homomorphism requirement. Notice that Equation 4.3 accepts both e and e^{-1} in the target graph. Which is another way of saying that homomorphisms consider edges and their inverses to be equivalent. An *isomorphism* in this context is now a homomorphism whose inverse is also a homomorphism.

Definition 4.1.12 (Resource Graph Isomorphism)**.** Let G and G' be two resource graphs. A bijective function $f\colon \bar{V}(G) \to \bar{V}(G')$ is called an *isomorphism* between G and G' if both f and f^{-1} are homomorphisms. If f is an isomorphism between G and G', then G and G' are said to be *isomorphic*. ⊣

Resource Trees The queries supported by our system are tree-shaped, and they will be based on tree-shaped resource graphs. It is not obvious what the term tree-shaped actually means in the context of resource graphs, so now we will describe it formally. This requires formal definitions of *walks* and *paths* in resource graphs.

Definition 4.1.13 (Walk)**.** Let G be a resource graph. A walk from a vertex v to a vertex v' is a sequence of vertices $v_0, v_1, \ldots, v_n \in \bar{V}(\mathcal{Q})$ and a sequence of edges $e_0, e_1, \ldots, e_{n-1} \in \bar{E}(G)$ that satisfies the following three conditions:

$$v_0 = v \tag{4.4}$$
$$v_n = v' \tag{4.5}$$
$$v_i = \mathsf{src}(e_i) \text{ and } v_{i+1} = \mathsf{tar}(e_i) \text{ for all } i = 0, 1, \ldots, n-1 \tag{4.6}$$

A walk is called *direction-ignorant* if it ignores the direction of the edges. A walk becomes direction-ignorant if we replace Equation 4.6 above with

$$\begin{gathered} v_i = \mathsf{src}(e_i) \text{ and } v_{i+1} = \mathsf{tar}(e_i) \\ \text{or} \\ v_i = \mathsf{tar}(e_i) \text{ and } v_{i+1} = \mathsf{src}(e_i) \\ \text{for all } i = 0, 1, \ldots, n-1 \end{gathered}$$

⊣

Intuitively, a walk is given by process of starting in a vertex and transition along the direction of the edges in the graph. A direction-ignorant walk is the same thing, except that it allows transitions along edges in reversed direction. Notice that walks can both re-visit vertices multiple times, and transition along the same edge multiple times. Notice also that it is possible to construct a walk without any edges. This is then just a walk from a single vertex to itself. Now, we define a path to be a walk where edges are not repeated.

Definition 4.1.14 (Path)**.** A *path* is a walk where the sequence of visited edges contains no duplicates. A *direction-ignorant path* is a direction-ignorant walk where the sequence of visited edges contains no duplicates or inverses. ⊣

Now we can use direction-ignorant paths to formally define resource trees.

Definition 4.1.15 (Resource Tree)**.** A *resource tree* is a resource graph R with exactly one unique direction-ignorant path between each pair of vertices. $\dashv$

If there exists exactly one vertex $v_r \in \bar{V}(R)$, called the root of R, that can reach all other vertices in R when walking along the edges, then we call R a *rooted resource trees.*

Definition 4.1.16 (Rooted Resource Tree)**.** A *rooted resource tree* is a resource tree R that contains exactly one vertex, $v_r = \mathsf{root}(R)$, called the *root* of R, such that there is a unique path from v_r to every vertex in $\bar{V}(R)$. $\dashv$

In a rooted resource tree, every vertex except for the root has exactly one incoming edge. We define the source of this edge to be the *parent* of the vertex.

Definition 4.1.17 (Parent of Vertex)**.** Let R be a rooted resource tree, and let $v \in \bar{V}(R) \setminus \{\mathsf{root}(R)\}$ be a vertex other than the root. We define the *parent* of v, denoted $\mathsf{parent}(v)$ to be the unique vertex v_p such that $(v_p, l, v) \in \bar{E}(R)$ for some label l. $\dashv$

We also need to refer to all the *children* of a given vertex.

Definition 4.1.18 (Children of Vertex)**.** Let R be a rooted resource tree, and let $v \in \bar{V}(R)$ be a vertex in R. We define the *children* of v, denoted $\mathsf{children}(v)$ to be the set of all vertices v_c such that $\mathsf{parent}(v_c) = v$. $\dashv$

Since resource trees are a special case of resource graphs, all the definitions we have presented related to resource graphs, will also be applicable to resource trees.

4.2 Main Structures

Before we can formally define the navigation graph $\mathcal{N}$, the dataset $\mathcal{D}$, and the partial query $\mathcal{Q}$, we have to specify the sets of components to be used in these structures. We already know that $\mathcal{N}$, $\mathcal{D}$, and $\mathcal{Q}$ will be based on resource graphs, and the components we now present are important because they will be used to define the universes that these resource graphs are defined over.

More specifically, we are going to assume that there exists a set of *classes* Γ_c, a set of *datatypes* Γ_d, a set of *instances* Γ_i, a set of *data values* Γ_v, a set of *object variables* Γ_{ov}, a set of *data variables* Γ_{dv}, and a set of *properties* Γ_p.

The navigation graph $\mathcal{N}$ is a specific type of resource graph over the universe $(\Gamma_c, \Gamma_d, \Gamma_p)$. In other words, $\mathcal{N}$ is a resource graph where the object vertices are classes, the data vertices are datatypes, and the edges are labeled with properties. When we refer to something that is either a class or a datatype, i.e., a general element in $\Gamma_c \cup \Gamma_d$, we use the term *type*.

The most essential part of the dataset $\mathcal{D}$ is a resource graph $G_{\mathcal{D}}$ over the universe $(\Gamma_i, \Gamma_v, \Gamma_p)$, called the *data graph* of $\mathcal{D}$. In this graph, the object vertices are instances, the data vertices are data values, while the edges are labeled with

Term	Sub Term	Symbol	Used in
Properties	Properties	Γ_p	$\mathcal{N}$, $\mathcal{D}$, $\mathcal{Q}$
Types	Classes	Γ_c	$\mathcal{N}$
	Datatypes	Γ_d	$\mathcal{N}$
Entities	Instances	Γ_i	$\mathcal{D}$
	Data values	Γ_v	$\mathcal{D}$
Variables	Object variables	Γ_{ov}	$\mathcal{Q}$
	Data variables	Γ_{dv}	$\mathcal{Q}$

Table 4.1: Summary of the sets of components needed to model $\mathcal{N}$, $\mathcal{D}$, and $\mathcal{Q}$.

properties. When we refer to something that is either an instance or a data value, i.e., a general element in $\Gamma_i \cup \Gamma_v$, we use the term *entity*.

Finally, the core of a query $\mathcal{Q}$ is a resource tree $R_\mathcal{Q}$ over the universe $(\Gamma_{ov}, \Gamma_{dv}, \Gamma_p)$, called the *query tree* of $\mathcal{Q}$. In this tree, the object vertices are object variables, data vertices are data variables, and the edges are labeled with properties. When we refer to a general element of $\Gamma_{ov} \cup \Gamma_{dv}$, we use the term *variable*.

An overview of all seven component sets is presented in Table 4.1. Components cannot belong to more than one of these sets at the same time, i.e., we assume that all the seven sets are pairwise disjoint. This means that we can determine the type of a component by checking which set it belongs to. Furthermore, since the properties in Γ_p will be used as labels in all the resource graphs we consider, it is required that each property has an inverse.

As long as all the seven component sets are pairwise disjoint and every property has an inverse, we do not put any other limitations on the components. However, in order to make a useful system, it is important that the user recognizes classes, datatypes, and properties from the domain. In practice, this means that every class, datatype, and property from the domain should be included in Γ_c, Γ_d, and Γ_p respectively. Similarly, in order to make a precise model of the RDF dataset, it is important that every class, datatype, property, instance, and data value is available in their corresponding component set.

Example 4.2.1. These seven sets, combined with the inverse relationships defined below, defines a valid collection of component sets.

$$\begin{aligned}
\Gamma_p &= \{visited, visitedBy, knows, borders, name, age, population, \dots\} \\
\Gamma_c &= \{Person, Country, Actor, \dots\} \\
\Gamma_d &= \{Integer, String, Double, Boolean, \dots\} \\
\Gamma_i &= \{P1, P2, P3, P4, \dots, C1, C2, \dots\} \\
\Gamma_v &= \text{every integer, string, double, datetime etc.} \\
\Gamma_{ov} &= \{?o_1, ?o'_1, ?o_2, ?o'_2, ?o_3, ?o'_3, \dots\} \\
\Gamma_{dv} &= \{?d_1, ?d'_1, ?d_2, ?d'_2, ?d_3, ?d'_3, \dots\}
\end{aligned}$$

$$visited^{-1} = visitedBy$$
$$knows^{-1} = knows$$
$$borders^{-1} = borders$$
$$\dots$$

These sets are all pairwise disjoint, and every property has an inverse. Both *knows* and *borders* are their own inverses. ♦

In all the examples later in this thesis where a set of components is needed, we will use the components from Example 4.2.1.

4.2.1 The Navigation Graph

The navigation graph is a structure that defines how classes, datatypes, and properties can be combined into queries in our system. We model it as a resource graph $\mathcal{N}$, where object vertices are classes from Γ_c, data vertices are datatypes from Γ_d, and edges are labeled with properties from Γ_p. In addition, we require it to be finite, and that the inverse of each object edge is included.

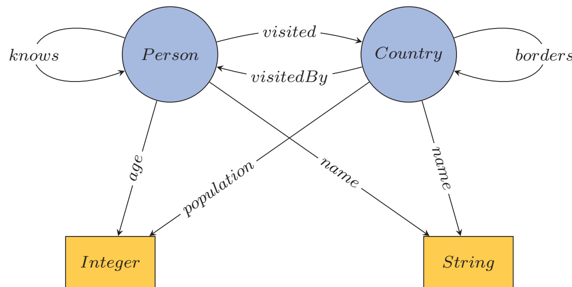
Definition 4.2.2 (Navigation Graph)**.** A *Navigation Graph* is a finite resource graph $\mathcal{N}$ over $(\Gamma_c, \Gamma_d, \Gamma_p)$, where the inverse of each object edge is included. I.e.,

$$e \in \bar{E}_o(\mathcal{N}) \implies e^{-1} \in \bar{E}_o(\mathcal{N})$$

⊣

Each edge (t, p, t') in $\mathcal{N}$ where t' is an object vertex, is called an object edge, and we will use the term *object property* when we refer to the property p of this edge. Similarly, we will use the term *data property* when we refer to properties used in the data edges of $\mathcal{N}$.

Example 4.2.3. Given Γ_c, Γ_d, and Γ_p from Example 4.2.1, the figure below depicts a valid navigation graph $\mathcal{N}$. Notice that this is exactly the same resource graph as we presented in Example 4.1.4.



$\mathcal{N}$ has two classes, given by $\bar{V}_o(\mathcal{N}) = \{Person, Country\}$, and two datatypes, given by $\bar{V}_d(\mathcal{N}) = \{Integer, String\}$. Furthermore, it has a total of eight edges. The properties *age*, *population*, and *name* are all data properties since the targets of their edges are all datatypes. The properties *knows*, *visited*, *visitedBy*, and *borders* are all object properties. ♦

This is the only navigation graph we will present in this thesis, and it will be used in every example throughout this thesis, where a navigation graph is needed.

The purpose of the navigation graph is to have a structure that clearly states which types the user is allowed to include in their query, and how these types can be connected by properties. More precisely, a property between two types in $\mathcal{N}$ indicates that the same property can be used to connect variables of the two types in the query. For example, if one constructs a query over the navigation graph from Example 4.2.3, then *visited* is the only allowed property from a variable of type *Person* to a variable of type *Country*.

The reason why we require each object edge to have an inverse is that this allows users to navigate in both directions between classes when they construct queries. I.e., if a user wants all persons and the countries each of them has visited, they may either start with Person and connect it to Country, or start with Country and connect it to Person. Why this is only needed on object edges, and not data edges, becomes clear when we define queries in Section 4.2.3, and query extensions in Chapter 5.

4.2.2 The Dataset

To model the dataset $\mathcal{D}$, we need both a *data graph* $G_{\mathcal{D}}$, and a *typing function* $T_{\mathcal{D}}$. The data graph $G_{\mathcal{D}}$ specifies which instances from Γ_i and data values from Γ_v that are included in the dataset, and how they are connected by properties from Γ_p. This can be modeled by a finite resource graph defined over the universe $(\Gamma_i, \Gamma_v, \Gamma_p)$. The typing function $T_{\mathcal{D}}$ is used to specify which classes and datatypes each entity in $G_{\mathcal{D}}$ is typed to. In other words, it maps each entity in $G_{\mathcal{D}}$ to a subset of $\Gamma_c \cup \Gamma_d$. In the definition below, we use $\mathcal{P}(\Gamma_c \cup \Gamma_d)$ to denote the power set of $\Gamma_c \cup \Gamma_d$.

Definition 4.2.4 (Dataset). A *dataset* $\mathcal{D}$ is a pair $\mathcal{D} = (G_{\mathcal{D}}, T_{\mathcal{D}})$ where:

- $G_{\mathcal{D}}$ is a finite resource graph over the universe $(\Gamma_i, \Gamma_v, \Gamma_p)$, called the *data graph* of $\mathcal{D}$.
- $T_{\mathcal{D}} \colon \bar{V}(G_{\mathcal{D}}) \to \mathcal{P}(\Gamma_c \cup \Gamma_d)$ is a function, called the *typing function* of $\mathcal{D}$.

⊣

This dataset definition is very flexible. It does not restrict or limit which instances, data values, or properties to use, and it allows typing to any subset of the available types. The dataset could, for example, have an instance without types, or a data value typed to five different classes and one datatype.

However, we will be using queries that follow the structure of $\mathcal{N}$, and when these queries are executed over the dataset, they will only extract data that matches the structure given by the query, and hence $\mathcal{N}$. (See upcoming Definitions 4.2.9 and 4.3.1 for details.) In other words, if parts of the dataset do not conform to $\mathcal{N}$, then it will simply be ignored by the system, so indirectly, the dataset will have a connection to $\mathcal{N}$. A result of this is that all typing from instances to datatypes, and data values to classes, will be ignored. This is in accordance with how classes and datatypes should be used in a clean dataset: instances should be typed to classes, while data values should be typed to datatypes.

Since the most important part of a dataset $\mathcal{D} = (G_{\mathcal{D}}, T_{\mathcal{D}})$ is its data graph $G_{\mathcal{D}}$, we will often refer to entities, edges, or properties, in this graph directly, as if they were components of $\mathcal{D}$ itself. In particular, we define

$$\begin{aligned} \bar{V}(\mathcal{D}) &= \bar{V}(G_{\mathcal{D}}) & \bar{E}(\mathcal{D}) &= \bar{E}(G_{\mathcal{D}}) \\ \bar{V}_o(\mathcal{D}) &= \bar{V}_o(G_{\mathcal{D}}) & \bar{E}_o(\mathcal{D}) &= \bar{E}_o(G_{\mathcal{D}}) \\ \bar{V}_d(\mathcal{D}) &= \bar{V}_d(G_{\mathcal{D}}) & \bar{E}_d(\mathcal{D}) &= \bar{E}_d(G_{\mathcal{D}}) \end{aligned}$$

Example 4.2.5. Given the setup of components from Example 4.2.1, Figure 4.2 shows a valid dataset $\mathcal{D}$. It contains eight instances (blue circles) and 16 data values (yellow rectangles) in total. The types corresponding to each entity is listed inside its vertex, below its name. Notice that each entity is associated with exactly one single type, except for P_5, which is typed to both $Person$ and $Actor$, and 16, which is not typed to any class. The dataset presents a small group of six persons, and the countries they have visited. It is relatively simple and well-structured, but observe that two of the persons share the same name: $Alice$, and that P_4 is not connected to the other persons and countries, which makes the data graph disconnected. ♦

The dataset in Figure 4.2 is the only dataset we will present in this thesis, and every example that requires a dataset $\mathcal{D}$, will use this dataset.

So far we have given definitions describing the structure of both the navigation graph $\mathcal{N}$ and the dataset $\mathcal{D}$. These two structures are both very central parts of our work, and they will be used in many of the definitions presented throughout the thesis. Hence, from this point on, we will just assume that $\mathcal{N}$ refers to the only relevant navigation graph and that $\mathcal{D}$ refers to the only relevant dataset.

4.2.3 Queries

The core of a query $\mathcal{Q}$ is a tree-shaped query graph that specifies which variables to include, and how they are connected by properties. Each of these variables will be associated with a certain type. If the variable is typed to a class, it is defined to be an object variable, and if it is typed to a datatype, it is defined to be a data variable. This can be modeled by a resource tree $R_{\mathcal{Q}}$ over the universe $(\Gamma_{ov}, \Gamma_{dv}, \Gamma_p)$, and a *typing function* $T_{\mathcal{Q}}$, which maps each object variable in

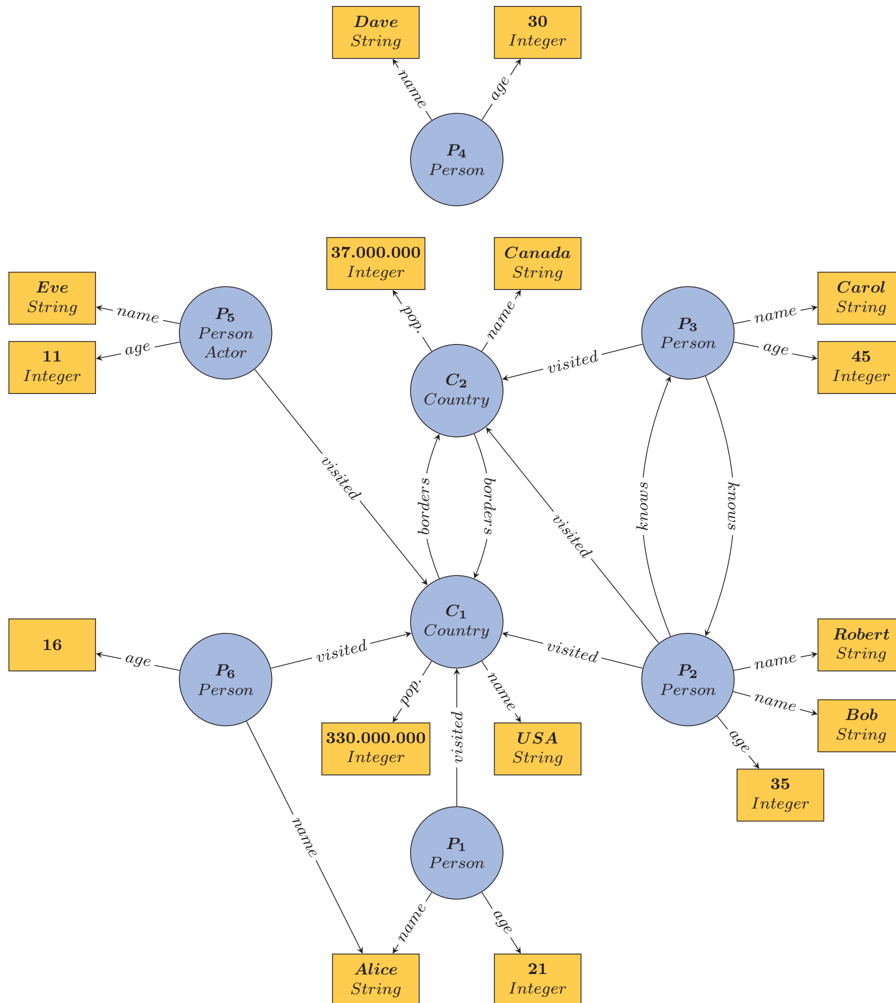


Figure 4.2: A visual representation of a dataset. The types of each instance and data value is given below its name.

$\bar{V}_o(R_\mathcal{Q})$ to a class in Γ_c, and each data variable in $\bar{V}_d(R_\mathcal{Q})$ to a datatype in Γ_d. In addition to $R_\mathcal{Q}$ and $T_\mathcal{Q}$, $\mathcal{Q}$ must also contain a *filter function* $F_\mathcal{Q}$, which maps each data variable v to its set of legal data values, i.e., $F_\mathcal{Q}$ must map the variable v to a subset of the possible data values Γ_v.

Definition 4.2.6 (Query)**.** A query $\mathcal{Q}$ is a triple $(R_\mathcal{Q}, T_\mathcal{Q}, F_\mathcal{Q})$ where:

- $R_\mathcal{Q}$ is a resource tree over $(\Gamma_{ov}, \Gamma_{dv}, \Gamma_p)$, called the *query tree* of $\mathcal{Q}$.
- $T_\mathcal{Q}\colon \bar{V}(R_\mathcal{Q}) \to (\Gamma_c \cup \Gamma_d)$ is a function, called the *typing function* of $\mathcal{Q}$, which satisfies both of the following requirements:
  - $T_\mathcal{Q}(v) \in \Gamma_c$ for every $v \in \bar{V}_o(R_\mathcal{Q})$
  - $T_\mathcal{Q}(v) \in \Gamma_d$ for every $v \in \bar{V}_d(R_\mathcal{Q})$
- $F_\mathcal{Q}\colon \bar{V}_d(R_\mathcal{Q}) \to \mathcal{P}(\Gamma_v)$ is a function, called the *filter function* of $\mathcal{Q}$.

$\dashv$

In the next chapter, where we describe how to extend queries, we need a way to represent queries where a particular variable v_r is in focus. To do this, we use queries where the query tree $R_\mathcal{Q}$ is rooted, and where the root of $R_\mathcal{Q}$ equals this special variable v_r. Since the query tree of such a query is rooted, we call it *rooted query*. Furthermore, we will use the term *root variable* when we refer to v_r, and we will use the function root to access this root.

Definition 4.2.7 (Rooted Query)**.** A *rooted query* is a query $\mathcal{Q} = (R_\mathcal{Q}, T_\mathcal{Q}, F_\mathcal{Q})$ were $R_\mathcal{Q}$ is a rooted resource tree. The *root variable* of $\mathcal{Q}$, denoted $\mathsf{root}(\mathcal{Q})$, is defined to be the root of $R_\mathcal{Q}$, i.e., $\mathsf{root}(\mathcal{Q}) = \mathsf{root}(R_\mathcal{Q})$. $\dashv$

Example 4.2.8. Consider the two queries:

$$\mathcal{Q}_1 = (R_{\mathcal{Q}_1}, T_{\mathcal{Q}_1}, F_{\mathcal{Q}_1}) \text{ and } \mathcal{Q}_2 = (R_{\mathcal{Q}_2}, T_{\mathcal{Q}_2}, F_{\mathcal{Q}_2})$$

where

$$\begin{aligned}
R_{\mathcal{Q}_1} &= (\{?o_1\}, \{?d_1\}, \{(?o_1, name, ?d_1)\}) \\
T_{\mathcal{Q}_1} &= \{?o_1 \mapsto Person, ?d_1 \mapsto String\} \\
F_{\mathcal{Q}_1} &= \{?d_1 \mapsto \Gamma_v\}
\end{aligned}$$

$$\begin{aligned}
R_{\mathcal{Q}_2} &= (\{?o_1\}, \{?d_1, ?d_2\}, \{(?o_1, name, ?d_1), (?o_1, age, ?d_2)\}) \\
T_{\mathcal{Q}_2} &= \{?o_1 \mapsto Person, ?d_1 \mapsto String, ?d_2 \mapsto Integer\} \\
F_{\mathcal{Q}_2} &= \{?d_1 \mapsto \Gamma_v, ?d_2 \mapsto \{u \in \Gamma_v \mid u \geq 18\}\}
\end{aligned}$$

Since both $R_{\mathcal{Q}_1}$ and $R_{\mathcal{Q}_2}$ are rooted resource trees, $\mathcal{Q}_1$ and $\mathcal{Q}_2$ are in fact rooted queries where $\mathsf{root}(\mathcal{Q}_1) = ?o_1$ and $\mathsf{root}(\mathcal{Q}_2) = ?o_2$. Visual representations of both $\mathcal{Q}_1$ and $\mathcal{Q}_2$ are given in Figure 4.3, where each vertex contains the name, type, and potential filter of the variable it corresponds to, and where the arrow below each query points at the root.

♦

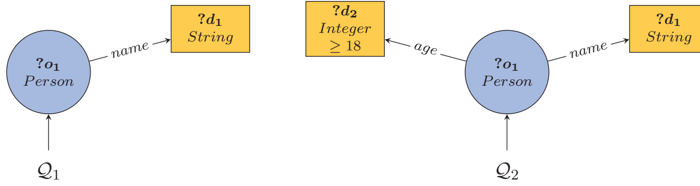


Figure 4.3: The two rooted queries $\mathcal{Q}_1$ and $\mathcal{Q}_2$.

The way we have chosen to model filters is very general: the function $F_{\mathcal{Q}}$ links each data variable v to the set of values it can be mapped to. We call this the *filter set* of v. These filter sets can be used to model all kinds of user interface filter elements over the system, as each such element essentially just defines a set of values the variable can take. By default, we consider a variable to be unrestricted, which corresponds to a filter set equal to Γ_v. Any significant restriction to a variable will then correspond to a filter set equal to a proper subset of the values in Γ_v.

Notice that in Figure 4.3 we did not include the filter set of the variable $?d_1$ in query $\mathcal{Q}_2$. This is because it is unrestricted. Variable $?d_2$, on the other hand, is restricted to values not below 18, and since this is not the default filter, we have to specify it explicitly in the visual representation of the query.

If all the data variables of a query are unrestricted, the query is called a *filterless query*. There is no need to specify the filter function of such a query since it will just map every variable to Γ_v. Hence, if a query is filterless, we will often just exclude the entire filter function from the query specification, and represent a query by only its query tree and typing function. Using this shorthand notation, we can represent query $\mathcal{Q}_1$ again like this:

$$\mathcal{Q}_1 = (R_{\mathcal{Q}_1}, T_{\mathcal{Q}_1})$$

where

$$R_{\mathcal{Q}_1} = (\{?o_1\}, \{?d_1\}, \{(?o_1, name, ?d_1)\})$$
$$T_{\mathcal{Q}_1} = \{?o_1 \mapsto Person, ?d_1 \mapsto String\}$$

Since the most important part of a query $\mathcal{Q} = (R_{\mathcal{Q}}, T_{\mathcal{Q}}, F_{\mathcal{Q}})$ is its query tree $R_{\mathcal{Q}}$, we will often refer to variables, edges, or properties in this tree directly, as if they were components of $\mathcal{Q}$ itself. In particular, we define

$$\bar{V}(\mathcal{Q}) = \bar{V}(R_{\mathcal{Q}}) \qquad \bar{E}(\mathcal{Q}) = \bar{E}(R_{\mathcal{Q}})$$
$$\bar{V}_o(\mathcal{Q}) = \bar{V}_o(R_{\mathcal{Q}}) \qquad \bar{E}_o(\mathcal{Q}) = \bar{E}_o(R_{\mathcal{Q}})$$
$$\bar{V}_d(\mathcal{Q}) = \bar{V}_d(R_{\mathcal{Q}}) \qquad \bar{E}_d(\mathcal{Q}) = \bar{E}_d(R_{\mathcal{Q}})$$

Legal Queries The queries we have considered so far do not have any connection at all to the navigation graph $\mathcal{N}$. But as indicated earlier, the purpose of having

the navigation graph is to control which queries the user is allowed to make. More precisely, the system should only allow queries where the typing function $T_{\mathcal{Q}}$ is a homomorphism from the query tree of $\mathcal{Q}$ to $\mathcal{N}$. When this is the case, we say that the query *conforms* to $\mathcal{N}$, or that the query is *legal* with respect to $\mathcal{N}$.

Definition 4.2.9 (Legal Queries)**.** A query $\mathcal{Q} = (R_{\mathcal{Q}}, T_{\mathcal{Q}}, F_{\mathcal{Q}})$ is *legal* with respect to $\mathcal{N}$ if $T_{\mathcal{Q}}$ is a homomorphism from $R_{\mathcal{Q}}$ to $\mathcal{N}$. ⊣

Given a legal query $\mathcal{Q} = (R_{\mathcal{Q}}, T_{\mathcal{Q}}, F_{\mathcal{Q}})$, the typing function $T_{\mathcal{Q}}$ can be applied to any variable $v \in \bar{V}(\mathcal{Q})$ to get the type $T_{\mathcal{Q}}(v) \in \bar{V}(\mathcal{N})$ that corresponds to v. But, this typing function indirectly also maps each edge $e = (v_s, p, v_t) \in \bar{E}(\mathcal{Q})$ to a corresponding edge in $\mathcal{N}$, given by $T_{\mathcal{Q}}(e) = (T_{\mathcal{Q}}(v_s), p, T_{\mathcal{Q}}(v_t))$.

Example 4.2.10. Consider the two queries $\mathcal{Q}_1$ and $\mathcal{Q}_2$ from Example 4.2.8. Both of them are legal with respect to $\mathcal{N}$ from Example 4.2.3. To prove that $\mathcal{Q}_1$ is legal, we need to show that $T_{\mathcal{Q}_1}$ is a homomorphism from $R_{\mathcal{Q}_1}$ to $\mathcal{N}$. $R_{\mathcal{Q}_1}$ only has one edge, $e = (?o_1, name, ?d_1)$, and in order for $T_{\mathcal{Q}_1}$ to be a homomorphism, there must a corresponding edge $T_{\mathcal{Q}_1}(e) = (T_{\mathcal{Q}_1}(?o_1), name, T_{\mathcal{Q}_1}(?d_1)) = (Person, name, String)$ in $\mathcal{N}$, which is indeed the case. To prove that $\mathcal{Q}_2$ conforms to $\mathcal{N}$, one has to do a similar check for $T_{\mathcal{Q}_2}$ and the two edges in $\mathcal{Q}_2$. ♦

Simple Queries A rooted query is called *simple* if it does not have two of the same outgoing properties. Simple queries will play a central role later in the thesis.

Definition 4.2.11 (Simple Rooted Queries)**.** A rooted query $\mathcal{Q} = (R_{\mathcal{Q}}, T_{\mathcal{Q}}, F_{\mathcal{Q}})$ is *simple* if no variable has more than one outgoing edge with the same label and target type. I.e.,

$$(v_s, p, v_t), (v_s, p, v'_t) \in \bar{E}(R_{\mathcal{Q}}) \text{ and } T_{\mathcal{Q}}(v_t) = T_{\mathcal{Q}}(v'_t) \implies v_t = v'_t$$

⊣

Conversely, a query that is not simple is called *non-simple*.

Subqueries If a rooted query $\mathcal{Q}_s$ contains only a subset of the variables of another query $\mathcal{Q}$, and they both have the same root, then we say that $\mathcal{Q}_s$ is a subquery $\mathcal{Q}$. The typing function and filter function of $\mathcal{Q}$ can be used also in $\mathcal{Q}_s$, but it must be restricted to the variables of $\mathcal{Q}_s$ first.

Definition 4.2.12 (Subquery)**.** Let $\mathcal{Q} = (R_{\mathcal{Q}}, T_{\mathcal{Q}}, F_{\mathcal{Q}})$ and $\mathcal{Q}_s = (R_{\mathcal{Q}_s}, T_{\mathcal{Q}_s}, F_{\mathcal{Q}_s})$ be two rooted queries. $\mathcal{Q}_s$ is a *subquery* of $\mathcal{Q}$, denoted $\mathcal{Q}_s \sqsubseteq \mathcal{Q}$, if all the following statements are true:

- $\mathsf{root}(R_{\mathcal{Q}}) = \mathsf{root}(R_{\mathcal{Q}_s})$
- $R_{\mathcal{Q}_s}$ is a subgraph of $R_{\mathcal{Q}}$

- $T_{\mathcal{Q}_s} = T_{\mathcal{Q}}\restriction_{R_{\mathcal{Q}_s}}$
- $F_{\mathcal{Q}_s} = F_{\mathcal{Q}}\restriction_{R_{\mathcal{Q}_s}}$

⊣

Query Renaming The names of the query variables does not contain any useful meaning in themselves – they are essentially just identifiers of the variables. Hence, if two queries have the same structure, types, and filters, but they differ on one or more of the variable names, they will still have the exact same semantic meaning. This kind of *renaming* from one query to another can be described by what we call a *renaming function*, which is a function that maps each variable in the first query to a corresponding variable in the other query.

Definition 4.2.13 (Query Renaming)**.** A *renaming function* f_r from a rooted query $\mathcal{Q} = (R_{\mathcal{Q}}, T_{\mathcal{Q}}, F_{\mathcal{Q}})$ to a rooted query $\mathcal{Q}' = (R_{\mathcal{Q}'}, T_{\mathcal{Q}'}, F_{\mathcal{Q}'})$, is an isomorphism from $R_{\mathcal{Q}}$ to $R_{\mathcal{Q}'}$ that preserves the types, the filters, and the root, i.e.,

$$\begin{aligned} T_{\mathcal{Q}}(v) &= T_{\mathcal{Q}'}(f_r(v)) \text{ for each variable } v \in \bar{V}(R_{\mathcal{Q}}) \\ F_{\mathcal{Q}}(v) &= F_{\mathcal{Q}'}(f_r(v)) \text{ for each data variable } v \in \bar{V}_d(R_{\mathcal{Q}}) \\ \mathsf{root}(R_{\mathcal{Q}'}) &= f_r(\mathsf{root}(R_{\mathcal{Q}})) \end{aligned}$$

If f_r is a renaming function from $\mathcal{Q}$ to $\mathcal{Q}'$, then its inverse, f_r^{-1}, will be a renaming function in the opposite direction, from $\mathcal{Q}'$ to $\mathcal{Q}$. If there exists a renaming function from $\mathcal{Q}$ to $\mathcal{Q}'$, then $\mathcal{Q}'$ is will be called a *renaming* of $\mathcal{Q}$ (and vice versa). ⊣

Notice that renaming functions can only be found between queries that are actually semantically identical. Also observe that if the two queries are non-simple, then there may be more than one renaming function between them.

4.3 Query Answers

When executing a query $\mathcal{Q} = (R_{\mathcal{Q}}, T_{\mathcal{Q}}, F_{\mathcal{Q}})$ over the dataset $\mathcal{D} = (G_{\mathcal{D}}, T_{\mathcal{D}})$, we are interested in all parts of the data graph $G_{\mathcal{D}}$ that matches the query pattern defined by $R_{\mathcal{Q}}$. Such a match can be represented by a homomorphism π from $R_{\mathcal{Q}}$ to $G_{\mathcal{D}}$, which also preserves the types. Furthermore, to ensure the filters are satisfied, π must map variables to values accepted by $F_{\mathcal{Q}}$.

Definition 4.3.1 (Query Answers)**.** Let $\mathcal{Q} = (R_{\mathcal{Q}}, T_{\mathcal{Q}}, F_{\mathcal{Q}})$ be a query. The *answers* of $\mathcal{Q}$ over $\mathcal{D}$, denoted $\mathsf{ans}(\mathcal{Q}, \mathcal{D})$, is the set of all homomorphisms $\pi\colon \bar{V}(R_{\mathcal{Q}}) \to \bar{V}(G_{\mathcal{D}})$ from $R_{\mathcal{Q}}$ to $G_{\mathcal{D}}$ that satisfies:

$$\begin{aligned} T_{\mathcal{Q}}(v) &\in T_{\mathcal{D}}(\pi(v)) && \text{for each } v \in \bar{V}(R_{\mathcal{Q}}) \\ \pi(v) &\in F_{\mathcal{Q}}(v) && \text{for each } v \in \bar{V}_d(R_{\mathcal{Q}}) \end{aligned}$$

⊣

π	$?o_1$	$?d_1$
π_1	$P1$	*Alice*
π_2	$P2$	*Robert*
π_3	$P2$	*Bob*
π_4	$P3$	*Carol*
π_5	$P4$	*Dave*
π_6	$P5$	*Eve*
π_7	$P6$	*Alice*

Table 4.2: Answers returned by $\mathsf{ans}(\mathcal{Q}_1, \mathcal{D})$.

π	$?o_1$	$?d_1$	$?d_2$
π_1	$P1$	*Alice*	21
π_2	$P2$	*Robert*	35
π_3	$P2$	*Bob*	35
π_4	$P3$	*Carol*	45
π_5	$P4$	*Dave*	30

Table 4.3: Answers returned by $\mathsf{ans}(\mathcal{Q}_2, \mathcal{D})$.

Example 4.3.2. If we execute queries $\mathcal{Q}_1$ and $\mathcal{Q}_2$ from Example 4.2.8 over the dataset $\mathcal{D}$ from Example 4.2.5, we get the results presented in Table 4.2 and Table 4.3, respectively. In both of these tables, each row corresponds to a homomorphism π_i, while each column corresponds to a query variable in the relevant query. Each cell corresponds to the entity one gets by applying the row's homomorphism to the column's variable.

♦

Projected Answers The answer function $\mathsf{ans}(\mathcal{Q}, \mathcal{D})$ gives us the full set of answers to the query $\mathcal{Q}$ over the dataset $\mathcal{D}$, but we will often only be interested in the entities assigned to one specific variable of $\mathcal{Q}$, i.e., the *projected answers* onto a variable v.

Definition 4.3.3. Let $\mathcal{Q}$ be a query and let v be a variable in $\bar{V}(\mathcal{Q})$. We define the *projected answers* of $\mathcal{Q}$ with respect to the variable v, denoted $\mathsf{ans}_P(\mathcal{Q}, \mathcal{D}, v)$, to be the set of distinct entities assigned to v by functions in $\mathsf{ans}(\mathcal{Q}, \mathcal{D})$. I.e.,

$$\mathsf{ans}_P(\mathcal{Q}, \mathcal{D}, v) = \{\pi(v) \mid \pi \in \mathsf{ans}(\mathcal{Q}, \mathcal{D})\}$$

⊣

Example 4.3.4. In Example 4.3.2 we executed $\mathcal{Q}$ over $\mathcal{D}$ using the standard answer function ans. If we gather all distinct values from each column in the result table of this operation, we get the projected answers:

$$\mathsf{ans}_P(\mathcal{Q}_1, \mathcal{D}, ?o_1) = \{P1, P2, P3, P4, P5, P6\}$$
$$\mathsf{ans}_P(\mathcal{Q}_1, \mathcal{D}, ?d_1) = \{Alice, Bob, Carol, Dave, Eve\}$$

♦

Productive Queries Now when we have defined what it means to be an answer to a query $\mathcal{Q}$, we can formally define what our system needs to detect dead-ends. We define a query to be *productive* if it returns answers over $\mathcal{D}$, and *unproductive* if this is not the case. The task of detecting dead-end queries is then equivalent to the task of finding unproductive queries.

Definition 4.3.5 (Productive Query). A query $\mathcal{Q}$ is *productive* with respect to $\mathcal{D}$ if $|\,\mathsf{ans}(\mathcal{Q},\mathcal{D})| > 0$. If the query is not productive, it is called *unproductive.* $\dashv$

For example, query $\mathcal{Q}_2$ is productive, because when executing it over $\mathcal{D}$, five answers are returned. If, however, we changed the filter on $?d_2$ to $\{u \in \Gamma_v \mid u \geq 50\}$, it would become unproductive, because every person in the dataset is younger than 50 years.

Theorems Finally, we present some useful theorems about queries and their answers. Given two queries $\mathcal{Q}$ and $\mathcal{Q}_s \sqsubseteq \mathcal{Q}$, an answer π to $\mathcal{Q}$ is also an answer to $\mathcal{Q}_s$, after restricting it to the variables of $\mathcal{Q}_s$.

Theorem 4.3.6. If $\mathcal{Q}_s = (R_{\mathcal{Q}_s}, T_{\mathcal{Q}_s}, F_{\mathcal{Q}_s})$ is a subquery of $\mathcal{Q} = (R_{\mathcal{Q}}, T_{\mathcal{Q}}, F_{\mathcal{Q}})$, and $\pi \in \mathsf{ans}(\mathcal{Q},\mathcal{D})$ then $\pi\restriction \bar{V}(R_{\mathcal{Q}_s}) \in \mathsf{ans}(\mathcal{Q}_s,\mathcal{D})$. $\dashv$

Proof. In order to prove this, we must check that all requirements from Definition 4.3.1 holds for $\pi_s = \pi\restriction \bar{V}(R_{\mathcal{Q}_s})$. Since π is a homomorphism from $\mathcal{Q}$ to $\mathcal{D}$, then π_s must be a homomorphism from $R_{\mathcal{Q}_s}$ to $\mathcal{D}$. Furthermore, since $T_{\mathcal{Q}}(v) \in T_{\mathcal{D}}(\pi(v))$, $T_{\mathcal{Q}}(v) = T_{\mathcal{Q}_s}(v)$, and $T_{\mathcal{D}}(\pi(v)) = T_{\mathcal{D}}(\pi_s(v))$ for all $v \in \bar{V}(R_{\mathcal{Q}_s})$, then $T_{\mathcal{Q}_s}(v) \in T_{\mathcal{D}}(\pi_s(v))$. Finally, since $\pi(v) \in F_{\mathcal{Q}}(v)$, $\pi(v) = \pi_s(v)$, and $F_{\mathcal{Q}_s}(v) = F_{\mathcal{Q}}(v)$ for all $v \in \bar{V}_d(R_{\mathcal{Q}_s})$, then we also get $\pi_s(v) \in F_{\mathcal{Q}_s}(v)$. ■

If we project answers onto a variable v that exists in both $\mathcal{Q}_s$ and $\mathcal{Q}$, then we get the following theorem.

Theorem 4.3.7. If $\mathcal{Q}_s = (R_{\mathcal{Q}_s}, T_{\mathcal{Q}_s}, F_{\mathcal{Q}_s})$ is a subquery of $\mathcal{Q} = (R_{\mathcal{Q}}, T_{\mathcal{Q}}, F_{\mathcal{Q}})$, then

$$\mathsf{ans}_P(\mathcal{Q},\mathcal{D},v) \subseteq \mathsf{ans}_P(\mathcal{Q}_s,\mathcal{D},v)$$

for each variable $v \in \bar{V}(R_{\mathcal{Q}_s})$. $\dashv$

Proof. If $u \in \mathsf{ans}_P(\mathcal{Q},\mathcal{D},v)$, then there must be a $\pi \in \mathsf{ans}(\mathcal{Q},\mathcal{D})$ such that $\pi(v) = u$. From Theorem 4.3.6, we then know that $\pi\restriction R_{\mathcal{Q}_s} \in \mathsf{ans}(\mathcal{Q}_s,\mathcal{D})$, and since $v \in \bar{V}(R_{\mathcal{Q}_s})$, then $u = \pi(v) = \pi\restriction R_{\mathcal{Q}_s}(v) \in \mathsf{ans}_P(\mathcal{Q}_s,\mathcal{D},v)$. ■

In this chapter, we have presented the most essential structures and definitions that our VQS model relies on. In the next chapter, we will complete this VQS model, by defining how the user can extend the partial query. We will also define how the system can provide query extension suggestions to the user.

Chapter 5

Query Extensions

In the previous chapter, we formally defined the most central parts of our VQS model: the navigation graph $\mathcal{N}$, the dataset $\mathcal{D}$, and the type of queries it supports. Furthermore, we defined the two answer functions ans and ans_P, which returns respectively the full set of answers, and the projected answers of a query when it is executed over $\mathcal{D}$. Finally, we defined a query to be legal when it follows the structure outlined by $\mathcal{N}$, and we defined it to be productive when it returns non-empty answers over $\mathcal{D}$.

We also described briefly how a session can be described by the sequence of queries the user makes: $\mathcal{Q}_1, \mathcal{Q}_2, \ldots, \mathcal{Q}_k$. In this chapter, we will take a closer look at the transition between these queries, i.e., how the user modifies a query $\mathcal{Q}_{i-1}$ into a new version $\mathcal{Q}_i$, and in particular, we are interested in *query extensions* since they can potentially lead to dead-ends.

This chapter contains four sections. In Section 5.1, we describe the three user actions our VQS supports: *delete*, *refocus*, and *extend*. All of them make modifications to the partial query, but extend is the most interesting one in the context of our work since it is the only action that can possibly lead to dead-end queries. Based on the definitions of legal and productive queries from the last chapter, we define legal and productive extensions in Sections 5.2 and 5.3 respectively. Then, in Section 5.4 we discuss different approaches to detect such productive extensions.

5.1 User Actions

We assume that the VQS supports three actions that allow the user to modify the partial query:

- Delete: Remove a given leaf variable from the query.
- Refocus: Change the focus variable of the query.
- Extend: Extend the query with a new variable.

Multiple uses of these three actions can be done in sequence, and this allows the user to formulate any legal query. They can also be combined to make other more advanced user actions, like editations. For example, if the user requests the system to edit the filter set of a variable, this can be done by deleting the whole variable, and then replace it with a new variable coupled with the correct set of filters. The three actions presented above are not necessarily the same as those that are presented to the user, but since they make a complete set of actions, it is sufficient to only consider them.

Delete The delete action allows the user to select and delete an existing leaf variable from the query, i.e., remove a variable that is only connected to one other variable. This action also deletes the edge that connects it to the rest of the query, and it discards both the type and the filter of the variable. Only leaf variables can be deleted, because if internal variables (i.e., non-leaf variables) are deleted, then the resulting query would become disconnected, which is illegal. If the user really wants to delete an internal variable, they need to first delete other leaf variables, one by one, until the internal variable becomes a leaf variable. After that, they can delete it directly.

We define the deletion process formally by a function del, which takes as input a query $\mathcal{Q}$ and the variable v to delete, and returns the parts of $\mathcal{Q}$ that are left after v has been removed.

Definition 5.1.1 (Delete Function)**.** Let $\mathcal{Q} = (R_\mathcal{Q}, T_\mathcal{Q}, F_\mathcal{Q})$ be a query, and let $v \in \bar{V}(R_\mathcal{Q})$ be a leaf variable of $\mathcal{Q}$. The query $\mathcal{Q}'$ that remains after removing both v and the only edge e connected to v from $\mathcal{Q}$, denoted $\mathsf{del}(\mathcal{Q}, v)$, is defined as $\mathcal{Q}' = \mathsf{del}(\mathcal{Q}, v) = ((V'_o, V'_d, E'), T_{\mathcal{Q}'}, F_{\mathcal{Q}'})$ where

$$\begin{aligned}
V'_o &= V_o \setminus \{v\} \\
V'_d &= V_d \setminus \{v\} \\
E' &= E \setminus \{e\} \\
T_{\mathcal{Q}'} &= T_\mathcal{Q}\restriction_{(V'_o \cup V'_d)} \\
F_{\mathcal{Q}'} &= F_\mathcal{Q}\restriction_{V'_d}
\end{aligned}$$

⊣

In the definition above, v should be removed from V_o if it is an object variable, and from V_d if it is a data variable, but since we do not want to make one delete function for each of the two cases, we just try to remove it from both sets, and then it will be removed from the set where it actually exists.

Refocus Before the user can extend the query with a new variable through the extend action, they must first decide which variable to extend from. This is done through a process called *focusing* or *refocusing*, where the user just selects one of the variables, which is then set to be the *focus variable*.

After a variable has been selected, it is always possible to turn the partial query into a rooted query where the focus variable is the root, by replacing a particular subset of the edges with their inverse. Technically, this process makes a new version of the query, but since ans considers each edge and its corresponding inverse to be equivalent, this new query will return the exact same answers as the original query over every dataset. In other words, this flipping process does not change the semantic meaning of the query. At the same time, this representation is very convenient, because it standardizes the form of queries, by making sure that every edge points away from the focus variable. Because of this, we will be using this rooted version of the partial query frequently in this and the remaining chapters.

It should always be clear to the user which variable is in focus at any time, so it is important that the system labels the focus variable in some way. However, the user will not necessarily be interested in the rooted version of the query, where some of the edges have been flipped, so this rooted version should only be used internally by the system and kept hidden to the user.

Selecting a focus variable is then, from a technical perspective, just the process of defining which variable to use as root in the query. We now formalize this *rooting process* by the function setRoot, which turns a query $\mathcal{Q}$ into the semantically equivalent rooted query $\mathcal{Q}'$ where the given variable v_r is the root.

Definition 5.1.2 (Set Root Function). Let $\mathcal{Q} = (R_{\mathcal{Q}}, T_{\mathcal{Q}}, F_{\mathcal{Q}})$ be a query, and let $v_r \in \bar{V}_o(R_{\mathcal{Q}})$ be an object variable of $\mathcal{Q}$. The rooted query $\mathcal{Q}'$ returned after rooting on v_r, denoted $\mathsf{setRoot}(\mathcal{Q}, v_r)$, is the query that keeps all variables, types, and filters of $\mathcal{Q}$, but where every edge $e = (v_s, p, v_t) \in \bar{V}(R_{\mathcal{Q}})$ is replaced by e^{-1} if the unique direction-ignorant path from v_r to v_s contains v_t. ⊣

If the direction-ignorant path from v_r to v_s contains v_t, it means that v_t is closer to v_r than v_s, which means that e points towards v_r. This is not what we want, so if that is the case, e has to be flipped, i.e., it has to be replaced by e^{-1}.

Since we are representing the partial query with focus as a rooted query, most of the remaining work of this thesis will actually be related to rooted queries. In fact, there will just be a few cases where we consider queries where a root variable is not set, so instead of specifying every time that a query is rooted, we will instead specify when this is not the case – we then use the term *unrooted query*. When we are working with rooted queries, we will use both of the two terms *root variable* and *focus variable* when we refer to the root of the query. We will use the terms *root class* or *focus class* when we refer to the type of the root variable.

Example 5.1.3. Consider the unrooted query $\mathcal{Q}_9$ in Figure 5.1. It can be turned into the rooted query $\mathcal{Q}_{9a}$ by focusing on the variable $?o_1$, i.e., $\mathcal{Q}_{9a} = \mathsf{setRoot}(\mathcal{Q}_9, ?o_1)$. Similarly, it can be turned into $\mathcal{Q}_{9b}$ and $\mathcal{Q}_{9c}$ by focusing on $?o_2$ and $?o_3$ respectively. Notice how the edge $(?o_2, visitedBy, ?o_1)$ in $\mathcal{Q}_9$ got replaced by its inverse edge $(?o_1, visited, ?o_2)$ in $\mathcal{Q}_{9a}$. This was done because all edges have to point away from the root in a rooted tree.

♦

Extend The third, and most important user action we consider, is the extend action, which allows the user to add a new *extension variable* v_e to the query by connecting it to the current focus variable v_r via a property p_e. The extension variable v_e also needs a type t_e, and a filter set X_e if it is a data variable. This gives us two types of extensions: *object property extensions* when v_e is an object variable, and *data property extensions* when v_e is a data variable. These two types of extensions are quite different due to the way the navigation graph and queries are defined. The most notable difference is that data property extensions require a filter set that must be attached to the extension variable, while this is not required for object property extensions. This actually makes dead-end

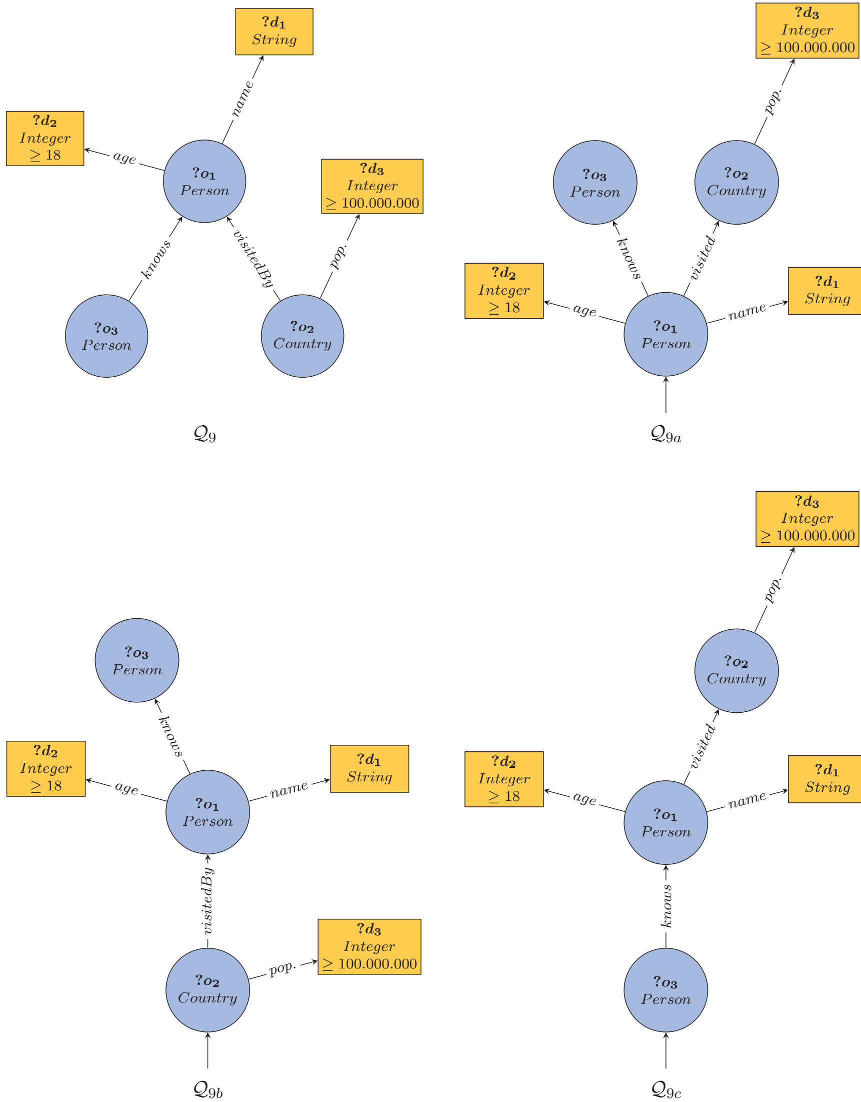


Figure 5.1: Query $\mathcal{Q}_9$ and how focusing on each of its three object variables results in three different rooted queries.

detection much harder in the case of data property extensions. In fact, object property extensions and data property extensions are so different, that it is hard to make a unified extensions model for them, so we will for now just consider dead-end detection over data property extensions, and then, later, we will discuss how the methods we develop can be used to also detect dead-ends over object property extensions.

A data property extension can be specified by the triple consisting of the *extension property* p_e, the *extension type* t_e, and the *extension filter set* X_e. We group them together into what we call an *extension specification* (p_e, t_e, X_e).

Definition 5.1.4 (Extension Specification)**.** An *extension specification* is a triple $(p_e, t_e, X_e) \in \Gamma_p \times \Gamma_d \times \mathcal{P}(\Gamma_v)$, where p_e is called the *extension property*, t_e is called the *extension type*, and X_e is called the *extension filter set*. $\dashv$

When we use the term *extension*, we usually refer to the whole process of extending $\mathcal{Q}$ into a new version $\mathcal{Q}'$, as described above. But, since there is a one-to-one correspondence between extensions and extension specifications, we may occasionally use the term *extension* also when we refer to specifications.

We can now define the *extension function* ext, which formalizes the process of extending our query $\mathcal{Q}$ according to an extension specification σ.

Definition 5.1.5 (Extension Function)**.** Let $\mathcal{Q} = (R_\mathcal{Q}, T_\mathcal{Q}, F_\mathcal{Q})$ be a query, where $R_\mathcal{Q} = (V_o, V_d, E)$ and $v_r = \mathsf{root}(R_\mathcal{Q})$, and let $\sigma = (p_e, t_e, X_e)$ be an extension specification. The query $\mathcal{Q}'$ resulting from extending a query $\mathcal{Q}$ according to the extension specification σ, into a new extension variable $v_e \in (\Gamma_{dv} \setminus \bar{V}(\mathcal{Q}))$, denoted $\mathsf{ext}(\mathcal{Q}, \sigma, v_e)$, is given by $\mathcal{Q}' = \mathsf{ext}(\mathcal{Q}, \sigma, v_e) = ((V_o', V_d', E'), T_{\mathcal{Q}'}, F_{\mathcal{Q}'})$ where

$$V_o' = V_o \tag{5.1}$$

$$V_d' = V_d \cup \{v_e\} \tag{5.2}$$

$$E' = E \cup \{(v_r, p_e, v_e)\} \tag{5.3}$$

$$T_{\mathcal{Q}'} = T_\mathcal{Q} \cup \{v_e \mapsto t_e\} \tag{5.4}$$

$$F_{\mathcal{Q}'} = F_\mathcal{Q} \cup \{v_e \mapsto X_e\} \tag{5.5}$$

$\dashv$

Even though this definition is quite extensive, it still just describes the relatively simple process of adding one new data variable to the query. The new query $\mathcal{Q}'$ keeps everything from the original query, but adds what is needed to include the new variable. All object variables are kept unchanged (Equation 5.1), while both a new data variable and an edge are added (Equation 5.2 and Equation 5.3). The types and filter sets are kept unchanged for all variables, except for v_e, which is mapped to the type t_e and filter set X_e (Equation 5.4 and Equation 5.5).

Example 5.1.6. Let $\mathcal{Q}_1$ be the query defined in Example 4.2.8, which is presented again in Figure 5.2. Furthermore, let $\sigma = (age, Integer, \{u \in \Gamma_v \mid u \geq 18\})$ be an extension specification. Using the extension function ext, we can construct the

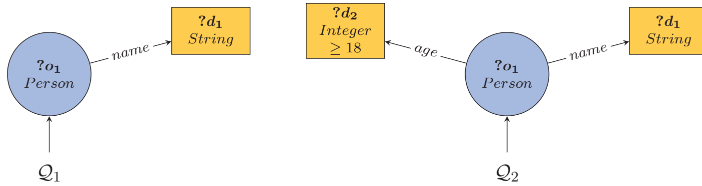


Figure 5.2: The query $\mathcal{Q}_1$, and the resulting query $\mathcal{Q}_2$ after extension.

query we get by extending $\mathcal{Q}_1$ according to the specification σ, into a new variable $?d_2$. This gives us $\mathcal{Q}_2 = \mathsf{ext}(\mathcal{Q}_1, \sigma, ?d_2)$, which is also defined in Example 4.2.8, and presented visually in Figure 5.2, next to $\mathcal{Q}_1$.

♦

Notice that the extension variable v_e is not included in the specification σ in Definition 5.1.4. This is because it is just a reference to the variable, and this does not affect the meaning of the resulting query. Or, said in another way, the query $\mathcal{Q}' = \mathsf{ext}(\mathcal{Q}, \sigma, v_e)$ is just a renaming of the query $\mathcal{Q}' = \mathsf{ext}(\mathcal{Q}, \sigma, v'_e)$, which we get if we extend $\mathcal{Q}$ into another variable v'_e. In particular, we are very interested in the projected answers of $\mathcal{Q}'$ with respect to the extension variable, and those do not depend on the extension variable's name.

Among the three actions we now have considered, only the extend action makes the query more restrictive, which means that this is the only action that may lead the user into queries that are illegal or unproductive. Hence, we will only focus on extensions in the remainder of this thesis.

5.2 Legal Extensions

In Definition 4.2.9 we defined what it means for a query to be legal with respect to $\mathcal{N}$. Now we extend this term to also cover extension specifications: we consider them to be *legal* if they lead to legal queries.

Definition 5.2.1 (Legal Extension Specification)**.** Let $\mathcal{Q}$ be a query, and let σ be an extension specification. σ is a *legal extension specification* if $\mathsf{ext}(\mathcal{Q}, \sigma, v_e)$ is legal for each data variable $v_e \in (\Gamma_{dv} \setminus \bar{V}(\mathcal{Q}))$. ⊣

If we know that $\mathcal{Q}$ is already a legal query, then it is relatively easy to determine if σ is a legal extension: we have to consider the focus concept of the partial query and check that one of its outgoing edges in $\mathcal{N}$ has a property and target type that matches the property and type of σ. For example, if we work over the navigation graph from Example 4.2.3, and have a query with root class *Person*, then legal extensions must be of the form $(name, String, X_e)$ or $(age, Integer, X_e)$ because the only two outgoing data edges of *Person* are $(Person, name, String)$ and $(Person, age, Integer)$.

Theorem 5.2.2. Let $\mathcal{Q} = (R_\mathcal{Q}, T_\mathcal{Q}, F_\mathcal{Q})$ be a legal query, let t_r be the root class of $\mathcal{Q}$, and let $\sigma = (p_e, t_e, X_e)$ be an extension specification. σ is legal if and only if $(t_r, p_e, t_e) \in \bar{E}(\mathcal{N})$. ⊣

Proof. Since $\mathcal{Q}$ is a legal query, we know that $T_\mathcal{Q}$ is a homomorphism from $R_\mathcal{Q}$ to $\mathcal{N}$, i.e., there exists a corresponding edge in $\mathcal{N}$ for each edge in $R_\mathcal{Q}$. The only difference between the edges of $\mathcal{Q}$ and $\mathcal{Q}' = \mathsf{ext}(\mathcal{Q}, \sigma, v_e)$, is the new edge (v_r, p_e, v_e), so we just need to show that this edge has a corresponding edge in $\mathcal{N}$. From Definition 4.2.9 and Definition 4.1.11, this edge has to equal

$$(T_{\mathcal{Q}'}(v_r), p_e, T_{\mathcal{Q}'}(v_e)) = (t_r, p_e, t_e)$$

so $\mathcal{Q}'$ is legal if and only if $(t_r, p_e, t_e) \in \bar{E}(\mathcal{N})$. ■

Notice that we only need to consider p_e and t_e, and not X_e, in order to determine whether an extension is legal or not. This complies with Definition 4.2.9 of legal queries, where the filter of the query is not even considered. It also makes sense if we highlight the fact that $\mathcal{N}$ does not contain any data values, and hence, cannot dictate which data values to filter on. We will combine the property p_e and the type t_e into a pair $\tau = (p_e, t_e)$, called an *extension pair*, and we say that an extension is *based on* τ if it uses the property and type specified by τ, i.e., if it has the form (p_e, t_e, X_e).

Definition 5.2.3 (Extension Pair). An *extension pair* is a pair $\tau = (p_e, t_e) \in \Gamma_p \times \Gamma_d$. An extension specification $\sigma = (p'_e, t'_e, X_e)$ is *based on* τ if both $p_e = p'_e$ and $t_e = t'_e$. ⊣

The system should only allow legal queries, hence, it should only allow legal extension specifications. Based on our findings from Theorem 5.2.2, we can define the set of all *legal extension pairs*, i.e., all pairs such that any specification based on such a pair is legal.

Definition 5.2.4 (Legal Extension Pairs of Class). Let $t_r \in \bar{V}_o(\mathcal{N})$ be a class. The set of all *legal extension pairs* of t_r with respect to $\mathcal{N}$, denoted $J(t_r, \mathcal{N})$ is defined as

$$J(t_r, \mathcal{N}) = \{(p_e, t_e) \mid (t_r, p_e, t_e) \in \bar{E}(\mathcal{N})\}$$

⊣

We extend the definition of legal extension pairs to also cover queries.

Definition 5.2.5 (Legal Extension Pairs of Query). Let $\mathcal{Q} = (R_\mathcal{Q}, T_\mathcal{Q}, F_\mathcal{Q})$ be a legal query, and let t_r be the root class of $\mathcal{Q}$. The set of all *legal extension pairs* of $\mathcal{Q}$, denoted $J(\mathcal{Q}, \mathcal{N})$, is equal to $J(t_r, \mathcal{N})$. ⊣

Example 5.2.6. Given the navigation graph $\mathcal{N}$ from Example 4.2.3 and the query $\mathcal{Q}_1$ from Example 5.1.6 with root class $Person$, we get the following set of legal

extension pairs:

$$\begin{aligned} J(\mathcal{Q}_1, \mathcal{N}) &= J(Person, \mathcal{N}) \\ &= \{(p_e, t_e) \mid (Person, p_e, t_e) \in \bar{E}(\mathcal{N})\} \\ &= \{(name, String), (age, Integer)\} \end{aligned}$$

♦

The VQS should never allow the user to construct illegal queries, but this can easily be avoided by only allowing extension specifications that are legal. In the remainder of the thesis, where we focus mostly on productive queries, we will just assume that the partial query is legal and that only legal extensions will be considered.

5.3 Productive Extensions

Similar to how we defined and analyzed legal extensions in the previous section, we are now going to consider *productive extensions*, i.e., extensions that lead to productive queries. In Chapter 4, we defined a productive query to be a query $\mathcal{Q}$ that returns at least one answer when executed over $\mathcal{D}$. In other words, productive extensions are the extensions we want the system to suggest and allow because they are exactly the ones that are not leading to dead-end queries. In order to detect if a query $\mathcal{Q}$ is productive, we have to execute it over $\mathcal{D}$ and check if it returns any answers. This is a much more demanding task than checking if a query is legal because the dataset is much larger than the navigation graph.

Definition 5.3.1 (Productive Extension Specification)**.** Let $\mathcal{Q}$ be a query, and let σ be an extension specification. σ is *productive* if the query $\mathcal{Q}' = \mathsf{ext}(\mathcal{Q}, \sigma, v_e)$ is productive for all data variables $v_e \in (\Gamma_{dv} \setminus \bar{V}(\mathcal{Q}))$, i.e., if $\mathsf{ans}(\mathcal{Q}', \mathcal{D}) \neq \emptyset$. ⊣

Example 5.3.2. Consider query $\mathcal{Q}_1$ and the extension specification $\sigma = (age, Integer, \{u \in \Gamma_v \mid u \geq 18\})$ from Example 5.1.6. This is a productive extension, because if we extend $\mathcal{Q}_1$ according to σ, we get $\mathcal{Q}_2 = \mathsf{ext}(\mathcal{Q}_1, \sigma, ?d_2)$, and $\mathcal{Q}_2$ returns five answers over $\mathcal{D}$ (see Example 4.3.2). However, the extension specification $\sigma' = (age, Integer, \{u \in \Gamma_v \mid u \geq 50\})$, is not productive, because the query $\mathsf{ext}(\mathcal{Q}_1, \sigma', ?d_2)$, which asks for all named persons with age greater or equal to 50, returns no answers. The filter on the variable $?d_2$ is simply too strict to make the resulting query productive. ♦

If we only need to determine whether one single specification σ is productive or not, then we only need to calculate $\mathsf{ans}(\mathsf{ext}(\mathcal{Q}, \sigma, v_e), \mathcal{D})$ and check how many answers it returns, like we just did in Example 5.3.2. But, if we have to check multiple extension specifications based on the same extension pair (p_e, t_e), then it is better to first collect the set of all data values that can be assigned to the extension variable . For example, if we re-visit query $\mathcal{Q}_1$ from Example 5.3.2, and consider extensions based on the extension pair $\tau = (age, Integer)$, there are only five different data values the extension variable can take, given by the

ages of the people in the dataset: $\{11, 21, 30, 35, 45\}$. We call this the set of *productive values*. When we have this set, it is, for example, very easy to see that $\{u \in \Gamma_v \mid u \geq 50\}$ is a too restrictive filter set.

Definition 5.3.3 (Productive Values)**.** Let $\mathcal{Q}$ be a query, and let $\tau = (p_e, t_e)$ be an extension pair. The set of *productive values* of τ, denoted $X_o(\mathcal{Q}, \tau)$, is the set of all data values $u \in \Gamma_v$ such that the query we get by extending $\mathcal{Q}$ according to the extension $(p_e, t_e, \{u\})$ returns at least one answer. I.e.,

$$X_o(\mathcal{Q}, \tau) = \{u \in \Gamma_v \mid \mathsf{ans}(\mathsf{ext}(\mathcal{Q}, (p_e, t_e, \{u\}), v_e), \mathcal{D}) \neq \emptyset\}$$

$\dashv$

Theorem 5.3.4. Let $\mathcal{Q}$ be a query, and let $\tau = (p_e, t_e)$ be an extension pair. Furthermore, let $\mathcal{Q}_e$ be the query produced by extending $\mathcal{Q}$ according to the extension specification (p_e, t_e, Γ_v), i.e., $\mathcal{Q}_e = \mathsf{ext}(\mathcal{Q}, (p_e, t_e, \Gamma_v), v_e)$. Then the following statement is true.

$$X_o(\mathcal{Q}, \tau) = \mathsf{ans}_P(\mathcal{Q}_e, \mathcal{D}, v_e)$$

$\dashv$

Proof. Select an arbitrary data value $u \in \Gamma_v$, and let $\mathcal{Q}'_e = \mathsf{ext}(\mathcal{Q}, (p_e, t_e, \{u\}), v_e)$. Notice that $\mathcal{Q}'_e$ is identical to $\mathcal{Q}_e$, except for the filter set on v_e, where $\mathcal{Q}'_e$ is more restrictive. Hence, the two following statements are true.

1. $\pi \in \mathsf{ans}(\mathcal{Q}'_e, \mathcal{D}) \implies \pi \in \mathsf{ans}(\mathcal{Q}_e, \mathcal{D})$.
2. If $\pi \in \mathsf{ans}(\mathcal{Q}_e, \mathcal{D})$ and $\pi(v_e) = u$, then $\pi \in \mathsf{ans}(\mathcal{Q}'_e, \mathcal{D})$.

We start by proving that $X_o(\mathcal{Q}, \tau)$ is included in $\mathsf{ans}_P(\mathcal{Q}_e, \mathcal{D}, v_e)$. If $u \in X_o(\mathcal{Q}, \tau)$, then by Definition 5.3.3, there has to exist a function $\pi \in \mathsf{ans}(\mathcal{Q}'_e, \mathcal{D})$, such that $\pi(v_e) = u$. From the first statement above, we then know that $\pi \in \mathsf{ans}(\mathcal{Q}_e, \mathcal{D})$. If we now project onto v_e we get that $u = \pi(v_e) \in \mathsf{ans}_P(\mathcal{Q}_e, \mathcal{D}, v_e)$.

We can make a similar argument to prove inclusion the other way. If $u \in \mathsf{ans}_P(\mathcal{Q}_e, \mathcal{D}, v_e)$, then there exists a function $\pi \in \mathsf{ans}(\mathcal{Q}_e, \mathcal{D})$ such that $\pi(v_e) = u$. But then π must also be in $\mathsf{ans}(\mathcal{Q}'_e, \mathcal{D})$, according to the second statement above, which means that u must be in $X_o(\mathcal{Q}, \tau)$. ∎

Given the set of productive values of a given extension pair τ, it is very easy to determine if an extension specification based on τ is productive or not: it is productive if the extension filter set contains at least one productive value.

Theorem 5.3.5. Let $\mathcal{Q}$ be a query, and let $\tau = (p_e, t_e)$ be an extension pair. An extension specification $\sigma = (p_e, t_e, X_e)$ based on τ is productive if and only if $X_e \cap X_o(\mathcal{Q}, \tau) \neq \emptyset$. $\dashv$

Proof. Let $X_e \in \mathcal{P}(\Gamma_v)$ be an arbitrary subset of Γ_v, let $\mathcal{Q}_e = \mathsf{ext}(\mathcal{Q}, (p_e, t_e, \Gamma_v), v_e)$, and let $\mathcal{Q}'_e = \mathsf{ext}(\mathcal{Q}, \sigma, v_e)$. Notice that $\mathcal{Q}'_e$ is identical to $\mathcal{Q}_e$, except for the filter set of v_e, where $\mathcal{Q}'_e$ is more restrictive. From this we know that the two following statements are true.

1. $\pi \in \mathsf{ans}(\mathcal{Q}'_e, \mathcal{D}) \implies \pi \in \mathsf{ans}(\mathcal{Q}_e, \mathcal{D})$.

2. If $\pi \in \mathsf{ans}(\mathcal{Q}_e, \mathcal{D})$ and $\pi(v_e) \in X_e$, then $\pi \in \mathsf{ans}(\mathcal{Q}'_e, \mathcal{D})$.

We start by proving that if σ is productive, then $X_e \cap X_o(\mathcal{Q}, \tau) \neq \emptyset$. σ is productive if and only if $\mathsf{ans}(\mathcal{Q}'_e, \mathcal{D}) \neq \emptyset$, and this only happens if there exists a function $\pi \in \mathsf{ans}(\mathcal{Q}'_e, \mathcal{D})$. Notice that $\pi(v_e) \in X_e$. Using the first statement above, we then know that $\pi \in \mathsf{ans}(\mathcal{Q}_e, \mathcal{D})$, which implies that $\pi(v_e) \in \mathsf{ans}_P(\mathcal{Q}_e, \mathcal{D}, v_e)$. Now we have proved that $\pi(v_e)$ is included in both X_e and $\mathsf{ans}_P(\mathcal{Q}_e, \mathcal{D}, v_e)$, hence $X_e \cap \mathsf{ans}_P(\mathcal{Q}_e, \mathcal{D}, v_e) = X_e \cap X_o(\mathcal{Q}, \tau) \neq \emptyset$.

To prove the statement in the other direction, we assume that $X_e \cap \mathsf{ans}_P(\mathcal{Q}_e, \mathcal{D}, v_e) = X_e \cap X_o(\mathcal{Q}, \tau) \neq \emptyset$. This means that there has to be at least one data value $u \in \Gamma_v$, which is included in both X_e, and $\mathsf{ans}_P(\mathcal{Q}_e, \mathcal{D}, v_e)$, which means that there exists a function $\pi \in \mathsf{ans}(\mathcal{Q}_e, \mathcal{D})$ such that $\pi(v_e) = u \in X_e$. From the second statement above, we then know that $\pi \in \mathsf{ans}(\mathcal{Q}'_e, \mathcal{D})$. Hence, $\mathsf{ans}(\mathcal{Q}'_e, \mathcal{D}) \neq \emptyset$, which means that σ is productive. ■

Example 5.3.6. Consider query $\mathcal{Q}_1$ from Example 5.1.6, and the extension pair $\tau = (age, Integer)$. The set of productive values of τ equals $X_o(\mathcal{Q}, \tau) = \{11, 21, 30, 35, 45\}$. ♦

The set of productive values $X_o(\mathcal{Q}, \tau)$ of an extension pair $\tau = (p_e, t_e)$ can either be used by the system to generate extension specifications, which the user can select from, or they can be used to validate one or more extensions suggested by the user. For example, if the system for some reason only wants to allow singleton filters, then it should make a drop-down list with one element for each productive value. Each item in this drop-down list would correspond to an extension $(p_e, t_e, \{u\})$, where u is a productive value of τ. The productive values could also be used to guide auto-completion in text-fields, to validate filter forms, to generate checkboxes, or to set upper and lower bounds on range-sliders. A special case occurs when there are no productive values for an extension pair τ, i.e., when $X_o(\mathcal{Q}, \tau) = \emptyset$. When this happens, the system should not suggest any extensions based on τ, since all of them will be dead-ends. Extension specifications can be generated or selected in many different ways, but if the system wants to detect dead-ends, i.e., detect unproductive extensions, it will almost always be an advantage to calculate the set of productive values first.

We know that we can get the full set of productive values $X_o(\mathcal{Q}, \tau)$ by executing the extended query $\mathcal{Q}_e = \mathsf{ext}(\mathcal{Q}, (p_e, t_e, \Gamma_v), v_e)$ over $\mathcal{D}$, and then project onto the extension variable v_e. But, there is no guarantee that this can be done fast enough for our needs since it requires querying over $\mathcal{D}$, so we need to consider other, faster solutions. In particular, we are going to look at efficient methods that return approximations of $X_o(\mathcal{Q}, \tau)$. We need a way to describe these approximations, which allows us to compare them to a perfect system, and this leads us over to *value functions*.

5.4 Value Functions

When we use the term *value function*, we refer to a function that attempts to calculate productive values based on an *extension case*, i.e., a partial query $\mathcal{Q}$ and a given legal extension pair $\tau \in J(\mathcal{Q}, \mathcal{N})$.

Definition 5.4.1 (Value Function). A *value function*, is a function S, which takes as input a query $\mathcal{Q}$ and a legal extension pair $\tau = (p_e, t_e) \in J(\mathcal{Q}, \mathcal{N})$ of $\mathcal{Q}$, and returns a set of data values $X \in \mathcal{P}(\Gamma_v)$. ⊣

5.4.1 Simple Value Functions

We now present three relatively simple value functions: the *productive value function* S_o, the *domain-based value function* S_d, and the *empty value function* S_e. The productive value function S_o returns the set of all productive values, the domain-based value function S_d returns the whole domain of possible data values: Γ_v, while the empty value function S_e returns just the empty set: $\emptyset$. All three value functions are defined formally below.

Definition 5.4.2 (The Productive Value Function: S_o). Let $\mathcal{Q}$ be a query, and let $\tau = (p_e, t_e) \in J(\mathcal{Q}, \mathcal{N})$ be a legal extension pair of $\mathcal{Q}$. The *productive value function*, denoted S_o, is defined as

$$S_o(\mathcal{Q}, \tau) = X_o(\mathcal{Q}, \tau)$$

⊣

Definition 5.4.3 (The Domain-based Value Function: S_d). Let $\mathcal{Q}$ be a query, and let $\tau = (p_e, t_e) \in J(\mathcal{Q}, \mathcal{N})$ be a legal extension pair of $\mathcal{Q}$. The *domain-based value function*, denoted S_d, is defined as

$$S_d(\mathcal{Q}, \tau) = \Gamma_v$$

⊣

Definition 5.4.4 (The Empty Value Function: S_e). Let $\mathcal{Q}$ be a query, and let $\tau = (p_e, t_e) \in J(\mathcal{Q}, \mathcal{N})$ be a legal extension pair of $\mathcal{Q}$. The *empty value function*, denoted S_e, is defined as

$$S_e(\mathcal{Q}, \tau) = \emptyset$$

⊣

Example 5.4.5. Let $\mathcal{Q}_3$ be the query in Figure 5.3, and let $\tau = (name, String) \in J(\mathcal{Q}_3, \mathcal{N})$ be a legal extension pair of $\mathcal{Q}_3$. Then the three value functions S_o, S_d, and S_e return the following three sets of values.

$$\begin{aligned}
S_o(\mathcal{Q}_3, \tau) &= X_o(\mathcal{Q}_3, \tau) = \{Alice, Bob, Robert\} \\
S_d(\mathcal{Q}_3, \tau) &= \Gamma_v \\
S_e(\mathcal{Q}_3, \tau) &= \emptyset
\end{aligned}$$

♦

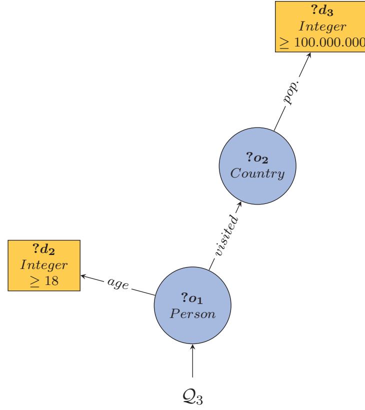


Figure 5.3: The query $\mathcal{Q}_3$.

5.4.2 Precision and Recall

We can evaluate a value function by measuring how close its set of suggested values is to the real set of productive values defined by X_o. This can be done with the two standard measures of precision and recall, where we consider $X_o(\mathcal{Q}, \tau)$ to be the set of relevant values.

Definition 5.4.6 (Precision)**.** Let S be a value function, and let $\tau \in J(\mathcal{Q}, \mathcal{N})$ be a legal extension pair of the query $\mathcal{Q}$. The *precision* of S with respect to $\mathcal{Q}$ and τ, denoted $\mathsf{prec}_{\mathrm{C}}(S, \mathcal{Q}, \tau)$, is the fraction of values returned by S that are productive, i.e.,

$$\mathsf{prec}_{\mathrm{C}}(S, \mathcal{Q}, \tau) = \frac{|S(\mathcal{Q}, \tau) \cap X_o(\mathcal{Q}, \tau)|}{|S(\mathcal{Q}, \tau)|}$$

⊣

Definition 5.4.7 (Recall)**.** Let S be a value function and let $\tau \in J(\mathcal{Q}, \mathcal{N})$ be a legal extension pair of the query $\mathcal{Q}$. The *recall* of S with respect to $\mathcal{Q}$ and τ, denoted $\mathsf{rec}(S, \mathcal{Q}, \tau)$, is the fraction of productive values that are also returned by S, i.e.,

$$\mathsf{rec}(S, \mathcal{Q}, \tau) = \frac{|S(\mathcal{Q}, \tau) \cap X_o(\mathcal{Q}, \tau)|}{|X_o(\mathcal{Q}, \tau)|}$$

⊣

The precision and recall of the three value functions presented so far: S_o, S_d, S_e, are calculated below.

$$\mathsf{prec}_{\mathrm{C}}(S_o, \mathcal{Q}, \tau) = \frac{|X_o(\mathcal{Q}, \tau) \cap X_o(\mathcal{Q}, \tau)|}{|X_o(\mathcal{Q}, \tau)|} = \frac{|X_o(\mathcal{Q}, \tau)|}{|X_o(\mathcal{Q}, \tau)|} = 1$$
$$\mathsf{rec}(S_o, \mathcal{Q}, \tau) = \frac{|X_o(\mathcal{Q}, \tau) \cap X_o(\mathcal{Q}, \tau)|}{|X_o(\mathcal{Q}, \tau)|} = \frac{|X_o(\mathcal{Q}, \tau)|}{|X_o(\mathcal{Q}, \tau)|} = 1$$

$$\mathsf{prec}_{\mathrm{C}}(S_d, \mathcal{Q}, \tau) = \frac{|\Gamma_v \cap X_o(\mathcal{Q}, \tau)|}{|\Gamma_v|} = \frac{|X_o(\mathcal{Q}, \tau)|}{|\Gamma_v|} \approx 0$$
$$\mathsf{rec}(S_d, \mathcal{Q}, \tau) = \frac{|\Gamma_v \cap X_o(\mathcal{Q}, \tau)|}{|X_o(\mathcal{Q}, \tau)|} = \frac{|X_o(\mathcal{Q}, \tau)|}{|X_o(\mathcal{Q}, \tau)|} = 1$$

$$\mathsf{prec}_{\mathrm{C}}(S_e, \mathcal{Q}, \tau) = \frac{|\emptyset \cap X_o(\mathcal{Q}, \tau)|}{|\emptyset|} = \frac{|\emptyset|}{|\emptyset|} = 1$$
$$\mathsf{rec}(S_e, \mathcal{Q}, \tau) = \frac{|\emptyset \cap X_o(\mathcal{Q}, \tau)|}{|X_o(\mathcal{Q}, \tau)|} = \frac{|\emptyset|}{|X_o(\mathcal{Q}, \tau)|} = 0$$

S_o gets perfect score on both precision and recall. This is not a surprise since it is defined to return exactly the data values we need. S_d returns all of Γ_v, which is too much in all non-trivial cases. This is reflected by its precision, which is the fraction of values in Γ_v that are productive. Since we assume that Γ_v is a relatively large set, maybe even infinitely large (if it contains e.g. all strings or all real numbers), while the set of productive values is finite and often relatively limited, the precision of S_d will in many cases be close to 0. However, since all productive values are in Γ_v, the recall of S_d is perfect. S_e, on the other hand, returns just the empty set, which results in perfect precision, but a recall of 0.

While S_o is the function that returns exactly what we want, S_d and S_e are the two extreme cases with respect to precision and recall: S_d suggests everything that can be suggested, while S_e does not suggest anything at all. S_d and S_e are interesting from an analytical perspective, but they should not be used to suggest values in a real-life system because this would lead to terrible extension suggestions: S_d would accept any extension, while S_e would instead reject them all.

Type I and Type II Errors In general, there are two types of errors that can occur when the system attempts to predict whether an extension σ is productive or not:

- **Type I:** The system classifies σ to be productive, but in reality it is unproductive.
- **Type II:** The system classifies σ to be unproductive, but in reality it is productive.

Both of these two types of errors have some unfortunate effects, which users of systems with perfect dead-end detection never experience: Type I errors may lead to some undetected dead-ends, and this may cause the user to make a query that they believe is productive, but which later appears to be unproductive when it is executed over the dataset, and no results are returned. A Type II error, on the other hand, may lead to a situation where a productive extension is classified as a dead-end. This may either discourage or prevent the user completely from

selecting this extension, which causes the user to miss out on interesting parts of the dataset.

These two types of errors are directly connected to the precision and recall of the value function S used by the system. To be more precise, in order to prevent Type I errors, S needs perfect precision, and in order to prevent Type II errors, S needs perfect recall. For example, if our system uses S_d to calculate productive values, it will, according to Theorem 5.3.5, always classify σ as productive. This will likely cause some Type I errors, but on the other hand, since the recall of S_d is perfect, all Type II errors will be completely eliminated. Since Type II errors in the worst case prevent the user from making queries and access the data they want, we consider them to be inferior to Type I errors. Therefore, we are going to focus mostly on value functions with perfect recall in the remainder of this thesis.

Precision of Multiple Extension Cases So far we have only defined the precision of a value function S over a single extension case, where a particular query $\mathcal{Q}$ and extension pair τ is given (see Definition 5.4.6). But a value function is not only going to be used once – it will be used multiple times in different cases over multiple sessions. Hence, to get an idea of how useful S is in general, we must calculate the average, or a similar aggregated value, over a set of cases that are representative for those cases where S will be used.

Representative cases can, for example, be extracted from a log of completed queries made in earlier query sessions. This assumes that future queries are built under the same conditions as the queries from the log, i.e., it assumes that the user base, the navigation graph, and the dataset have not changed significantly. There are multiple ways of extracting cases from a query log, and below we consider some of them, which we will make use of later in the thesis when we evaluate more sophisticated value functions.

We start by considering exactly one rooted query $\mathcal{Q}$ that has been made by a user earlier. The simplest way of generating cases from $\mathcal{Q}$ is to pair it with every possible extension pair in $J(\mathcal{Q}, \mathcal{N})$. We can average over all these cases to get the following precision formula:

$$\frac{1}{|J(\mathcal{Q}, \mathcal{N})|} \sum_{\tau \in J(\mathcal{Q}, \mathcal{N})} \mathsf{prec}_{\mathrm{C}}(S, \mathcal{Q}, \tau)$$

This way of generating cases assumes that the user has already made the query $\mathcal{Q}$ and that they need suggestions on how to extend it further, but this is not the case since $\mathcal{Q}$ in this formula refers to the final version made by the user.

A better approach is therefore to focus on the already existing outgoing data edges of the root $v_r = \mathsf{root}(\mathcal{Q})$. By removing such an edge $e = (v_r, p, v)$ and its corresponding data variable v, we get the resulting query $\mathsf{del}(\mathcal{Q}, v)$. We can then generate a case by pairing this query with the extension pair corresponding to the edge and variable that was just removed, i.e., $\tau = (p, T_{\mathcal{Q}}(v))$. When we do this for all the outgoing data edges of v_r, we get the following precision formula for S over a rooted query $\mathcal{Q}$.

Definition 5.4.8 (Precision over Rooted Query). Let S be a value function, let $\mathcal{Q}$ be a rooted query, and let E_{v_r} be the set of all outgoing data edges of $v_r = \mathsf{root}(\mathcal{Q})$. The precision of S over $\mathcal{Q}$, denoted $\mathsf{prec}_{\mathrm{Q}}(S, \mathcal{Q})$, is defined as

$$\mathsf{prec}_{\mathrm{Q}}(S, \mathcal{Q}) = \frac{1}{|E_{v_r}|} \sum_{(v_r, p, v) \in E_{v_r}} \mathsf{prec}_{\mathrm{C}}(S, \mathsf{del}(\mathcal{Q}, v), (p, T_{\mathcal{Q}}(v))) \tag{5.6}$$

⊣

This way of calculating the precision over a single rooted query was used in the two first experiments we did as a part of the thesis project. Both of them are presented in Chapter 6.

We also need to calculate the precision of a value function over a whole query log where the queries are unrooted. An unrooted query can be turned into several rooted queries by focusing on each of the object variables in $\mathcal{Q}$, one at a time. If we now use Equation 5.6 to calculate a precision over each such rooted query, and then calculate the average of all these precisions, we give equal weight to each possible root in the query, not each possible extension edge. So instead of doing this, we use a slightly different way of calculating the precision of an unrooted query, where each possible extension edge is weighted equally:

Definition 5.4.9 (Precision over Unrooted Query). Let S be a value function and let $\mathcal{Q}$ be an unrooted query. The precision of S over $\mathcal{Q}$, denoted $\mathsf{prec}_{\mathrm{U}}(S, \mathcal{Q})$, is defined as

$$\mathsf{prec}_{\mathrm{U}}(S, \mathcal{Q}) = \frac{1}{|\bar{E}_d(\mathcal{Q})|} \sum_{(v_r, p, v) \in \bar{E}_d(\mathcal{Q})} \mathsf{prec}_{\mathrm{C}}(S, \mathcal{Q}_d, (p, T_{\mathcal{Q}}(v)))$$

where

$$\mathcal{Q}_d = \mathsf{del}(\mathsf{setRoot}(\mathcal{Q}, v_r), v)$$

⊣

We will represent a query log as a set of pairs $(w_i, \mathcal{Q}_i)$, where w_i is a number that specifies the significance of the unrooted query $\mathcal{Q}_i$. This weight can, for example, correspond to the frequency of a query in the log. So if two queries appear five and nine times, they get a weight of 5.0 and 9.0 respectively.

Definition 5.4.10 (Query Log). A *query log* $\mathcal{L}$ is a finite set of pairs $\mathcal{L} = \{(w_1, \mathcal{Q}_1), (w_2, \mathcal{Q}_2), \ldots, (w_k, \mathcal{Q}_k)\}$, where every $w_i \in \mathbb{R}_{\geq 0}$ is a non-negative real number and every $\mathcal{Q}_i$ is an unrooted query. ⊣

We can now calculate the weighted average over the precisions of each unrooted query in $\mathcal{L}$ to get a precision over the whole query log.

Definition 5.4.11 (Precision over Query Log). Let S be a value function, and let $\mathcal{L} = \{(w_1, \mathcal{Q}_1), (w_2, \mathcal{Q}_2), \ldots, (w_k, \mathcal{Q}_k)\}$ be a query log. The precision of S over $\mathcal{L}$ is defined as the weighted average of precisions for each of the queries in $\mathcal{L}$,

i.e.,

$$\mathsf{prec}_{\mathrm{L}}(S, \mathcal{L}) = \frac{\sum_{1 \le i \le k} w_i \cdot \mathsf{prec}_{\mathrm{U}}(S, \mathcal{Q}_i)}{\sum_{1 \le i \le k} w_i}$$

⊣

If two value functions S_1 and S_2 are designed such that S_2 is more precise than S_1 for all extension cases, then S_2 will also be more precise than S_1 when they both are evaluated over rooted queries, unrooted queries, and even query logs.

Theorem 5.4.12. Let S_1 and S_2 be two value functions. If $\mathsf{prec}_{\mathrm{C}}(S_1, \mathcal{Q}, \tau) \le \mathsf{prec}_{\mathrm{C}}(S_2, \mathcal{Q}, \tau)$ for every extension case of $\mathcal{Q}$ and τ, then all three of the following statements will be true:

$$\mathsf{prec}_{\mathrm{Q}}(S_1, \mathcal{Q}_r) \le \mathsf{prec}_{\mathrm{Q}}(S_2, \mathcal{Q}_r)$$
$$\mathsf{prec}_{\mathrm{U}}(S_1, \mathcal{Q}_u) \le \mathsf{prec}_{\mathrm{U}}(S_2, \mathcal{Q}_u)$$
$$\mathsf{prec}_{\mathrm{L}}(S_1, \mathcal{L}) \le \mathsf{prec}_{\mathrm{L}}(S_2, \mathcal{L})$$

For all rooted queries $\mathcal{Q}_r$, unrooted queries $\mathcal{Q}_u$, and query logs $\mathcal{L}$. ⊣

Proof. Since all of these three advanced precision metrics are just linear combinations of the precision of a basic case, they all must be true. ∎

5.4.3 Advanced Value Functions

We are now going to consider two more sophisticated value functions: the *range-based value function* S_r and the *local value function* S_l.

The Range-Based Value Function: S_r We start by defining the *range-based* value function S_r to be the value function that only suggests values that actually exist in $\mathcal{D}$. More precisely, it considers every instance in $\mathcal{D}$ typed to the root class t_r of $\mathcal{Q}$, and each of its connected edges that corresponds to the extension pair $\tau = (p_e, t_e)$. Each data value in the target position of such an edge will be suggested by S_r. For example, if the root class is $Person$, and the extension pair is $(name, String)$, then S_r will return the full list of all names that are assigned to at least one person in the dataset. The values produced by this process can also be obtained by calculating the projected answers of $\mathcal{Q}_r$ onto v_e, where $\mathcal{Q}_r$ is the query that consists of two variables: v_r and v_e, typed to t_r and t_e respectively, and one edge (v_r, p_e, v_e).

Definition 5.4.13 (The Range-based Value Function: S_r)**.** Let $\mathcal{Q}$ be a query with root variable v_r and let $\tau = (p_e, t_e) \in J(\mathcal{Q}, \mathcal{N})$. The *range-based value function*, denoted S_r, is defined as $S_r(\mathcal{Q}, \tau) = \mathsf{ans}_P(\mathcal{Q}_r, \mathcal{D}, v_e)$ where

$$\mathcal{Q}_r = ((\{v_r\}, \{v_e\}, \{(v_r, p, v_e)\}), \{v_r \mapsto t_r, v_e \mapsto t_e\})$$

⊣

Example 5.4.14. Let $\mathcal{Q}_3$ be the query in Figure 5.3, and let $\tau = (name, String) \in J(\mathcal{Q}_3, \mathcal{N})$. Then the value function S_r returns the following set of values:

$$S_r(\mathcal{Q}_3, \tau) = \{Alice, Bob, Robert, Carol, Dave, Eve\}$$

♦

This way of suggesting values is very simple. It does not consider anything from the actual query, except for the root class and the extension pair. Hence, given an extension pair, the list of suggested values will always be the same. The small query $\mathcal{Q}_r$ will probably return answers fast when executed over $\mathcal{D}$, but since the lists it suggests are static, it is better to calculate all these lists offline and store them in an index that guarantees fast lookup. In order to support every legal extension pair τ, this index then needs one list for each such pair τ.

The Local Value Function: S_l The next value function we are going to present is the *local value function* S_l. S_l modifies the extended query $\mathcal{Q}_e$ into a new version $\mathcal{Q}_l$, by removing every variable that is more than one edge away from the root of $\mathcal{Q}_e$, i.e., it keeps only the local properties. This pruned query can then be executed over $\mathcal{D}$, with projection on the extension variable, to get a set of values to suggest.

Definition 5.4.15 (The Local Value Function: S_l)**.** Let $\mathcal{Q}$ be a rooted query with root variable v_r, and let $\tau = (p_e, t_e) \in J(\mathcal{Q}, \mathcal{N})$. Let $\mathcal{Q}_e = \mathsf{ext}(\mathcal{Q}, (p_e, t_e, \Gamma_v), v_e)$. The *local value function*, denoted S_l is defined as:

$$S_l(\mathcal{Q}, \tau) = \mathsf{ans}_P(\mathcal{Q}_l, \mathcal{D}, v_e)$$

where $\mathcal{Q}_l$ is the subquery of $\mathcal{Q}_e$ that only contains v_r and all variables of $\mathcal{Q}_e$ that have v_r as parent. ⊣

Example 5.4.16. Let $\mathcal{Q}_3$ be the query in Figure 5.3, and let $\tau = (name, String) \in J(\mathcal{Q}_3, \mathcal{N})$. Then the value function S_l returns the following set of values

$$S_l(\mathcal{Q}_3, \tau) = \{Alice, Bob, Robert, Carol\}$$

♦

The result of the pruning done by S_l is a star-shaped query $\mathcal{Q}_l$. If $\mathcal{Q}$, and hence also $\mathcal{Q}_e$, is star-shaped, then $\mathcal{Q}_l$ will not remove anything from $\mathcal{Q}_e$, which means that $\mathcal{Q}_l$ will equal $\mathcal{Q}_e$. When this is the case, then S_l considers all parts of the query and will return exactly $X_o(\mathcal{Q}, \tau)$, just like S_o does. On the other hand, if $\mathcal{Q}_e$ is a very deep query, such that a significant amount of its variables are far away from the root, then $\mathcal{Q}_l$ will differ a lot from $\mathcal{Q}_e$, and the values it suggests will not be as precise.

It makes sense to compare S_l with faceted search over one class since both of them generate dead-ends based on only the properties connected directly to the root variable. They both require answers to star-shaped queries, but in FS these queries are always simple, while in our case, $\mathcal{Q}_l$ may not be simple. This

can be solved by further removing variables from $\mathcal{Q}_l$ until the query becomes simple. This may reduce the precision of S_l, but will be necessary if we want to use faceted search engines to calculate the answers of $\mathcal{Q}_l$ efficiently.

5.4.4 Comparison of Value Functions

We have now defined five different value functions in this section, all of which have different approaches to the task of calculating and suggesting values. They have already been used to suggest values over the same example case (see Example 5.4.5, Example 5.4.14, and Example 5.4.16), where they returned the following sets of values:

$$
\begin{aligned}
S_e(\mathcal{Q}_3, \tau) &= \emptyset \\
S_o(\mathcal{Q}_3, \tau) &= \{Alice, Bob, Robert\} \\
S_l(\mathcal{Q}_3, \tau) &= \{Alice, Bob, Robert, Carol\} \\
S_r(\mathcal{Q}_3, \tau) &= \{Alice, Bob, Robert, Carol, Dave, Eve\} \\
S_d(\mathcal{Q}_3, \tau) &= \Gamma_v
\end{aligned}
$$

Notice that each of the value functions in the list above returns a larger set of values than the value function above it. This is not a coincidence, this is true for any possible extension case.

Theorem 5.4.17. The five value functions S_e, S_o, S_l, S_r, and S_d generate sets of values that satisfy

$$\emptyset = S_e(\mathcal{Q}, \tau) \subseteq S_o(\mathcal{Q}, \tau) \subseteq S_l(\mathcal{Q}, \tau) \subseteq S_r(\mathcal{Q}, \tau) \subseteq S_d(\mathcal{Q}, \tau) = \Gamma_v \tag{5.7}$$

for any query $\mathcal{Q}$ and extension pair τ. ⊣

Proof. All value functions must return a subset of the data values in Γ_v, see Definition 5.4.1). Therefore, $S_o(\mathcal{Q}, \tau)$, $S_l(\mathcal{Q}, \tau)$, and $S_r(\mathcal{Q}, \tau)$ must all be both supersets of $S_e(\mathcal{Q}, \tau)$, and subsets of $S_d(\mathcal{Q}, \tau)$. To prove that $S_o(\mathcal{Q}, \tau) \subseteq S_l(\mathcal{Q}, \tau) \subseteq S_r(\mathcal{Q}, \tau)$, we can use the fact that each of these three sets can be expressed as the result of evaluating a query over $\mathcal{D}$, and projecting onto the variable v_e. In particular, we know that

$$
\begin{aligned}
S_o(\mathcal{Q}, \tau) &= \mathsf{ans}_P(\mathcal{Q}_e, \mathcal{D}, v_e) \\
S_l(\mathcal{Q}, \tau) &= \mathsf{ans}_P(\mathcal{Q}_l, \mathcal{D}, v_e) \\
S_r(\mathcal{Q}, \tau) &= \mathsf{ans}_P(\mathcal{Q}_r, \mathcal{D}, v_e)
\end{aligned}
$$

where $\mathcal{Q}_r \sqsubseteq \mathcal{Q}_l \sqsubseteq \mathcal{Q}_e$. By using Theorem 4.3.7, we then get

$$S_o(\mathcal{Q}, \tau) \subseteq S_l(\mathcal{Q}, \tau) \subseteq S_r(\mathcal{Q}, \tau)$$

■

Since both S_r and S_l always produce a superset of the values defined by S_o, they must both have perfect recall. They will also have non-perfect precision, unless they are able to produce the same set of values as S_o does, i.e.,

$$\mathsf{rec}(S_r, \mathcal{Q}, \tau) = \mathsf{rec}(S_l, \mathcal{Q}, \tau) = \mathsf{rec}(S_o, \mathcal{Q}, \tau) = 1.$$

$$\mathsf{prec}_{\mathrm{C}}(S_r, \mathcal{Q}, \tau) \leq \mathsf{prec}_{\mathrm{C}}(S_l, \mathcal{Q}, \tau) \leq \mathsf{prec}_{\mathrm{C}}(S_o, \mathcal{Q}, \tau) = 1$$

Efficiency So far, we have been focusing on the values produced by each of the value functions, but before these value functions can be used in a real-life system, we must ensure that they can produce values efficiently. In general, an algorithm is considered to be efficient if it scales well with respect to its input. This definition fits our case too: we consider our dead-end detection algorithms to be efficient if the time it takes to calculate dead-ends scales well with respect to the size of the navigation graph, the dataset, and the partial query. But, what really matters in practice, is that dead-ends are calculated fast enough for the user to take advantage of them (see Section 3.3). In other words, we are looking for value functions that can be calculated in less than a few seconds, and ideally in less 200ms [36].

Three of the value functions we have presented, S_r, S_l, and S_o, all depend on the answers of $\mathsf{ans}_P(\mathcal{Q}_s, \mathcal{D}, v_e)$ for some query $\mathcal{Q}_s$, which means that they will be too slow when used on large databases, where complex queries with many joins may take minutes, or even hours. The two remaining value functions, S_e and S_d, on the other hand, are very efficient, since they simply return the same predefined set every time.

The only thing we actually need in order to calculate $S_r(\mathcal{Q}, \tau)$ is the root class t_r of the partial query $\mathcal{Q}$ and the extension pair τ. There is a limited number of possible such pairs (t_r, τ), and we can compute all pairs by considering every possible root class t_r in $\mathcal{N}$, and every legal extension pair of this class, given by $\tau = (p_e, t_e) \in J(t_r, \mathcal{N})$. For each such pair, we can construct $\mathcal{Q}_r$ as described in Definition 5.4.13, and then calculate the values that S_r should return for a query with root class t_r. All of this can be done offline, i.e., before any query session starts, and the computed sets of values for every pair (t_r, τ) can be stored in a small index. Now, given a partial query $\mathcal{Q}$ and an extension pair τ, $S_r(\mathcal{Q}, \tau)$ can be computed efficiently by looking up the set of values corresponding to the pair (t_r, τ) in the index. Essentially, we only need to loop over all instances of type t_r in the dataset to collect all the data values they are connected to with a property p_e. For example, if the root class t_r equals Person, and the extension pair τ is $(name, String)$, then we have to collect the set of all names given to a person in the dataset, and store this in the index.

It is also possible to construct an index that will make S_l efficient, but this index will be more expensive. $\mathcal{Q}_l$ is made by removing all parts of $\mathcal{Q}$ that are not connected directly to the root, i.e., $\mathcal{Q}_l$ is a star-shaped query. This is exactly the type of queries that can be answered by a standard faceted search index, so if we make one such index for each possible root class $t_r \in \bar{V}_o(\mathcal{N})$ and define $J(t_r, \mathcal{N})$

S	Precision	Recall	Eff. w.o. Index	Index Size
S_e	1	0	Instant	0
S_o	1	1	Slow	∞
S_l	Medium	1	Slow	Medium
S_r	Low	1	Slow	Small
S_d	≈ 0	1	Instant	0

Table 5.1: Summary of the five value functions S_d, S_r, S_l, S_o, and S_e.

to be the set of facets, then we have a way to calculate $S_r(\mathcal{Q}, \tau)$ efficiently for all $\mathcal{Q}$ and $\tau \in J(t_r, \mathcal{N})$.

Since $\mathcal{Q}$ is unbounded in size, $\mathcal{Q}_e$ from Theorem 5.3.4 will also be unbounded, which means that it is impossible to make a finite index that can be used to calculate $S_o(\mathcal{Q}, \tau)$ efficiently. This is exactly the same conclusion we got when we discussed dead-ends in Section 3.3.

Table 5.1 presents a summary of the five value functions we have considered in this chapter. S_d and S_e can both be calculated very efficiently, but the sets of values they produce are not useful. S_o has perfect precision and recall, but it is too slow in the worst-case scenario, and it is not possible to improve its efficiency with an index, because this index would need to be infinitely large. Both S_l and S_r are also inefficient, but it is possible to make them efficient by using finite indices. Without any knowledge about the concrete dataset and extension case, it is impossible to quantify the precision and required index size of S_l and S_r. The only thing we know, which we proved above, is that S_l is more precise than S_r and that it needs a larger index than S_r.

Notice how both S_l and S_r are pruning away certain parts of the partial query $\mathcal{Q}_e$ before running the remaining query over $\mathcal{D}$ to find the values to return. This pruning process reduces their precisions, but it also allows them to be computed efficiently with a precomputed index. In the next chapter, we generalize this idea, and we present a framework that enables us to construct value functions based on arbitrary pruning boundaries.

Chapter 6

The Index-Based Extension Framework

In the previous chapter, we defined general query extensions, and how one can easily detect or generate dead-end extensions among such extensions by first calculating the set of productive values $X_o(\mathcal{Q}, \tau)$ for a given partial query $\mathcal{Q}$ and extension pair τ. Unfortunately, this set of productive values can in general not be computed efficiently. However, it can be approximated efficiently, and we demonstrated this by presenting five different value functions with an analysis of their accuracy and efficiency (see Section 5.4).

Two of these value functions, S_r and S_l, use similar approaches to generate values: instead of returning the full set of productive values, given by $\mathsf{ans}_P(\mathcal{Q}_e, \mathcal{D}, v_e)$ they replace $\mathcal{Q}_e$ with a query $\mathcal{Q}_s \sqsubseteq \mathcal{Q}_e$, and return $\mathsf{ans}_P(\mathcal{Q}_s, \mathcal{D}, v_e)$. S_r and S_l use two different ways of determining what to remove from $\mathcal{Q}_e$: S_r removes everything except for the root and the extension variable, while S_l removes everything that is not connected directly to the root variable. The result of doing this kind of replacement is a value function that returns a superset of all productive values. This value function will then have perfect recall, but a precision that is less than perfect in most cases. The process of pruning $\mathcal{Q}_e$ may seem like a bad approach since it reduces the precision of the results. But, by doing it, the value function defines a boundary on the type of queries that needs to be answered, and this makes it possible to construct an index based on $\mathcal{D}$ that ensures efficiency of the value function.

We are going to generalize this idea, and consider other value functions that prune $\mathcal{Q}_e$ into such a subquery $\mathcal{Q}_s$. More precisely, in this chapter, we will define a framework that allows us to generate value functions based on a description of which parts of $\mathcal{Q}_e$ to keep. We will represent such a description by a simple, filterless query $\mathcal{Z}$, called a *configuration query* since it needs to be specified during the configuration phase of the VQS. The value function defined by $\mathcal{Z}$, denoted $S_a^{\mathcal{Z}}$, will then return the values given by $\mathsf{ans}_P(\mathcal{Q}_s, \mathcal{D}, v_e)$, where $\mathcal{Q}_s$ is the subquery of $\mathcal{Q}_e$ that remains after it has been pruned with respect to $\mathcal{Z}$. Furthermore, we will describe how to set up an index $\mathcal{I}_{\mathcal{Z}}$ based on $\mathcal{Z}$ and $\mathcal{D}$, which will be able to efficiently provide answers to $\mathcal{Q}_s$, regardless of how complex the extension query $\mathcal{Q}_e$ is. This index has to be set up during the configuration phase of the VQS and will be used to support multiple sessions.

6.1 The Configuration-based Value Function: $S_a^{\mathcal{Z}}$

In this section we formally define configuration queries, and how each such configuration query $\mathcal{Z}$ can be used to generate a concrete value function $S_a^{\mathcal{Z}}$.

6.1.1 Configuration Queries

A configuration query $\mathcal{Z}$ is a special type of query that has to be defined during the configuration phase of the VQS, i.e., outside all of the query sessions, where the system only has access to $\mathcal{N}$ and $\mathcal{D}$, but not the partial query $\mathcal{Q}$. $\mathcal{Z}$ has two main purposes:

- $\mathcal{Z}$ will be used to set up an index $\mathcal{I}_{\mathcal{Z}}$ during the configuration phase, based on the data of $\mathcal{D}$.
- $\mathcal{Z}$ defines a value function $S_a^{\mathcal{Z}}$, which can be computed efficiently if the system has access to $\mathcal{I}_{\mathcal{Z}}$.

Formally, a configuration query is a simple, filterless query that is legal with respect to $\mathcal{N}$.

Definition 6.1.1 (Configuration Query)**.** A *configuration query* $\mathcal{Z}$ is a simple, legal, rooted, filterless query $\mathcal{Z} = (R_{\mathcal{Z}}, T_{\mathcal{Z}})$. ⊣

Since a configuration query is just a special type of rooted query, every definition we have presented so far related to rooted queries will also apply to configuration queries. But, configuration queries will have a special role in our system, and there will be a need to distinguish them from other more general queries. In general, when we use the term *configuration query*, we refer to the query used to set up the system, which has to be constructed or generated by an administrator before query sessions can even start. We will never use the term configuration query when we refer to a query made during a query session by an end-user, even when it satisfies the requirements in the above definition. Actually, it would have been possible to present and use tree-shaped configurations without defining them as queries, but since we want to compare configurations and queries later, it is convenient that they are defined in the same way.

Now, we define a query $\mathcal{Q}_s$ to be *covered* by a configuration query $\mathcal{Z}$ if it has a similar, but smaller graph structure than $\mathcal{Z}$. Below we formally define what we mean by this.

Definition 6.1.2 (Query Covered by Configuration)**.** Let $\mathcal{Q}_s = (R_{\mathcal{Q}_s}, T_{\mathcal{Q}_s}, F_{\mathcal{Q}_s})$ be a rooted query, and let $\mathcal{Z}$ be a configuration query. $\mathcal{Q}_s$ is *covered* by $\mathcal{Z}$ if there exists a configuration query $\mathcal{Z}_s$ such that:

- $\mathcal{Z}_s \sqsubseteq \mathcal{Z}$
- $\mathcal{Z}_s$ is a renaming of $(R_{\mathcal{Q}_s}, T_{\mathcal{Q}_s})$

⊣

The renaming from $(R_{\mathcal{Q}_s}, T_{\mathcal{Q}_s})$ to $\mathcal{Z}_s$ in the definition above is in fact just a renaming from $\mathcal{Q}_s$ to $\mathcal{Z}_s$ where the filters are ignored. We call such a renaming a *filter-ignorant* renaming.

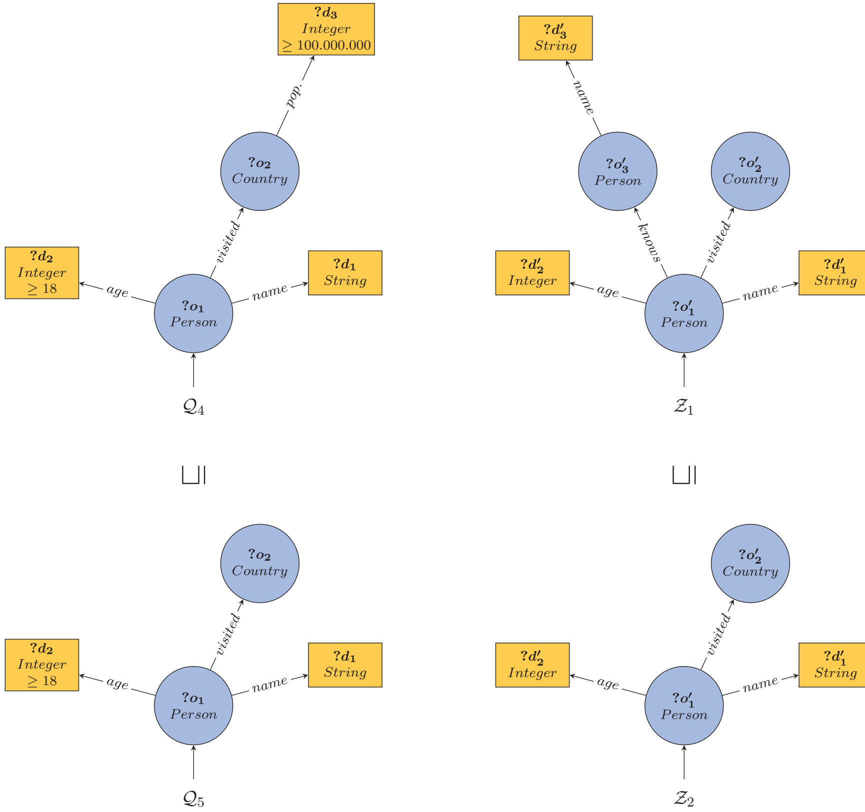


Figure 6.1: Two queries, $\mathcal{Q}_4$ and $\mathcal{Q}_5$, to the left, and two configuration queries, $\mathcal{Z}_1$, and $\mathcal{Z}_2$ to the right. $\mathcal{Q}_4$ is not covered by $\mathcal{Z}_1$, but its subquery $\mathcal{Q}_5$ is covered by $\mathcal{Z}_1$. $\mathcal{Z}_2$ is the subquery of $\mathcal{Z}_1$ that corresponds to $\mathcal{Q}_5$.

Example 6.1.3. Figure 6.1 displays two queries to the left: $\mathcal{Q}_4$ and $\mathcal{Q}_5$, and two configuration queries to the right: $\mathcal{Z}_1$ and $\mathcal{Z}_2$. $\mathcal{Q}_4$ is not covered by $\mathcal{Z}_1$, because $\mathcal{Q}_4$ contains a variable referring to the population of the visited country, while $\mathcal{Z}_1$ does not. However, $\mathcal{Q}_5$, which is a subquery of $\mathcal{Q}_4$, is indeed covered by $\mathcal{Z}_1$, since $\mathcal{Z}_1$ contains variables corresponding to all of the four variables of $\mathcal{Q}_5$ (a person with name, age, and visited country). To prove formally that $\mathcal{Q}_5$ is covered by $\mathcal{Z}_1$, we must show that there exists a subquery of $\mathcal{Z}_1$ that is also a filter-ignorant renaming of $\mathcal{Q}_5$. $\mathcal{Z}_2$ is such a configuration query. ♦

In order for a query $\mathcal{Q}_s$ to be covered by a configuration query $\mathcal{Z}$, each of its variables must correspond to a particular variable in $\mathcal{Z}$, given by the filter-ignorant renaming from $\mathcal{Q}_s$ to $\mathcal{Z}_s$, which is then also a filter-ignorant renaming from $\mathcal{Q}_s$ to some of the variables in $\mathcal{Z}$. This renaming preserves the type of each variable in $\mathcal{Q}_s$, including the root variable, so, if the root classes of $\mathcal{Q}_s$ and $\mathcal{Z}$

differ, then $\mathcal{Q}_s$ can never be covered by $\mathcal{Z}$.

Theorem 6.1.4 (Unique $\mathcal{Z}_s$)**.** The configuration query $\mathcal{Z}_s$, and the filter-ignorant renaming function from $(R_{\mathcal{Q}_s}, T_{\mathcal{Q}_s})$ to $\mathcal{Z}_s$ in Definition 6.1.2 is unique. In addition, the query $\mathcal{Q}_s$ must be simple. ⊣

Proof. Assume that there are two distinct configuration queries $\mathcal{Z}'_s$ and $\mathcal{Z}''_s$ satisfying the two requirements from the theorem. Since they are both filter-ignorant renamings of $(R_{\mathcal{Q}_s}, T_{\mathcal{Q}_s})$, there has to be a renaming $f_r \colon \bar{V}(\mathcal{Z}'_s) \to \bar{V}(\mathcal{Z}''_s)$ from $\mathcal{Z}'_s$ to $\mathcal{Z}''_s$. We will use structural induction to prove that $f_r(v) = v$ for each varible $v \in \bar{V}(\mathcal{Z}'_s)$, which then shows that $\mathcal{Z}'_s$ must be identical to $\mathcal{Z}''_s$. Base case: every renaming maps the root of the source query to the root of the target query, hence $f_r(\mathsf{root}(\mathcal{Z}'_s)) = \mathsf{root}(\mathcal{Z}''_s)$. Inductive step: assume that $f_r(v) = v$ for some variable $v \in \bar{V}(\mathcal{Z}'_s)$, and assume that $e = (v, p, v_c) \in \bar{E}(\mathcal{Z}'_s)$ is an edge going out from v. If we define $v'_c = f_r(v_c)$, then edge $e' = (v, p, v'_c)$ must be in $\bar{E}(\mathcal{Z}''_s)$, but since both $\mathcal{Z}'_s$ and $\mathcal{Z}''_s$ are subqueries of $\mathcal{Z}$, then both e and e' are also in $\mathcal{Z}$, which is only possible if $e = e'$, and hence $v_c = v'_c$, since $\mathcal{Z}$ is simple.

We can prove that the filter-ignorant renaming function from $(R_{\mathcal{Q}_s}, T_{\mathcal{Q}_s})$ to $\mathcal{Z}_s$ must be unique in a similar fashion, again by using structural induction. Assume that there are two filter-ignorant renaming functions, f'_r and f''_r, from $(R_{\mathcal{Q}_s}, T_{\mathcal{Q}_s})$ to $\mathcal{Z}_s$. We need to prove that $f'_r(v) = f''_r(v)$ for each $v \in \bar{V}(R_{\mathcal{Q}_s})$. Base case: both f'_r and f''_r map the root of $(R_{\mathcal{Q}_s}, T_{\mathcal{Q}_s})$ to the root of $\mathcal{Z}_s$. Inductive step: assume that $v' = f'_r(v) = f''_r(v)$ for some variable $v \in \bar{V}(R_{\mathcal{Q}_s})$, and assume that $e = (v, p, v_c) \in \bar{E}(R_{\mathcal{Q}_s})$ is an edge going out from v. If $f'_r(v_c) = v'_c$, while $f''_r(v_c) = v''_c$, then both (v', p, v'_c) and (v', p, v''_c) must be edges in $\mathcal{Z}_s$, which is impossible unless $v'_c = v''_c$, since $\mathcal{Z}_s$ is simple.

Since $\mathcal{Q}_s$ is a filter-ignorant renaming of $\mathcal{Z}_s$, which is simple because it is a subquery of the configuration query $\mathcal{Z}$, $\mathcal{Q}_s$ must also be simple. ∎

The reason why we are particularly interested in queries covered by $\mathcal{Z}$ is because these queries can be answered efficiently by the index $\mathcal{I}_{\mathcal{Z}}$ defined by $\mathcal{Z}$ (see Section 6.2), and they will have a central role in the new value function we are going to define in the next few paragraphs.

If a query $\mathcal{Q}$ is not covered by the configuration $\mathcal{Z}$ but has the same root class, then it is always possible to prune $\mathcal{Q}$, i.e., remove some of its branches, to make a smaller query, $\mathcal{Q}_s$, that is covered by $\mathcal{Z}$. In the most extreme case, $\mathcal{Q}$ must be pruned down to only the root variable before it is covered by $\mathcal{Z}$, but often it is possible to just prune relatively small parts of $\mathcal{Q}$ in order to get coverage. In Example 6.1.3, we considered query $\mathcal{Q}_4$, which is not covered by $\mathcal{Z}_1$. But, the query $\mathcal{Q}_5$, which we get by pruning away variable $?d_3$ from $\mathcal{Q}_4$, is indeed covered by $\mathcal{Z}_1$. It would also be possible to remove any subset of the three variables $?d_1$, $?d_2$, and $?o_2$ from $\mathcal{Q}_5$, and still have a query covered by $\mathcal{Z}_1$. Even the smallest possible subquery of $\mathcal{Q}_4$, consisting of only the root variable, is a query covered by $\mathcal{Z}_1$.

Given a query $\mathcal{Q}$ and a configuration query $\mathcal{Z}$, we define a pruned version of $\mathcal{Q}$ with respect to $\mathcal{Z}$ to be a subquery of $\mathcal{Q}$ that is covered by $\mathcal{Z}$. There may be many such queries, and they define a set, which we will denote $\mathsf{prune}(\mathcal{Q}, \mathcal{Z})$.

Definition 6.1.5 (Query Pruning)**.** Let $\mathcal{Q}$ be a query and let $\mathcal{Z}$ be a configuration query. A *pruned version of* $\mathcal{Q}$ *with respect to* $\mathcal{Z}$, is a query $\mathcal{Q}_s \sqsubseteq \mathcal{Q}$ that is covered by $\mathcal{Z}$. The set of all pruned versions of $\mathcal{Q}$ with respect to $\mathcal{Z}$ is denoted $\mathsf{prune}(\mathcal{Q}, \mathcal{Z})$. ⊣

We will also need a variant of the pruning function that preserves a given variable v connected to the root of $\mathcal{Q}$.

Definition 6.1.6 (Variable-Preserving Pruning)**.** Let $\mathcal{Q}$ be a query and let $\mathcal{Z}$ be a configuration query. The set of all pruned versions of $\mathcal{Q}$ with respect to $\mathcal{Z}$, where the variable $v \in \bar{V}(\mathcal{Q})$ is preserved, denoted $\mathsf{prune}(\mathcal{Q}, \mathcal{Z}, v)$, is defined as

$$\mathsf{prune}(\mathcal{Q}, \mathcal{Z}, v) = \{\mathcal{Q}_s \in \mathsf{prune}(\mathcal{Q}, \mathcal{Z}) \mid v \in \bar{V}(\mathcal{Q}_s)\}$$

⊣

Later, we will compare configuration queries with queries that have a different root class, and when this is the case, then both $\mathsf{prune}(\mathcal{Q}, \mathcal{Z})$ and $\mathsf{prune}(\mathcal{Q}, \mathcal{Z}, v)$ will be empty. But for now, we are going to focus mostly on configuration queries and queries with the same root class.

Example 6.1.7. Query $\mathcal{Q}_4$ in Figure 6.1 has a total of eight subqueries that are covered by $\mathcal{Z}_1$, so $\mathsf{prune}(\mathcal{Q}_4, \mathcal{Z}_1)$ contains eight queries. $\mathsf{prune}(\mathcal{Q}_4, \mathcal{Z}_1, ?d_1)$ contains every query in $\mathsf{prune}(\mathcal{Q}_4, \mathcal{Z}_1)$ that includes $?d_1$, and there are four such queries. Both $\mathsf{prune}(\mathcal{Q}_4, \mathcal{Z}_1)$ and $\mathsf{prune}(\mathcal{Q}_4, \mathcal{Z}_1, ?d_1)$ contain the query $\mathcal{Q}_5$, which is also the largest query in both these sets. ♦

6.1.2 The Configuration-based Value Function: $S_a^{\mathcal{Z}}$

Now, given the partial query $\mathcal{Q}$ and an extension pair $\tau = (p_e, t_e)$, we can formulate the extended query $\mathcal{Q}_e = \mathsf{ext}(\mathcal{Q}, (p_e, t_e, \Gamma_v), v_e)$ as before. If we prune $\mathcal{Q}_e$ with respect to $\mathcal{Z}$, while preserving the variable v_e, we get a set containing multiple subqueries of $\mathcal{Q}_e$. Each such subquery, $\mathcal{Q}_s$, corresponds to a set of data values given by $\mathsf{ans}_P(\mathcal{Q}_s, \mathcal{D}, v_e)$, which must be a superset of the productive values defined by $X_o(\mathcal{Q}, \tau)$, since $\mathcal{Q}_s \sqsubseteq \mathcal{Q}_e$. Hence, if we intersect the sets corresponding to each subquery $\mathcal{Q}_s$, we get a smaller set that is still a superset of $X_o(\mathcal{Q}, \tau)$. Since this final intersection is a relatively small superset of $X_o(\mathcal{Q}, \tau)$, it should make a relatively precise set of values, so we define the configuration-based value function based on $\mathcal{Z}$, denoted $S_a^{\mathcal{Z}}$, to return this set.

Definition 6.1.8 (The Configuration-based Value Function: $S_a^{\mathcal{Z}}$)**.** Let $\mathcal{Q}$ be a query, let $\tau = (p_e, t_e) \in J(\mathcal{Q}, \mathcal{N})$ be a legal extension pair of $\mathcal{Q}$, and let $\mathcal{Q}_e = \mathsf{ext}(\mathcal{Q}, (p_e, t_e, \Gamma_v), v_e)$. The *configuration-based value function* based on

the configuration query $\mathcal{Z}$, denoted $S_a^{\mathcal{Z}}$, is defined as

$$S_a^{\mathcal{Z}}(\mathcal{Q}, \tau) = \Gamma_v \cap \left(\bigcap_{\mathcal{Q}_s \in \mathsf{prune}(\mathcal{Q}_e, \mathcal{Z}, v_e)} \mathsf{ans}_P(\mathcal{Q}_s, \mathcal{D}, v_e) \right)$$

⊣

Notice that we only consider the pruned versions of $\mathcal{Q}_e$ where v_e is preserved. This is important because $\mathsf{ans}_P(\mathcal{Q}_s, \mathcal{D}, v_e)$ is undefined when v_e is not contained in $\mathcal{Q}_s$ – it is not possible to project onto a variable that does not exist in the query. It may seem pointless to intersect all the projected answers with Γ_v since $\mathsf{ans}_P(\mathcal{Q}_s, \mathcal{D}, v_e)$ is always a subset of Γ_v, but we need it when $\mathcal{Z}$ does not cover the extension pair τ, i.e., when there is no variable in $\mathcal{Z}$ that corresponds to v_e. When this is the case, then $\mathsf{prune}(\mathcal{Q}_e, \mathcal{Z}, v_e)$ will be empty, and there will be no pruned versions of $\mathcal{Q}_e$ that can contribute with useful values. When this happens, then $S_a^{\mathcal{Z}}$ should fall back to just suggesting Γ_v. From a technical point of view, we are only going to allow the use of the nullary intersection, i.e., the intersection of zero sets, when it is intersected with Γ_v, and then we define it to equal Γ_v, i.e., we always define $\Gamma_v \cap (\bigcap_{A \in \emptyset} A) = \Gamma_v$. Suggesting Γ_v is not ideal, but the system cannot do anything better, since $\mathcal{Z}$, and its corresponding index $\mathcal{I}_{\mathcal{Z}}$ is not set up to efficiently return any better answers. The only way to prevent $S_a^{\mathcal{Z}}$ from returning Γ_v is to make sure that $\mathcal{Z}$ covers τ, and the only way to prevent it from returning Γ_v for any of the legal extension pairs $\tau \in J(\mathcal{Q}, \mathcal{N})$, is to make sure that $\mathcal{Z}$ covers all of them. Since the root class of the partial query changes as the user changes focus, one single configuration query is not enough to prevent $S_a^{\mathcal{Z}}$ from returning Γ_v in every case. In fact, to cover every potential root class of $\mathcal{Q}$, and every possible extension pair of such a query, we need a set with multiple configuration queries, at least one for each class in $\mathcal{N}$. We will discuss this later in Section 6.4, where we extend S_a to take advantage of multiple configuration queries, but before that, we will need to make a proper analysis of the case with only one configuration query.

In general, when $\mathcal{Q}_s$ is a small query without many restrictions, the resulting set of values $\mathsf{ans}_P(\mathcal{Q}_s, \mathcal{D}, v_e)$ will be relatively large, and hence it will not affect the final set of values returned by $S_a^{\mathcal{Z}}$ so much. On the other hand, if $\mathcal{Q}_s$ is a large and more restrictive query, then $\mathsf{ans}_P(\mathcal{Q}_s, \mathcal{D}, v_e)$ will be a smaller set, which will likely have a higher impact on the set of returned values. In particular, if $\mathcal{Q}_s$ and $\mathcal{Q}_s'$ are two queries in $\mathsf{prune}(\mathcal{Q}_e, \mathcal{D}, v_e)$, and $\mathcal{Q}_s' \sqsubseteq \mathcal{Q}_s$, then we know from Definition 4.3.7 that $\mathsf{ans}_P(\mathcal{Q}_s, \mathcal{D}, v_e) \subseteq \mathsf{ans}_P(\mathcal{Q}_s', \mathcal{D}, v_e)$, which means that we can completely ignore $\mathcal{Q}_s'$, without changing the final results produced by $S_a^{\mathcal{Z}}$.

Now, consider the partially ordered set defined by the subquery relation over all the queries in $\mathsf{prune}(\mathcal{Q}_e, \mathcal{Z}, v_e)$. If $\mathcal{Z}$ covers the relevant extension pair, then $\mathsf{prune}(\mathcal{Q}_e, \mathcal{Z}, v_e)$ will at least contain the query consisting of only the root variable $v_r = \mathsf{root}(\mathcal{Q})$ and v_e. Since v_r and v_e are both required to be a part of $\mathsf{prune}(\mathcal{Q}_e, \mathcal{Z}, v_e)$, this query is a smallest query in the set, i.e., it is a subquery of every other query in $\mathsf{prune}(\mathcal{Q}_e, \mathcal{Z}, v_e)$. There will also be a set of maximal queries,

i.e., queries which are not a subquery of any other query in $\mathsf{prune}(\mathcal{Q}_e, \mathcal{Z}, v_e)$. But, since every $\mathcal{Q}_s \in \mathsf{prune}(\mathcal{Q}_e, \mathcal{D}, v_e)$ is either a maximal query, or a subquery of a maximal query, we actually only need to consider the maximal queries of $\mathsf{prune}(\mathcal{Q}_e, \mathcal{D}, v_e)$ when calculating the values in $S_a^{\mathcal{Z}}(\mathcal{Q}, \tau)$.

Theorem 6.1.9. Let $S_a^{\mathcal{Z}}$ be the value function defined in Definition 6.1.8, and let $\mathsf{prune}_M(\mathcal{Q}_e, \mathcal{D}, v_e)$ be the set of all maximal queries in $\mathsf{prune}(\mathcal{Q}_e, \mathcal{D}, v_e)$. Then

$$S_a^{\mathcal{Z}}(\mathcal{Q}, \tau) = \Gamma_v \cap \left(\bigcap_{\mathcal{Q}_s \in \mathsf{prune}_M(\mathcal{Q}_e, \mathcal{D}, v_e)} \mathsf{ans}_P(\mathcal{Q}_s, \mathcal{D}, v_e) \right)$$

⊣

Proof. This follows directly from the definition of maximal queries, Definition 4.3.7, and our arguments above. ■

From this theorem, it should be clear that if there exists a maximum query in $\mathsf{prune}(\mathcal{Q}_e, \mathcal{D}, v_e)$, i.e., a query $\mathcal{Q}_m$ such that $\mathcal{Q}_s \sqsubseteq \mathcal{Q}_m$ for all queries in $\mathsf{prune}(\mathcal{Q}_e, \mathcal{D}, v_e)$, then we can calculate the values of $S_a^{\mathcal{Z}}(\mathcal{Q}, \tau)$ only based on $\mathcal{Q}_m$, i.e., we get $S_a^{\mathcal{Z}}(\mathcal{Q}, \tau) = \mathsf{ans}_P(\mathcal{Q}_m, \mathcal{D}, v_e)$. For example, if $\mathcal{Q}_e$ is already covered by $\mathcal{Z}$, then $\mathsf{prune}(\mathcal{Q}_e, \mathcal{D}, v_e)$ will contain $\mathcal{Q}_e$ itself, which must then be both simple and maximum in $\mathsf{prune}(\mathcal{Q}_e, \mathcal{D}, v_e)$. This means that $S_a^{\mathcal{Z}}(\mathcal{Q}, \tau) = \mathsf{ans}_P(\mathcal{Q}_e, \mathcal{D}, \tau) = X_o(\mathcal{Q}, \tau)$. In other words, if the extended query $\mathcal{Q}_e$ is covered by $\mathcal{Z}$, then $S_a^{\mathcal{Z}}$ will return perfect results. If $\mathcal{Q}$ is a simple query, then there will be a maximum query in $\mathsf{prune}(\mathcal{Q}_e, \mathcal{Z}, v_e)$.

Theorem 6.1.10. If $\mathcal{Q}$ from Definition 6.1.8 is simple, then there exists a maximum query in $\mathsf{prune}(\mathcal{Q}_e, \mathcal{Z}, v_e)$, i.e., a query $\mathcal{Q}_m$ such that $\mathcal{Q}_s \sqsubseteq \mathcal{Q}_m$ for all $\mathcal{Q}_s \in \mathsf{prune}(\mathcal{Q}_e, \mathcal{Z}, v_e)$. ⊣

Proof. Let the union superquery of two queries $\mathcal{Q}'_s \sqsubseteq \mathcal{Q}_e$ and $\mathcal{Q}''_s \sqsubseteq \mathcal{Q}_e$, be the query $\mathcal{Q}_u$ that contains all variables and edges from both $\mathcal{Q}'_s$ and $\mathcal{Q}''_s$, and which uses the types and filters given by $\mathcal{Q}_e$, restricted to the included variables. Now, if both $\mathcal{Q}'_s$ and $\mathcal{Q}''_s$ are in $\mathsf{prune}(\mathcal{Q}_e, \mathcal{Z}, v_e)$, then we can prove that $\mathcal{Q}_u$ will also be in $\mathsf{prune}(\mathcal{Q}_e, \mathcal{Z}, v_e)$. The union superquery of all queries in $\mathsf{prune}(\mathcal{Q}, \mathcal{Z})$ will then be the maximal query $\mathcal{Q}_m$.

If $\mathcal{Z}'_s$ and $\mathcal{Z}''_s$ are the two configuration queries corresponding to $\mathcal{Q}'_s$ and $\mathcal{Q}''_s$ respectively, and f'_r, f''_r are the filter-ignorant renaming functions from $\mathcal{Q}'_s$ to $\mathcal{Z}'_s$, and $\mathcal{Q}''_s$ to $\mathcal{Z}''_s$ respectively, then the union superquery of $\mathcal{Z}'_s$ and $\mathcal{Z}''_s$, denoted $\mathcal{Z}_u$, will correspond to $\mathcal{Q}_u$, and by combining f'_r and f''_r, we get a filter-ignorant renaming function from $\mathcal{Q}_u$ to $\mathcal{Z}_u$. Since $\mathcal{Q}$ is simple, it is impossible to pick two different variables $v' \in \bar{V}(\mathcal{Q}'_s)$ and $v'' \in \bar{V}(\mathcal{Q}''_s)$ such that $f'_r(v') = f''_r(v'')$. This can happen if $\mathcal{Q}$ is non-simple. ■

Now, we will consider two examples where the two above theorems are used when calculating values based on S_a.

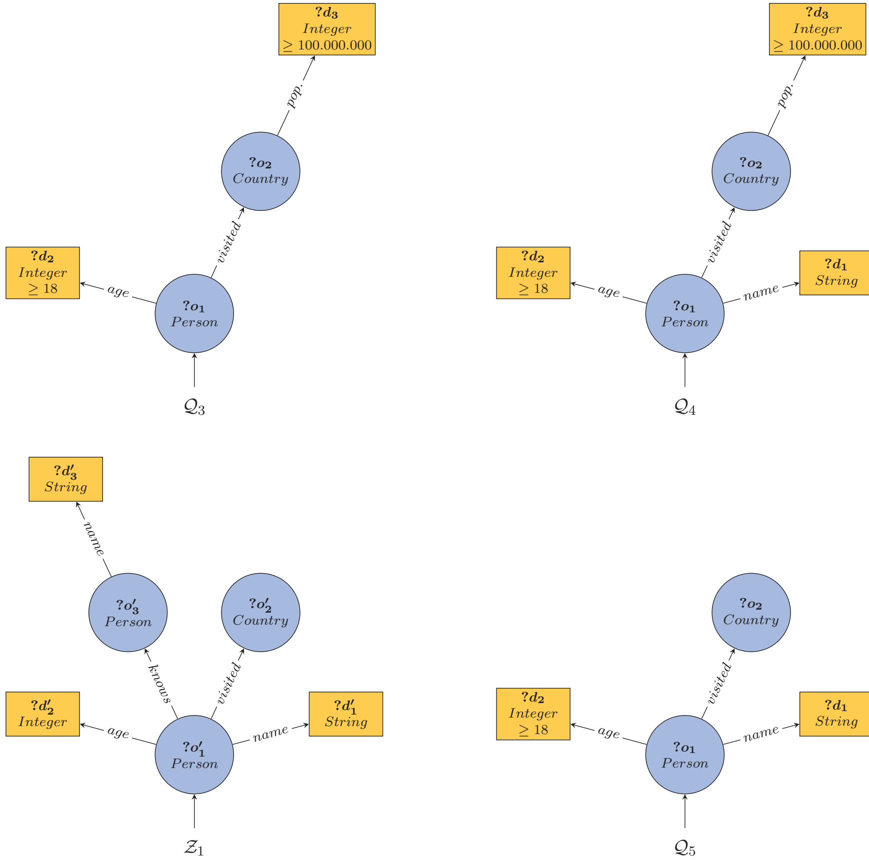


Figure 6.2: The partial query $\mathcal{Q}_3$, its extended version $\mathcal{Q}_4$, the configuration query $\mathcal{Z}_1$, and query $\mathcal{Q}_5$, which is the result after pruning $\mathcal{Q}_4$ with respect to $\mathcal{Z}_1$.

Example 6.1.11. Let us define a value function $S_a^{\mathcal{Z}_1}$ based on the configuration query $\mathcal{Z}_1$ in Figure 6.2, and let us go back to the case we considered in Example 5.4.5, where $\mathcal{Q}_3$ is the partial query, and values are requested for the extension pair $(name, String)$. We start by extending $\mathcal{Q}_3$ with $(name, String, \Gamma_v)$ into a the new variable $?d_1$, which gives us the extended query $\mathcal{Q}_4 = \mathsf{ext}(\mathcal{Q}_3, (name, String, \Gamma_v), ?d_1)$, depicted in Figure 6.2. Since $\mathcal{Q}_4$ is simple, there exists a maximum query in $\mathsf{prune}(\mathcal{Q}_4, \mathcal{Z}_1, ?d_1)$, and this query is $\mathcal{Q}_5$. Hence we do not need to consider any other of the queries in $\mathsf{prune}(\mathcal{Q}_4, \mathcal{Z}_1, ?d_1)$, and $S_a^{\mathcal{Z}_1}$ will return the following values:

$$S_a^{\mathcal{Z}_1}(\mathcal{Q}_3, (name, String)) = \mathsf{ans}_P(\mathcal{Q}_5, \mathcal{D}, ?d_1) = \{Alice, Bob, Robert, Carol\}$$

♦

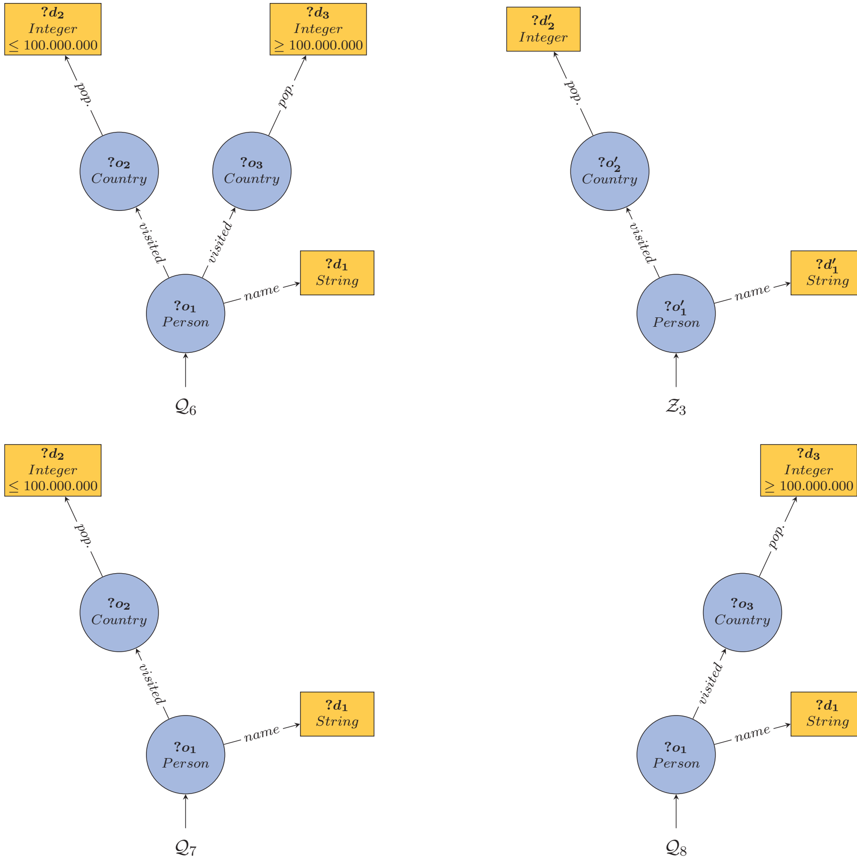


Figure 6.3: The two queries $\mathcal{Q}_7$ and $\mathcal{Q}_8$ are the only two maximal subqueries in $\mathsf{prune}(\mathcal{Q}_6, \mathcal{Z}_2)$.

If $\mathcal{Q}_e$ is not simple, there may be several maximal pruned versions of $\mathcal{Q}_e$, which all need to be evaluated using ans_P in order to calculate the full set of values to return. We demonstrate this in the example below.

Example 6.1.12. Consider the non-simple extended query $\mathcal{Q}_6$, which has just been extended with the variable $?d_1$, in Figure 6.3, and the configuration query $\mathcal{Z}_3$ next to it. There are two maximal queries in $\mathsf{prune}(\mathcal{Q}_6, \mathcal{Z}_3, ?d_1)$: $\mathcal{Q}_7$ and $\mathcal{Q}_8$. We calculate the projected values given by each of them:

$$\mathsf{ans}_P(\mathcal{Q}_7, \mathcal{D}, ?d_1) = \{Bob, Robert, Carol\}$$
$$\mathsf{ans}_P(\mathcal{Q}_8, \mathcal{D}, ?d_1) = \{Alice, Bob, Robert, Eve\}$$

By intersecting these two sets, we get the values generated by $S_a^{\mathcal{Z}_3}$: $\{Bob, Robert\}$. ♦

In some cases, it is not even necessary to check all maximal queries of $\mathsf{prune}(\mathcal{Q}_e, \mathcal{D}, v_e)$. For example, if two maximal queries are renamings of each other, they will evaluate to the same set of values, hence we do not need to calculate ans_P for more than one of them. Another case that can occur, is that two maximal queries have the same overall shape, but they differ on the filters applied to two corresponding variables, and one of the filters is more restrictive than the other. For example, if the filter set on $?d_2$ was changed to $\{u \in \Gamma_v \mid u \geq 200.000.000\}$ in query $\mathcal{Q}_6$, and hence also in $\mathcal{Q}_7$, then $\mathcal{Q}_7$ would become an overall more restrictive query than $\mathcal{Q}_8$, so there would be no need to calculate $\mathsf{ans}_P(\mathcal{Q}_8, \mathcal{D}, ?d_2)$ at all, since $\mathsf{ans}_P(\mathcal{Q}_7, \mathcal{D}, ?d_2) \subseteq \mathsf{ans}_P(\mathcal{Q}_8, \mathcal{D}, ?d_2)$.

In the case we presented in Example 6.1.12, the results produced by $S_a^{\mathcal{Z}_3}$ are optimal, i.e., they equal the complete set of productive values. But, $S_a^{\mathcal{Z}}$ does not always generate optimal results, as we saw in Example 6.1.11, where $S_a^{\mathcal{Z}_1}$ produced a proper superset of the productive values.

In general, the results produced by $S_a^{\mathcal{Z}}$ are closer to the true set of productive values when $\mathcal{Z}$ is large. This is because it then leads to some large, restrictive queries in the set of pruned queries. Furthermore, $S_a^{\mathcal{Z}}$ will, in general, give more precise results when $\mathcal{Q}$ is small, because this increases the chance that $\mathcal{Q}$ is covered by $\mathcal{Z}$, which again decreases the chance of missing restrictions that lead to dead-ends. But it is not only the size of $\mathcal{Q}$ and $\mathcal{Z}$ that matters: it is also important to consider how much they overlap. It does not help to have a small $\mathcal{Q}$ and a large $\mathcal{Z}$ if $\mathcal{Z}$ mostly covers parts outside of $\mathcal{Q}$. That said, one configuration query gives origin to a value function that will be used to generate values for multiple cases with different queries $\mathcal{Q}$, hence, it is important to consider the whole collection of queries, and not only individual queries when deciding which configuration queries to use. We do this in Chapter 8, where we explain how to generate configurations that lead to high precision over a whole set of different queries.

Since there is a one-to-one correspondence between each configuration query $\mathcal{Z}$ and its corresponding value function $S_a^{\mathcal{Z}}$, we are going to allow $S_a^{\mathcal{Z}}$ to be replaced by $\mathcal{Z}$ in all the precision formulas we presented in Chapter 5. I.e., for each rooted query $\mathcal{Q}$, unrooted query $\mathcal{Q}_u$, extension pair τ, and query log $\mathcal{L}$, we define:

$$
\begin{aligned}
\mathsf{prec}_{\mathrm{C}}(\mathcal{Z}, \mathcal{Q}, \tau) &= \mathsf{prec}_{\mathrm{C}}(S_a^{\mathcal{Z}}, \mathcal{Q}, \tau) \\
\mathsf{prec}_{\mathrm{Q}}(\mathcal{Z}, \mathcal{Q}) &= \mathsf{prec}_{\mathrm{Q}}(S_a^{\mathcal{Z}}, \mathcal{Q}) \\
\mathsf{prec}_{\mathrm{U}}(\mathcal{Z}, \mathcal{Q}_u) &= \mathsf{prec}_{\mathrm{U}}(S_a^{\mathcal{Z}}, \mathcal{Q}_u) \\
\mathsf{prec}_{\mathrm{L}}(\mathcal{Z}, \mathcal{L}) &= \mathsf{prec}_{\mathrm{L}}(S_a^{\mathcal{Z}}, \mathcal{L})
\end{aligned}
$$

Our discussion and examples above show how much the results of $S_a^{\mathcal{Z}}$ depends on $\mathcal{Z}$. In the worst-case scenario, $\mathcal{Z}$ does not cover the relevant extension pair τ, which leads $S_a^{\mathcal{Z}}$ to return Γ_v. On the other hand, if $\mathcal{Z}$ is large, and covers all of $\mathcal{Q}_e$, then $S_a^{\mathcal{Z}}$ will instead return the set of productive values. This is summarized in the statement below, which holds for every configuration query

$\mathcal{Z}$ and extension pair τ.

$$X_o(\mathcal{Q},\tau) = S_o(\mathcal{Q},\tau) \subseteq S_a^{\mathcal{Z}}(\mathcal{Q},\tau) \subseteq S_d(\mathcal{Q},\tau) = \Gamma_v$$

If we know that $\mathcal{Z}$ covers the extension pair τ, then $S_a^{\mathcal{Z}}$ will not suggest a larger set of values than S_r, i.e.,

$$X_o(\mathcal{Q},\tau) = S_o(\mathcal{Q},\tau) \subseteq S_a^{\mathcal{Z}}(\mathcal{Q},\tau) \subseteq S_r(\mathcal{Q},\tau) \subseteq S_d(\mathcal{Q},\tau) = \Gamma_v \tag{6.1}$$

A consequence of this is that $S_a^{\mathcal{Z}}$ always will have perfect recall:

$$\mathsf{rec}(S_a^{\mathcal{Z}},\mathcal{Q},\tau) = \frac{|S_a^{\mathcal{Z}}(\mathcal{Q},\tau) \cap X_o(\mathcal{Q},\tau)|}{|X_o(\mathcal{Q},\tau)|} = \frac{|X_o(\mathcal{Q},\tau)|}{|X_o(\mathcal{Q},\tau)|} = 1$$

S_l is not included in the analysis above because it is impossible to compare $S_a^{\mathcal{Z}}$ to S_l without access to $\mathcal{Z}$.

In Section 5.4.2 we discussed how Type I errors and Type II errors are directly related to the precision and recall of the value function that is used. Since the value function $S_a^{\mathcal{Z}}$ has perfect recall, our system will never classify a productive extension as a dead-end (Type II), but since the precision is not necessarily perfect, it may fail to detect some dead-ends (Type I). The user may be confused if they are not aware of the fact that Type I errors are possible, so any system that uses our approximation should make sure to tell the user about this. But, it is easy for the system to compare $\mathcal{Q}_e$ with $\mathcal{Z}$, and if $\mathcal{Q}_e$ is covered by $\mathcal{Z}$, then the system will know for sure that its precision will equal 1, which means that all predictions will be correct. Conversely, if $\mathcal{Q}_e$ is not covered by $\mathcal{Z}$, then the system may want to notify the user about the fact that some of the true dead-ends likely have not been detected. This would then make a system that is quite different from standard dead-end detection with respect to user experience.

6.1.3 Experiment 1: Precision of $S_a^{\mathcal{Z}}$

In order to get an idea about how well $S_a^{\mathcal{Z}}$ performs compared to the other value functions we have defined, and how the shape and size of $\mathcal{Z}$ affect the overall precision, we decided to conduct a simple experiment over a real-life dataset, where we compared the methods. The experiment was done relatively early in the thesis project, and the results have been presented in three of the papers related to the thesis: Paper P1 [26], Paper P6 [27], and Paper P8 [29] (see Section 1.2).

Experiment Setup In this experiment, we used the RDF version of the NPD Factpages,[1] which contains data about oil and gas drilling activities in Norway. The dataset has over 2 million triples, and its corresponding OWL ontology contains 209 classes and 375 properties. This RDF dataset contains data extracted from the original NPD Factpages, which is a relational database (RDB)

[1] https://gitlab.com/logid/npd-factpages

that every oil company in Norway are legally required to report to. This means that both the original RDB version and the generated RDF version, are fairly complete and homogeneous. This is an optimal environment for our system since we target large, complex queries, which require long chains of connected entities. The classes in this dataset have, on average, 14.1 different outgoing data properties, and 6.4 outgoing object properties. The number of distinct entities each such property leads to, is 572 on average, with a median of 12.

Unfortunately, the provided SPARQL query log over the NPD dataset was not suited for our experiment: just a few of the queries were typed and tree-shaped, as required by our system, and none of them contained more than a few variables.[2] So instead of using this, we constructed a new query log, consisting of 29 simple (see Definition 4.2.11) queries.[3] They ranged from 5 to 8 object variables and 0 to 12 data variables, and their corresponding result sets over the NPD dataset ranged from just 12 tuples to over 5 million tuples. This setup over NPD Factpages was also used in Experiment 2, which is presented in Section 6.3.1.

We ran multiple test cases, where each test case was based on one of the 29 queries, $\mathcal{Q}$, from the query log, and one generated configuration query $\mathcal{Z}$. For every query, we selected the variable $?c1$ to be the focus variable, i.e., we turned each of the queries into rooted queries by always focusing on the same variable. From this focus variable of $\mathcal{Q}$, we determined its root class, which was also used as the root class of $\mathcal{Z}$, to make pruning possible. When generating $\mathcal{Z}$, we tried to avoid cases where $\mathcal{Q}$ was covered completely by $\mathcal{Z}$, because we knew that those cases would lead to perfect precision, hence, it was not necessary to actually evaluate them. In fact, all branches of $\mathcal{Z}$ that exceeded the boundary defined by $\mathcal{Q}$ were discarded, because we knew these branches would never lead to higher precision anyway. Instead, we focused on configurations $\mathcal{Z}$ with a shape covered by $\mathcal{Q}$, as these would result in pruning, and hence a lower precision compared to the case without pruning. For each test case, we calculated the precision of $S_a^{\mathcal{Z}}$ with respect to the selected query $\mathcal{Q}$, using $\mathsf{prec}_{\mathrm{Q}}$ presented in Definition 5.4.8. The reason why we calculated the precision over only one rooted query, and not the whole query log, for example, is because a single configuration query and its corresponding value function can only generate reasonable values if it has the same root class as the query it evaluating precision over. Hence, if we calculated the precision over the whole query log using $\mathsf{prec}_{\mathrm{L}}$, then the resulting precision would be unreasonably low.

When generating configuration queries for the test cases, we first made a configuration query $\mathcal{Z}_c$, called the *core*, consisting of only object variables. We explored the set of all such $\mathcal{Z}_c$, including the smallest core, containing only the root variable, and the largest possible core (not exceeding the current query $\mathcal{Q}$). The cores were not used directly. Instead, they were used as a basis for two other configuration queries each:

- $\mathcal{Z}_c^{Dat}$: The configuration query where $\mathcal{Z}_c$ is fully saturated with data

[2] https://gitlab.com/logid/npd-factpages/-/tree/develop/query/sparql

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

properties, i.e., where every *data property* from the navigation graph is added to the variables of $\mathcal{Z}_c$, where the type matches.

- $\mathcal{Z}_c^{ObjDat}$: The configuration query where $\mathcal{Z}_c$ is fully saturated with data properties and object properties, i.e., where every property from the navigation graph is added to the variables of $\mathcal{Z}_c$, where the type matches.

Since $\mathcal{Z}_c^{ObjDat}$ saturates variables of $\mathcal{Z}_c$ with both object properties and data properties, while $\mathcal{Z}_c^{Dat}$ only saturates them with data properties, we know that $\mathcal{Z}_c^{Dat} \sqsubseteq \mathcal{Z}_c^{ObjDat}$. These two ways of saturating the core define two groups, and this distinction was made to explore how the addition of object properties affects the precision.

After running sufficiently many test cases, we grouped them by the query $\mathcal{Q}$ that was used, the size (number of variables) of the core $\mathcal{Z}_c$, and which type they belonged to: Dat or ObjDat. Then, we calculated the average of each group's precisions, and made one plot per query, with the size of $\mathcal{Z}_c$ on the x-axis, and the average precision on the y-axis. After adding the precision of S_r to these charts, they all presented the results of three different series: Dat, $ObjDat$, and S_r. Notice that since S_r is independent of the configuration query $\mathcal{Z}_c$, it gives the same results for all sizes of $\mathcal{Z}_c$.

For each choice of $\mathcal{Q}$, τ, and $\mathcal{Z}_c$, both of the value sets returned by $S_a^{\mathcal{Z}_c^{Dat}}$ and $S_a^{\mathcal{Z}_c^{ObjDat}}$ will be a subset of the values returned by S_r, and a superset of the values suggested by S_o (see Equation 6.1), i.e.,

$$S_r(\mathcal{Q},\tau) \supseteq S_a^{\mathcal{Z}_c^{Dat}}(\mathcal{Q},\tau) \supseteq S_a^{\mathcal{Z}_c^{ObjDat}}(\mathcal{Q},\tau) \supseteq S_o(\mathcal{Q},\tau)$$

This leads to the following relationship between their precisions:

$$0 < \mathsf{prec}_{\mathsf{Q}}(S_r,\mathcal{Q}) \le \mathsf{prec}_{\mathsf{Q}}(S_a^{\mathcal{Z}_c^{Dat}},\mathcal{Q}) \le \mathsf{prec}_{\mathsf{Q}}(S_a^{\mathcal{Z}_c^{ObjDat}},\mathcal{Q}) \le \mathsf{prec}_{\mathsf{Q}}(S_o,\mathcal{Q}) = 1. \tag{6.2}$$

Results Figures 6.4, 6.5 and 6.6 present results for three selected individual queries, while Figure 6.7 shows the average precision for all the queries of size 6. (15 of the 29 queries have exactly 6 object variables.) The same type of charts for queries with 5, 7, and 8 object variables are omitted from the thesis since they look very similar to the chart in Figure 6.7, but they can be found on Github[4] together with charts for every individual query used in the experiment.

The yellow line in each chart shows the precision of the range-based value function S_r. Notice that since it does not depend on the configuration query, it stays constant as the size of the configuration core increases. Since this is the value function with the lowest precision we consider, it acts as a baseline – marking the worst-case scenario for S_a. The blue and red curves represent the average precision of the value functions in the groups Dat and $ObjDat$ respectively. As expected, these two curves are non-decreasing, since larger configuration queries

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

Query 2.6

```
1   PREFIX ns1: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
2   PREFIX ns2: <http://sws.ifi.uio.no/vocab/npd-v2#>
3
4   SELECT * WHERE {
5   ?c1 ns1:type ns2:ExplorationWellbore.
6   ?c2 ns1:type ns2:Field.
7   ?c5 ns1:type ns2:ProductionLicence.
8   ?c3 ns1:type ns2:Company.
9   ?c4 ns1:type ns2:FieldStatus.
10  ?c8 ns1:type ns2:Discovery.
11  ?c6 ns1:type ns2:ProductionLicenceStatus.
12  ?c7 ns1:type ns2:ProductionLicenceArea.
13
14  ?c1 ns2:explorationWellboreForField ?c2.
15  ?c1 ns2:explorationWellboreForLicence ?c5.
16  ?c2 ns2:currentFieldOperator ?c3.
17  ?c4 ns2:statusForField ?c2.
18  ?c8 ns2:includedInField ?c2.
19  ?c6 ns2:statusForLicence ?c5.
20  ?c7 ns2:isGeometryOfFeature ?c5.
21
22  ?c3 ns2:name ?a6.
23  ?c4 ns2:status ?a5.
24  ?c6 ns2:status ?a4.
25  ?c7 ns2:isStratigraphical ?a1.
26  ?c1 ns2:wellboreBottomHoleTemperature ?a7.
27  ?c2 ns2:name ?a8.
28  ?c2 ns2:status ?a9.
29  ?c5 ns2:isActive ?a10.
30  ?c5 ns2:name ?a11.
31  ?c5 ns2:originalAreaSize ?a12.
32
33  FILTER(?a7 >= 150).
34  FILTER(regex(?a8, "TAMBAR", "i")).
35
36  }
```

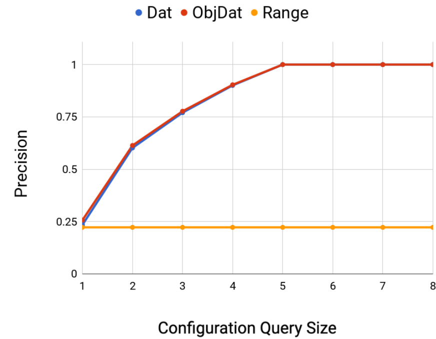


Figure 6.4: Precisions for Query 2.6.

lead to higher precision. We also see that the curve representing *ObjDat* always beats the curve of *Dat*, which is also as expected (see Equation 6.2).

The precision given by each of the three curves very much depends on how many of the important key restrictions of $\mathcal{Q}$ they are able to capture. A key restriction is a restriction that drastically reduces the number of data values one could assign to the extension variable, and hence, missing such a restriction, will lead to low precision. For example, Query 2.6 has only one key restriction on the data property *name* of the *Field* concept variable in depth 2 of $\mathcal{Q}$. Since this key restriction is associated with a datatype variable, both *Dat* and *ObjDat* perform about equally well. The slight difference between them is caused by other less important restrictions, which *ObjDat* manages to capture earlier than *Dat*. If this chart had shown the maximum precision of each test case, both *Dat* and *ObjDat* would have been close to perfect already at size 2 because that is the smallest size required for each of them to reach the field name. However, since we average over multiple differently shaped configurations for each size, and the branching factor of $\mathcal{Q}$ equals 2, both *Dat* and *ObjDat* will instead grow slowly

Query 2.8

```
1   PREFIX ns1: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
2   PREFIX ns2: <http://sws.ifi.uio.no/vocab/npd-v2#>
3
4   SELECT * WHERE {
5   ?c1 ns1:type ns2:ExplorationWellbore.
6   ?c2 ns1:type ns2:Field.
7   ?c5 ns1:type ns2:ProductionLicence.
8   ?c3 ns1:type ns2:Company.
9   ?c4 ns1:type ns2:FieldStatus.
10  ?c8 ns1:type ns2:Discovery.
11  ?c6 ns1:type ns2:ProductionLicenceStatus.
12  ?c7 ns1:type ns2:ProductionLicenceArea.
13
14  ?c1 ns2:explorationWellboreForField ?c2.
15  ?c1 ns2:explorationWellboreForLicence ?c5.
16  ?c2 ns2:currentFieldOperator ?c3.
17  ?c4 ns2:statusForField ?c2.
18  ?c8 ns2:includedInField ?c2.
19  ?c6 ns2:statusForLicence ?c5.
20  ?c7 ns2:isGeometryOfFeature ?c5.
21
22  ?c1 ns2:wellboreBottomHoleTemperature ?a7.
23
24  FILTER (?a7 >= 190).
25
26  }
```

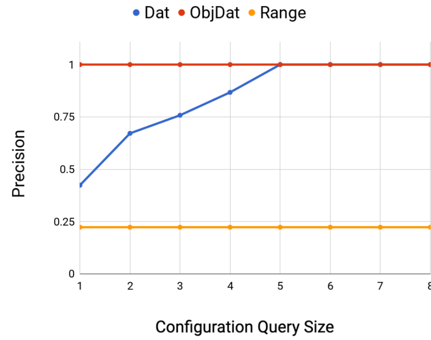


Figure 6.5: Precisions for Query 2.8.

Query 3.5

```
1   PREFIX ns1: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
2   PREFIX ns2: <http://sws.ifi.uio.no/vocab/npd-v2#>
3
4   SELECT * WHERE {
5
6   ?c1 ns1:type ns2:ExplorationWellbore.
7   ?c2 ns1:type ns2:Field.
8   ?c3 ns1:type ns2:FieldOperator.
9   ?c4 ns1:type ns2:Company.
10  ?c5 ns1:type ns2:BAA.
11  ?c7 ns1:type ns2:BAAArea.
12
13  ?c1 ns2:explorationWellboreForField ?c2.
14  ?c3 ns2:operatorForField ?c2.
15  ?c3 ns2:fieldOperator ?c4.
16  ?c5 ns2:baaOperatorCompany ?c4.
17  ?c7 ns2:isGeometryOfFeature ?c5.
18
19  ?c7 ns2:areaSize ?a3.
20
21  FILTER(?a3 >= 300).
22  }
```

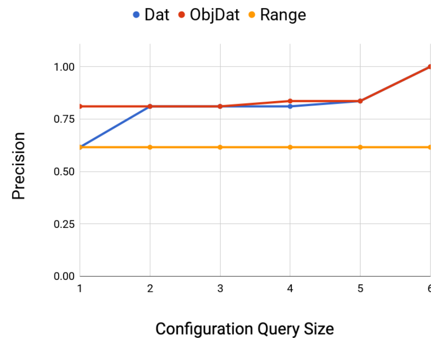


Figure 6.6: Precisions for Query 3.5.

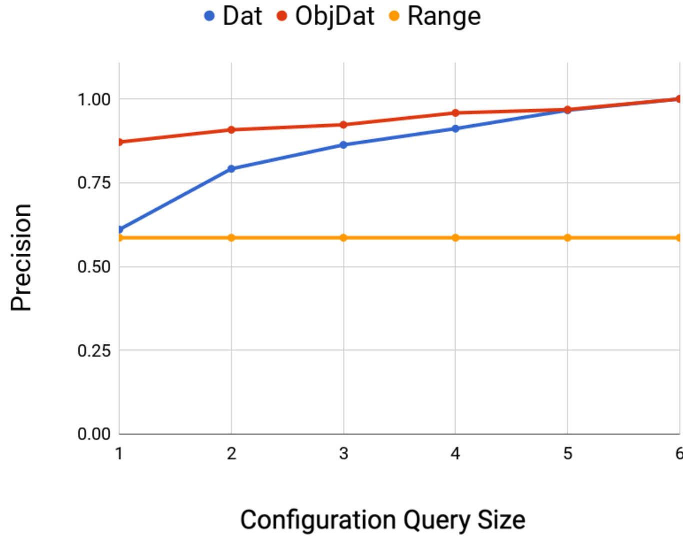


Figure 6.7: Average precision of all queries of size 6.

upwards until they reach size 5. At this point, the configuration is guaranteed to cover the key restriction regardless of its shape.

In general, it is not easy to know which parts of $\mathcal{Q}$ that are important key restrictions, because this requires a complete overview of the whole dataset. In some cases, it may also be the combination of several parts of $\mathcal{Q}$ that limits the result set, which means that a configuration query needs to cover all of them to achieve high precision.

Query 2.8, used in Figure 6.5, has two key restrictions. The first restriction is associated with a data property filter on the root node (wellboreTemperature $\geq$ 190). This is captured by all the configuration queries we used in the experiment, and the difference between S_r and *Dat* at size 1 shows the effect of capturing it. The other key restriction is associated with the *Field* concept variable in depth 2. Since *ObjDat* includes one additional layer of concept variables, it captures this already from size 1, which gives it perfect precision. *Dat*, on the other hand, needs to be of the correct shape in order to capture it, hence the steadily rising curve, similar to the one we saw in Figure 6.4.

Query 3.5, used in Figure 6.6, is a linear query (the tree has only one branch), which means that there only exists one possible core for each size. This query also has two key restrictions: The first one is an object property restriction in depth 2 of the query – the effect of capturing this restriction is shown by the precision increase of *ObjDat* between size 1 and 2. The second restriction is a data property restriction associated with the only concept variable in depth 6 of the query. This restriction is very hard to capture for both *ObjDat* and *Dat*,

since it is so far away from the root, but when the core reaches size 6, which means that both *ObjDat* and *Dat* covers the whole query, they both get perfect results.

The rules that control *ObjDat* and *Dat* also applies to S_r: it only performs well if it is able to capture all of the important key restrictions. But since S_r never considers $\mathcal{Q}$, except for its root and the extension variable, it will always perform poorly if one or more such key restrictions exist. In both Query 2.6 and Query 2.8, we see that S_r only achieves a precision of less than 0.25 because of this. In query 3.5, the two key restrictions mentioned above are not as significant, since S_r gets a precision above 0.6 without catching any of them.

The chart in Figure 6.7 shows the average precision over all queries of size 6 used in the experiment. Notice the relatively high precision of the range-based function S_r, equal to 0.58. Since precisions are always between 0 and 1, 0.58 is reasonable, at least when we consider how simple it is. But, the range-based solution is not sufficient, according to feedback from the user studies performed over OptiqueVQS. I.e., the users expect a higher precision.

In the cases where key restrictions are associated with object properties, *ObjDat* performs much better than *Dat*. In fact, it quite often returns suggestions with perfect precision, like it did for Query 2.8 in Figure 6.5. The average difference between *ObjDat* and *Dat*, we see in Figure 6.7, tells us that object variables indeed should be considered since they have the potential to increase precision quite a lot.

6.2 The Extension Index

So far we have been focusing on the configuration query $\mathcal{Z}$ and its relation to the value function $S_a^{\mathcal{Z}}$. In this section, we describe how the same configuration query can be used to set up the index $\mathcal{I}_{\mathcal{Z}}$, and how this index can be used to efficiently return answers to every query $\mathcal{Q}_s$ covered by $\mathcal{Z}$, which is necessary to ensure sufficient efficiency of $S_a^{\mathcal{Z}}$.

Ideally, we would like to make an index that can return efficient answers to $\mathcal{Q}_e$ directly, as this would allow us to just calculate the set of productive values. But, since we do not have access to $\mathcal{Q}_e$ at the point in time when the index has to be constructed, we would have to build an index that can support any such query, and since both $\mathcal{Q}$ and $\mathcal{Q}_e$ are unbounded, this index would be of infinite size. But, we do have access to $\mathcal{Z}$, and we know that there will be a filter-ignorant renaming function f_r from the pruned query $\mathcal{Q}_s$ to a subquery $\mathcal{Z}_s$ of $\mathcal{Z}$. Hence, if we store answers of $\mathcal{Z}_s$ over $\mathcal{D}$ in the index, we can combine them with f_r and the filters of $\mathcal{Q}_s$ to generate the answers of $\mathcal{Q}_s$. The theorem below formalizes this idea.

Theorem 6.2.1 (Answers Renaming)**.** Let $\mathcal{Q}_s = (R_{\mathcal{Q}_s}, T_{\mathcal{Q}_s}, F_{\mathcal{Q}_s})$ be a query, and let f_r be a filter-ignorant renaming from $\mathcal{Q}_s$ to a configuration query $\mathcal{Z}_s$. Then $\mathsf{ans}(\mathcal{Q}_s, \mathcal{D})$ is equal to the set of every $\pi \in \{\phi \circ f_r \mid \phi \in \mathsf{ans}(\mathcal{Z}_s, \mathcal{D})\}$ that satisfies:

$$\pi(v) \in F_{\mathcal{Q}_s}(v) \text{ for all } v \in \bar{V}_d(\mathcal{Q}_s)$$

⊣

Proof. If we ignore the variable names, then $\mathcal{Z}_s$ is just $\mathcal{Q}_s$ without filters, and when we add the filter requirement on the renamed answers we get from $\mathsf{ans}(\mathcal{Z}, \mathcal{D})$, they must equal the answers of $\mathsf{ans}(\mathcal{Q}_s, \mathcal{D})$. ■

Based on this, we want to make an index that contains the answers of $\mathcal{Z}_s$ over $\mathcal{D}$, for every $\mathcal{Z}_s \sqsubseteq \mathcal{Z}$. To unify all these answers, which are functions from $\mathcal{Z}_s$ to $\mathcal{D}$, we extend their domain to the union of all variables they are defined for: $\cup_{\mathcal{Z}_s \sqsubseteq \mathcal{Z}} \bar{V}(\mathcal{Z}_s) = \bar{V}(\mathcal{Z})$. For answers of any subquery $\mathcal{Z}_s$ where not every variable of $\mathcal{Z}$ is included, these excess variables have to be mapped to a special *null symbol* ω. In addition, we introduce a special symbol χ, which will be used to mark the existence of an entity from $\mathcal{D}$. It will play a central role later. An index over $\mathcal{Z}$ is then a set $\mathcal{I}_{\mathcal{Z}}$ of functions from $\bar{V}(\mathcal{Z})$ to $\bar{V}(\mathcal{D}) \cup \{\omega, \chi\}$.

Definition 6.2.2 (Index over $\mathcal{Z}$)**.** Let $\mathcal{Z}$ be a configuration query. An index over $\mathcal{Z}$ is a set $\mathcal{I}_{\mathcal{Z}}$ of functions $\phi\colon \bar{V}(\mathcal{Z}) \to (\bar{V}(\mathcal{D}) \cup \{\omega, \chi\})$. ⊣

Inspired by Theorem 6.2.1, we now define how to use this type of index to answer queries.

Definition 6.2.3 (ans_I and ans_{PI})**.** Let $\mathcal{Z}$ be a configuration query, and let $\mathcal{I}_{\mathcal{Z}}$ be an index over $\mathcal{Z}$. Let $\mathcal{Q}_s$ be a query covered by $\mathcal{Z}$, and let f_r be the filter-ignorant renaming function from $\mathcal{Q}_s$ to its corresponding subquery $\mathcal{Z}_s$ of $\mathcal{Z}$. The set of answers of $\mathcal{Q}_s$ over $\mathcal{I}_{\mathcal{Z}}$, denoted $\mathsf{ans}_I(\mathcal{Q}_s, \mathcal{I}_{\mathcal{Z}})$, is defined to be the set of all functions $\pi\colon \bar{V}(\mathcal{Q}_s) \to \bar{V}(\mathcal{D})$ from $M = \{(\phi\restriction_{\bar{V}(\mathcal{Z}_s)} \circ f_r) \mid \phi \in \mathcal{I}_{\mathcal{Z}}\}$ that satisfy both of the following requirements:

$$\pi(v) \neq \omega \qquad \forall v \in \bar{V}(\mathcal{Q}_s) \tag{6.3}$$

$$\pi(v) \in F_{\mathcal{Q}_s}(v) \quad \forall v \in \bar{V}_d(\mathcal{Q}_s) \tag{6.4}$$

The set of projected answers of $\mathcal{Q}_s$ over $\mathcal{I}_{\mathcal{Z}}$, onto the variable $v \in \bar{V}(\mathcal{Q}_s)$, is defined as follows:

$$\mathsf{ans}_{PI}(\mathcal{Q}_s, \mathcal{I}_{\mathcal{Z}}, v) = \{\pi(v) \mid \pi \in \mathsf{ans}_I(\mathcal{Q}_s, \mathcal{I}_{\mathcal{Z}})\}$$

⊣

In the definition above, each function $\pi \in M$ is a transformed version of a function $\phi \in \mathcal{I}_{\mathcal{Z}}$, where the restriction of ϕ to $\bar{V}(\mathcal{Z}_s)$ gives a function that can legally be composed with f_r, so each function in M is defined from $\bar{V}(\mathcal{Q}_s)$ to $\bar{V}(\mathcal{D})$. Any answer to $\mathcal{Q}_s$ should not map any of its variables to ω, and it should satisfy all the filters defined by $\mathcal{Q}_s$, and this is expressed by Equation 6.3 and Equation 6.4.

Our goal is to construct an index that gives the same projected answers to $\mathcal{Q}_s$ as the original dataset $\mathcal{D}$ does. If the index is populated in a way that ensures this, then it is considered to be *correct*.

Definition 6.2.4 (Correct Index)**.** Let $\mathcal{Z}$ be a configuration query, and let $\mathcal{I}_{\mathcal{Z}}$ be an index over $\mathcal{Z}$. $\mathcal{I}_{\mathcal{Z}}$ is *correct* if $\mathsf{ans}_{PI}(\mathcal{Q}_s, \mathcal{I}_{\mathcal{Z}}, v) = \mathsf{ans}_P(\mathcal{Q}_s, \mathcal{D}, v)$ for every query $\mathcal{Q}_s$ covered by $\mathcal{Z}$, and every variable $v \in \bar{V}(\mathcal{Q}_s)$. ⊣

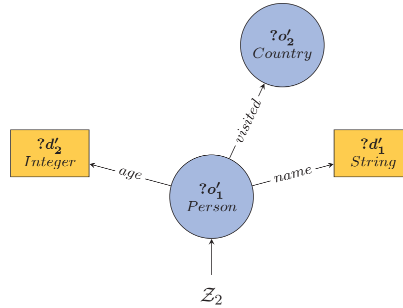


Figure 6.8: The configuration query $\mathcal{Z}_2$.

6.2.1 Index Construction

There are many ways to construct an index from $\mathcal{D}$, but not all of them are correct. We will start by presenting an index that is easy to prove correct: the index that contains all the padded answers to every subquery of $\mathcal{Z}$. We formalize this by introducing the function ans_S in the following definition.

Definition 6.2.5 (Subquery Answers of $\mathcal{Z}$)**.** Let $\mathcal{Z}$ be a configuration query. The set of *subquery answers* of $\mathcal{Z}$, denoted $\mathsf{ans}_S(\mathcal{Z}, \mathcal{D})$, is the set of all padded answers to any subquery $\mathcal{Z}_s$ of $\mathcal{Z}$, i.e.,

$$\mathsf{ans}_S(\mathcal{Z}, \mathcal{D}) = \bigcup_{\mathcal{Z}_s \sqsubseteq \mathcal{Z}} \{f_p(\phi) \mid \phi \in \mathsf{ans}(\mathcal{Z}_s, \mathcal{D})\}$$

where f_p transforms ϕ into the padded function $\phi_p\colon \bar{V}(\mathcal{Z}) \to (\bar{V}(\mathcal{D}) \cup \{\omega, \chi\})$ defined by:

$$\phi_p(v) = [f_p(\phi)](v) = \begin{cases} \phi(v) & \text{if } v \in \bar{V}(\mathcal{Z}_s) \\ \omega & \text{otherwise} \end{cases}$$

⊣

Example 6.2.6 shows how to use ans_S to construct an index over $\mathcal{D}$.

Example 6.2.6. Let us reconsider configuration query $\mathcal{Z}_2$, which is presented again in Figure 6.8. $\mathcal{Z}_2$ has eight different subqueries, and the set of all 50 (padded) answers they produce is given by $\mathcal{I}_{\mathcal{Z}_2} = \mathsf{ans}_S(\mathcal{Z}_2, \mathcal{D})$, which is presented in Table 6.1.

♦

The following example shows how we can use the index to produce correct answers to a query.

Example 6.2.7. Consider the index $\mathcal{I}_{\mathcal{Z}_2}$ in Example 6.2.6, and let us revisit query $\mathcal{Q}_2$, which is presented again in Figure 6.9. If we compute the projected answers

ϕ	$?o'_1$	$?d'_1$	$?o'_2$	$?d'_2$	ϕ	$?o'_1$	$?d'_1$	$?o'_2$	$?d'_2$
ϕ_1	$P1$	ω	ω	ω	ϕ_{28}	$P1$	ω	ω	21
ϕ_2	$P2$	ω	ω	ω	ϕ_{29}	$P2$	ω	ω	35
ϕ_3	$P3$	ω	ω	ω	ϕ_{30}	$P3$	ω	ω	45
ϕ_4	$P4$	ω	ω	ω	ϕ_{31}	$P4$	ω	ω	30
ϕ_5	$P5$	ω	ω	ω	ϕ_{32}	$P5$	ω	ω	11
ϕ_6	$P6$	ω	ω	ω					
ϕ_7	$P1$	$Alice$	ω	ω	ϕ_{33}	$P1$	$Alice$	ω	21
ϕ_8	$P2$	Bob	ω	ω	ϕ_{34}	$P2$	Bob	ω	35
ϕ_9	$P2$	$Robert$	ω	ω	ϕ_{35}	$P2$	$Robert$	ω	35
ϕ_{10}	$P3$	$Carol$	ω	ω	ϕ_{36}	$P3$	$Carol$	ω	45
ϕ_{11}	$P4$	$Dave$	ω	ω	ϕ_{37}	$P4$	$Dave$	ω	30
ϕ_{12}	$P5$	Eve	ω	ω	ϕ_{38}	$P5$	Eve	ω	11
ϕ_{13}	$P6$	$Alice$	ω	ω					
ϕ_{14}	$P1$	ω	$C1$	ω	ϕ_{39}	$P1$	ω	$C1$	21
ϕ_{15}	$P2$	ω	$C1$	ω	ϕ_{40}	$P2$	ω	$C1$	35
ϕ_{16}	$P2$	ω	$C2$	ω	ϕ_{41}	$P2$	ω	$C2$	35
ϕ_{17}	$P3$	ω	$C2$	ω	ϕ_{42}	$P3$	ω	$C2$	45
ϕ_{18}	$P5$	ω	$C1$	ω	ϕ_{43}	$P5$	ω	$C1$	11
ϕ_{19}	$P6$	ω	$C1$	ω					
ϕ_{20}	$P1$	$Alice$	$C1$	ω	ϕ_{44}	$P1$	$Alice$	$C1$	21
ϕ_{21}	$P2$	Bob	$C1$	ω	ϕ_{45}	$P2$	Bob	$C1$	35
ϕ_{22}	$P2$	$Robert$	$C1$	ω	ϕ_{46}	$P2$	$Robert$	$C1$	35
ϕ_{23}	$P2$	Bob	$C2$	ω	ϕ_{47}	$P2$	Bob	$C2$	35
ϕ_{24}	$P2$	$Robert$	$C2$	ω	ϕ_{48}	$P2$	$Robert$	$C2$	35
ϕ_{25}	$P3$	$Carol$	$C2$	ω	ϕ_{49}	$P3$	$Carol$	$C2$	45
ϕ_{26}	$P5$	Eve	$C1$	ω	ϕ_{50}	$P5$	Eve	$C1$	11
ϕ_{27}	$P6$	$Alice$	$C1$	ω					

Table 6.1: All the 50 functions in $\mathcal{I}_{\mathcal{Z}_2} = \mathsf{ans}_S(\mathcal{Z}_2, \mathcal{D})$.

of $\mathcal{Q}_2$ over $\mathcal{D}$, onto the variable $?d_1$ we get

$$\mathsf{ans}_P(\mathcal{Q}_2, \mathcal{D}, ?d_1) = \{Alice, Bob, Robert, Carol, Dave\}$$

We get the same data values if we execute $\mathcal{Q}_2$ over the index $\mathcal{I}_{\mathcal{Z}_2}$: Let us consider every $\phi \in \mathcal{I}_{\mathcal{Z}_2}$ from Table 6.1, and find out which of them will result in a function $\pi = \phi\restriction_{\bar{V}(\mathcal{Q}_2)} \circ f_r$ that satisfies both Equation 6.3 and Equation 6.4. We can for example discard functions ϕ_1 to ϕ_{27}, because they all map $?d'_2$ to ω, which gives $\pi(?d_2) = \phi(f_r(?d_2)) = \phi(?d'_2) = \omega$. We can do the same with $?d'_1$ to discard all functions ϕ_{28} to ϕ_{32}, and the functions ϕ_{39} to ϕ_{43}. The remaining 13 functions (ϕ_{33} to ϕ_{38} and ϕ_{44} to ϕ_{50}) result in the following 13 answers of $\mathcal{Q}_2$ after renaming, and before filtering.

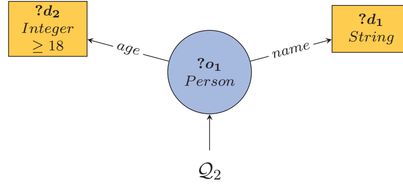


Figure 6.9: The query $\mathcal{Q}_2$.

π	$?o_1$	$?d_1$	$?d_2$
π_{33}	$P1$	$Alice$	21
π_{34}	$P2$	Bob	35
π_{35}	$P2$	$Robert$	35
π_{36}	$P3$	$Carol$	45
π_{37}	$P4$	$Dave$	30
π_{38}	$P5$	Eve	11
π_{44}	$P1$	$Alice$	21
π_{45}	$P2$	Bob	35
π_{46}	$P2$	$Robert$	35
π_{47}	$P2$	Bob	35
π_{48}	$P2$	$Robert$	35
π_{49}	$P3$	$Carol$	45
π_{50}	$P5$	Eve	11

Many of these functions are duplicates, and both π_{38} and π_{50} fail to satisfy the filter of age equal or higher than 18. After merging duplicates and projecting onto the variable $?d_1$, only five distinct values remain: $\{Alice, Bob, Robert, Carol, Dave\}$, which is exactly $\mathsf{ans}_P(\mathcal{Q}_2, \mathcal{D}, ?d_1)$. ♦

Based on our arguments up to this point, it should not be a surprise that the index $\mathcal{I}_\mathcal{Z} = \mathsf{ans}_S(\mathcal{Z}, \mathcal{D})$ is correct.

Theorem 6.2.8. The index $\mathcal{I}_\mathcal{Z} = \mathsf{ans}_S(\mathcal{Z}, \mathcal{D})$ over a configuration query $\mathcal{Z}$ is correct. ⊣

Proof. In order to prove this we are going to show that $\mathsf{ans}(\mathcal{Q}_s, \mathcal{D}) = \mathsf{ans}_I(\mathcal{Q}_s, \mathcal{I}_\mathcal{Z})$, for any $\mathcal{Q}_s$ covered by $\mathcal{Z}$, which implies that $\mathsf{ans}_P(\mathcal{Q}_s, \mathcal{D}, v) = \mathsf{ans}_{PI}(\mathcal{Q}_s, \mathcal{I}_\mathcal{Z}, v)$ for any variable $v \in \mathcal{Q}_s$. We assume that f_r is the filter-ignorant renaming from $\mathcal{Q}_s$ to its corresponding configuration query $\mathcal{Z}_s \sqsubseteq \mathcal{Z}$. Any function π contained in either $\mathsf{ans}(\mathcal{Q}_s, \mathcal{D})$ or $\mathsf{ans}_I(\mathcal{Q}_s, \mathcal{I}_\mathcal{Z})$ has to satisfy the filters defined by $\mathcal{Q}_s$, so we do not need to pay attention to the filters. Now, pick any $\pi \in \mathsf{ans}(\mathcal{Q}_s, \mathcal{D})$. According to Theorem 6.2.1, there has to be a function $\phi \in \mathsf{ans}(\mathcal{Z}_s, \mathcal{D})$ such that $\pi = \phi \circ f_r$. Since $\phi \in \mathsf{ans}(\mathcal{Z}_s, \mathcal{D})$, and $\mathcal{Z}_s \sqsubseteq \mathcal{Z}$, then the padded version of ϕ, denoted ϕ_p, will be in $\mathcal{I}_\mathcal{Z}$. Now, consider $\pi' = (\phi_p \restriction_{\bar{V}(\mathcal{Z}_s)} \circ f_r)$. Since $\phi_p \restriction_{\bar{V}(\mathcal{Z}_s)} = \phi$, we get $\pi' = \phi \circ f_r = \pi$. Furthermore, $\pi'(v) = \phi(f_r(v))$ cannot equal ω, since $f_r(v)$ is in $\bar{V}(\mathcal{Z}_s)$, and $\phi \in \mathsf{ans}(\mathcal{Z}_s, \mathcal{D})$, so

ϕ	$?o'_1$	$?d'_1$	$?o'_2$	$?d'_2$
ϕ_{27}	$P6$	$Alice$	$C1$	ω
ϕ_{37}	$P4$	$Dave$	ω	30
ϕ_{44}	$P1$	$Alice$	$C1$	21
ϕ_{45}	$P2$	Bob	$C1$	35
ϕ_{46}	$P2$	$Robert$	$C1$	35
ϕ_{47}	$P2$	Bob	$C2$	35
ϕ_{48}	$P2$	$Robert$	$C2$	35
ϕ_{49}	$P3$	$Carol$	$C2$	45
ϕ_{50}	$P5$	Eve	$C1$	11

Table 6.2: Index $\mathcal{I}'_{\mathcal{Z}_2} = \mathsf{ans}_O(\mathcal{Z}_2, \mathcal{D})$, which only includes the maximal functions from $\mathcal{I}_{\mathcal{Z}_2} = \mathsf{ans}_S(\mathcal{Z}_2, \mathcal{D})$.

$\pi' = \pi$ satisfies all requirements from Definition 6.2.3, and hence it must be in $\mathsf{ans}_I(\mathcal{Q}_s, \mathcal{I}_\mathcal{Z})$.

The argument the other way is very similar. Assume that $\pi \in \mathsf{ans}_I(\mathcal{Q}_s, \mathcal{I}_\mathcal{Z})$. Then there must exist some $\phi_p \in \mathcal{I}_\mathcal{Z}$ such that $\pi = (\phi_p\restriction_{\bar{V}(\mathcal{Z}_s)} \circ f_r)$. Now, $\phi_p\restriction_{\bar{V}(\mathcal{Z}_s)}$ must be in $\mathsf{ans}(\mathcal{Z}_s, \mathcal{D})$, because for every $v' \in \bar{V}(\mathcal{Z}_s)$, we get $\phi_p(v') = \phi_p(f_r(v)) = \pi(v) \neq \omega$, where v is a variable in $\bar{V}(\mathcal{Q}_s)$. From Theorem 6.2.1 follows that π must be in $\mathsf{ans}(\mathcal{Q}_s, \mathcal{D})$. ■

Eliminate Redundancy The index based on ans_S is correct, but it contains many redundant functions. In $\mathcal{I}_{\mathcal{Z}_2}$, for example, all the entities ϕ_7 refers to are also included in ϕ_{20}, so if we remove ϕ_7 from the index, it will still be correct. This relationship between two functions is called a *subfunction* relationship.

Definition 6.2.9 (Subfunction)**.** Let $\phi\colon X \to Y$ and $\phi_s\colon X \to Y$ be two functions. ϕ_s is a *subfunction* of ϕ, denoted $\phi_s \subseteq \phi$, if $\phi_s(x) = \phi(x)$ or $\phi_s(x) = \omega$ for each $x \in X$. ⊣

Since the subfunction relationship is transitive, we can improve the index by removing every subfunction from $\mathsf{ans}_S(\mathcal{Z}, \mathcal{D})$. This is equivalent to keeping all maximal functions with respect to the subfunction relationship. The result is a subset of the answers in $\mathsf{ans}_S(\mathcal{Z}, \mathcal{D})$, which we denote $\mathsf{ans}_O(\mathcal{Z}, \mathcal{D})$.

Example 6.2.10. Let us reconsider the index $\mathcal{I}_{\mathcal{Z}_2}$ from Example 6.2.6. If we extract all maximal functions from $\mathcal{I}_{\mathcal{Z}_2}$, we get the much smaller index $\mathcal{I}'_{\mathcal{Z}_2} = \mathsf{ans}_O(\mathcal{Z}_2, \mathcal{D})$, presented in Table 6.2. ♦

We can generate the index of maximal functions iteratively by starting with the column of the root variable, and then add the column of one more variable at a time, moving away from the root, until every variable is covered.

For example, to generate the index $\mathcal{I}'_{\mathcal{Z}_2}$ given in Table 6.2 from scratch, we start with all instances matching the root of $\mathcal{Z}_2$: $?o'_1$. This gives us 6 functions, one for each person in the dataset (see Table 6.3). Next, we add the variable $?d'_1$,

$?o'_1$
$P1$
$P2$
$P3$
$P4$
$P5$
$P6$

Table 6.3: Step 1.

$?o'_1$	$?d'_1$
$P1$	$Alice$
$P2$	Bob
$P2$	$Robert$
$P3$	$Carol$
$P4$	$Dave$
$P5$	Eve
$P6$	$Alice$

Table 6.4: Step 2.

$?o'_1$	$?d'_1$	$?o'_2$
$P1$	$Alice$	$C1$
$P2$	Bob	$C1$
$P2$	Bob	$C2$
$P2$	$Robert$	$C1$
$P2$	$Robert$	$C2$
$P3$	$Carol$	$C2$
$P4$	$Dave$	ω
$P5$	Eve	$C1$
$P6$	$Alice$	$C1$

Table 6.5: Step 3.

$?o'_1$	$?d'_1$	$?o'_2$	$?d'_2$
$P1$	$Alice$	$C1$	21
$P2$	Bob	$C1$	35
$P2$	Bob	$C2$	35
$P2$	$Robert$	$C1$	35
$P2$	$Robert$	$C2$	35
$P3$	$Carol$	$C2$	45
$P4$	$Dave$	ω	30
$P5$	Eve	$C1$	11
$P6$	$Alice$	$C1$	ω

Table 6.6: Step 4.

and this extends each function from Table 6.3 with the names of each person. Since $P2$ has two names, we need to create two copies of $P2$, one for each of the names *Bob* and *Robert* (see Table 6.4). Next, we add the variable $?o'_2$, which includes every country a person has visited. Notice that $P2$ has visited two countries, so again, each function from Table 6.4 containing $P2$ must be duplicated. $P5$, on the other hand, has not visited any countries, so $?o'_2$ must be mapped to ω to indicate that this branch has ended (see Table 6.5). Finally, the last variable, d'_2 is added, and the full set of answers are given in Table 6.6. Notice that $P6$ is not registered with any age in the dataset, and again we use ω to indicate this. As expected, the functions in Table 6.6 perfectly matches the functions given in Table 6.2.

The results of the construction above can be obtained by running a single SPARQL query over the dataset, where every branch is enclosed by the OPTIONAL keyword. Figure 6.10 presents the two configuration queries $\mathcal{Z}_1$ and $\mathcal{Z}_2$ again, while Figure 6.11 shows the two SPARQL queries needed to calculate $\mathsf{ans}_O(\mathcal{Z}_1, \mathcal{D})$ and $\mathsf{ans}_O(\mathcal{Z}_2, \mathcal{D})$. Notice the nested optionals that occur in the SPARQL query corresponding to $\mathcal{Z}_1$. Since the answers of $\mathsf{ans}_O(\mathcal{Z}, \mathcal{D})$ are so closely related to `OPTIONAL`s in SPARQL, we call them the *optional answers* of $\mathcal{Z}$.

Definition 6.2.11 (Optional Answers)**.** Let $\mathcal{Z}$ be a configuration query. The *optional answers* of $\mathcal{Z}$ over $\mathcal{D}$, denoted $\mathsf{ans}_O(\mathcal{Z}, \mathcal{D})$, is all maximal functions

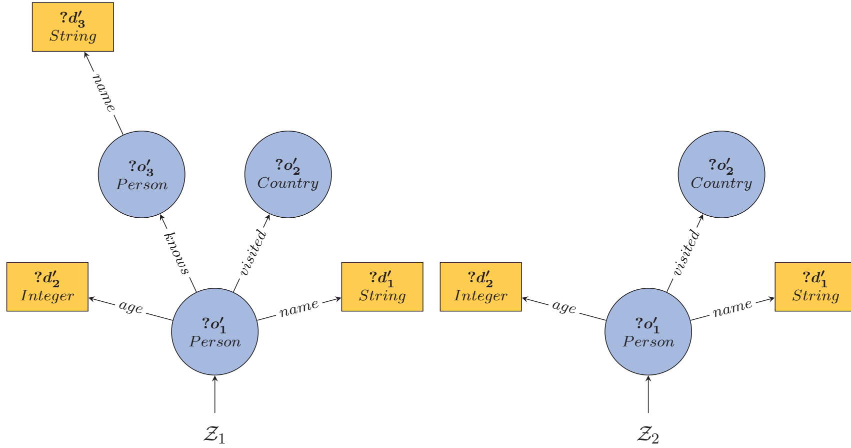


Figure 6.10: Configuration queries $\mathcal{Z}_1$ and $\mathcal{Z}_2$.

Query returning $\mathsf{ans}_O(\mathcal{Z}_1, \mathcal{D})$.

```
SELECT * WHERE {
  ?o1 rdf:type ex:Person.
  OPTIONAL {
    ?o2 rdf:type ex:Country.
    ?o1 ex:visited ?o2.
  }
  OPTIONAL {
    ?o3 rdf:type ex:Person.
    ?o1 ex:knows ?o3.
    OPTIONAL {
      ?o3 ex:name ?d3.
    }
  }
  OPTIONAL {
    ?o1 ex:age ?d1.
  }
  OPTIONAL {
    ?o1 ex:name ?d2.
  }
}
```

Query returning $\mathsf{ans}_O(\mathcal{Z}_2, \mathcal{D})$.

```
SELECT * WHERE {
  ?o1 rdf:type ex:Person.
  OPTIONAL {
    ?o2 rdf:type ex:Country.
    ?o1 ex:visited ?o2.
  }
  OPTIONAL {
    ?o1 ex:age ?d1.
  }
  OPTIONAL {
    ?o1 ex:name ?d2.
  }
}
```

Figure 6.11: SPARQL queries with nested optionals used to calculate $\mathsf{ans}_O(\mathcal{Z}_1, \mathcal{D})$ and $\mathsf{ans}_O(\mathcal{Z}_2, \mathcal{D})$.

$\phi \in \mathsf{ans}_S(\mathcal{Z}, \mathcal{D})$ with respect to the subfunction relationship. $\dashv$

Now we just need to prove formally that the index $\mathcal{I}_\mathcal{Z} = \mathsf{ans}_O(\mathcal{Z}, \mathcal{D})$ is correct.

Theorem 6.2.12. The index defined by $\mathcal{I}_\mathcal{Z} = \mathsf{ans}_O(\mathcal{Z}, \mathcal{D})$ over the configuration query $\mathcal{Z}$ is correct. $\dashv$

Since $\mathsf{ans}_O(\mathcal{Z}, \mathcal{D})$ is just $\mathsf{ans}_S(\mathcal{Z}, \mathcal{D})$ where subfunctions are removed, we can show that Theorem 6.2.12 is true by proving a more general theorem, which states that the act of removing a subfunction from an index does not change the results of $\mathsf{ans}_I(\mathcal{Q}, \mathcal{I}_\mathcal{Z})$.

Theorem 6.2.13. Let $\mathcal{I}_\mathcal{Z}$ be an index over a configuration query $\mathcal{Z}$, and let $\mathcal{Q}_s$ be a query covered by $\mathcal{Z}$. Furthermore, let ϕ and $\phi_s \subseteq \phi$ be two functions in $\mathcal{I}_\mathcal{Z}$. Then

$$\mathsf{ans}_I(\mathcal{Q}_s, \mathcal{I}_\mathcal{Z}) = \mathsf{ans}_I(\mathcal{Q}_s, \mathcal{I}_\mathcal{Z} \setminus \{\phi_s\})$$

$\dashv$

Proof. Assume that we use the same notation as in Definition 6.2.3, and assume that $\pi \in \mathsf{ans}_I(\mathcal{Q}_s, \mathcal{I}_\mathcal{Z})$ is an answer that depends on ϕ_s, i.e., $\pi = (\phi_s\restriction_{\bar{V}(\mathcal{Z}_s)} \circ f_r)$. Now, for every variable $v' \in \mathcal{Z}_s$, we get $\phi_s(v') = \phi_s(f_r(v)) = \pi(v) \neq \omega$, where v is the variable in $\mathcal{Q}_s$ that corresponds to v'. But since $\phi_s \subseteq \phi$, we then know that $\phi_s(v') = \phi(v')\ \forall v' \in \bar{V}(\mathcal{Z}_s)$, which leads to the fact that $\phi_s\restriction_{\bar{V}(\mathcal{Z}_s)} = \phi\restriction_{\bar{V}(\mathcal{Z}_s)}$. So even if ϕ_s is removed from the index, we still get π as an answer because $\pi = (\phi\restriction_{\bar{V}(\mathcal{Z}_s)} \circ f_r)$. ∎

Existential Object Variables We can further reduce the size of the index by utilizing the fact that our queries do not apply any filters on the object variables. In Definition 6.2.3, the only requirement on object variables is that they should not be assigned to ω. In other words, we do not need to store the particular instances in the index, we just need to know whether such an instance exists or not. For example, consider the functions in Table 6.2 again, and in particular ϕ_{45} and ϕ_{47}, which are identical for all variables except for the object variable $?o'_2$, which tells which countries the person has visited. When answering queries, it is not relevant whether the person visited $C1$ or $C2$ – all our system needs to know, is that there exists a country visited by the person. Hence, our index will still be correct if we replace each variable assignment to an instance with a similar assignment to the special *existence symbol* χ, which indicates to the system that such an instance exists. Some functions in the index will then become equal, and they can together be represented by only one function. In our example, this will happen to ϕ_{45} and ϕ_{47}, but also ϕ_{46} and ϕ_{48}. By merging these function, we get the index presented in Table 6.7.

Notice that by introducing existential object variables, the function ϕ'_{27} suddenly becomes a subquery of ϕ'_{44}. This problem can be solved by first introducing existential object variables, and then removing subfunctions. Let us formalize this process into a new type of answer function: the *existential answers* ans_E.

ϕ	$?o_1'$	$?d_1'$	$?o_2'$	$?d_2'$
ϕ_{27}'	χ	*Alice*	χ	ω
ϕ_{37}'	χ	*Dave*	ω	30
ϕ_{44}'	χ	*Alice*	χ	21
ϕ_{45}'	χ	*Bob*	χ	35
ϕ_{46}'	χ	*Robert*	χ	35
ϕ_{49}'	χ	*Carol*	χ	45
ϕ_{50}'	χ	*Eve*	χ	11

Table 6.7: Every function remaining from $\mathsf{ans}_O(\mathcal{Z}_2, \mathcal{D})$ after introducing existential object variables, and before removing subfunctions one more time.

ϕ	$?o_1'$	$?d_1'$	$?o_2'$	$?d_2'$
ϕ_{37}'	χ	*Dave*	ω	30
ϕ_{44}'	χ	*Alice*	χ	21
ϕ_{45}'	χ	*Bob*	χ	35
ϕ_{46}'	χ	*Robert*	χ	35
ϕ_{49}'	χ	*Carol*	χ	45
ϕ_{50}'	χ	*Eve*	χ	11

Table 6.8: Every function in $\mathsf{ans}_E(\mathcal{Z}_2, \mathcal{D})$.

Definition 6.2.14 (Existential Answers ans_E)**.** Let $\mathcal{Z}$ be a configuration query. The set of every *existential answer* of $\mathcal{Z}$ over $\mathcal{D}$, denoted $\mathsf{ans}_E(\mathcal{Z}, \mathcal{D})$, is the set of all maximal functions in $\{f_E(\phi) \mid \phi \in \mathsf{ans}_S(\mathcal{Z}, \mathcal{D})\}$ with respect to the subfunction relation, where f_E is a function that transforms every function ϕ into ϕ' given by

$$\phi'(v) = \begin{cases} \chi & \text{if } v \in \bar{V}_o(\mathcal{Z}) \text{ and } \phi(v) \neq \omega \\ \phi(v) & \text{otherwise} \end{cases} \tag{6.5}$$

⊣

Example 6.2.15. If we revisit the configuration query $\mathcal{Z}_2$ used in the previous examples, and compute $\mathsf{ans}_E(\mathcal{Z}_2, \mathcal{D})$ we get the functions presented in Table 6.8 ♦

Notice that the first column in Table 6.8 that corresponds to the root variable of $\mathcal{Z}_2$ only contains the symbol χ. This is not a coincidence – every function $\pi \in \mathsf{ans}_E(\mathcal{Z}, \mathcal{D})$ will map the root of $\mathcal{Z}$ to χ, since it is an object variable that cannot be mapped to ω.

Theorem 6.2.16. The index defined by $\mathcal{I}_{\mathcal{Z}} = \mathsf{ans}_E(\mathcal{Z}, \mathcal{D})$ over the configuration query $\mathcal{Z}$ is correct. ⊣

Proof. We already proved that $\mathsf{ans}_S(\mathcal{Z}, \mathcal{D})$ is correct in Theorem 6.2.8, and that removing subfunctions does not make an index incorrect in Theorem 6.2.13,

so we only need to show that the introduction of the existence symbol does not make the index incorrect. In other words, if we let $\mathcal{Q}_s$ be a query covered by $\mathcal{Z}$, and we consider a function $\pi \in \mathsf{ans}_I(\mathcal{Q}_s, \mathcal{I}_\mathcal{Z})$, where $\mathcal{I}_\mathcal{Z} = \mathsf{ans}_S(\mathcal{Z}, \mathcal{D})$, then we only need to show that $f_E(\pi)$ still qualifies as an answer, i.e., that it satisfies both requirements given by Equation 6.3 and Equation 6.4. If this is true, then projection on a given variable still returns the same results, since f_E does not change the values assigned to data variables of $\mathcal{Z}$. But $f_E(\pi)$ satisfies Equation 6.3, because it does not reassign any variables to ω, and it satisfies Equation 6.4, because it does not reassign any of the data variables. ■

This final way of setting up the index based on ans_E is the best we can achieve under the index definition we use, so we will use this index from here on. I.e., we assume that $\mathcal{I}_\mathcal{Z} = \mathsf{ans}_E(\mathcal{Z}, \mathcal{D})$ unless stated otherwise.

Object Property Extensions Early in Chapter 5, we presented two different types of extensions: object property extensions and data property extensions. So far we have only been focusing on data property extensions, but now we will briefly explain how to use the index to also detect dead-ends among object property extensions. The approach is very similar to the one we use to calculate dead-ends among data property extensions.

Since object variables can not have filters, an object property extension can be specified by just an extension pair $\tau = (p_e, t_e) \in \Gamma_p \times \Gamma_c$, where p_e is an object property and t_e is a class. All legal such pairs can be determined from the navigation graph, and to check if τ is productive, we just have to extend the partial query with respect to τ and execute the resulting query $\mathcal{Q}_e$ over the index with projection on the extension variable v_e. Since v_e now is an object variable, and every column corresponding to an object variable in the index only contains values from $\{\chi, \omega\}$, the projected result of $\mathcal{Q}_e$ will be a subset of $\{\chi, \omega\}$. The extension should be classified as productive if there exists at least one answer, i.e., if χ is contained in the answers of $\mathcal{Q}_e$. For each case with a partial query and extension pair, we then just get a binary prediction, and the precision of this prediction is perfect if the extension is classified correctly, and 0 otherwise.

6.2.2 Index Efficiency

So far we have described how to set up an index based on ans_E, and how to use it correctly with ans_I and ans_{PI}. Now we will explain how to implement it in a way that ensures sufficient efficiency.

The simplest way to do it is to use state-of-the-art faceted search systems, like the ones we presented in Chapter 2. Earlier we claimed that these systems cannot be used in the multi-class setting we work in because they require both queries and a dataset that is limited by a predefined schema. But this is more or less exactly what we have left after introducing the configuration query $\mathcal{Z}$: both the index and the queries we work with are limited by the variables and structure of $\mathcal{Z}$. Or, phrased in another way: we can turn our configuration-based solution into standard faceted search by considering every data property in $\mathcal{Z}$ to be a

facet of the root. By implementing our index-based solution in state-of-the-art search systems, we are guaranteed both sufficient efficiency and scalability.

Another way to implement our system is to materialize $\mathcal{I}_{\mathcal{Z}}$ into its corresponding *index table*, which is the table with one row per function in $\mathcal{I}_{\mathcal{Z}}$, and one column per variable in $\mathcal{Z}$, and where each cell contains the value one gets when applying the row's function on the column's variable. This is similar to the tables of functions we have presented earlier in this chapter, like Table 6.8. Think of the index table as a large, denormalized table where all possible joins are precomputed.

This index table can be stored in a relational database, under the schema defined by $\mathcal{Z}$. The process defined by ans_I is then just a table scan, where only rows that satisfy the filters of the relevant query are returned. Such table scans can be done efficiently (logarithmic with respect to the number of rows) if database indices are added to each column. Table scans can also be parallelized easily, i.e., it is easy to distribute the work over multiple machines if needed.

The time it takes to build the index based on $\mathcal{Z}$ is mostly consumed by the time it takes for the query engine to calculate $\mathsf{ans}_E(\mathcal{Z}, \mathcal{D})$. How fast this actually is, depends on the database, and how well it handles the nested optionals that occur in the SPARQL query corresponding to $\mathsf{ans}_E(\mathcal{Z}, \mathcal{D})$. Index construction should be as efficient as possible, but it is not as time-critical as dead-end detection. If data is updated daily, then index construction can be done overnight, for example.

6.2.3 Index Cost

One of the main drawbacks of using index-based solutions is that the index has to be stored somewhere. We have to include this aspect in our model, i.e., we need to somehow quantify the size of the index. Ideally, we would like to use the actual size in bytes of the index as a cost measure, but this depends too much on the implementation, and it is too complicated for our model. Instead, we define the cost of an index to equal the number of cells in its corresponding index table. This is a reasonable cost function, which grows at about the same rate as the size of the index in bytes, at least if we assume that the index is stored in its tabular form. Since we assume that $\mathcal{I}_{\mathcal{Z}} = \mathsf{ans}_E(\mathcal{Z}, \mathcal{D})$, and since we know that the column corresponding to the root variable only includes χ (see Table 6.8 for an example), we get the following cost function:

Definition 6.2.17 (Configuration Query Cost)**.** Let $\mathcal{Z}$ be a configuration query. The *cost* of $\mathcal{Z}$ over $\mathcal{D}$, denoted $\mathsf{cost}(\mathcal{Z}, \mathcal{D})$, is given by

$$\mathsf{cost}(\mathcal{Z}, \mathcal{D}) = (|\bar{V}(\mathcal{Z})| - 1) \cdot |\,\mathsf{ans}_E(\mathcal{Z}, \mathcal{D})| \tag{6.6}$$

⊣

In Equation 6.6, $|\bar{V}(\mathcal{Z})|$ equals the number of columns in the index table, while $|\,\mathsf{ans}_E(\mathcal{Z}, \mathcal{D})|$ is the number rows. We subtract 1 from the number of columns $|\bar{V}(\mathcal{Z})|$ in the cost function since the column of the root only contains χ, and hence, does not need to be stored.

6.2.4 Bucketing

Some properties in the dataset may have a large set of distinct values they can refer to. One example of this is the fortune of a person, which can be any amount of money between \$0 and \$100 billion. Few persons have the exact same fortune, and even if we reduce the precision to whole dollars, there are still extremely many possible fortunes left, if we consider all living persons. Properties with many possible values are problematic, because they require a large amount of storage space in general, and because they may cause information overload when presented to the user. In our context, they are also problematic because they lead to many rows in the index table, which again results in a high cost.

In order to eliminate the problems mentioned above, we can use a common technique called *bucketing*, which combines similar data values or entities into one group, called a bucket. For example, one could make a set of four buckets $B = \{b_1, b_2, b_3, b_4\}$ for the following four fortune ranges:

- b_1: $<$ \$100
- b_2: \$100 - \$10.000
- b_3: \$10.000 - \$1.000.000
- b_4: $>$ \$1.000.000

One way to integrate these buckets into our system, is to transform the dataset by replacing each individual's fortune with the bucket it belongs to. This reduces the number of distinct fortunes in the dataset, which may be millions, down to only four buckets.

The drawback of introducing buckets as described above is that it reduces the quality of the data, which again affects which queries the dataset can provide answers to. For example, if a user wants the list of all persons with a fortune between \$4000 and \$9000, then the dataset will not be able to provide a precise answer, because this fortune range does not comply with the buckets given above. In other words, if buckets are used to reduce the dataset, then they also have to be incorporated in the query filters. So, instead of allowing filters that are specified by a subset of the values in Γ_v, filters must instead specify a subset of the given buckets. The reduction from multiple data values to just a few buckets in query filters must be implemented also in the user interface of the relevant system. I.e., instead of providing a list with millions of possible fortunes, the user interface will instead only show 4 fortune buckets. In other words, bucketing is an efficient way of reducing the problem of information overload.

Bucketing is most commonly used for numerical datatypes, because the range of possible numbers is often infinitely large, and since they have a natural ordering that makes it easy to define buckets and place values in them. It is also possible to define buckets for any other type of data, but it is less common. For example, if the dataset uses a fine-grained set of food categories, it may be possible to define buckets based on categories higher up in the food category hierarchy.

This way of implementing buckets, where buckets are used everywhere, is not the only way of implementing them into our system. For example, it is possible to only use buckets in the index, but this requires a new definition of ans_I that still works for arbitrary filters in the queries. The use of buckets to simplify datasets is nothing new. For example, they are commonly used by e-commerce services like eBay to specify price ranges of products. Buckets are also a central part of the construction and usage of histograms internally in databases, which are often used to estimate the cardinality of queries [21, 42, 20]. In this thesis project, we have not been focusing on buckets, but, we know that bucketing is a useful technique, which our system would likely benefit from (index reduction), and we know that there exist research and methods that would allow us to use them [21, 42, 20]. Furthermore, in Section 8.1, we are going to estimate the cardinality of queries, and in parts of that estimation process, we assume that the distinct values associated with each property can be described by a finite, discrete set, as this allows us to construct histograms based on the dataset.

6.3 Optimal Configuration Queries

Given a configuration query $\mathcal{Z}$, we now have a definition of its corresponding value function, $S_a^{\mathcal{Z}}$, which we can evaluate over one or more different extension cases using any of the precision formulas from Section 5.4.2. In Section 6.1.2, we tied this precision directly to the configuration query $\mathcal{Z}$ itself. In other words, given $\mathcal{Z}$, we have methods to calculate the precision it leads to over single extension cases, rooted queries, unrooted queries, or query logs. We also have a cost function that allows us to express how costly $\mathcal{Z}$ is with respect to $\mathcal{D}$ (see Definition 6.2.17).

In general, a large configuration query tends to result in both high cost and high precision, while a small configuration query tends to result in both low precision and low cost. This is interesting because we are obviously interested in configuration queries with high precision, but low cost. In this section, we are going to explore the trade-off between the two measures. In particular, we are going to discuss *Pareto optimal configuration queries*, and present the results of an Experiment related to them (see Section 6.3.1).

If $\mathcal{Z}$ is a configuration query and $\mathcal{Z}_s$ is a subquery of $\mathcal{Z}$, then $\mathcal{Z}$ will be at least as costly as $\mathcal{Z}_s$ and at least as precise as $\mathcal{Z}_s$ with respect to all the four possible precision functions we have defined.

Theorem 6.3.1. Let $\mathcal{Z}$ and $\mathcal{Z}_s$ be two configuration queries. If $\mathcal{Z}_s \sqsubseteq \mathcal{Z}$, then all of the following statements are true:

$$\mathsf{cost}(\mathcal{Z}_s, \mathcal{D}) \leq \mathsf{cost}(\mathcal{Z}, \mathcal{D}) \tag{6.7}$$

$$\mathsf{prec}_{\mathrm{C}}(\mathcal{Z}_s, \mathcal{Q}, \tau) \leq \mathsf{prec}_{\mathrm{C}}(\mathcal{Z}, \mathcal{Q}, \tau) \tag{6.8}$$

$$\mathsf{prec}_{\mathrm{Q}}(\mathcal{Z}_s, \mathcal{Q}) \leq \mathsf{prec}_{\mathrm{Q}}(\mathcal{Z}, \mathcal{Q}) \tag{6.9}$$

$$\mathsf{prec}_{\mathrm{U}}(\mathcal{Z}_s, \mathcal{Q}_u) \leq \mathsf{prec}_{\mathrm{U}}(\mathcal{Z}, \mathcal{Q}_u) \tag{6.10}$$

$$\mathsf{prec}_{\mathrm{L}}(\mathcal{Z}_s, \mathcal{L}) \leq \mathsf{prec}_{\mathrm{L}}(\mathcal{Z}, \mathcal{L}) \tag{6.11}$$

where $\mathcal{Q}$ is any rooted query, and τ is a legal extension pair, $\mathcal{Q}_u$ is an unrooted query, and $\mathcal{L}$ is a query log. ⊣

Proof. If we can prove Equation 6.8, then the three equations below it will also be true, according to Theorem 5.4.12. Hence, we only need to prove Equation 6.7 and Equation 6.8

To prove Equation 6.7, we have to consider both of the factors in Equation 6.6, and show that they both are smaller when we evaluate the cost of $\mathcal{Z}_s$ compared to the cost of $\mathcal{Z}$. $|\bar{V}(\mathcal{Z}_s)| - 1$ is obviously smaller than $|\bar{V}(\mathcal{Z})| - 1$, since the subquery $\mathcal{Z}_s$ has fewer variables than $\mathcal{Z}$ (unless $\mathcal{Z}_s = \mathcal{Z}$). To prove that $|\,\mathsf{ans}_E(\mathcal{Z}_s, \mathcal{D})|$ is smaller than $|\,\mathsf{ans}_E(\mathcal{Z}, \mathcal{D})|$, we are going to show that for each $\pi \in \mathsf{ans}_E(\mathcal{Z}_s, \mathcal{D})$, there exists a corresponding function in $\mathsf{ans}_E(\mathcal{Z}, \mathcal{D})$. Notice that since subfunctions will always be removed from $\mathsf{ans}_E(\mathcal{Z}_s, \mathcal{D})$ and $\mathsf{ans}_E(\mathcal{Z}, \mathcal{D})$, both of these sets only contain maximal functions with respect to the subfunction relation. Now, let ϕ be the padded version of π where each variable in $\bar{V}(\mathcal{Z}) \setminus \bar{V}(\mathcal{Z}_s)$ is mapped to ω. If ϕ is maximal in $\mathsf{ans}_E(\mathcal{Z}, \mathcal{D})$, then we are done with π, since we have found a corresponding function. If not, then select one maximal function $\phi^m \in \mathsf{ans}_E(\mathcal{Z}, \mathcal{D})$ such that $\phi \subseteq \phi^m$, and let ϕ^m be the query that corresponds to π. Now we only need to show that no two functions π_1 and π_2 in $\mathsf{ans}_E(\mathcal{Z}_s, \mathcal{D})$ correspond to the same function in $\mathsf{ans}_E(\mathcal{Z}, \mathcal{D})$. So let us assume that there exists a maximal query $\phi^m \in \mathsf{ans}_E(\mathcal{Z}, \mathcal{D})$ such that $\phi_1 \subseteq \phi^m$ and $\phi_2 \subseteq \phi^m$, where ϕ_1 and ϕ_2 are the padded versions of π_1 and π_2 respectively. But then $\pi^m = \phi^m\!\restriction_{\bar{V}(\mathcal{Z}_s)}$ will be a superfunction of both π_1 and π_2, which is only possible if $\pi_1 = \pi_2 = \pi^m$, since both π_1 and π_2 are maximal in $\mathsf{ans}_E(\mathcal{Z}_s, \mathcal{D})$.

To prove Equation 6.8, it is enough to show that $S_a^{\mathcal{Z}_s}(\mathcal{Q}, \tau) \supseteq S_a^{\mathcal{Z}}(\mathcal{Q}, \tau)$. But this is true, because $\mathsf{prune}(\mathcal{Q}_e, \mathcal{Z}_s, v_e) \subseteq \mathsf{prune}(\mathcal{Q}_e, \mathcal{Z}, v_e)$ (see Definition 6.1.8), so every $\mathcal{Q}_s$ and its corresponding $\mathsf{ans}_P(\mathcal{Q}_s, \mathcal{D}, v_e)$ that limits the set returned by $S_a^{\mathcal{Z}_s}(\mathcal{Q}, \tau)$, will also be limiting $S_a^{\mathcal{Z}}(\mathcal{Q}, \tau)$, but $S_a^{\mathcal{Z}}(\mathcal{Q}, \tau)$ may also contain other subqueries that limits it even more. ■

Pareto Optimal Configurations A configuration query is considered to be good if it has high precision, but also if it has a low cost. But, as we have already established, there is a natural trade-off between these two measures, so usually, we will not be able to find a single configuration query that is simply better than every other configuration query. Instead, there will be a set of what is called *Pareto optimal* configuration queries, i.e., configurations that are better than each other configuration with respect to either cost or precision.

Definition 6.3.2 (Pareto Optimal Configuration Queries)**.** Let X be a set of configuration queries. A configuration query $\mathcal{Z} \in X$ is *Pareto optimal* in X if for every configuration query $\mathcal{Z}' \in X$, at least one of the following statements is true:

1. $\mathsf{prec}_C(\mathcal{Z}', \mathcal{Q}, \tau) < \mathsf{prec}_C(\mathcal{Z}, \mathcal{Q}, \tau)$
2. $\mathsf{cost}(\mathcal{Z}', \mathcal{D}) > \mathsf{cost}(\mathcal{Z}, \mathcal{D})$

3. $\mathsf{cost}(\mathcal{Z}, \mathcal{D}) = \mathsf{cost}(\mathcal{Z}', \mathcal{D})$ and $\mathsf{prec}_{\mathrm{C}}(\mathcal{Z}, \mathcal{Q}, \tau) = \mathsf{prec}_{\mathrm{C}}(\mathcal{Z}', \mathcal{Q}, \tau)$

$\dashv$

The definition above is defined using $\mathsf{prec}_{\mathrm{C}}$, the precision over one single case, but it can also be applied to all the other three types of precisions we have defined. The third requirement covers the special case when two different configurations have the exact same cost and precision. In that case, we consider both to be Pareto optimal. The set of Pareto optimal configuration queries is also called the *Pareto frontier*.

Definition 6.3.2 defines Pareto optimality over a set of configuration queries X. This can be any set of configuration queries, but we are mostly interested in configuration queries that are globally Pareto optimal, i.e., Pareto optimal configuration queries with respect to the set of all possible configuration queries.

6.3.1 Experiment 2: Pareto Optimal Configuration Queries

In order to get an idea about what the relationship between cost and precision looks like in practice, we do a second experiment where we explore the full set of configuration queries and their corresponding cost and precision over one rooted query.

The results of this experiment have been published and used in two of the papers related to this thesis: Paper P6 [27] and Paper P8 [29].

We use the same setup as in Experiment 1 (see Section 6.1.3), based on the NPD Factpages dataset, and we calculate precisions based on only one rooted query at a time (see Definition 5.4.8), which is also what we did in Experiment 1. For each rooted query $\mathcal{Q}$, we generate the set of all configuration queries with the same root class as $\mathcal{Q}$, which are also covered by $\mathcal{Q}$. Then we calculate the precision and cost of each such configuration query.

Figure 6.12 presents the results of Query 6.2. It contains a chart with precision on the x-axis, and cost on the y-axis, and each considered configuration query is represented by a point in the position that corresponds to its cost and precision. All points close to the bottom right corner are good because they have both high precision and low cost. Some of these configuration queries are Pareto optimal, and to highlight those, we have connected them with lines.

Notice how the different configuration queries tend to group together on a limited set of fixed precisions. This is because the precision does not always increase when a new variable is added to a configuration query, but the cost will always increase.

Since we evaluate the precision over a rooted query $\mathcal{Q}$, we want configuration queries that at least contains all the local data properties in $\mathcal{Q}$. In other words, the smallest configuration query in Figure 6.12, denoted $\mathcal{Z}_{min}$, which is the one closest to the bottom left corner, is a star-shaped configuration query containing just the root variable and all local data properties that are also in $\mathcal{Q}$. The largest configuration query we use, denoted $\mathcal{Z}_{max}$, is the configuration query that is a filter-ignorant renaming of $\mathcal{Q}$ itself. The precision of this configuration query is 1.0 because it fully covers $\mathcal{Q}$, and it is represented by the point closest

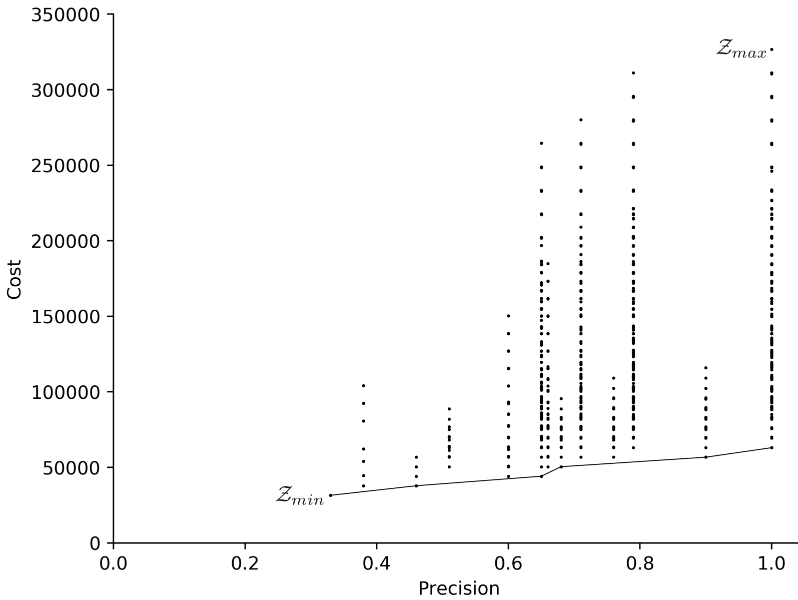


Figure 6.12: Chart showing all considered configuration queries when using Query 6.2 to calculate precision. The connected configurations are Pareto optimal.

to the upper right corner. While $\mathcal{Z}_{min}$ is Pareto optimal over all queries we consider, this is not always the case for $\mathcal{Z}_{max}$. For example, in Figure 6.12 we see that there exists another Pareto optimal configuration query below $\mathcal{Z}_{max}$ with a precision of 1.0 but lower cost than $\mathcal{Z}_{max}$.

If we were to also explore configuration queries with branches outside of $\mathcal{Q}$, which would not contribute to higher precision, only higher cost, we would get configurations placed right above the points we already have. Such configuration queries would never be Pareto optimal, so we are not missing any Pareto optimal configuration queries by not considering them.

If we only consider the Pareto optimal configuration queries over Query 6.2, they make a non-decreasing sequence of points in the chart, where the first point corresponds to $\mathcal{Z}_{min}$. $\mathcal{Z}_{min}$ has a cost of about 30000, and a precision of 0.33, while the Pareto optimal configuration query with perfect precision has a cost of 63000. In other words, with about twice the index size, the precision increases from 0.33 to 1.0.

We want to do a similar analysis over all queries, but in order to compare the results, we need to first normalize the cost axis by dividing each cost by the cost of $\mathcal{Z}_{min}$. When we do this for the results of Query 6.2, we get the chart in Figure 6.13. Since $\mathcal{Z}_{min}$ only contains the root variable and all local properties, we can then interpret the normalized cost as to how large the index will be compared to the index needed in the case of standard faceted search.

Figure 6.14 presents the normalized results of all the queries in the query log

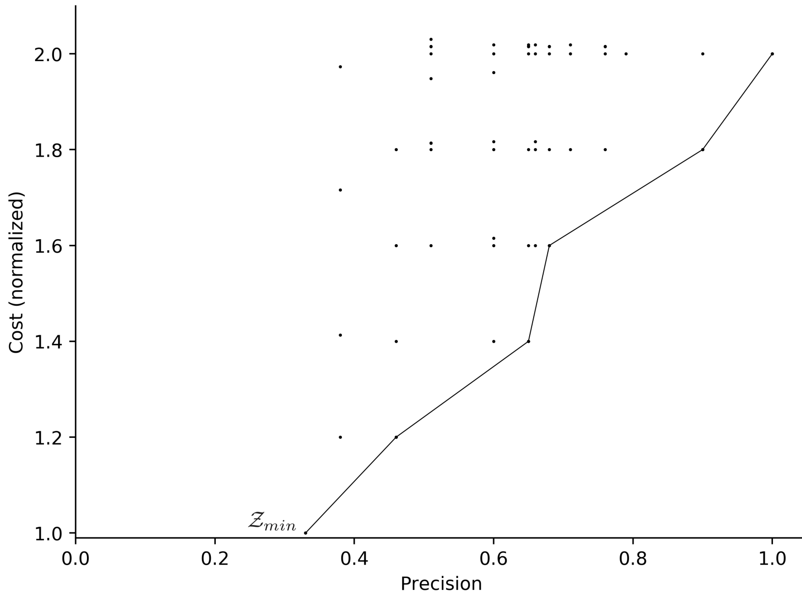


Figure 6.13: The same results as in Figure 6.12, just with a normalized y-axis.

in the same chart.

The overall results from Figure 6.14 seems promising, as most of the transitions between Pareto optimal points (black line segments) are more horizontal than vertical. This indicates that with a clever selection of branches in the configuration query, one can transition to much higher precisions without having to increase the cost very much. The median (red curve) and upper quartile (blue curve) have similar horizontal profiles, but with a slight increase as they approach the perfect precision, resulting in a more convex curve. In other words, the last 10% of precision is more expensive than any other 10% increase.

When interpreting these results, it is important to remember that we here only evaluate over one query at a time and that in real life, one configuration query and its corresponding value function will need to provide values for multiple different queries. All these queries are likely to have many different root classes, and if the root class of a query does not match the root class of the configuration query $\mathcal{Z}$, then the value function $S_a^{\mathcal{Z}}$ will not be able to provide any better set of values than just Γ_v. In order to solve this problem, we are going to extend our system to use more than one configuration query at a time.

6.4 Configuration Sets

So far we have only considered one particular configuration query at a time and seen how its corresponding value function $S_a^{\mathcal{Z}}$ can be used to suggest values. This value function may return reasonable values when the root class of $\mathcal{Q}$ matches

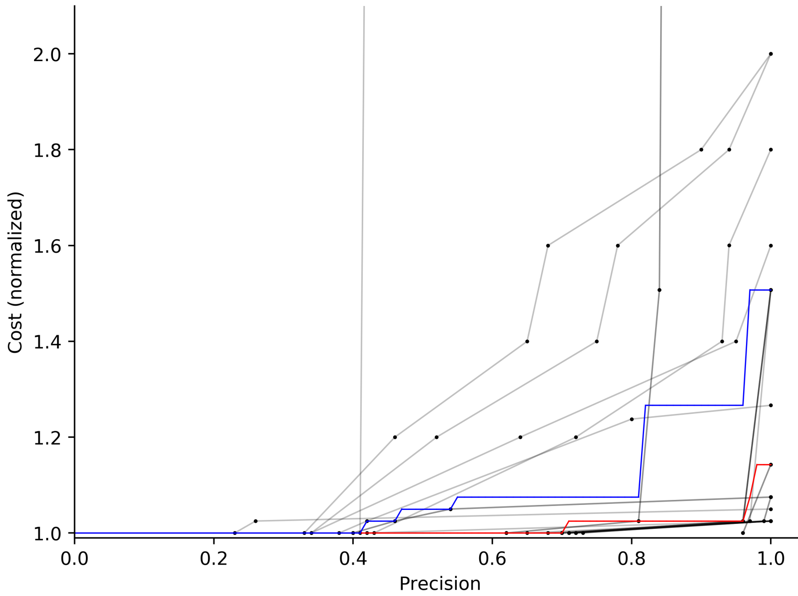


Figure 6.14: Pareto optimal configuration queries for all 29 queries with a normalized cost. The red curve shows the median, and the blue curve is the upper quartile.

the root class of $\mathcal{Z}$. However, when this is not the case, it will just return Γ_v, which is far from optimal.

In this section we define *configurations sets*, or just *configurations*, which are sets containing multiple configuration queries, possibly with multiple different root classes. Such a configuration set defines multiple value functions with corresponding indices, and all of these value functions can be combined into a complex value function with higher precision, but also higher cost than the value functions induced by each individual configuration query. This extension to configuration sets is essential when the goal is to achieve high precision over multiple real-life query construction sessions, where there are multiple extension cases to consider, usually with many different root classes.

Definition 6.4.1 (Configuration Set)**.** A *configuration set* is a finite set $\mathcal{W} = \{\mathcal{Z}_1, \mathcal{Z}_2, \ldots, \mathcal{Z}_n\}$ of configuration queries $\mathcal{Z}_1, \mathcal{Z}_2, \ldots, \mathcal{Z}_n$. $\dashv$

Each configuration query $\mathcal{Z} \in \mathcal{W}$ corresponds to a concrete value function $S_a^{\mathcal{Z}}$, which can be used to produce a set of values to suggest. If the root class of $\mathcal{Z}$ is not the same as the root class of $\mathcal{Q}$, or if $\mathcal{Z}$ does not cover the relevant extension pair, then it will default to only suggest the full set of data values Γ_v. But, if it overlaps to a high degree with $\mathcal{Q}$, then the resulting set of values will often be much closer to the actual set of productive values defined by X_o. If there is more than one configuration query with significant overlap, then their

corresponding value functions may all produce different value sets, which can be intersected to get results even closer to what X_o would produce. Based on this idea, we now formally define the value function $S_a^{\mathcal{W}}$ based on a configuration set $\mathcal{W}$.

Definition 6.4.2 (Configuration-based Value Function: $S_a^{\mathcal{W}}$)**.** Let $\mathcal{Q}$ be a query, let $\tau \in J(\mathcal{Q}, \mathcal{N})$, and let $\mathcal{W}$ be a configuration set. The configuration-based value function based on the configuration set $\mathcal{W}$, denoted $S_a^{\mathcal{W}}$, is defined as

$$S_a^{\mathcal{W}}(\mathcal{Q}, \tau) = \Gamma_v \cap \bigcap_{\mathcal{Z} \in \mathcal{W}} S_a^{\mathcal{Z}}(\mathcal{Q}, \tau)$$

⊣

This value function is well-defined, because $S_a^{\mathcal{Z}}(\mathcal{Q}, \tau)$ is defined to return a subset of Γ_v even in the cases when $\mathcal{Z}$ does not cover τ (see Definition 6.1.5). Also, note that we intersect with Γ_v, just like we did in Definition 6.1.8, which means that S_a will just return Γ_v when $\mathcal{W}$ is empty.

All the precision formulas (see Section 5.4.2) we have defined for general value functions will also, of course, be valid for $S_a^{\mathcal{W}}$, and since there is a correspondence between $\mathcal{W}$ and $S_a^{\mathcal{W}}$ we will link each of these precisions directly to $\mathcal{W}$ itself. In other words, for each rooted query $\mathcal{Q}$, extension pair τ, unrooted query $\mathcal{Q}_u$ and query log $\mathcal{L}$, we define:

$$\begin{aligned} \mathsf{prec}_{\mathrm{C}}(\mathcal{W}, \mathcal{Q}, \tau) &= \mathsf{prec}_{\mathrm{C}}(S_a^{\mathcal{W}}, \mathcal{Q}, \tau) \\ \mathsf{prec}_{\mathrm{Q}}(\mathcal{W}, \mathcal{Q}) &= \mathsf{prec}_{\mathrm{Q}}(S_a^{\mathcal{W}}, \mathcal{Q}) \\ \mathsf{prec}_{\mathrm{U}}(\mathcal{W}, \mathcal{Q}_u) &= \mathsf{prec}_{\mathrm{U}}(S_a^{\mathcal{W}}, \mathcal{Q}_u) \\ \mathsf{prec}_{\mathrm{L}}(\mathcal{W}, \mathcal{L}) &= \mathsf{prec}_{\mathrm{L}}(S_a^{\mathcal{W}}, \mathcal{L}) \end{aligned}$$

The indices generated by the configuration queries in $\mathcal{W}$ are all independent, so the cost of $\mathcal{W}$ and its corresponding value function $S_a^{\mathcal{W}}$ must equal the sum of the costs of each individual configuration query in $\mathcal{W}$.

Definition 6.4.3 (Cost of Configuration Set)**.** The cost of a configuration set $\mathcal{W}$, denoted $\mathsf{cost}(\mathcal{W}, \mathcal{D})$, is the sum of the cost of each of its configuration queries. I.e.,

$$\mathsf{cost}(\mathcal{W}, \mathcal{D}) = \sum_{\mathcal{Z} \in \mathcal{W}} \mathsf{cost}(\mathcal{Z}, \mathcal{D})$$

⊣

We already know that large configuration queries in general result in both high precision, and high cost. There are similar general rules that hold for configuration sets:

- Configuration sets with *large configuration queries* will in general have both high precision and high cost.
- Configuration sets with *many configuration queries* will in general have both high precision and high cost.

In some cases, one can directly compare the configuration queries contained in two configuration sets, and infer from that a relationship between their costs and precisions. For example, if we construct a *successor* of $\mathcal{W}$, which is the new configuration set $\mathcal{W}'$ we get after extending one of the configuration queries in $\mathcal{W}$, then both the precision and cost of $\mathcal{W}'$ will be equal or higher than the precision and cost of $\mathcal{W}$. I.e.,

$$\begin{aligned}
\mathsf{prec}_{\mathrm{C}}(\mathcal{W}, \mathcal{Q}, \tau) &\leq \mathsf{prec}_{\mathrm{C}}(\mathcal{W}', \mathcal{Q}, \tau) \\
\mathsf{prec}_{\mathrm{Q}}(\mathcal{W}, \mathcal{Q}) &\leq \mathsf{prec}_{\mathrm{Q}}(\mathcal{W}', \mathcal{Q}) \\
\mathsf{prec}_{\mathrm{U}}(\mathcal{W}, \mathcal{Q}_u) &\leq \mathsf{prec}_{\mathrm{U}}(\mathcal{W}', \mathcal{Q}_u) \\
\mathsf{prec}_{\mathrm{L}}(\mathcal{W}, \mathcal{L}) &\leq \mathsf{prec}_{\mathrm{L}}(\mathcal{W}', \mathcal{L}) \\
\mathsf{cost}(\mathcal{W}, \mathcal{D}) &\leq \mathsf{cost}(\mathcal{W}', \mathcal{D})
\end{aligned}$$

for each rooted query $\mathcal{Q}$, unrooted query $\mathcal{Q}_u$, extension pair τ, and query log $\mathcal{L}$.

6.4.1 Special Configuration Sets

In Chapter 5, we defined five different value functions: S_o, S_l, S_r, S_d, and S_e. With our new extended framework, which uses configuration sets, we will now attempt to model these value functions. In order to do this, we need to introduce some special configuration sets, which we will also use in Chapter 8 when we consider the problem of generating optimal configuration queries.

$\boldsymbol{S_e}$ S_a is only able to produce value functions with perfect recall, but S_e has a recall of 0, which means that it is impossible to model S_e with S_a.

$\boldsymbol{S_d}$ S_d is supposed to return Γ_v in every case, which can be achieved by using the empty configuration set $\mathcal{W}_d = \emptyset$ over S_a. Without any configuration queries, $S_a^{\mathcal{W}_d}$ will not be able to suggest anything better than Γ_v itself, which is exactly what S_d does. Since $\mathcal{W}_d = \emptyset$ leads to a value function that always returns Γ_v, its precision will be close to 0. The cost of $\mathcal{W}_d$ will also be 0 since it does not contain any configuration queries, and hence indices are not required either. This cost and precision match the cost and precision we calculated for S_d in Chapter 5.

$\boldsymbol{S_r}$ S_r is defined to only keep the parts of the query that match the extension pair itself. This can be achieved by using a configuration set $\mathcal{W}_r$, consisting of many small configuration queries, one for each edge in $\mathcal{N}$, i.e.,

$$\mathcal{W}_r = \bigcup_{e=(t_s,p,t_t)\in \bar{E}(\mathcal{N})} \{\mathcal{Z}_e\} \tag{6.12}$$

where

$$\mathcal{Z}_e = ((\{v_r, v\}, \{(v_r, p, v)\}), \{v_r \mapsto t_s, v \mapsto t_t\})$$

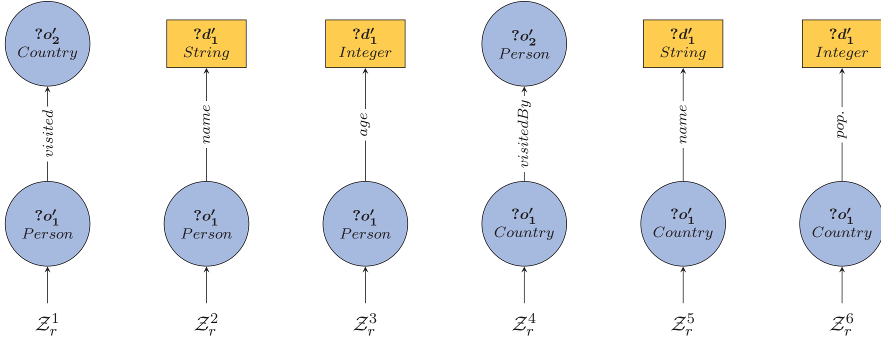


Figure 6.15: The six configuration queries in $\mathcal{W}_r$ when using the navigation graph from Example 4.2.3.

In the declaration of $\mathcal{Z}_e$ above, we have used $R_{\mathcal{Z}_e} = (\{v_r, v\}, \{(v_r, p, v)\})$ to represent the query tree of $\mathcal{Z}_e$. This notation differs from the notation we have used earlier in the thesis, but it is just shorthand notation for

$$R_{\mathcal{Z}_e} = \begin{cases} (\{v_r, v\}, \{\}, \{(v_r, p, v)\}) & \text{if } v \text{ is an object variable} \\ (\{v_r\}, \{v\}, \{(v_r, p, v)\}) & \text{if } v \text{ is a data variable} \end{cases}$$

This notation is more convenient to use in the above formula because whether v is an object variable or a data variable depends on the particular edge e in $\mathcal{N}$ that is considered. We will also use this notation later in the thesis.

To show what $\mathcal{W}_r$ may look like, consider the following example: if we use the navigation graph from Example 4.2.3, then all the six configuration queries of $\mathcal{W}_r$ are presented in Figure 6.15.

Now, given an extension case with a query $\mathcal{Q}$ and an extension pair $\tau \in J(\mathcal{Q}, \mathcal{N})$, there will only exist one single configuration query in $\mathcal{W}_r$ that covers τ, and after pruning, it will only return one pruned query, which consists of only the root of $\mathcal{Q}$ and the extension variable.

S_l In Definition 5.4.15, we defined the value function S_l to remove every part of $\mathcal{Q}_e$ that is not connected directly to the root variable of $\mathcal{Q}_e$, and then calculate values based on the remaining query. We are not able to model S_l exactly with S_a, because we only allow simple configuration queries, but by using a configuration set with multiple star-shaped, configuration queries, one for each class in $\mathcal{N}$, we get something that is relatively close to S_l. I.e., we define $\mathcal{W}_l$ to be the configuration set

$$\mathcal{W}_l = \bigcup_{t \in \bar{V}_o(\mathcal{N})} \{\mathcal{Z}_t\}$$

where $\mathcal{Z}_t$ is the star-shaped configuration query with a root variable v_r of type t, and one edge of the form (v_r, p, v) from v_r to v of type t' for each outgoing

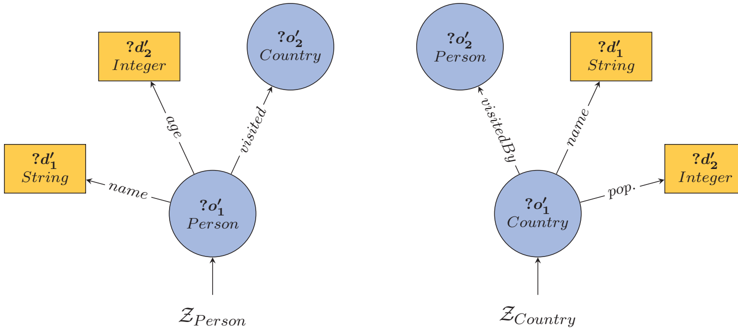


Figure 6.16: The two configuration queries in $\mathcal{W}_l$ when using the navigation graph from Example 4.2.3.

edge (t, p, t') of t in $\mathcal{N}$. For example, if we use the navigation graph from Example 4.2.3, then both of the configuration queries of $\mathcal{W}_l$ are presented in Figure 6.16.

$S_a^{\mathcal{W}_l}$ prunes away everything of $\mathcal{Q}_e$ that is not connected directly to the root, which is exactly what S_l does. But, in addition, it will need to prune away all additional branches if $\mathcal{Q}_e$ is non-simple. So, in general we have that $S_l(\mathcal{Q}, \tau) \subseteq S_a^{\mathcal{W}_l}(\mathcal{Q}, \tau)$, and if $\mathcal{Q}$ is simple, then $S_l(\mathcal{Q}, \tau) = S_a^{\mathcal{W}_l}(\mathcal{Q}, \tau)$.

$\boldsymbol{S_o}$ S_o is able to deal perfectly with non-simple queries, which is not something we can do with S_a, and because of this, we will not be able to model S_o with S_a. Additionally, since S_o never prunes away anything from $\mathcal{Q}_e$, we would need very large configuration queries in the configuration set. In fact, in order to support arbitrary large queries, they would need to be infinitely large, which is not possible in practice.

However, we can still make a configuration-based value function that approximates S_o quite well, by using a configuration set $\mathcal{W}$ with very large configuration queries. For example, if we for each class in $\mathcal{N}$ added one configuration query with all possible branches given by $\mathcal{N}$ down to a given depth δ, then this value function would return exactly the same as S_o for all non-simple queries $\mathcal{Q}$ of depth δ or less, and it would also give reasonable results for all the queries deeper than δ. This is likely not something that would work in practice though, since the corresponding index would be extremely large, even for small navigation graphs.

Another possibility, which may lead to very large configuration sets, is to generate the configuration set $\mathcal{W}_m$ that only extends as far as the set of queries in the query log requires. More precisely, for every branch from a potential root variable v_r in one of the queries of the query log, the configuration query of the type v_r must have a corresponding branch (there should only be one configuration query for each type). This configuration set can still not include

Set	Description
$\mathcal{W}_d$	The empty configuration set: $\mathcal{W}_d = \emptyset$.
$\mathcal{W}_r$	One small configuration query for each edge in $\mathcal{N}$.
$\mathcal{W}_r^d$	Variant of $\mathcal{W}_r$ where only data properties are included.
$\mathcal{W}_l$	One fully saturated configuration query for each class.
$\mathcal{W}_l^d$	Variant of $\mathcal{W}_l$ where only data properties are included.
$\mathcal{W}_m$	The maximum configuration set containing all paths in the query log.

Table 6.9: Summary of the six special configuration sets we have considered.

non-simple configuration queries, so it will not be able to achieve perfect precision for non-simple queries. But despite this, $S_a^{\mathcal{W}_m}$ is going to return very good results. Just like the configuration set covering queries down to a given depth δ, $\mathcal{W}_m$ will also lead to a very large index if the query log is large, so in practice, $\mathcal{W}_m$ will likely not be a good solution.

To summarize, we have now presented four different configuration sets: $\mathcal{W}_d = \emptyset$, $\mathcal{W}_l$, and $\mathcal{W}_r$, which both can be generated directly from $\mathcal{N}$, and $\mathcal{W}_m$, which must be generated from a query log. $\mathcal{W}_d$ can be used to model $S_d = S_a^{\mathcal{W}_d}$ and $\mathcal{W}_r$ can be used to model $S_r = S_a^{\mathcal{W}_r}$. $S_a^{\mathcal{W}_l}$ is not equivalent to S_l in general, only if the partial query is simple. $\mathcal{W}_m$ is also not a perfect copy of S_o in general, only if the partial query is non-simple, and covered by the query log used to generate $\mathcal{W}_m$.

In addition to these four configuration sets, we define $\mathcal{W}_r^d$ and $\mathcal{W}_l^d$ to be the two versions of $\mathcal{W}_r$ and $\mathcal{W}_l$ respectively, where all object properties have been removed, such that only data properties remain. For example, if we use the navigation graph from Example 4.2.3, then $\mathcal{W}_r^d$ would be the set containing $\{\mathcal{Z}_r^2, \mathcal{Z}_r^3, \mathcal{Z}_r^5, \mathcal{Z}_r^6\}$ from Figure 6.15, and $\mathcal{W}_l^d$ would be the set containing $\mathcal{Z}_{Person}$ and $\mathcal{Z}_{Country}$ from Figure 6.16, after removing the object variable $?o_2'$ from both of them. The reason why we define variants of $\mathcal{W}_r$ and $\mathcal{W}_l$ that only focus on data properties is that we want to see how good they are compared to $\mathcal{W}_l$ and $\mathcal{W}_r$ themselves. I.e., we would like to measure the effect of removing all object properties, similar to what we did in Experiment 1, where we defined two groups of configuration queries: one saturated with both data properties and object properties (ObjDat), and one saturated with only data properties (Dat).

A summary of the six configuration sets we have just presented is given in Table 6.9.

6.4.2 The Configuration Generation Problem

We have now completed the description of our configuration-based framework S_a, where a configuration set $\mathcal{W}$ defines a value function $S_a^{\mathcal{W}}$, which again can be used to make an estimate of the productive values when a partial query $\mathcal{Q}$ and extension pair τ is given. In order to ensure a high efficiency of $S_a^{\mathcal{W}}$, it is important that the index corresponding to $\mathcal{W}$ is constructed in advance of

all query sessions where $S_a^{\mathcal{W}}$ is going to be used. In other words, $\mathcal{W}$ must be specified during the configuration phase of the VQS it will be used for.

We do not know how to select a successful configuration query $\mathcal{W}$, because we do not know which queries the users are going to construct in the VQS. But, we have access to the dataset $\mathcal{D}$ during the configuration phase, and we also assume that we have access to a query log $\mathcal{L}$ with representative queries. A good configuration set is then one that has low cost with respect to $\mathcal{D}$, and high precision over $\mathcal{L}$.

For every realistic dataset $\mathcal{D}$, there will not only be one such configuration set but many, given by the frontier of all Pareto optimal configuration sets. In Section 6.3, we only discussed Pareto optimal *configuration queries*, but since we extended the cost and precision measures to also cover configuration sets, the principles of Pareto optimality will still apply to configuration sets.

Which of all these Pareto optimal configuration sets we actually want to use, depends on how much space we allow the index to use, or how precise we want the system to be. We can either specify a maximum cost, M, and then use the most precise configuration set with cost less or equal to M, or we can specify a minimum required precision P, and then use the cheapest configuration set with precision higher or equal to P. We will assume the former of these two cases, which leads us to the *configuration generation problem*:

Definition 6.4.4 (Configuration Generation Problem)**.** Let $\mathcal{D}$ be a dataset, let $\mathcal{L}$ be a query log, and let M be a maximum allowed cost. The *configuration generation problem* is to find the configuration set $\mathcal{W}$ with the highest precision, which also has cost less or equal to M. I.e., to calculate

$$\underset{\substack{\mathcal{W} \\ \mathsf{cost}(\mathcal{W},\mathcal{D}) \leq M}}{\arg\max} \quad \mathsf{prec}_{\mathrm{L}}(\mathcal{W}, \mathcal{L})$$

$\dashv$

This optimization problem is trivial to solve when the number of configuration sets is so small that one has time to loop over each of them and calculate the precision and cost, but this is almost never the case, due to the rapid growth of possible configuration sets. In addition, it will take a lot of time to actually calculate the cost and precision of any single configuration set, given our assumption that executing one single query may take minutes. Below we discuss these two challenges in detail.

The Search Space In order to find the optimal solution to the configuration generation problem, we need to consider every configuration set in the search space, so let us take a look at how many such sets there are. As long as $\mathcal{N}$ contains at least one object property, the set of possible configuration sets over $\mathcal{N}$ is infinite because it is possible to construct arbitrarily long configuration queries by traversing back and forth between the two types that are connected by the object property. We can reduce this to a finite number by only considering configuration queries where the branches are contained in the query log. Or,

phrased differently, if $\mathcal{W}_m$ is the configuration set generated from the provided query log, then we only need to consider configuration queries that are subqueries of a configuration query in $\mathcal{W}_m$.

The number of subqueries N of a particular configuration query $\mathcal{Z}$ with root variable v_r can be calculated exactly by using the following recursive formula: $N = n(v_r)$, where $n(v) = 1$ if v is a leaf variable, and

$$n(v) = \prod_{(v,p,v') \in \bar{E}(\mathcal{Z})} (n(v') + 1)$$

if v is an internal variable. We can make a rough estimate of N if we assume that $\mathcal{Z}$ has a branching factor of b and a depth of d: $N \approx 2^{(b^d)}$. A configuration set may select any subset of these configuration queries, which gives us approximately $2^N = 2^{(2^{(b^d)})}$ possibilities. Now, if we assume that there are c classes, and hence c configuration queries in $\mathcal{W}_m$ to select subqueries from, then the total number of possible configuration sets we have to consider becomes approximately $(2^{(2^{(b^d)})})^c = (2^{c \cdot (2^{(b^d)})})$. With a branching factor of $b = 2$, a depth of $d = 3$ and $c = 10$ classes, for example, the number of possible configuration sets will already be larger than 10^{700}. In other words, it will not be possible to consider every possible configuration set in the search space given a query log of say 1000 queries, with queries of decent size (5–10 variables), and a navigation graph with maybe 30 classes. In the calculations above we have not been considering the redundant configuration queries that are subqueries of other configuration queries, but even after removing these, there will still be too many possible configuration sets to consider.

In Experiment 2, we were able to calculate the cost and precision of every configuration query, which allowed us to both make the chart in Figure 6.12, and calculate all Pareto optimal configuration queries. The problem we have now looks similar, but it is much harder: now we work with configuration sets instead of configuration queries, and we evaluate precision over a full query log, not a single query.

Efficiency of Cost and Precision The second problem that makes the configuration generation problem hard, is the fact that calculating the precision and cost of a single configuration set $\mathcal{W}$ is too inefficient. In Section 8.1 we cover this in detail, but essentially the problem is that we have to execute multiple queries over $\mathcal{D}$ in order to calculate both the exact precision and cost.

In the two following chapters, we are going to work on the configuration generation problem. In order to solve the problem with the search space that is too large, we are going to use heuristics to select which configuration sets to consider, and which ones to discard (see Section 8.2). This process may discard the true optimal configuration set, but with good heuristics, the result should still be very good. In order to solve the problem with inefficient cost and precision calculation, we are going to use estimates of these measures (see Section 8.1).

The configuration generation problem requires a large query log $\mathcal{L}$ as input, and in order to make a good evaluation of the configuration generation algorithms, we need a more extensive query log than the one we used for Experiment 1 and Experiment 2, which only contains 29 hand-made queries for the NPD Factpages dataset. So, for the configuration generation evaluation, we make a benchmark based on Wikidata, which we present in the next chapter.

Chapter 7

The Wikidata Benchmark

In order to evaluate the configuration generation methods presented in the next chapter, we need a setup to evaluate over. This setup must consist of the following three parts:

- A large dataset $\mathcal{D}$, preferably about a well-known domain.
- A navigation graph $\mathcal{N}$ that matches $\mathcal{D}$.
- An extensive query log $\mathcal{L}$ with complex queries over $\mathcal{N}$, which reflects the information need of human users.

The NPD Factpages setup, which we used for both of the experiments presented in Chapter 6, is not extensive enough for our needs. The main problem with this setup is the query log, which only contains 29 queries hand-made by the author. We need a query log that captures the information need of real human users, and this query log must contain many queries, and large queries.

Dead-end detection is not the only research problem related to OptiqueVQS and similar systems that requires a query log like this. For example, query extension ranking algorithms need realistic queries to check if a particular suggested order is helpful for the user. Ideally, a set of queries should be harvested from real-life use of the VQS itself, but there is a chicken and egg problem here: until the usability of the VQS improves, e.g., by providing suggestions that avoid dead-ends, there will be no intensive use of the VQS that could generate such queries.

In this chapter we present the *Wikidata benchmark*, which is an extensive setup based on Wikidata[1] (WD). While this benchmark was mainly constructed to be used for the evaluation of our configuration generation methods, it is of course well suited for evaluation of other systems related to ontology-based VQSs. In fact, the WD benchmark has already been used in another project, to evaluate an adaptive ranking method that attempts to place suggested object and data properties in VQSs in the right order [13].

Desirable Properties Since we do not have access to any in-use VQS to make a benchmark from, we need to construct it from some other source. Before we do that, let us first describe in detail the properties we are looking for in such a benchmark.

First of all, the benchmark should be as **realistic** as possible: we want a dataset and a corresponding navigation graph that has real-life applications, and we want queries over this navigation graph that matches real information needs

[1] https://www.wikidata.org/

of humans. In other words, we want to avoid a computer-generated or fabricated setup.

The benchmark must obviously also **fit the formal models** we have defined in this thesis. Any RDF graph fits our dataset model, and any ontology can be transformed into a navigation graph by using the projection algorithm of OptiqueVQS [50], or similar algorithms. We do not require that the dataset fully conforms to the navigation graph in our setup, but in order to make the evaluation interesting, at least some of the queries must have non-empty answers. In addition, we want a relatively complete dataset – the data quality must be so high that it makes sense to pose large, complex queries and expect useful answers. With respect to queries, we are more demanding, because our system does not support all features of SPARQL. Specifically, we need queries that are both typed and tree-shaped, without any of the complex SPARQL operators that we do not support, like `OPTIONAL` and `UNION`. In addition, the queries have to be legal with respect to the navigation graph.

Furthermore, the dataset, the navigation graph, and the query log must have a **suitable size**. None of them should be so small that the configuration generation problem becomes trivial. The dataset must contain enough triples to demonstrate that scalability with respect to data is sufficiently good, i.e., we want millions of triples. The actual number of queries in the query log is not very important, as long as it contains a representative sample of the queries that are going to be made in the future. But, this is often easier to achieve with many queries. The size of each individual query is more important: our system is intended for relatively complex information needs, so a query log consisting of queries with five or more variables would be ideal. For the configuration generation problem, the size of the navigation graph is not as important, but it should contain enough classes and properties to allow for a good variety of possible queries. On the other hand, if the navigation graph is very large, then more queries are needed in order to make sure that we have a representative sample.

Wikidata The largest query log we were able to find, was the query log over Wikidata (WD)[2] provided by the Knowledge-Based Systems group at TU Dresden [35].[3] This log consists of 3.5 million human-made queries from 2017 and 2018. Most of these queries are very small, with only one or two variables, and few of them have exactly the typed, tree-shaped form that we require. But, the query log also contains many queries with more than five variables, and some of the queries even have more than ten variables, so we decided to attempt to rewrite these queries into the typed, tree-shaped form we require, and use the resulting query log in the benchmark. The Wikidata dataset is a collaboratively edited knowledge base hosted by the Wikimedia Foundation.[4] The RDF version of the dataset contains 11 billion triples, pertaining to large parts of human

[2] https://www.wikidata.org/

[3] https://iccl.inf.tu-dresden.de/web/Wikidata_SPARQL_Logs/en

[4] https://wikimediafoundation.org/

knowledge. It is used actively by several projects, including Wikipedia, and all of the data is available under a free license. Wikidata does not provide any ontology to generate a navigation graph from, so in order to make a complete benchmark, we had to construct this navigation graph based on the WD dataset and the WD query log.

In the following three sections, we describe each of the three parts that constitute the WD benchmark: the WD navigation graph, the WD dataset, and the WD query log.

7.1 WD Navigation Graph

Wikidata does not provide a suitable ontology to generate a navigation graph from, so we have to construct a navigation graph by hand, based on the given WD query log and WD dataset.

In a complete setup, every query in the query log must conform to $\mathcal{N}$ (see Definition 4.2.9), so after making the navigation graph, the whole query log must be filtered with respect to it. Hence, if we construct a navigation graph that only contains classes and properties that occur infrequently in the original query log, then the resulting log after filtering will become very small.

The original WD query log contains many queries about persons and the properties and classes related to them (films, TV-series, countries, cities, etc.), so we decided to build a navigation graph based on this.

The resulting WD navigation graph contains 15 classes, 5 datatypes, and a total of 107 properties. All the 15 classes are presented in Table 7.1, together with an overview of how many incoming and outgoing properties each of them has. Based on the sum of incoming and outgoing properties, Person is the most central class, followed by Country and City. Both Film and Television Series are also central classes, while Capital, on the other hand, is not connected to any other classes. Notice that the navigation graph contains both City USA and Capital, which are both subclasses of City in real life. If the navigation graph had support for subclasses, then subclass relationships could have been added where it makes sense, but this is not the case, so our system just considers all of these three classes to be regular classes without any special relationship.

The datatypes only have incoming data edges, naturally. String is the most popular of the five datatypes, with 12 incoming data properties, while Double is not used at all. Notice also that six of the classes only have one outgoing object property, i.e., they are only connected to one other class. When the user focuses on one of these six classes, there will be no legal data property extensions, which means that dead-end detection will not be relevant. Hence, it is pointless to construct a configuration query rooted in one of these classes.

7.2 WD Dataset

The Wikimedia Foundation provides a public SPARQL endpoint to WD, but the server's timeout settings prevent us from using it as a part of the setup.

Type Name	Category	In	Out	DP	OP
Person	Class	30	37	7	30
Gender	Class	1	1	0	1
Profession	Class	1	1	0	1
Eye Color	Class	1	1	0	1
Hair Color	Class	1	1	0	1
Country	Class	11	16	5	11
City	Class	7	11	4	7
City USA	Class	6	10	4	6
Continent	Class	3	5	2	3
Capital	Class	0	0	0	0
Film	Class	7	8	1	7
Filmography	Class	1	1	0	1
Film Genre	Class	1	1	0	1
Television Series	Class	5	9	4	5
Award	Class	5	5	0	5
String	Datatype	12	0	0	0
Integer	Datatype	5	0	0	0
Datetime	Datatype	6	0	0	0
Location	Datatype	4	0	0	0
Double	Datatype	0	0	0	0
Sum	**-**	**107**	**107**	**27**	**80**

Table 7.1: The number of incoming and outgoing properties of each class and datatype in the WD navigation graph. In: Incoming properties. Out: Outgoing properties. DP: Outgoing data properties. OP: Outgoing object properties.

Furthermore, this version is not static: WD is an open, collaborative project, so the data in this database changes every day. The dataset is also not from the same time period as the queries in the query log (2017–2018). Therefore, instead of relying on this public SPARQL endpoint, we decided to run queries on a local copy of the dataset. Ideally, we would like to use the version of WD from the same period as that when the query log was gathered, but the only database dump we found from this period was too large to work with, so we had to instead use a database dump from 2015. This is two years before the time period the queries were constructed, but with about 1 billion triples in total, the dataset should still contain enough relevant data to make a good benchmark.

When we first tried to load all these triples into Blazegraph,[5] which is the same graph database as the Wikimedia Foundation uses to host WD, it failed because of hardware limitations. In order to solve this problem, we decided to filter the dataset with respect to the navigation graph. This removed over 98% of the dataset, which left us with about 17 million triples. This dataset was so small that we managed to load it into Blazegraph without problems.

[5] https://blazegraph.com/

Class	Frequency
Person	2771131
Gender	5
Profession	2204
Eye Color	10
Hair Color	10
Country	143
City	19460
City USA	133
Continent	10
Capital	341
Film	151632
Filmography	641
Film Genre	252
Television Series	19269
Award	11734
Sum	**2976975**

Table 7.2: The number of instances in the WD dataset typed to each of the 15 classes.

By removing everything in the dataset that does not conform to the navigation graph, we do not change the answers of the queries in the query log, since they also conform to the navigation graph. This means that all the costs and precisions stay unchanged after data filtering, which is important since they are essential metrics used in the definition of the configuration generation problem.

Table 7.2 presents all classes in the WD navigation graph and the number of instances in the final dataset that is typed to the given class. In total, the dataset contains 2976649 distinct instances, and in total there are 2976975 typing relationships. In other words, each instance has slightly more than one type on average. This is natural since the classes we have chosen tend to be disjoint in real life. The exception to this is the three classes Capital, City, and City USA: all American cities and capitals are also cities in real life, so all American cities and capitals should also by typed to City in the dataset.

Among all the instances in the dataset, over 93% are of type Person. The second most frequent class is Film with about 150 thousand instances, which corresponds to about 5%. Classes with many instances will in general lead to more costly indices, but only if they are combined with data properties that lead to a large number of different values, like persons and their family name for example.

Table 7.3 shows all types of triples that occur more than 100000 times in the WD dataset sorted by frequency. Notice that three of them (given name, date of birth, family name) are data properties related to persons, while the remaining ones are object properties that connect persons to other classes. In other words, persons, and data related to them, make up a large part of the

Source Class	Property	Target Class	Frequency
Person	given name	String	1888994
Person	date of birth	Datetime	1832109
Person	occupation	Profession	1736987
Person	country of citizenship	Country	1234515
Film	cast member	Person	503050
Person	place of birth	City	339316
Person	award received	Award	109858
Person	family name	String	109340

Table 7.3: All types of triples that occur more than 100000 times in the WD dataset sorted by frequency.

dataset. This is of course not a problem, it is quite common that a few classes dominate in a dataset, but it is something to be aware of. It is also important to remember that only properties that correspond to edges in the navigation graph are included here, and we know from Section 7.1 that many of the properties are related to persons.

7.3 WD Query Log

The original WD query log contains about 3.5 million human-made queries, but many of them do not have the typed, tree-shaped form our system requires. Furthermore, not all of the queries conform to the navigation graph described in Section 7.1. To solve this problem, we need to transform the query log, i.e., rewrite the queries into the required form.

7.3.1 Query Transformation Process

In this section, we describe all the ten modification steps that are needed to transform the original WD query log into the required form. All of the ten steps are formulated as rules, and by applying these rules exhaustively to the query log, it eventually gets the correct form. Some of the modification steps have to be done in a particular order. For example, it is not possible to check whether a query is tree-shaped or not before it has been reduced to a basic graph pattern. Other modifications, like merging identical queries, for example, can be done to the query log regardless of its state. Below, the steps are listed in the most natural order to apply them.

Recall that our formal model of the query log associates a given weight to each query. Initially, we give each of the queries a weight of 1.0, as we want each of them to be equally significant. But, as we modify the log, we will sometimes need to change this weight. In particular, when we merge identical queries, we add up their weights and assign the sum of the weights to the merged version, and when we divide a query into multiple new versions, we distribute the weight of the original query equally over all of them. This ensures that the total weight

of the query log is always the same and that the weight distribution of the final query log matches the distribution of the original query log as much as possible. The only exception to this is step 9, where empty, disconnected, and acyclic queries are just removed from the query log – this reduces the total weight of the query log.

1. Remove Solution Sequence Modifiers A *solution sequence modifier* in SPARQL is an operator that changes the solution sequence before it is returned to the user. It includes modifiers like `LIMIT`, `OFFSET`, `ORDER BY`, `GROUP BY`, `DISTINCT`, and `REDUCE`, in addition to variable projections. None of these operators are supported by our system, so in this modification step, all of them are just removed from the query. These modifiers also appear outside the main clause of the query, so by removing them, the important core part still remains.

2. Split Queries with UNIONs Queries that contain a `UNION` of two or more clauses are in this step divided into multiple queries, one for each of the clauses that are combined by the `UNION`. The assumption here is that if a query with a `UNION` is of interest, then all of the clauses it is composed of must also be of interest. The resulting queries should not each get the same weight as the original query, because this would put too high significance on original queries with many unions. Instead, the weight of the original query must be distributed equally over all the new versions.

3. Split Queries with OPTIONALs Queries that contain `OPTIONAL` clauses are not supported by our system, so they must be modified. We divide them into two new queries: one where both the `OPTIONAL` keyword and its corresponding clause are removed, and one where only the `OPTIONAL` keyword is removed, while everything inside the corresponding clause is kept. Each of the two new queries gets a weight equal to half of the weight of the original query in order to preserve the total weight of the query log.

4. Resolve Property Paths We expand finite property paths into their corresponding version of multiple triples and explicit variables. Property paths using the asterisk symbol (*) to denote paths of arbitrary length are modified into three different queries, using the property 0, 1, or 2 times respectively, and each of the three resulting queries gets a weight equal to one-third of the weight of the original query. The assumption here is that it is unlikely that a property is used three times or more, while it is about equally likely that it is used 0, 1, or 2 times. All other types of property paths are just discarded in this step.

5. Remove Remaining Unsupported Keywords All SPARQL keywords that are not resolved by any of the steps above, except for filters, are just removed in this step. This ensures that all the remaining queries only consist of basic triples and filters.

6. Resolve Unsupported Triples All triples in a query of the correct form must either connect two variables by a concrete property or type a variable to a concrete class. In this step, we remove all unsupported triples, except for those connecting a variable to a concrete entity, where we instead just replace the concrete entity with a new variable. E.g., a triple like $(P1, visited, ?v_1)$ is replaced by $(?v_2, visited, ?v_1)$, and a triple like $(?v_1, visited, Canada)$ is replaced by $(?v_1, visited, ?v_2)$, where $?v_2$ is a new variable.

7. Filter on Navigation Graph All queries in the final query log must conform to the navigation graph, so in this step, we remove every property triple that is not supported by a corresponding edge in the navigation graph and every typing triple that types a variable to a class that does not exist in the navigation graph.

8. Add Types to Variables Since our system requires types on all variables in the query, we need to add types to variables that are not typed. For each variable, we consider all incoming and outgoing edges, to determine the possible types for the variable. Then, for each possible assignment of types to the variables, we make a new, typed version of the query. The weight of the original query is distributed equally onto all of the new versions. In most cases, there is only one way to assign types to the variables, but if some classes have many of the same ingoing and outgoing properties, like City and City USA, then there may be multiple ways of adding types.

9. Remove Empty, Disconnected, and Cyclic Queries Our system does not support disconnected or cyclic queries, so this step removes those queries from the log. There will also be many empty queries in the log after all the modifications we have done in the above steps, and all of them must also just be discarded.

10. Merge Identical Queries Finally, we merge identical queries into one query with a new weight equal to the sum of the weights of each of the individual queries.

7.3.2 Transformed Queries

After transforming the query log according to the transformation process described above, we get a query log with 54195 unique queries and a total weight of 1415505.5. Since the total weight of the original query log was about 3.5 million, the transformation removed about 60% of the original weight. Whole queries are only removed in step 9, where empty, disconnected, and cyclic queries are discarded. Most of these discarded queries are empty, but they are not empty because they were empty in the original query log – they have been stripped down to nothing by the earlier steps.

Most of the transformed queries are still very small – almost 95% of the total weight comes from queries of size 1 and 2 (see Table 7.4). Queries with size one, i.e., queries with only one variable, does not lead to any extension cases,

Size	Weight	Weight (%)
1	684004.0	48.3
2	654876.8	46.3
3	44642.2	3.2
4	23778.6	1.7
5	6171.4	0.4
6	1588.5	0.1
7	317.1	≈0.0
8	76.8	≈0.0
9	32.1	≈0.0
10	8.8	≈0.0
>10	9.1	≈0.0
Sum	**1415505.4**	**100.0**

Table 7.4: Weight of the queries grouped by size after the transformation process.

since they do not have any data edges (see Definition 5.4.10), so they will not be useful, and can in fact just be ignored. But, even after removing queries of size 1, over 89% of the remaining queries have size 2. With queries of this size, there is no point in using our configuration-based value function, because a simple value function, like S_r, will be able to perform extremely well with a very low cost. Hence, we generate two logs based on the set of transformed queries: $\mathcal{L}_A$ (A for all queries), where all queries of size ≥ 2 are included, and $\mathcal{L}_B$ (B for big queries), where only queries of size ≥ 6 are included. The idea is that by only including large queries in $\mathcal{L}_B$, we get a query log that better simulates the scenario and the type of users our dead-end detection system is made for: trained professionals with complex information needs. $\mathcal{L}_A$, on the other hand, can be used to find out how useful dead-end detection is for the more general case where most of the queries are really simple.

After filtering on size, these two logs still contain 53822 and 7895 unique queries respectively, which is quite many, considering the fact that we need to loop over all of them to calculate the precision of a single configuration query (see Definition 5.4.11). In order to reduce the number of unique queries while keeping most of the weight, we, therefore, remove queries with low weight from both $\mathcal{L}_A$ and $\mathcal{L}_B$. More specifically, we construct the two query logs as follows:

- $\mathcal{L}_A$: Queries with 2 or more variables and weight ≥ 10.0
- $\mathcal{L}_B$: Queries with 6 or more variables and weight ≥ 0.1

Our choice of removing all queries with low weight in the two logs is a very effective way to reduce the number of unique queries, while at the same time preserving most of the weight. In $\mathcal{L}_A$, this reduction removes 95% of the unique queries while keeping 94% of the weight, and in $\mathcal{L}_B$, the reduction removes 69% of the unique queries while keeping 93% of the weight (see Table 7.5).

The size distributions of the two resulting logs, and to which degree the queries are simple or not, are presented in Table 7.6 and Table 7.7. Since 93%

Description	$\mathcal{L}_A$	$\mathcal{L}_B$
Total weight before reduction	731501.4	2032.5
Total weight after reduction	686757.3	1899.8
Unique queries before reduction	53822	7895
Unique queries after reduction	2608	2448

Table 7.5: The effect of removing queries with small weight from each of the two query logs.

Size	**Simple**	**Non-simple**	**Simple (%)**
2	638516.6	0.0	100.0
3	29012.5	1047.7	96.5
4	15641.1	389.5	97.6
5	1939.1	87.0	95.7
6	82.1	12.0	87.3
7	29.6	0.0	100.0
8	0.0	0.0	-
9	0.0	0.0	-
10	0.0	0.0	-
>10	0.0	0.0	-
Sum	**685221.1**	**1536.2**	**99.8**

Table 7.6: Overview of queries in $\mathcal{L}_A$ grouped by size.

Size	**Simple**	**Non-simple**	**Simple (%)**
6	1089.7	409.6	72.7
7	199.3	86.9	69.7
8	29.1	42.5	40.6
9	10.5	16.4	39.0
10	2.0	4.8	29.6
>10	7.0	2.0	77.8
Sum	**1337.7**	**562.2**	**70.4**

Table 7.7: Overview of queries in $\mathcal{L}_B$ grouped by size.

Class	$\mathcal{L}_A$ (%)	$\mathcal{L}_B$ (%)
City	28.6	6.1
Person	21.9	31.6
Country	15.5	15.5
Continent	11.1	0.3
City USA	11.0	6.1
Profession	7.8	21.8
Gender	1.8	15.9
Television Series	1.2	≈0.0
Film	0.6	≈0.0
Award	0.2	2.4
Eye Color	0.2	≈0.0
Hair Color	≈0.0	0.2
Capital	0.0	0.0
Film Genre	0.0	0.0
Filmography	0.0	0.0
Sum	**100.0**	**100.0**

Table 7.8: Popularity of each class in the two query logs $\mathcal{L}_A$ and $\mathcal{L}_B$.

of the query weight in $\mathcal{L}_A$ has size 2, and since all of this weight corresponds to simple queries, the total fraction of simple queries in $\mathcal{L}_A$, measured by weight, is 99.8%, which is very high. $\mathcal{L}_B$ has an overall lower fraction of simple queries, of 70.4%.

Table 7.8 shows to which degree the variables in the two query logs are typed to the different classes. For each class, we have computed the weighted sum of all instances that are typed to this class. The table shows the percentage of the weighted sum over all instances in the whole log. The two query logs have a similar focus: persons and different places (cities, countries, continents) are the most popular classes, while film genres, capitals, filmographies, are all non-existent. There are also classes that are just barely represented, like films, TV-series, and awards.

It might seem unlikely that a company invests in the curation of data that is not used in queries. On the other hand, it might be the case that this data is used in predefined dashboards or reports, and just not in the ad-hoc queries we consider, so keeping all the infrequent classes should not be a problem. A good configuration generation algorithm should not focus on these infrequent classes, and by including them in the navigation graph, we can see if this is indeed the case. Furthermore, the WD benchmark may be used to evaluate other systems where these infrequent classes play a more central role.

The class frequency in $\mathcal{L}_A$ and $\mathcal{L}_B$ indirectly tells us which properties the two query logs contain. For example, the property that relates a Person to the city they live in will probably appear very frequently in both query logs, while the property that links a film to its film genre will not be present at all. This gives us an idea of what a useful configuration set may look like: it should include

properties and classes that are used frequently in the queries, so configuration queries about cities, persons, and countries should do well.

The benchmark we have created and presented in this chapter is not perfect. It does not use the original WD query log, but instead, it uses a transformed version where queries have the required form. In addition, the navigation graph is made by hand, based on the query log and the data. But, the setup does fulfill most of our requirements, and based on the assets we had access to, we are satisfied with the final result. And, most importantly, the benchmark is sufficient for the evaluation of the configuration generation, which we present in the next chapter.

Chapter 8

Configuration Generation

In Chapter 6 we described the index-based query extension framework and how it can be used to turn any predefined configuration set into a value function that can be efficiently calculated by using the corresponding index. Given a fixed dataset and query log, we then described how to calculate the cost and precision of such a configuration set, and how both precision and cost tend to increase with respect to its size. Furthermore, we presented the configuration generation problem, which is to find the most precise configuration set with a cost less than or equal to a given cost threshold M.

In this chapter, we present a collection of different search methods, which all attempt to solve the configuration generation problem. The search space of possible configurations these methods have to search over is very large in general (see Section 6.4.2), so the search methods will need to prioritize which of the configurations to actually consider. Furthermore, with a setup of decent size, it will take too much time to actually evaluate each configuration exactly as defined by the precision and cost formulas in Definition 5.4.11 and Definition 6.2.17. More precisely: it will be possible to calculate the exact cost and precision of a single configuration set, but since the search needs to evaluate many configurations, this evaluation has to be very efficient. In order to overcome this problem, we introduce efficient cost and precision estimates that can be used instead of the exact measures.

This chapter has three sections. Section 8.1 describes in detail the cost and precision estimates required to evaluate single configuration sets efficiently, while Section 8.2 presents all the different search methods we consider. Finally, in Section 8.3, we present the evaluation of our methods, which is based on the WD benchmark from Chapter 7.

8.1 Cost and Precision Estimation

In this section, we present algorithms that allow us to estimate the cost and precision of a given configuration set $\mathcal{W}$ efficiently. More specifically, we will consider the cost and precision formulas we have already presented, and detect which parts of them that are expensive to compute, and then we will present efficient alternatives to those parts.

Cost Estimation According to Definition 6.4.3, the cost of $\mathcal{W}$ equals the sum of the costs of each of its included configuration queries:

$$\mathsf{cost}(\mathcal{W}, \mathcal{D}) = \sum_{\mathcal{Z} \in \mathcal{W}} \mathsf{cost}(\mathcal{Z}, \mathcal{D}) = \sum_{\mathcal{Z} \in \mathcal{W}} (|\bar{V}(\mathcal{Z})| - 1) \cdot |\,\mathsf{ans}_E(\mathcal{Z}, \mathcal{D})|$$

The process of calculating this for any $\mathcal{Z}$ is not going to be efficient enough for our search, because it requires the exact size of $\mathsf{ans}_E(\mathcal{Z}, \mathcal{D})$, which can only be obtained after querying over $\mathcal{D}$. This is the most time-consuming part of the cost calculation process, so if we can find an efficient way to estimate $|\mathsf{ans}_E(\mathcal{Z}, \mathcal{D})|$, for any arbitrary configuration query $\mathcal{Z}$, then we also have an efficient way of estimating $\mathsf{cost}(\mathcal{Z}, \mathcal{D})$, and hence $\mathsf{cost}(\mathcal{W}, \mathcal{D})$.

Precision Estimation The precision of a configuration set $\mathcal{W}$ over a query log $\mathcal{L}$ is a weighted average of all the specific extension cases that are generated from the query log (see Definition 5.4.11). If $\mathcal{Q}$ is the rooted query and τ is the extension pair of such an extension case, then the precision of this case is given by Definition 5.4.6:

$$\mathsf{prec}_{\mathrm{C}}(S_a^{\mathcal{W}}, \mathcal{Q}, \tau) = \frac{|S_a^{\mathcal{W}}(\mathcal{Q}, \tau) \cap X_o(\mathcal{Q}, \tau)|}{|S_a^{\mathcal{W}}(\mathcal{Q}, \tau)|} = \frac{|X_o(\mathcal{Q}, \tau)|}{|S_a^{\mathcal{W}}(\mathcal{Q}, \tau)|} \tag{8.1}$$

We start by considering the numerator of this fraction. From Theorem 5.3.4, we know that $|X_o(\mathcal{Q}, \tau)|$ equals $|\mathsf{ans}_P(\mathcal{Q}_e, \mathcal{D}, v_e)|$, where $\mathcal{Q}_e$ is the extended version of $\mathcal{Q}$ with respect to τ, extended into a new, filterless data variable v_e. In order to calculate this exactly, we have to execute a query over $\mathcal{D}$, which is something we want to avoid. In other words, we need a way to efficiently estimate the cardinality of $\mathsf{ans}_P(\mathcal{Q}_e, \mathcal{D}, v_e)$.

Furthermore, we must consider the denominator in Equation 8.1, which can be expanded into this formula:

$$|S_a^{\mathcal{W}}(\mathcal{Q}, \tau)| = \left| \Gamma_v \cap \left(\bigcap_{\substack{\mathcal{Z} \in \mathcal{W} \\ \mathcal{Q}_s \in \mathsf{prune}(\mathcal{Q}_e, \mathcal{Z}, v_e)}} \mathsf{ans}_P(\mathcal{Q}_s, \mathcal{D}, v_e) \right) \right| \tag{8.2}$$

This is an intersection of all the value sets produced by the possible pruned versions of $\mathcal{Q}$ with respect to each configuration query $\mathcal{Z} \in \mathcal{W}$ where v_e is preserved. We need the cardinality of this intersection. Many of the considered configuration queries will not even cover the relevant extension pair, so they will only suggest Γ_v, which will not have any effect on the final intersection. Among the possibly very few configuration queries that actually cover the relevant extension pair, there may also be so little partial overlap that using just the smallest of all the sets we intersect over will be a relatively good approximation. I.e., we can assume that

$$|S_a^{\mathcal{W}}(\mathcal{Q}, \tau)| \approx \min \left(|\Gamma_v|, \min_{\substack{\mathcal{Z} \in \mathcal{W} \\ \mathcal{Q}_s \in \mathsf{prune}(\mathcal{Q}_e, \mathcal{Z}, v_e)}} |\mathsf{ans}_P(\mathcal{Q}_s, \mathcal{D}, v_e)| \right) \tag{8.3}$$

Alternatively, one could make an extensive analysis of all the pruned queries that are considered, and see how much their tree-structures overlap with each other, and the original query $\mathcal{Q}$, in order to calculate a better cardinality estimate of the intersection, but this is future work.

Regardless of how we choose to estimate the intersection of these sets, we still need an efficient way to estimate $|\,\mathsf{ans}_P(\mathcal{Q}_s, \mathcal{D}, v_e)|$. This is exactly the same conclusion we got when we analyzed the numerator $|X_o(\mathcal{Q}, \tau)|$, except that now we have to calculate ans_P of $\mathcal{Q}_s \sqsubseteq \mathcal{Q}_e$. But this will not be harder than calculating or estimating $|\,\mathsf{ans}_P(\mathcal{Q}_e, \mathcal{D}, v_e)|$, so the takeaway from our analysis is that if we can estimate $|\,\mathsf{ans}_P(\mathcal{Q}_e, \mathcal{D}, v_e)|$ efficiently, where $\mathcal{Q}_e$ is a query with a filterless data variable v_e, then we can efficiently calculate the precision of a configuration set over a query log.

Cardinality Estimation The problems of estimating the number of results returned by ans_P and ans_E are both *cardinality estimation problems*. Cardinality estimation in general is the problem of efficiently estimating the number of answers returned by a query over a given database. It is useful in any setting where efficiency is crucial, and where only a rough estimate of the number of answers is sufficient. Its most well-known application is in the context of query planning, where estimated cardinalities are used to decide the optimal order to execute joins in. The database community has studied cardinality estimation for decades, mostly in the context of relational databases [12, 33]. In recent years, cardinality estimation has also been studied in the context of RDF and SPARQL [41, 56, 45].

The most common approach to cardinality estimation of queries over a relational database is to base it on relatively simple statistics about the dataset, which is often gathered as new data is added. This could for example be the number of rows or columns in a table, the number of distinct values of a column, the number of null values in a column, or the distribution of values in one or more columns. Cardinality estimates can also be based on constraints over the database, like uniqueness constraints or key relationships between columns. Similarly, statistics can be prefetched from RDF graphs in order to support cardinality estimation for SPARQL queries. For example, one could count how often different entities occur in the subject or object positions in triples, which equals the number of incoming and outgoing edges their corresponding vertices have in the RDF graph.

By making certain assumptions, it is possible to calculate the number of answers to a query based on these statistics alone, but because these assumptions are almost never completely true, the calculated cardinality will just be an estimate. One common such assumption is the *independence assumption*, which states that any pair of joins affects the number of answers independently. Another common assumption is the *uniformity assumption*, which asserts that all values are distributed uniformly over the set of possible values. Another more advanced variant of this is to assume that the values follow another form of distribution. For example, many real-life datasets follow the skewed Zipfian distribution. We consider different such distributions in Section 8.1.2.

The queries we consider, where the structure is tree-shaped, and every variable is typed exactly once, are quite specific. Furthermore, when these queries are fed into ans_E, the resulting SPARQL query that actually is executed over the data

becomes very distinctive, with nested `OPTIONAL` clauses and `BOUND` on all object variables. We were not able to find a suitable method, or an existing system, that allows us to estimate the cardinality of these queries. Therefore, we had to develop a set of new cardinality estimation techniques, tailored for our special queries. In the remainder of this section, we present these cardinality estimation methods.

8.1.1 Basic Counts

Before we present how to estimate the cardinalities of the results produced by ans_P and ans_E, we need to gather some statistics from the dataset. More specifically, we will gather four kinds of counts, called *basic counts*, related to the classes and edges of $\mathcal{N}$:

- Class count
- Edge count
- Edge source count
- Edge target count

Class Count We start by defining the *class count* of a class $t \in \bar{V}_o(\mathcal{N})$, which is the number of distinct instances in $\mathcal{D}$ that are typed to t. If an instance belongs to multiple classes, that is fine, but in order to included in the count, t must be one of these classes. The class count can be calculated using ans_P on a query consisting of only one variable of type t, as described in the following definition:

Definition 8.1.1 (Class Count). Let $t \in \bar{V}_o(\mathcal{N})$ be a class in $\mathcal{N}$, and let $\mathcal{Q} = ((\{v\}, \{\}, \{\}), \{(v \mapsto t)\})$ for some variable $v \in \Gamma_{ov}$. The *class count* of t, denoted $C_c(t)$, is given by

$$C_c(t) = |\,\mathsf{ans}_P(\mathcal{Q}, \mathcal{D}, v)|$$

⊣

Edge counts The three remaining basic counts are all related to the edges of $\mathcal{N}$. Given any edge e of $\mathcal{N}$, we first consider the set of all e-edges in $\mathcal{D}$, which is the set of every edge $e' \in \mathcal{D}$ that satisfy $\mathsf{lab}(e) = \mathsf{lab}(e')$, $\mathsf{src}(e) \in T_{\mathcal{D}}(\mathsf{src}(e'))$, and $\mathsf{tar}(e) \in T_{\mathcal{D}}(\mathsf{tar}(e'))$. The number of such edges is called the *edge count* of e. Furthermore, we count the distinct sources and targets that occur in any of these edges, which gives us what we call the *edge source count* and *edge target count* of e. The definition below formalizes this and describes how each of the edge counts can be computed by using the answer functions we have defined earlier.

Definition 8.1.2 (Edge Counts). Let $e = (t_s, p, t_t) \in \bar{E}(\mathcal{N})$ be an edge in $\mathcal{N}$, and let $\mathcal{Q}$ be the filterless query defined by

$$\mathcal{Q} = ((\{v_s, v_t\}, \{(v_s, p, v_t)\}), \{v_s \mapsto t_s, v_t \mapsto t_t\})$$

The *edge count* of e, denoted $C_e(e)$, the *edge source count* of e, denoted $C_{es}(e)$, and the *edge target count* of e, denoted $C_{et}(e)$, are defined as:

$$C_e(e) = |\,\mathsf{ans}(\mathcal{Q}, \mathcal{D})|$$
$$C_{es}(e) = |\,\mathsf{ans}_P(\mathcal{Q}, \mathcal{D}, v_s)|$$
$$C_{et}(e) = |\,\mathsf{ans}_P(\mathcal{Q}, \mathcal{D}, v_t)|$$

⊣

Each basic count can be calculated easily by executing a simple query over $\mathcal{D}$, as described in the definitions above. Each of them will also likely be used many times, so if we cache these numbers, there will be no need to execute queries over the dataset anymore, which will ensure good performance of the estimation algorithms. A cache containing all basic counts will need to store one number per class in $\mathcal{N}$, and three numbers for each edge in $\mathcal{N}$, so the memory footprint of this cache will be about as large as the memory footprint of $\mathcal{N}$ itself.

Given how the basic counts are defined, we can infer some simple relationships between them. For example, the distinct source count of an edge must be lower than the class count of the edge's source – there cannot be more distinct persons who have visited any country than there are persons in total. The following theorem formalizes this and some other relationships that are always true for the basic counts.

Theorem 8.1.3. For every edge $e \in \bar{E}(\mathcal{N})$, the following two statements are always true:

$$C_{es}(e) \leq C_c(\mathsf{src}(e)) \tag{8.4}$$
$$C_e(e) \leq C_{et}(e) \cdot C_{es}(e) \tag{8.5}$$

If e is an object edge, then these statements are also true:

$$C_{et}(e) \leq C_c(\mathsf{tar}(e)) \tag{8.6}$$
$$C_e(e) = C_e(e^{-1}) \tag{8.7}$$
$$C_{es}(e) = C_{et}(e^{-1}) \tag{8.8}$$
$$C_{et}(e) = C_{es}(e^{-1}) \tag{8.9}$$

⊣

Proof. Both sides of Equation 8.4 are calculated by counting the number of projected answers returned by queries, but $C_c(\mathsf{src}(e))$ uses a query that is a subquery of the query used to calculate $C_{es}(e)$, hence, $C_c(\mathsf{src}(e))$ will be at least as large as $C_{es}(e)$. Equation 8.5 holds because with $C_{es}(e)$ possible source instances, and $C_{et}(e)$ possible target instances, the number of possible e-edges is at most $C_{es}(e) \cdot C_{et}(e)$: when every source is connected to every target. Equation 8.6 holds by the same argument as we used for Equation 8.4, but it can only be used when e is an object edges, because unless, $C_c(\mathsf{tar}(e))$ is not defined. Equations 8.7, 8.8, and 8.9 all holds because the answer function considers edges and their inverses to be equivalent. ■

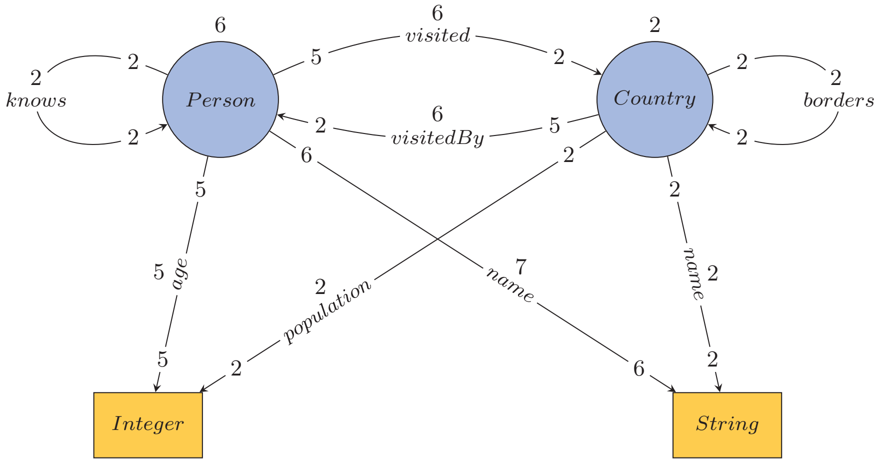


Figure 8.1: The navigation graph from Example 4.2.3 with basic counts included.

Example 8.1.4. Given the navigation graph $\mathcal{N}$ from Example 4.2.3 and the dataset $\mathcal{D}$ from Example 4.2.5, all the basic counts are given in Figure 8.1. The numbers above each of the two classes are their class counts: there are a total of 6 distinct persons, and 2 distinct countries in $\mathcal{D}$. Each edge is labeled with three numbers: the number closest to the source of the edge is the edge source count, the number near the middle of the edge is the edge count, while the number close to the target of the edge is the edge target count. For example, if $e = (Person, visited, Country)$, then $C_e(e) = 6$, $C_{es}(e) = 5$, and $C_{et}(e) = 2$. Notice that $e^{-1} = (Country, visitedBy, Person)$ has the same edge count as e, and that $C_{es}(e) = C_{et}(e^{-1})$, and that $C_{et}(e) = C_{es}(e^{-1})$. This is in accordance with Theorem 8.1.3. Notice also that the edge $(Person, knows, Person)$ is its own inverse, so its edge source count is equal to its edge target count.

♦

8.1.2 Edge Target Distributions

In addition to the four basic counts presented above, our cardinality estimation algorithms need access to an *edge target distribution function* H_e for each edge $e \in \bar{E}(\mathcal{N})$. The purpose of this function is to provide the estimation algorithms with a distribution that is supposed to match the distribution of entities in the target position of all e-edges in $\mathcal{D}$. H_e will be used both to estimate the effect of query filters and to estimate the number of distinct values in multisets, which is needed to estimate both $\texttt{ans}_P$ and $\texttt{ans}_E$.

Definition 8.1.5 (Edge Target Distribution Function)**.** Let $e \in \bar{E}(\mathcal{N})$ be an edge in $\mathcal{N}$. An *edge target distribution function* of e, is a probability distribution over $\Gamma_i \cup \Gamma_v$, i.e., a function $H_e : (\Gamma_i \cup \Gamma_v) \to [0, 1]$ that satisfies

$$\sum_{u \in (\Gamma_i \cup \Gamma_v)} H_e(u) = 1$$

⊣

We are going to allow H_e to be any possible distribution. For example, if we want to rely on the uniformity assumption, then we just need to define H_e to be uniform. But, in order to get the best possible estimates, H_e should be based on the true distribution of the edge targets in $\mathcal{D}$.

Definition 8.1.6 (Edge Target Distribution Function Based on Dataset)**.** Let $e = (t_s, p, t_t) \in \bar{E}(\mathcal{N})$ be an edge in $\mathcal{N}$. The edge target distribution function based on $\mathcal{D}$ is defined by

$$H_e(u) = \frac{|\{\pi \in \mathsf{ans}(\mathcal{Q}, \mathcal{D}) \mid \pi(v_t) = u\}|}{|\mathsf{ans}(\mathcal{Q}, \mathcal{D})|}$$

for all $u \in (\Gamma_i \cup \Gamma_v)$, where

$$\mathcal{Q} = ((\{v_s, v_t\}, \{(v_s, p, v_t)\}), \{v_s \mapsto t_s, v_t \mapsto t_t\})$$

⊣

When distributions are collected directly from a dataset like this, they are also called *histograms* over the dataset. While the true histogram defined above is preferred, it is often sufficient with an approximation based on samples of the dataset.

Notice that if $\mathcal{D}$ does not contain any e-edges, then the above definition will not be able to produce any distribution function. If this is the case, then a target distribution cannot be made based on $\mathcal{D}$, and one would need to define H_e in another way, e.g., one can assume that H_e is the uniform distribution.

Example 8.1.7. The histogram of the data edge $e_1 = (Person, name, String)$ induced by the dataset from Example 4.2.5, denoted H_{e_1}, is given by

$$\begin{aligned} H_{e_1}(Alice) &= 2/7 \\ H_{e_1}(Bob) &= 1/7 \\ H_{e_1}(Robert) &= 1/7 \\ H_{e_1}(Carol) &= 1/7 \\ H_{e_1}(Dave) &= 1/7 \\ H_{e_1}(Eve) &= 1/7 \end{aligned}$$

For every remaining entity $u \in (\Gamma_i \cup \Gamma_v)$, $H_{e_1}(u)$ equals 0.

The histogram of the object edge $e_2 = (Person, visits, Country)$ is given by

$$H_{e_2}(C_1) = 4/6$$
$$H_{e_2}(C_2) = 2/6$$

For every remaining entity $u \in (\Gamma_i \cup \Gamma_v)$, $H_{e_2}(u)$ equals 0. ♦

Like the basic counts, each H_e should be cached, such that $H_e(u)$ for a given entity u can be calculated efficiently without the need to query or sample from the data. The most naive way to do this would be to store each pair of an entity and its corresponding probability, but this would become infinitely large if $\Gamma_i \cup \Gamma_v$ is infinite, which is the case if Γ_v contains all possible strings, for example. A much better approach is to only store the entities with nonzero probability. If H_e is a histogram based on $\mathcal{D}$, then the cache corresponding to H_e will be finite, since we assume that the dataset is finite. An alternative is to just store a function that corresponds to the distribution we want. For example, if we want to use a uniform distribution of the range of natural numbers from 1 to 10, then we can represent that as a function that returns probability 1/10 for all natural numbers between 1 and 10, and 0 otherwise. Other general distributions can also be represented like this: Zipfian distributions and Gaussian distributions can be cached by storing the parameters that define them, and possibly the order the values occur if that is not obvious. We also mentioned bucketing in Section 6.2.4 as a technique to reduce infinite continuous ranges to a small, finite set of buckets, and this may help if caching is problematic in practice. In two of the three cases where the distribution functions will be used, only the distribution of data edges will be needed, while in the third case, distributions over both data edges and object edges will be required.

8.1.3 Cardinality Estimation: ans

We will now estimate the cardinality of $\mathsf{ans}(\mathcal{Q}, \mathcal{D})$ when $\mathcal{Q}$ is a rooted, filterless query. Later we extend this to queries with filters, which again can be used to make a cardinality estimate of $\mathsf{ans}_P(\mathcal{Q}, \mathcal{D}, v)$ where v is a filterless data variable of $\mathcal{Q}$. In order to estimate the cardinality of $\mathsf{ans}(\mathcal{Q}, \mathcal{D})$, we first need what we call the *branching factor* of an edge $e \in \bar{E}(\mathcal{N})$:

Definition 8.1.8 (Branching Factor)**.** The *branching factor* of an edge $e \in \bar{E}(\mathcal{N})$, denoted $\mathsf{bf}(e)$, is defined as

$$\mathsf{bf}(e) = \frac{C_e(e)}{C_c(\mathsf{src}(e))} \tag{8.10}$$

⊣

$\mathsf{bf}(e)$ is the number of e-edges in the dataset divided by the number of instances of type $\mathsf{src}(e)$. This means that $\mathsf{bf}(e)$ equals the expected number of outgoing e-edges an instance of type $\mathsf{src}(e)$ has in $\mathcal{D}$. For example, the branching factor of the edge $e = (Person, visited, Country)$, in the dataset of

Example 4.2.5, is equal to

$$\mathsf{bf}(e) = \frac{C_e(e)}{C_c(\mathsf{src}(e))} = \frac{C_e((Person, visited, Country))}{C_c(Person)} = \frac{6}{6} = 1$$

and this tells us that a person on average has visited exactly one country, i.e., the expected number of visited countries for a person is 1.

Now, let us go back to the filterless, rooted query $\mathcal{Q}$. If we only consider the root variable v_r of $\mathcal{Q}$, it can be assigned to exactly $C_c(T_\mathcal{Q}(v_r)) = C_c(T_\mathcal{Q}(\mathsf{root}(\mathcal{Q})))$ instances in $\mathcal{D}$, by the definition of the class count. I.e., we can exactly calculate the cardinality of the query consisting of only the root variable. We will use this as our initial estimate, and then we will update it as we consider the remaining variables of $\mathcal{Q}$. As we introduce new variables to the query, propagating out from the root, and assume that their instantiations will be independent of each other, we can update our cardinality estimate by multiplying the current estimate with the branching factor of $T_\mathcal{Q}(e)$, where e is the edge in $\mathcal{Q}$ that leads to the new variable. When doing this, we assume that each of the possible entities assigned to the source of e leads to $\mathsf{bf}(T_\mathcal{Q}(e))$ new target entities each. This is a fair assumption, but it is not always true, hence the new number we get by multiplying with the branching factor will be an estimate. For example, 70% of all persons overall may have a driving license, but if we only consider persons who own a car, then the fraction of persons with a driving license should be closer to 100%.

If we multiply our estimate by the branching factor corresponding to each edge e in $\mathcal{Q}$, we get the following cardinality estimate of $\mathsf{ans}(\mathcal{Q}, \mathcal{D})$, denoted $\widehat{\mathsf{ans}}(\mathcal{Q}, \mathcal{D})$:

$$\widehat{\mathsf{ans}}(\mathcal{Q}, \mathcal{D}) = C_c(T_\mathcal{Q}(\mathsf{root}(\mathcal{Q}))) \cdot \prod_{e \in \bar{E}(\mathcal{Q})} \mathsf{bf}(T_\mathcal{Q}(e))$$

We can modify this equation into a more convenient form, and the first step in this process is to replace $\mathsf{bf}(T_\mathcal{Q}(e))$ with a fraction of basic counts according to Equation 8.10.

$$\widehat{\mathsf{ans}}(\mathcal{Q}, \mathcal{D}) = C_c(T_\mathcal{Q}(\mathsf{root}(\mathcal{Q}))) \cdot \prod_{e \in \bar{E}(\mathcal{Q})} \frac{C_e(T_\mathcal{Q}(e))}{C_c(\mathsf{src}(T_\mathcal{Q}(e)))} \tag{8.11}$$

The denominator of the product in this equation can be rewritten to $C_c(T_\mathcal{Q}(\mathsf{src}(e)))$. Here $\mathsf{src}(e)$ refers to source variables in the query, so we can turn the product over edges into a product over vertices if we are able to calculate how often each of them is the source of an edge. But this is relatively easy to do since $\mathcal{Q}$ is tree-shaped: a variable v is the source of $n_v - 1$ edges, where n_v is the sum of incoming and outgoing edges of v. The exception to this is of course the root variable v_r, which appears as the source of an edge exactly n_{v_r} times.

Using this, we can pull out the denominator product from Equation 8.11, and replace it with a product over the vertices. After canceling the factor containing

the root, we get a cleaner cardinality estimate of $\mathsf{ans}(\mathcal{Q}, \mathcal{D})$:

$$\widehat{\mathsf{ans}}(\mathcal{Q}, \mathcal{D}) = \prod_{v \in \bar{V}_o(\mathcal{Q})} \left(\frac{1}{C_c(T_\mathcal{Q}(v))}\right)^{n_v - 1} \cdot \prod_{e \in \bar{E}(\mathcal{Q})} C_e(T_\mathcal{Q}(e))$$

This equation is independent of both the root of the query and the direction of its edges, which means that it can be used also on the unrooted version of $\mathcal{Q}$ and every other rooted version of $\mathcal{Q}$ where any of the object variables are set to be the root. In other words, it is an estimate that can be used for any filterless query.

Definition 8.1.9 (Estimated Cardinality of ans)**.** The estimated cardinality of a filterless query $\mathcal{Q} = (R_\mathcal{Q}, T_\mathcal{Q})$ over $\mathcal{D}$, denoted $\widehat{\mathsf{ans}}(\mathcal{Q}, \mathcal{D})$ is defined as

$$\widehat{\mathsf{ans}}(\mathcal{Q}, \mathcal{D}) = \prod_{v \in \bar{V}_o(\mathcal{Q})} \left(\frac{1}{C_c(T_\mathcal{Q}(v))}\right)^{n_v - 1} \cdot \prod_{e \in \bar{E}(\mathcal{Q})} C_e(T_\mathcal{Q}(e))$$

where n_v is the sum of incoming and outgoing edges of v. ⊣

Queries with Filters In order to estimate the cardinality of a query $\mathcal{Q} = (R_\mathcal{Q}, T_\mathcal{Q}, F_\mathcal{Q})$ with filters, we first need to estimate the cardinality of its filterless version $(R_\mathcal{Q}, T_\mathcal{Q})$ as described above. Then we can consider each data edge e in the query, one at a time, and multiply our estimate by a factor f_e representing the effect of the filters of $v = \mathsf{tar}(e)$. If we assume that the values that are assigned to v are distributed according to $H_{T_\mathcal{Q}(e)}$, then we want f_e to equal the fraction of answers accepted by the filter set of v, which we get by summing up $H_{T_\mathcal{Q}(e)}(u)$ for each data value $u \in F_\mathcal{Q}(v)$. I.e., we define f_e to be

$$f_e = \sum_{u \in F_\mathcal{Q}(v)} H_{T_\mathcal{Q}(e)}(u)$$

This gives us the following cardinality estimate of queries with filters:

Definition 8.1.10 (Estimated Cardinality of ans)**.** The estimated cardinality of a query $\mathcal{Q} = (R_\mathcal{Q}, T_\mathcal{Q}, F_\mathcal{Q})$ over $\mathcal{D}$, is defined as

$$\widehat{\mathsf{ans}}(\mathcal{Q}, \mathcal{D}) = \widehat{\mathsf{ans}}((R_\mathcal{Q}, T_\mathcal{Q}), \mathcal{D}) \cdot \prod_{e \in \bar{E}_d(\mathcal{Q})} \left(\sum_{u \in F_\mathcal{Q}(\mathsf{tar}(e))} H_{T_\mathcal{Q}(e)}(u) \right)$$

⊣

8.1.4 Cardinality Estimation: ans_P

Our estimate of $|\mathsf{ans}(\mathcal{Q}, \mathcal{D})|$ can now be used to estimate $|\mathsf{ans}_P(\mathcal{Q}, \mathcal{D}, v)|$, where v is a filterless data variable of $\mathcal{Q}$: First we define Y to be the multiset of data values defined by $\pi(v)$, for each $\pi \in \mathsf{ans}(\mathcal{Q}, \mathcal{D})$. $|\mathsf{ans}_P(\mathcal{Q}, \mathcal{D}, v)|$ is then just the

number of distinct data values in Y. We cannot calculate the data values in Y efficiently, because this requires that we actually calculate $\mathsf{ans}(\mathcal{Q}, \mathcal{D})$. But, we can estimate the number of distinct data values in Y by considering both its size and the distribution that determines which values it contains. This requires the following general theorem about how to estimate the number of distinct elements in a multiset:

Theorem 8.1.11. Let Y be a multiset with k elements sampled with replacement from the set U according to a probability distribution $H : U \to [0, 1]$. Then the expected number of distinct values in Y equals

$$\sum_{u \in U} 1 - (1 - H(u))^k$$

⊣

Proof. Let A_u be the random variable that equals 1 if element u is in Y, and 0 otherwise. The expected value of A_u, i.e., the probability that u is included in Y at least once, is equal to $\mathbf{E}(A_u) = 1 - (1 - H(u))^k$. The expected number of distinct values included in Y is now the sum of the expected value of each A_u, which equals

$$\mathbf{E}(\sum_{u \in U} A_u) = \sum_{u \in U} \mathbf{E}(A_u) = \sum_{u \in U} 1 - (1 - H(u))^k$$

■

Now, if e is the incoming edge of the projection variable v in $\mathcal{Q}$, then we can assume that the values in Y are sampled from Γ_v according to the distribution defined by $H_{T_\mathcal{Q}(e)}$. We also have an estimate of the number of sampled values k, because this equals the number of answers in $\mathsf{ans}(\mathcal{Q}, \mathcal{D})$, which we have an estimate for: $\widehat{\mathsf{ans}}(\mathcal{Q}, \mathcal{D})$. If we insert all of this into the formula in Theorem 8.1.11, we get the following cardinality estimate of ans_P.

Definition 8.1.12 (Cardinality Estimate: ans_P)**.** Let $\mathcal{Q} = (R_\mathcal{Q}, T_\mathcal{Q}, F_\mathcal{Q})$ be a query and let $k = \widehat{\mathsf{ans}}(\mathcal{Q}, \mathcal{D})$. Let v be a data variable in $\mathcal{Q}$, and let e be the data edge of $\mathcal{Q}$ where v is the target variable. The estimated cardinality of $\mathsf{ans}_P(\mathcal{Q}, \mathcal{D}, v)$, denoted $\widehat{\mathsf{ans}}_P(\mathcal{Q}, \mathcal{D}, v)$, is given by

$$\widehat{\mathsf{ans}}_P(\mathcal{Q}, \mathcal{D}, v) = \sum_{u \in \Gamma_v} 1 - (1 - H_{T_\mathcal{Q}(e)}(u))^k$$

⊣

8.1.5 Cardinality Estimation: ans_O

While we had to calculate estimates for ans and ans_P for arbitrary queries $\mathcal{Q}$, we only need estimates for ans_O and ans_E when they are applied to configuration queries, i.e., simple, rooted queries $\mathcal{Z} = (R_\mathcal{Z}, T_\mathcal{Z})$.

In Section 6.2 where we introduced ans_O, we saw how one can construct the index table of $\mathsf{ans}_O(\mathcal{Z}_2, \mathcal{D})$ by starting with a table corresponding to just the root variable (see Table 6.3), and then expand this by adding one new variable at a time, propagating away from the root. During this whole process, the number of rows (i.e., answer functions) in the index table increased from 6 to 9. This is mostly due to the person $P2$, who has visited two countries, and who has two names. In the final index table (Table 6.6), P2 is assigned to the root variable 4 times, while the other persons are only assigned to the root once each. We call this number the *expansion factor* of the given entity, so $P2$ has an expansion factor of 4, while $P1$ has an expansion factor of 1, for example. If we calculate the average expansion factor of all persons, we get 1.5, and we will assign this expansion factor to the variable $?o'_1$.

If we have the expansion factor of the root variable of $\mathcal{Z}$, then we can easily calculate the number of rows in $\mathsf{ans}_O(\mathcal{Z}, \mathcal{D})$ by multiplying the expansion factor with the number of instances that can be assigned to the root. In our example this would be $|\,\mathsf{ans}_O(\mathcal{Z}_2, \mathcal{D})| = 6 \cdot 1.5 = 9$.

We can extend the concept of an expansion factor to any other variable v of $\mathcal{Z}$ by defining it to be the expansion factor of the root variable (v) of the query that is formed by only considering v and all its descendants. This tells us how much we can expect the index table to expand when we include the descendants of v. We are going to use m_v to denote our estimate of the expansion factor of a given variable $v \in \mathcal{Z}$.

If v has exactly one outgoing edge e to another variable v_c, then we can calculate m_v by using the estimated expansion factor of v_c and basic counts. In particular, we get

$$m_v = (1-p) + p \cdot \frac{C_e(T_{\mathcal{Z}}(e))}{C_{es}(T_{\mathcal{Z}}(e))} \cdot m_{v_c} \quad \text{where} \quad p = \frac{C_{es}(T_{\mathcal{Z}}(e))}{C_c(\mathsf{src}(T_{\mathcal{Z}}(e)))}$$

Here p is the probability that a randomly chosen entity of type $T_{\mathcal{Z}}(\mathsf{src}(e)) = \mathsf{src}(T_{\mathcal{Z}}(e)) = T_{\mathcal{Z}}(v)$ has at least one outgoing e-edge. When this happens, then the relevant row will expand into a number of new rows, and the expected number of such rows is equal to $\frac{C_e(T_{\mathcal{Z}}(e))}{C_{es}(T_{\mathcal{Z}}(e))}$. Then, each of these new rows may expand even more when we consider the descendants of v_c, and therefore we also have to multiply by the expansion factor of v_c, which we have estimated to be m_{v_c}. There is then a probability of $(1-p)$ that the entity is not the source of any e-edges in $\mathcal{D}$, and when this is the case, we just get one resulting row where v_c is assigned to ω. All the descendants of v_c will then also be assigned to ω later, and this stops all expansion from this row.

If v has more than one outgoing edge, then we will assume that the expansion of each edge is independent of the expansion of all the other edges, i.e., we get a product over all the edges. Finally, if a variable has no outgoing edges, then it will not lead to any more expansion, i.e., its expansion factor is equal to 1.

Now we can calculate m_v for every variable in the query recursively, and finally multiply the expansion factor of the root with the number of possible root instances in order to get an estimate of the cardinality of $\mathsf{ans}_O(\mathcal{Z}, \mathcal{D})$.

Definition 8.1.13 (Cardinality Estimation ans_O)**.** Let $\mathcal{Z}$ be a configuration query with root variable v_r. The estimated cardinality of $\mathsf{ans}_O(\mathcal{Z}, \mathcal{D})$, denoted $\widehat{\mathsf{ans}}_O(\mathcal{Z}, \mathcal{D})$, is defined as

$$\widehat{\mathsf{ans}}_O(\mathcal{Z}, \mathcal{D}) = C_c(T_{\mathcal{Q}}(v_r)) \cdot m_{v_r}$$

where m_v is defined as follows

$$m_v = \prod_{e=(v,p,v_c)\in \bar{E}(\mathcal{Z})} 1 - \frac{C_{es}(T_{\mathcal{Z}}(e))}{C_c(\mathsf{src}(T_{\mathcal{Z}}(e)))} + \frac{C_e(T_{\mathcal{Z}}(e))}{C_c(\mathsf{src}(T_{\mathcal{Z}}(e)))} \cdot m_{v_c} \qquad \text{for all } v \in \bar{V}(\mathcal{Z})$$

$\dashv$

Notice that $m_v \geq 1$ for all variables $v \in \mathcal{Z}$, since $C_e(T_{\mathcal{Z}}(e))$ is always larger than $C_{es}(T_{\mathcal{Q}}(e))$. This makes sense when we think about $\mathsf{ans}_O(\mathcal{Z}, \mathcal{D})$ intuitively: adding variables to $\mathcal{Z}$ should never reduce the number of rows in $\mathsf{ans}_O(\mathcal{Z}, \mathcal{D})$.

8.1.6 Cardinality Estimation: ans_E

Given our estimate of $|\mathsf{ans}_O(\mathcal{Z}, \mathcal{D})|$, we are now going to make an estimate of $|\mathsf{ans}_E(\mathcal{Z}, \mathcal{D})|$. Every function $\phi \in \mathsf{ans}_O(\mathcal{Z}, \mathcal{D})$ corresponds to a particular $\phi' = f_E(\phi)$, where f_E is defined in Definition 6.2.14. Two functions $\phi_1, \phi_2 \in \mathsf{ans}_O(\mathcal{Z}, \mathcal{D})$ will both be mapped to the same function ϕ' if they map each data variable of $\mathcal{Z}$ to the same value, and if they map each object variable to ω only when the other function does it too. If this happens, then we get a reduction in the number of answer functions compared to $\mathsf{ans}_O(\mathcal{Z}, \mathcal{D})$.

In order to estimate the amount of distinct target functions left after the transformation defined by f_E, we can use the number of source functions in $\mathsf{ans}_O(\mathcal{Z}, \mathcal{D})$, which is approximately $k = \widehat{\mathsf{ans}}_O(\mathcal{Z}, \mathcal{D})$, and the number of possible target functions n, which we can also estimate, as described below.

First, we will let d_v denote our estimate of the number of possible ways of assigning entities to the variable v and all its descendants in $\mathsf{ans}_E(\mathcal{Z}, \mathcal{D})$. For each data variable v with incoming edge e, the number of possible assignments is equal to $C_{et}(T_{\mathcal{Z}}(e)) + 1$, where one is added to include the null symbol ω. I.e., we set $d_v = C_{et}(T_{\mathcal{Z}}(e)) + 1$ when v is a data variable. When v is an object variable, then it can be assigned to either χ or ω, and when it is assigned to χ, then each of its descendants v_c will contribute with d_{v_c} possible entities. All of them are independent, so we can multiply them to get

$$d_v = 1 + \prod_{v_c \in children(v)} d_{v_c}$$

When this formula is used on v_r, it will be off by one, since v_r cannot be assigned to ω. We are not going to define a separate formula for d_{v_r}, but instead we will subtract one when we calculate n, i.e., we set $n = d_{v_r} - 1$. For the sake of simplicity, we assume that each of the n possible answer functions is equally likely to occur. Now we can use Theorem 8.1.11 with a uniform distribution to calculate an estimate of $\mathsf{ans}_E(\mathcal{Z}, \mathcal{D})$, as described by the following definition.

Definition 8.1.14 (Cardinality Estimate ans_E)**.** Let $\mathcal{Z}$ be a configuration query with root variable v_r and let $k = \widehat{\mathsf{ans}}_O(\mathcal{Z}, \mathcal{D})$. The estimated cardinality of $\mathsf{ans}_E(\mathcal{Z}, \mathcal{D})$, denoted $\widehat{\mathsf{ans}}_E(\mathcal{Z}, \mathcal{D})$, is given by

$$\widehat{\mathsf{ans}}_E(\mathcal{Z}, \mathcal{D}) = n - n\left(1 - \frac{1}{n}\right)^k$$

where $n = d_{v_r} - 1$, and d_v is defined for every variable $v \in \mathcal{Z}$ below:

$$d_v = 1 + \prod_{v_c \in \mathsf{children}(v)} d_{v_c} \quad \text{when } v \in \bar{V}_o(\mathcal{Z})$$

$$d_v = C_{et}(e) + 1 \quad \text{when } v \in \bar{V}_d(\mathcal{Z})$$

where $e \in \bar{E}(\mathcal{Z})$ is the edge from $\mathsf{parent}(v)$ to v. $\dashv$

All the estimates presented in this section are relatively simple, but they allow us to make reasonable estimates of the cost and precision, which is what we need to perform a proper search. There are several ways to improve these estimates if needed. One possibility is to gather more detailed statistics about the dataset, and in particular, one should consider multidimensional histograms, which is a natural next step that has been studied before [38, 21, 58]. If this is too costly, one could instead analyze the data to discover which distribution each property is most similar too. As already mentioned, many real-life properties follow Zipfian or Gaussian distributions, and not uniform distributions, which are often assumed. Different constraints from the database schema can also be used, if this is given, to outline the distributions that are used. Finally, one could improve the main algorithms used to calculate the estimates or consider other angles of attack when estimating. For example, with the precision estimate we have presented, there is still a need to consider every possible extension case induced by the query log, one by one, and if there are many such cases, then the overall precision estimation will take a long time to compute. In order to improve this, one could for example make a summary of the query log, and compare the configuration set to this summary instead of comparing it to every query in the log, one at a time.

But, even if the cost and precision estimates we use are imprecise, our configuration generation methods may still be able to find optimal, or close to optimal configuration sets, since estimation errors often have similar effects on all configurations they are applied to. For example, if all estimated precisions are equal to 0.8 of their true precision, then the configuration set with the highest true precision will also have the highest estimated precision. The uncertainty of the cost function is a little more problematic because of the concrete cost threshold M we want to stay below. For example, if the cost estimates are too low, then the configuration generation methods may suggest a configuration with too high true cost, while if the cost estimates are too high, the configuration generation methods will fail in the sense that they are not using all the cost they are allowed to use, which will cause it to find a configuration with lower precision than the optimal one.

The actual time it takes to calculate the cost and precision depends on the dataset, the navigation graph, the query log, and of course, the configuration set itself. But, over the WD setup, where both of the query logs have between unique 2000 and 3000 queries, the evaluation time of a single configuration set was reduced from minutes and hours to about one second after introducing the estimates. Most of this time is consumed by the precision estimate because it has to consider each of the queries one by one. In the future, we would like to improve this by making a summary of the query log and compute a precision estimate much faster by comparing the configuration to only this summary. This would make the estimation time constant with respect to the number of queries in the log, i.e., it could reduce our estimates over WD from about a second down to milliseconds, which would allow us to search more extensively.

8.2 Search Methods

Now that we have established efficient cost and precision estimates, we are able to evaluate configuration sets fast enough to perform a proper search over the possible configurations. Recall what we are trying to achieve: we are looking for configuration sets with low cost and high precision. In particular, we are trying to solve the configuration generation problem defined in Section 6.4.2, i.e., we want to find the configuration set with the highest possible precision and a cost less than a given cost threshold M.

In this section, when we refer to the process of extending a configuration set $\mathcal{W}$, we mean to extend one of the configuration queries in $\mathcal{W}$ or to add a new configuration query with only one variable to $\mathcal{W}$. The resulting configuration set of this process is a new configuration set $\mathcal{W}'$. When $\mathcal{W}'$ can be produced by extending $\mathcal{W}$ once, we call $\mathcal{W}'$ a *successor* of $\mathcal{W}$, and we define $\mathsf{succ}(\mathcal{W}, \mathcal{N})$ to be the set of all successors of $\mathcal{W}$ with respect to the navigation graph $\mathcal{N}$, i.e., $\mathsf{succ}(\mathcal{W}, \mathcal{N})$ is the set of all configuration sets that can be made by adding just one more variable to $\mathcal{W}$.

For all non-trivial setups, the set of all possible configuration sets over $\mathcal{N}$ is infinitely large, because one can always produce a new configuration set by adding a new object edge to any of the configuration queries in an existing configuration set, or by adding a completely new configuration query to the set. But, in the context we have defined, the set of configuration sets we actually need to consider is limited by the maximum cost M. It is also limited by the given query log $\mathcal{L}$ and the maximal configuration set $\mathcal{W}_m$ that is defined by this query log (see Section 6.4): since $\mathcal{W}_m$ covers all non-simple parts of $\mathcal{L}$, we never need to consider any configuration sets larger than $\mathcal{W}_m$. More precisely, if there exists a sequence of extensions that transforms $\mathcal{W}_m$ into a configuration query $\mathcal{W}$, then the cost of $\mathcal{W}$ will be higher than the cost of $\mathcal{W}_m$, which means that it is worse than $\mathcal{W}_m$ with respect to both cost and precision.

If we place a directed transition edge from each configuration set to its successors, we get a directed acyclic graph (DAG) where $\mathcal{W}_d = \emptyset$ is the only configuration without any incoming transition edges. A natural way of

traversing this DAG is by doing a breadth-first-search starting from $\emptyset$, which first considers $\emptyset$, then all configurations in $\mathsf{succ}(\emptyset)$, then all configurations in $\{\mathsf{succ}(\mathcal{W}) \mid \mathcal{W} \in \mathsf{succ}(\emptyset)\}$, etc.

The number of outgoing transition edges of a given configuration set in this graph is determined by the navigation graph. In the beginning, when the configuration set is empty, then the number of possible successors is equal to the number of possible root classes. As more and more configuration queries are added, and each of them is populated with more variables, the number of possible successors tends to increase. More precisely: when a new data variable is added, then the number of possible successors decreases by 1 since we only allow simple queries, and since a data property does not lead to any new possible successors. But, when an object property is added, then the number of successors increases quite much instead. How much it increases depends on the type of the newly added variable, and how many outgoing properties its type has in the navigation graph. For example, if we consider the WD setup again, and a configuration set is extended with a variable of type Person, then the resulting configuration set will have $37 - 1$ new possible successors because Person has 37 outgoing properties, and -1 because this property cannot be added one more time (simplicity). On the other hand, if the type is Eye Color, then the number of new possible successors does not increase at all, because Eye Color can only lead to Person (see Table 7.1). The average number of outgoing properties of a class in the WD navigation graph is 7.1 if all the classes are weighted equally, but since the number of incoming edges is so unevenly distributed over the classes, there is a much higher chance of extending to a person than to an eye color, for example. If we take this into account, and if we consider both data properties and object properties, then we can calculate the expected increase in the number of successors after a new variable has been added. This is equal to 11.0 for the WD navigation graph. So after adding 10 variables, we should expect a configuration set with 110 successors, and after adding 10 more variables, then the expected number of successors is equal to 220. For comparison, $\mathcal{W}_l$ has 122 variables and 1585 successors.

If we discard all extensions that are not represented at all in the given query log, since we know that they will not lead to higher precision anyway, we both reduce the overall search space and the number of successors of each configuration. For the query log $\mathcal{L}_A$ (small queries) this reduction is about 10%, while it is about 50% for $\mathcal{L}_B$ (large queries). This still does not reduce the search space enough to allow brute-force search. For example, if we assume that the number of successors is reduced with about 50% such that the number of successors of a configuration with i variables equals $7 + 5i$, then the number of possible configuration sets of size 10 we need to consider is about

$$7 \cdot (7 + 5 \cdot 1) \cdot (7 + 5 \cdot 2) \cdot \dots \cdot (7 + 5 \cdot 9) \approx 1.0 \times 10^{14}$$

and this is too many configuration sets to evaluate, at least when we know that it takes about a second to estimate the cost and precision of a single configuration set.

There may exist some configuration queries in the configuration set that are redundant, in the sense that they are completely covered by at least one of the other configuration queries in the set. These configuration queries are completely useless because they only lead to indices that do not contribute to higher precision, only increased cost. In other words, these redundant configuration queries should be removed. Even if we prune away all these redundant configuration sets, and only consider configurations covered by $\mathcal{W}_m$ as described above, the search space will still be too large to do a brute-force search. So, in this section, we present some alternative heuristic-based search methods, which only consider parts of the search space. The actual results we got by applying these search methods to the WD setup are presented in Section 8.3.

Reference Configuration Sets In Section 6.4 we defined six different configuration sets: $\mathcal{W}_d$, $\mathcal{W}_r^d$, $\mathcal{W}_r$, $\mathcal{W}_l^d$, $\mathcal{W}_r$, and $\mathcal{W}_m$. Each of them is in themselves reasonable configuration sets, and since we have already discussed them quite extensively, it makes sense to use them as a frame of reference when we now present new methods and the configuration sets they generate. They are also natural default configuration choices if one does not have any knowledge about the data or the query log since they are symmetric with respect to the classes and properties in $\mathcal{N}$. In that sense they act as a good benchmark: if our search methods are not able to find any configurations that are better than the obvious defaults, then they are useless. The reference configurations can also be generated without much effort, i.e., we can generate them directly from $\mathcal{N}$ without considering $\mathcal{D}$ or the relevant query log. The two configuration queries $\mathcal{W}_d = \emptyset$ and $\mathcal{W}_m$ are also guaranteed to be Pareto optimal: all configuration sets with 0 cost will also have zero precision, just like $\mathcal{W}_d$, and $\mathcal{W}_m$ is the cheapest configuration query that is able to achieve the maximum precision (this may be less than 1 if there are non-simple queries in the query log). In other words, they are the two extreme endpoints of the Pareto frontier.

Greedy Query Weight Method If we consider the query log, there will be some properties that occur frequently, while others are not used at all. The *Greedy Query Weight Method* uses this to build a configuration set: it starts with one configuration query of only one variable for each class, and then it selects greedily the extension that corresponds to the most popular property in the query log not added yet. More precisely, the method calculates a score for every possible walk in the navigation graph, and this score is equal to the sum of the weights of the queries where a path corresponding to this walk is present. For example, to calculate the score of a walk from Person to Country via the property visits, the method must start with a total score of 0.0, and then it must loop over all queries in the query log, and check if it contains a variable typed to Person, which is also connected to a variable of type Country via the visit property. If so, then the weight of the query is added to the total score. The list of walks can then be sorted by their score, and this defines an order to add new variables and edges to the configuration set.

This method is not really a search method since it has to follow a certain sequence of extensions, which does not allow it to explore the search space. It also does not consider the data at all, it only assumes that properties that are used frequently in the query log lead to high precision, and it will include them in the index disregarding the cost they lead to.

If this method is given a maximum cost M, then it will only be allowed to extend if the result has a cost lower than M. This means that in some cases, it may have to skip the extension with the highest score, and rather select the one with the second-highest score. If the method is implemented without a maximum cost, then it will eventually reach the end of its extension sequence, at which point the configuration set includes every path that occurs in the query log and is equal to $\mathcal{W}_m$.

Random Method If we add new properties to the configuration set in random order instead of adding them by decreasing weight in the query log, then we get the *Random Method*. This method will, just like the Greedy Query Weight Method, eventually reach $\mathcal{W}_m$, it just picks a less optimal path to get there. The Random Method will for all non-trivial cases perform very poorly, so it should not be considered a real configuration generation alternative, but rather just a reference method that shows what random selection leads to.

Greedy Precision Method The *Greedy Precision Method* is also a greedy method, which starts with a configuration set $\mathcal{W} = \emptyset$, and then iteratively extends $\mathcal{W}$ by always selecting the successor of $\mathcal{W}$ with the highest precision. This method will rapidly find configurations of high precision if they exist in the set of possible successors. But, it does not consider the cost at all, so it may end up selecting a successor that leads to a very high cost. It is also not able to see more than one step forward, which means that it will fail to select successors with low precision, but high potential, in the sense that they lead to other configuration sets with very high precision. Just like the Greedy Query Weight Method, this method can be implemented with or without a maximum cost M.

None of the two greedy methods presented so far are guaranteed to return the optimal configuration set, but they work in a very intuitive way, and will often lead to reasonable solutions. Like any other greedy method, they will do well when the current best choice is also the choice that leads to the best result overall. Both methods will struggle when they have to make sacrifices in order to gain a large reward later.

Exploratory Method The problem with both of the greedy methods presented so far is that they are not able to look further than one step ahead, and hence will fail to select extensions that lead to large future rewards. This can be solved by looking further, and consider all configuration sets two or more steps ahead. But, this is problematic, because it increases the number of configuration queries that need to be evaluated by a large factor. For example, using query log $\mathcal{L}_B$ and after adding about 20 variables to the configuration set, the Greedy Precision

Method has to evaluate about 200 successors, which takes about 3 minutes. If it instead would have to consider all successors of these successors, which corresponds to about 40000 configuration sets, then this same evaluation task would take over 11 hours, which is too long for just one iteration.

There already exist heuristic-based search methods that are suited for search spaces with large branching factors. For example, Monte Carlo tree search (MCTS) [14, 47] has been used with success in game engines for Go and Chess, where the branching factor typically is very high. We did some initial experiments with MCTS over single configuration queries, but ended up discarding the method because it was not fast enough. We did not investigate in detail why this search method performed so poorly – it could be that our problem does not fit the algorithm, but we believe that it is more likely that the evaluation process is just too slow, or that the parameters we used for the search were too ambitious.

Based on the ideas of MCTS, we developed the *Exploratory Method*, which selects a successor $\mathcal{W}' \in \mathsf{succ}(\mathcal{W})$ based on the best precision among a set of 10 random extensions out from $\mathcal{W}'$. I.e., instead of considering every configuration two steps ahead as described above, we consider only ten random ones for each direct successor. This allows the method to find good configuration sets even if they are not directly connected to the current configuration, but it will only do so occasionally, when one of the random extensions are able to find a configuration with high precision. This method will only be successful if the precision improves after at most two steps, so if the configurations with high precision are three or more steps away, then this method will not be able to find them.

8.3 Evaluation

In this section, we evaluate the configuration generation methods presented in the previous section. For each of the search methods, we generate configuration sets over the WD setup presented in Chapter 7 and evaluate them using the cost and precision estimates described in Section 8.1. Recall that we defined two different query logs in Chapter 7: $\mathcal{L}_A$, which contains 2608 queries with seven or fewer variables, and $\mathcal{L}_B$, which contains 2448 queries, each with six or more variables. We first present the results based on the large queries in $\mathcal{L}_B$ in Section 8.3.1, and after that, we present the results based on the small queries in $\mathcal{L}_A$, in Section 8.3.2.

The way we have formulated the configuration generation problem in Definition 6.4.4 targets one particular optimal configuration set by asking for the most precise configuration set below a given maximum cost threshold M. All our methods can produce such a configuration in two different ways. Either the search method is given M before it starts, which prevents it from extending into a configuration set with a too high cost. Or, the method is applied without any cost threshold, and then, after it has produced enough results, the configuration with the best precision and a cost less than M is chosen. The former approach will give an overall better result because it actually uses M to make decisions. The latter approach allows us to make a sequence of configurations that tells

Set	Description
$\mathcal{W}_d$	The empty configuration set: $\mathcal{W}_d = \emptyset$.
$\mathcal{W}_r$	One small configuration query for each edge in $\mathcal{N}$.
$\mathcal{W}_r^d$	Variant of $\mathcal{W}_r$ where only data properties are included.
$\mathcal{W}_l$	One fully saturated configuration query for each class.
$\mathcal{W}_l^d$	Variant of $\mathcal{W}_l$ where only data properties are included.
$\mathcal{W}_m$	The maximum configuration set containing all paths in the query log.

Table 8.1: Summary of the six special configuration sets we have considered.

approximately how well the method performs for many different cost thresholds, which is overall more interesting. Hence, most of the runs we present in this section are done without a built-in cost threshold.

Adding a property to a configuration that is not used in the query log will not increase the precision, only the cost. Therefore, when we implemented the search methods, we limited them to only build configurations with paths that occur in the given query log. So, instead of considering every possible successor in $\mathsf{succ}(\mathcal{W}, \mathcal{N})$, the methods only look at successors that are covered by the query log. This improves the overall results because the search methods can spend time on the good candidate configurations. Two of the methods we consider, the Random Method and the Greedy Query Weight Method, are also programmed to only produce at most one configuration query per class.

Each method we consider produces multiple configuration sets, and we are going to present them all by their estimated precision and cost in the same kind of cost/precision diagram as we used in Experiment 2 (see Sec. 6.3.1). The estimates we use are the same as the ones presented in Section 8.1, but we only used uniform edge target distribution functions for the estimates.

All parts of the evaluation were done on a personal laptop computer with a 2.2 GHz processor and 16GB of RAM. Some of the runs, like the ones we did with the Random Method and the Greedy Query Weight Method, completed in less than a minute. Other methods, like the Exploratory Method, spent so much time that it had to be terminated after about 100 iterations and 10 hours.

8.3.1 Evaluation based on $\mathcal{L}_B$

Table 8.2 and Figure 8.2 present the precision and cost of each of the six reference configurations we defined in Section 6.4: $\mathcal{W}_d$, $\mathcal{W}_l$, $\mathcal{W}_l^d$, $\mathcal{W}_r$, $\mathcal{W}_r^d$, and $\mathcal{W}_m$, when evaluating them over the WD setup and query log $\mathcal{L}_B$. Table 8.1 is a copy of Table 6.9, to help recall the structure of all the six configurations.

Notice first that two of these configurations, $\mathcal{W}_r$ and $\mathcal{W}_r^d$, are presented with the same cost and precision. The only structural difference between these two configurations, is the extra configuration queries $\mathcal{W}_r$ has for each of the 80 object properties in $\mathcal{N}$. These configurations will not lead to any higher precision, because they do not contain any data variables. So $\mathcal{W}_r$ and $\mathcal{W}_r^d$ should indeed

Set	Precision	Cost
$\mathcal{W}_d$	0.00	0
$\mathcal{W}_r^d$	0.14	1.6×10^5
$\mathcal{W}_r$	0.14	1.6×10^5
$\mathcal{W}_l^d$	0.17	2.0×10^7
$\mathcal{W}_l$	0.72	8.4×10^{10}
$\mathcal{W}_m$	0.89	8.6×10^8

Table 8.2: The precision and cost of the six reference configuration sets $\mathcal{W}_d$, $\mathcal{W}_r^d$, $\mathcal{W}_r$, $\mathcal{W}_l^d$, $\mathcal{W}_l$, and $\mathcal{W}_m$ with respect to query log $\mathcal{L}_B$.

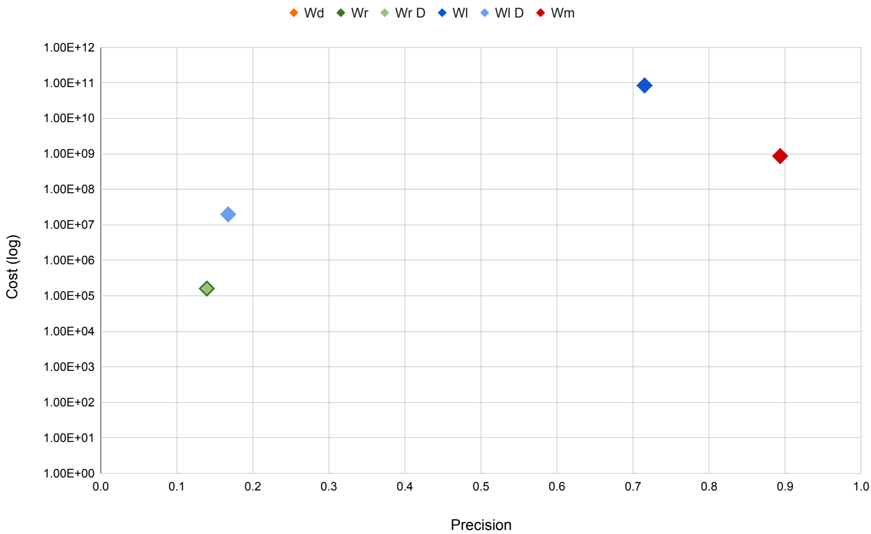


Figure 8.2: The precision and cost of the six reference configuration sets $\mathcal{W}_d$, $\mathcal{W}_r^d$, $\mathcal{W}_r$, $\mathcal{W}_l^d$, $\mathcal{W}_l$, and $\mathcal{W}_m$ over query log $\mathcal{L}_B$. $\mathcal{W}_d$ is not visible because the y-axis uses a logarithmic scale.

v_r	v
χ	χ

v_r	v
χ	ω

Table 8.3: The two types of index tables each of the 80 object property configuration queries in $\mathcal{W}_r$ can correspond to.

have the exact same precision. The cost of $\mathcal{W}_r^d$ and $\mathcal{W}_r$ is just *almost* the same: in reality, the cost of $\mathcal{W}_r$ is 80 more than the cost of $\mathcal{W}_r^d$, but this is so little that the difference is not visible in Table 8.2 with the precision we have used. This cost difference is a result of the extra object property configuration queries we just referred to. Each of them results in an index table similar to either of the two tables presented in Table 8.3, which both has cost 1 since the cell of the root column is not counted. With 80 such tables, the total cost of these extra configuration queries is equal to 80.

$\mathcal{W}_m$ contains large enough configuration queries to cover every non-simple query in the query log, so its precision of 0.89 is the highest that can be achieved with S_a over this query log. In order to beat this precision, we need another value function or framework that performs better on non-simple queries than what our framework does. Since $\mathcal{W}_m$ defines the highest possible precision, it also marks the highest possible cost that ever should be considered: it is pointless to use a more expensive configuration because then $\mathcal{W}_m$ will outperform it with respect to both cost and precision. But, $\mathcal{W}_l$ is exactly such a configuration, with its precision of 0.72 and cost of 8.4×10^{10}. $\mathcal{W}_l$ has such a high cost because it is generated based only on $\mathcal{N}$ and not the query log $\mathcal{L}_B$. In other words, $\mathcal{W}_l$ contains many local properties that are never actually used in the query log, and this gives it a very high cost and a lower precision than $\mathcal{W}_m$. A better alternative to $\mathcal{W}_l$ would be a version where all of these useless properties are removed. It would provide the exact same precision as $\mathcal{W}_l$, but with a lower cost than both $\mathcal{W}_l$ and $\mathcal{W}_m$. In fact, this is also the case with $\mathcal{W}_l^d$, $\mathcal{W}_r$, and $\mathcal{W}_r^d$: they all may contain properties that are useless because they are not used in any queries in $\mathcal{L}_B$. If we remove all these useless properties from these configurations, we get versions with the same precision but lower cost.

Since $\mathcal{W}_l^d$ is the variant of $\mathcal{W}_l$ where only data properties are included, it should have both lower cost and precision than $\mathcal{W}_l$. Our results show that this is indeed the case, but its precision is only 0.17, which is surprisingly low compared to the precision of $\mathcal{W}_l$. This shows the importance of including object variables in the configuration queries from a precision perspective, which is something we also discovered in Experiment 1 (see 6.1.3).

Both $\mathcal{W}_r$ and its variant $\mathcal{W}_r^d$ have very low precision scores of only 0.14, however, their cost is also very low: 1.6×10^5. Finally, $\mathcal{W}_d$ gets both a precision and cost of 0, as expected, since it is just the empty set.

Figure 8.3 shows the same reference configurations from Figure 8.2 together with 477 configurations generated by the Greedy Query Weight Method (yellow points), and 477 configurations generated by the Random Method (red crosses).

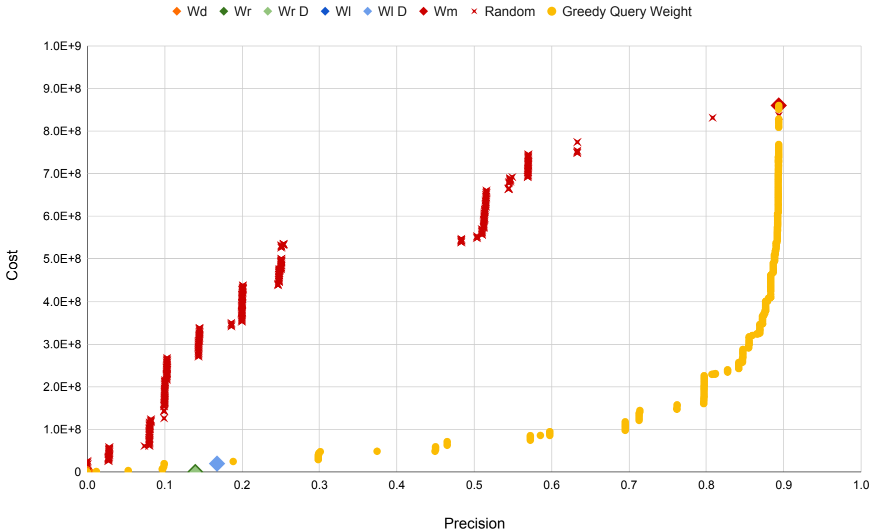


Figure 8.3: Configuration sets generated by the Greedy Query Weight Method (yellow) and the Random Method (red) over query log $\mathcal{L}_B$.

Notice that this chart uses a linear scale on the y-axis and a cost range that excludes $\mathcal{W}_l$ because of its high cost. Both of these methods make only one single configuration query per class, and they extend the set of all these classes by only adding properties that have been used in the query log. The Random Method adds these properties in random order, while the Greedy Query Weight Method adds them by decreasing popularity in $\mathcal{L}_B$. Regardless of how they choose to extend, they both eventually lead to the same configuration set, $\mathcal{W}_m$, where all paths from the query log are included. While the Greedy Query Weight Method should be considered to be a real attempt in generating useful configurations, the Random Method is just included for reference. The yellow points produced by the Greedy Query Weight Method form a convex function. I.e., the precision increases rapidly in the beginning when extensions with high weight are chosen, but it slows down as it approaches the maximum precision defined by $\mathcal{W}_m$. This is in contrast to the Random Method, which is slightly concave. This shows clearly that not every configuration query in the search space is useful, and that clever techniques are needed to find good configuration sets.

Since the Greedy Query Weight Method is not bounded by any maximum cost, it produces configuration queries with costs ranging from 0 and all the way up to the maximum cost defined by $\mathcal{W}_m$. Almost all of these configurations are Pareto optimal when we only consider the configurations generated by the Greedy Query Weight Method, since extending a configuration always increases the cost, and the precision never decreases (it may stay fixed for some datasets). But, if we also

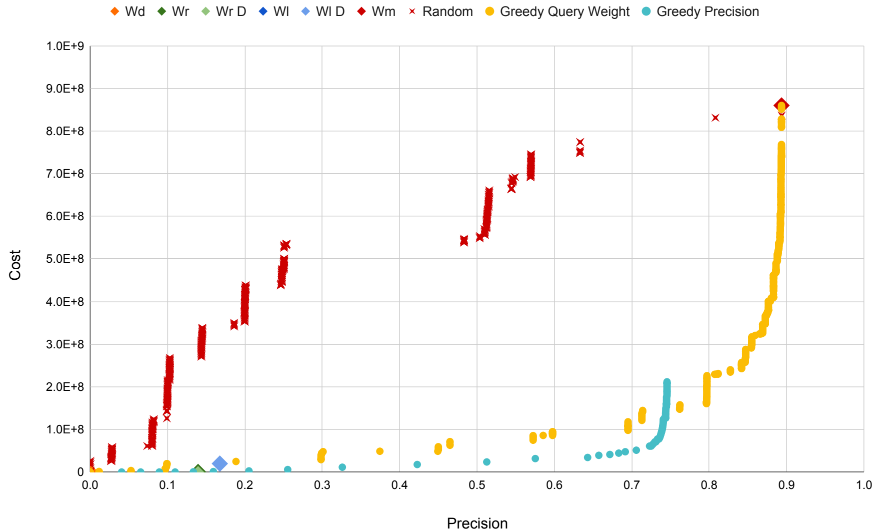


Figure 8.4: Configuration sets generated by the Greedy Query Weight Method (yellow), the Greedy Precision Method (teal), and the six reference configuration sets over query log $\mathcal{L}_B$.

consider all the six reference configurations, some of the greedy configurations will not be Pareto optimal anymore since they are weaker than either $\mathcal{W}_r$ or $\mathcal{W}_r^d$. The Greedy Query Weight Method produces numerous alternatives to the reference configuration sets, and given a maximum allowed cost M, one can just pick the most precise configuration set with a cost lower than M.

In Figure 8.4, we have added 72 configurations generated by the Greedy Precision Method (teal points). This method starts with the empty configuration set, and then it iterates by always transitioning to the successor with the highest precision. The method is very good at detecting and selecting properties that result in an immediate increase in precision, but it fails to select object properties that give it the potential to select valuable properties later. The Greedy Query Weight Method, on the other hand, is based on the assumption that extensions with high weight in the query log also leads to high precision. This causes it to make some suboptimal local choices with respect to precision, but it also leads it to select object properties with high future potential. We can see this effect in the results presented in Figure 8.4: the Greedy Precision Method performs better than the Greedy Query Weight Method in the beginning when there are still many good successors with high cost. But after reaching a precision of about 0.7, it runs out of good options and converges slowly towards a precision of 0.75, where it stagnates, because it keeps adding data properties with extremely little precision gain, instead of selecting an object property with no direct gain, but

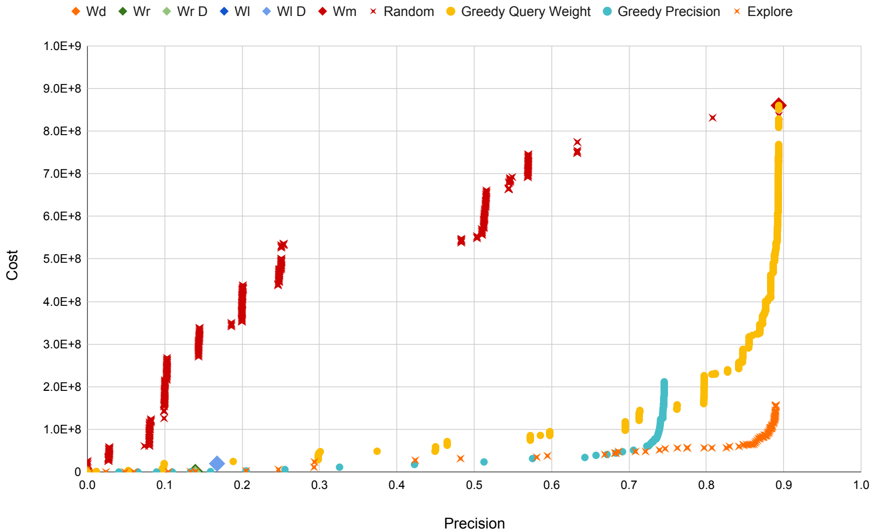


Figure 8.5: Configuration sets generated by the Greedy Query Weight Method (yellow), the Greedy Precision Method (teal), the Exploratory Method (orange), and the six reference configuration sets over query log $\mathcal{L}_B$.

high potential.

In Figure 8.5 we have included 94 configurations generated by the Exploratory Method, which selects how to extend the configuration set by evaluating ten random successors of each possible direct successor. It performs about equally good as the Greedy Precision Method up to a precision of 0.75, and after that, it continues to generate new Pareto optimal configurations until it reaches a precision of about 0.85, where it starts to bend upwards towards $\mathcal{W}_m$. We chose to terminate the method at this point, but if we had not done this, then the method would have continued to generate configurations until it eventually reached $\mathcal{W}_m$. These results demonstrate that exploring one more step ahead can solve the problems the Greedy Precision Method has, which is to choose successors that lead to good results in the long term. The Exploratory Method gives very good results, despite the fact that it only explores ten extensions out from each possible successor – all of the configurations it generates are Pareto optimal, or close to Pareto optimal, among the configurations generated in the experiment.

Finally, in Figure 8.6 we present the configurations generated by the Exploratory Method when it is enforced to only construct configurations that are cheaper than a maximum cost $M = 1.0 \times 10^7$.

This sequence more or less follows the path of the unrestricted Exploratory Method (orange crosses) until it is right below a cost of M, where it instead bends

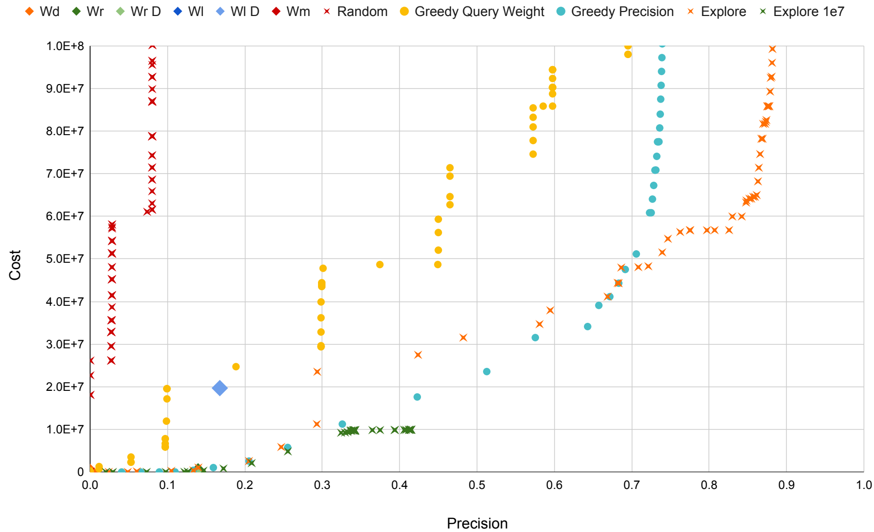


Figure 8.6: The sequence of configurations generated by the Exploratory Method over the query log $\mathcal{L}_B$, with a maximum cost of $M = 1.0 \times 10^7$ (green crosses).

slightly to form an almost straight horizontal line. It continues to improve until it reaches a configuration that cannot be extended anymore without becoming more expensive than M. The last configuration set in the generated sequence has a precision of 0.42, and a cost of 9928188, which is very close to $M = 1.0 \times 10^7$.

So far, we have only considered the cost and precision of the generated configurations, but now we are also going to take a look at the actual configurations that are generated. Let us consider the configuration set $\mathcal{W}^{53}$ generated by the Exploratory Method in its 53rd iteration, with a precision of 0.88 and a cost of 9.27×10^7. This is one of the best configurations generated by any of our methods, so it is interesting to see which configuration queries it actually contains. $\mathcal{W}^{53}$ only has three configuration queries, rooted in the three classes Person, City, and City USA. This is very few, considering the fact that there are 15 classes in total in the navigation graph, but it makes sense if we look back at the WD setup we are working with (see Chapter 7): 8 of the classes do not even have any outgoing data properties in the navigation graph, so there will not be any extension cases out from a variable of this class. In other words, a configuration query for any of them will be useless. Three more of the classes (Film, Award, and Continent) occur very infrequently in the query log, so having a configuration query for any of them would not increase the precision very much. The last class is Country, which has five outgoing data properties, and which is frequently used in the queries. This class should be a reasonable configuration root class, and in fact, if we look at the configuration generated 19 iterations

Class	Explanation
Person	Useful in configuration
City	Useful in configuration
City USA	Useful in configuration
Country	Useful in configuration
Film	Few queries in $\mathcal{L}_B$
Award	Few queries in $\mathcal{L}_B$
Continent	Few queries in $\mathcal{L}_B$
Gender	No data properties in $\mathcal{N}$
Profession	No data properties in $\mathcal{N}$
Eye Color	No data properties in $\mathcal{N}$
Hair Color	No data properties in $\mathcal{N}$
Filmography	No data properties in $\mathcal{N}$
Film Genre	No data properties in $\mathcal{N}$
Television Series	No data properties in $\mathcal{N}$
Capital	No data properties in $\mathcal{N}$

Table 8.4: Overview of the 15 classes and why most of our methods only construct configuration queries for four of them when calculating precision based on $\mathcal{L}_B$.

later by the Exploratory Method, then Country is also included. If we look at the last and 94th iteration $\mathcal{W}^{94}$ of the Exploratory Method, it also only contains four configuration queries, for the four same classes. In Table 8.4, we give an overview of the 15 classes in $\mathcal{N}$, and how useful their potential configuration queries are based on their roles in the navigation graph and the query log.

Let us inspect the configuration query with root class Person in $\mathcal{W}^{53}$. The root of this configuration query is connected to 14 object properties and 7 data properties, and 4 of the object properties have outgoing edges to 2, 2, 1, and 1 more variables, respectively. We have presented the whole configuration query as an indented tree below.

Configuration query generated for the class Person.

```
1  country of citizenship → Country
2      official name → String
3      part of → Continent
4  place of birth → City
5      population → Integer
6  child of → Person
7      occupation → Profession
8      has child → Person
9  has child → Person
10     child of → Person
11 occupation → Profession
12 award received → Award
13 sex or gender → Sex Or Gender
14 spouse → Person
15 number of children → Integer
16 height → Integer
17 date of birth → DateTime
18 place of birth → City
19 hair color → Hair Color
20 residence → City
21 residence → City USA
22 work location → City
23 work location → City USA
24 birth name → String
25 name native language → String
26 family name → String
27 given name → String
```

The configuration queries of City and City USA are actually identical, so we only present the configuration query of City below. It has two outgoing object properties and two outgoing data properties, and the Person it is the birthplace of is again connected to six more object properties and three more data properties.

Configuration query generated for the class City.

```
1  birthplace of → Person
2      country of citizenship → Country
3      place of birth → City
4      occupation → Profession
5      award received → Award
6      sex or gender → Sex Or Gender
7      spouse → Person
8      number of children → Integer
9      height → Integer
10     date of birth → DateTime
11 population → Integer
12 official name → String
13 coordinate location → Location
```

It is natural to question why the configuration queries of City and City USA are equal. We believe that this is because the two classes and their related

properties are equally weighted, or close to equally weighted in $\mathcal{L}_B$, and we think that this is a consequence of the query transformation process we presented in Section 7.3. If none or very few of the large queries in the original query log has assigned types to the variables, then any query with a variable that could match a city would also match an American city, since these two classes are connected to the exact same properties in the navigation graph. The transformation process described in Chapter 7 will then turn each such query into two versions where the variable is typed to City in the first one, and City USA in the other one. These two versions will be given equal weight, and hence give both City and City USA equal weight in the resulting query log.

The class City has 11 outgoing properties in total in the navigation graph, and only four of them have been included in the configuration query rooted in City in $\mathcal{W}^{53}$. Meanwhile, nine different properties are added to the person it is the birthplace of. This confirms the importance of the Person class in the WD setup. All nine of these properties are also connected directly to the root of the Person configuration query, which makes sense: paths that are useful in one configuration query are likely to be so also in another configuration query.

Both of these configuration queries show that the Exploratory Method is able to construct non-trivial configuration queries where some second-level properties have been prioritized before some of the local properties, i.e., it is able to construct uneven configuration queries that are good solutions for the given setup.

8.3.2 Evaluation based on $\mathcal{L}_A$

In Table 8.5, we present the precision and cost of the six reference configurations when they are evaluated using query log $\mathcal{L}_A$. These results show how easy it is to find precise configurations when the query log contains mostly small queries like $\mathcal{L}_A$ does. All of the reference configurations achieve close to perfect precisions, except for $\mathcal{W}_d$ of course. Even $\mathcal{W}_r$ and $\mathcal{W}_r^d$, which both only had a precision of 0.14 with respect to query log $\mathcal{L}_B$, now provide a precision of over 0.98 with query log $\mathcal{L}_A$. It is also worth pointing out that all of these configurations, except for $\mathcal{W}_m$, have the same cost as they had when we evaluated over $\mathcal{L}_B$. This makes sense since both the structure of these configurations and the dataset their cost is based on, are unchanged. $\mathcal{W}_m$, on the other hand, depends directly on the query log that is used, and both its cost and precision have increased compared to $\mathcal{W}_m$ based on $\mathcal{L}_B$. In fact, the cost of $\mathcal{W}_m$ is now so high that it exceeds the cost of $\mathcal{W}_l$.

In addition to evaluating the six reference configurations, we also generated configurations with the Random Method, the Greedy Query Weight Method, the Greedy Precision Method, and the Exploratory Method, all without any cost threshold, just like we did with query log $\mathcal{L}_B$. The results are all presented in Figure 8.7 (note the logarithmic scale on the y-axis). Since most of the precisions are above 0.95, we present an additional chart with only this range in Figure 8.8.

All search methods, except for the Random Method, are able to find very precise configurations after just a few iterations. Again, this is not very surprising,

Set	Precision	Cost
$\mathcal{W}_d$	0.000	0
$\mathcal{W}_r^d$	0.985	1.6×10^5
$\mathcal{W}_r$	0.985	1.6×10^5
$\mathcal{W}_l^d$	0.991	2.0×10^7
$\mathcal{W}_l$	1.000	8.4×10^{10}
$\mathcal{W}_m$	1.000	1.6×10^{14}

Table 8.5: The precision and cost of the six reference configuration sets $\mathcal{W}_d$, $\mathcal{W}_r$, $\mathcal{W}_r^d$, $\mathcal{W}_l$, $\mathcal{W}_l^d$, and $\mathcal{W}_m$ with respect to query log $\mathcal{L}_A$.

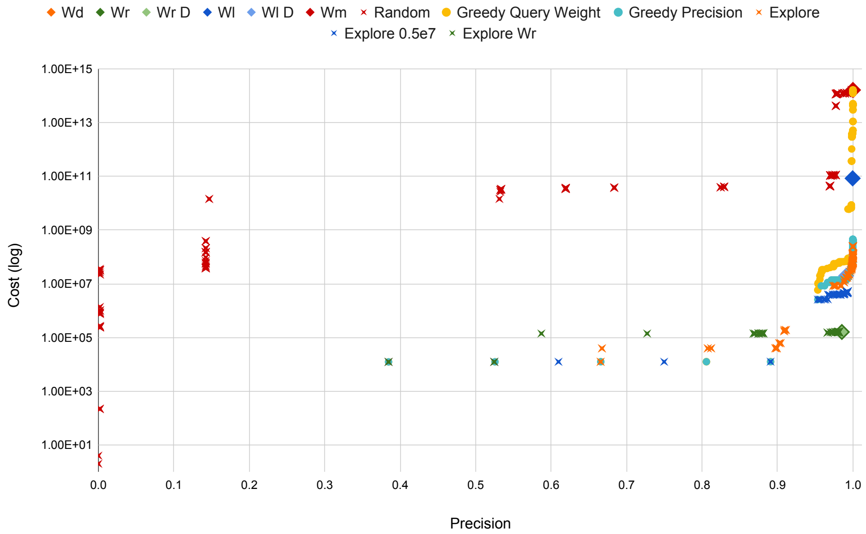


Figure 8.7: Precision and cost of every generated configuration when evaluated with respect to query log $\mathcal{L}_A$.

Figure 8.8: Precision and cost of every generated configuration with precision above 0.95 when evaluated with respect to query log $\mathcal{L}_A$.

since most of the queries in $\mathcal{L}_A$ are small: this makes it easier to find good configurations in each iteration, and it overall requires smaller configurations, which can be built with just a few extensions. Large parts of the results presented in Figure 8.7 are qualitatively the same as the results we got with $\mathcal{L}_B$: the Random Method is significantly worse than the Greedy Query Weight Method, which is again outperformed by the Greedy Precision Method and the Exploratory Method.

If we only consider the results of the Greedy Precision Method and the Exploratory Method, it looks like a configuration with cost above 1.0×10^7 is needed in order to achieve precisions above 0.995. We wanted to see if this was the case, so we tried to run the Exploratory Method with a smaller maximum cost of $M = 0.5 \times 10^7$ (blue crosses), to see if it still was able to produce a configuration with precision higher than 0.995. This is not the case, the Exploratory Method stops improving at a precision of 0.992.

We also did an exploratory run with a maximum cost of $M = \mathsf{cost}(\mathcal{W}_r, \mathcal{D}) = 1.6 \times 10^5$ (green crosses), in an attempt to beat the precision that $\mathcal{W}_r$ and $\mathcal{W}_r^d$ are able to provide. This run ends at a precision of 0.981, which is lower than the precision of $\mathcal{W}_r$. This probably happens because the Exploratory Method makes some suboptimal choices in one of the first iterations. For example, already in iteration number eight, it combines two properties into a V-shaped configuration query, which gives it a high cost compared to the alternative, which is two separate configuration queries with one property each. Since almost all queries

in $\mathcal{L}_A$ have size 2, V-shaped configuration queries are often not ideal. Even though the Exploratory Method was not able to find any better alternative to $\mathcal{W}_r$ and $\mathcal{W}_r^d$, it is still likely that there exists such a configuration set. It is, for example, possible that a given data property is totally non-existent in the query log, while it is still represented in the dataset, and if this is the case, then the configuration query corresponding to this property in $\mathcal{W}_r$ is totally useless and can be removed from $\mathcal{W}_r$ or $\mathcal{W}_r^d$ to achieve a lower cost, but the same precision.

To illustrate how easy it is to make a configuration set with high precision, let us consider $\mathcal{W}^5$, the fifth configuration set generated by the Greedy Precision Method, which has a precision of 0.981 and a cost of 1.26×10^4. This only contains 5 configuration queries with only one property each, and all of them are presented below.

Configuration query generated for the class City.

```
1  Person →  name native language → String
2  Continent → coordinate location → Location
3  Country → coordinate location → Location
4  City USA → coordinate location → Location
5  City → coordinate location → Location
```

According to this, there must be many queries of high weight asking for the location of cities, countries, and continents, in addition to the name of persons.

In this chapter, we have presented a set of different search methods that all attempt to find useful configuration sets given a particular setup of a dataset, a navigation graph, and a query log. All of these methods need to evaluate many different configurations during the search process, so they all rely on the efficient cost and precision estimates that we presented in the first section of this chapter.

Our evaluation of the search methods based on the WD benchmark shows that the methods we present, and in particular the Exploratory Method, are able to find useful configuration sets for settings with many large and complex queries (query log $\mathcal{L}_B$).

Chapter 9

Conclusion and Future Work

In this thesis project, we have been working on the problem of detecting dead-end query extensions in ontology-based VQSs that support tree-shaped, and arbitrarily large queries with typed variables and filters. For large datasets and queries, the process of calculating dead-ends perfectly in this setting is too slow, since it requires the system to execute complex queries over the database after each change to the partial query. Other systems with a more limited query expressivity solve this efficiency problem by constructing and using indices specialized for dead-end detection. This kind of solution cannot be used in the more complex setting, because that would require an infinitely large index. But, it is possible to use a variant of the solution if we allow some inaccuracy in the detected dead-ends, and this is the idea that our configurable dead-end detection system is based on.

The configuration used to set up our system determines both the structure of the index and how many dead-end extensions it detects, and based on this we can calculate both the precision and cost of the configuration. Obviously, we want a configuration with high precision and low cost, but finding such a configuration is non-trivial, because the search space of possible configurations is very large, and because it takes relatively long time to evaluate a single configuration. To solve this problem, we present a set of different configuration generation methods that use efficient cost and precision estimates and search heuristics to produce useful configurations. To evaluate these configuration generation methods, we constructed the Wikidata benchmark, which contains a large dataset and a corresponding navigation graph, in addition to a large collection of typed, tree-shaped queries that reflects human information needs. Our evaluation shows that the configuration generation methods are able to produce complex configurations with both low cost and high precision.

9.1 Conclusion

Since it is impossible to construct an index that can be used to efficiently compute dead-ends for queries of arbitrary size, the best alternative is to approximate them. The main idea of the approach presented in this thesis, which is to ignore the less important parts of the partial query when computing dead-ends, works because it limits the number of joins that needs to be pre-computed, which means that efficient results can be computed with the use of a finite index. The proposed way of defining which parts of the query to ignore with configuration queries is flexible enough to define any desired instance of the system since the tree shape it uses is essentially the set of paths to not ignore. Non-simple configuration queries are not supported in the presented framework, but we

believe it should be possible to support them without too much extra work (see Section 9.2). But it is worth questioning how useful such configurations will be since they will only result in higher precision when properties are repeated twice in the partial query.

The configuration query structure makes the pruning process intuitive, and it allows us to define index tables based on the answers to subqueries of the configuration query and the subfunction relationship (see Section 6.2). In addition, it can be extended easily to configuration sets, to support multiple different root classes (see Section 6.4). In this thesis, we prove theoretically that it is possible to construct an efficient index based on the configuration query and the dataset, and that this index returns the same results as the dataset to pruned queries. The recall of the proposed solution will always be perfect since all pruned versions of the partial query are less restrictive than the partial query itself. On the other hand, the precision depends on the dataset and the configuration, in addition to the particular query the user is making.

Chapter 7 and Chapter 8 are devoted to the problem of finding out what the best possible configuration is, and how useful this configuration will be. The result of the configuration generation process depends on which cost and precision measures that are used, and the measures we suggest in Chapter 6 are reasonable: the cost is based on how large the different index tables are, and the precision is based on how many of the suggested productive values that are truly productive. Comparing these measures to other cost and precision measures would be interesting, but it is hard to come up with reasonable measures that are fundamentally different from our proposed measures if the goal is to generate configurations that are useful in practice. The proposed way of using a query log to generate extension cases assumes that the queries in the query log are representative for future queries. This may not always be the case, but if the user base and time period are the same or almost the same, then it is safe to assume that there will be similarities. At least it should give better results than a method that does not take into account that some queries are more likely to be posed than others.

The cost and precision estimates presented in Section 8.1 are based on techniques that are used for cardinality estimation in the database community. But despite this, there is no guarantee that the estimates are sufficiently good, and a proper evaluation of the estimates we have used is future work. And, even though we have been evaluating many different search methods, they are all still based on the same approach, which is to start with an empty configuration and extend it based on heuristics. This probably misses some configurations of high quality, but how problematic this actually is, must be investigated in future work. But, it is important to remember that the purpose of the estimates and the search techniques is to sacrifice accuracy on configuration generation in order to make the process efficient. In other words, if the generated configurations are not good enough, it is always possible to use more complex cost and precision estimates and a more extensive search to find a better configuration, but then the search will also spend more time on the configuration generation process.

The results produced by the configuration generation methods in Section 8.3

are all based on only one setup: the WD benchmark, and it is important to remember that other setups could result in a completely different kind of result. In fact, by evaluating over both query log $\mathcal{L}_A$ and query log $\mathcal{L}_B$, we discovered how much the results can change by just replacing the query log. While the presented configuration generation methods are likely to be robust enough to find a configuration that is close to optimal, this does not mean that the resulting system will actually be useful. For example, the system will work well in cases where the users repeat the same classes and properties frequently, and where the index of these classes and properties does not explode when they are joined together. In contrast, if the queries are either not representative, or if they cover classes and properties that result in a large index, then it is impossible to find good configurations, at least if a small maximum cost threshold is set. In the concrete evaluation we did based on the WD benchmark, the results indicate that the exploratory method is able to produce the best configurations. This is not a surprise, due to the fact that it is more complex and time-consuming than the other methods, and it highlights the fact that a proper evaluation of all the search methods must be able to take both the resulting configuration and the efficiency into account. But, it is worth highlighting that this method was able to produce configurations that lead to a system that is objectively better than a system based on $\mathcal{W}_l$ and $\mathcal{W}_l^d$, which are the configurations that represent standard faceted search.

The popularity of dead-end detection in faceted search systems shows how useful this feature is during query construction. Current dead-end detection solutions used by systems that support complex queries suffer from inefficiency, and our solution is unique since it ensures efficiency by sacrificing precision of the detected dead-ends. Approximations to dead-end detection have, to our knowledge, not been explored much, so this thesis is a contribution to that area of research. The overall conclusion of our work is that the approximation approach we use to tackle the dead-end detection problem is practically possible to use, and that it will perform better than current solutions unless configuration generation fails due to hard setups.

9.2 Future work

In this final section, we highlight and discuss possible future projects related to our work. Most of them have already been discussed to some degree earlier in the thesis.

Changes to the Dataset Throughout the whole project, we have assumed that the dataset does not change very often and that when it does, all indices have to be rebuilt from scratch. But building an index from scratch may be very time consuming, so if the change in the dataset is small, it may be more efficient to instead update the index by this change. If we are able to develop methods that allow us to update indices efficiently, our system will become useful in situations where a full index re-creation is too slow.

Changes to the Navigation Graph Changes to the navigation graph are more problematic than changes to the dataset because they alter the foundation that both the configuration queries and the query log are based on. For example, if a new edge is added to the navigation graph, it must be decided if it should be added to any of the configuration queries. If so, then their corresponding indices must be re-constructed or updated. In order to determine if the new edge should be used by the system, one could re-run some of the configuration generation methods. But, since the query log does not contain any queries where the new edge has been used, this edge will not contribute at all to higher precision. Hence, none of the configuration generation methods will even consider to include it. This means that either a human has to decide whether the edge should be added, or that a new configuration generation method has to be developed, where new edges will be considered, even when they are not present in the query log.

Alternative Index Structures In Section 6.2.2, we suggested two ways of storing the extension index: either as tables in a relational database or as an instance of a search engine that supports faceted search. Both of these approaches are scalable, in the sense that the task of querying over them can be distributed efficiently over multiple machines. However, they are not especially memory efficient: each row in the index table corresponds to one way of assigning entities to the variables of the index table's corresponding configuration query, where variables that are not assigned to any entities are just filled with the null symbol ω. If the index was instead represented as a tree-structure, all these cells would not be needed. In the future, we want to consider other types of data structures, databases, or indexing systems, which take less space than our table-based approach, while still being scalable. We believe that some NoSQL databases, and in particular document-oriented databases, should be very suitable for the tree-shaped structures we are working with.

Bucketing We have already mentioned bucketing as a technique that can be used to both reduce the cost of the extension indices and minimize information overload in the user interface. But deciding how to use buckets, and when to use them is not completely clear and requires good insight about the navigation graph, the dataset, and the queries. There already exist techniques that generate buckets in ways that optimize for minimal memory footprint [21], and these techniques can probably be used with our system too, possibly with slight modifications.

Incorporate Subclass Relationships The navigation graph our system currently use does not include subclass relationships, even though such relationships are a central part of most ontologies. For example, if *Man*, *Woman*, and *Person* are three of the classes in the navigation graph, there is no way to state that both *Man* and *Woman* are subclasses of *Person*. It is not hard to extend the navigation graph with subclass relationships from the ontology, but if this is done, then we have to reconsider some parts of our system. In particular, we

would like to find out if it is possible to use the indices of *Man* and *Woman* to calculate values when the partial query is rooted in *Person*, or if a third separate index for *Person* is always needed. And, if classes can use the indices of their subclasses, then we would also like to find out how this affects the configuration generation methods we presented in Chapter 8.

Remove Redundancy over Multiple Indices We have already described different techniques that reduce the size of one individual index, but none of these techniques prevent redundancy over multiple indices. For example, if we have two indices over the class *Person*, and both of them include data about the person's name, in addition to other properties and classes, then there will be a significant amount of overlap between the two indices. But redundancy can also exist between indices over different classes. For example, if there is one index of *Person* that includes data about the countries any person has visited, and another index of *Country* that includes data about all persons that have been to any of the countries, then every person and each of their visited countries will be a part of both indices. In the future, we would like to see if it is possible to remove redundancy over multiple indices while still maintaining the scalability and efficiency we need.

Extend the Framework to Non-Simple Configuration Queries The requirement that configuration queries have to be simple, is a limitation of our system, which becomes very clear with setups where non-simple queries are frequently constructed. We believe that it is possible to extend our system to support non-simple configuration queries, which will then define indices that can be used to detect dead-ends perfectly for non-simple queries covered by this configuration query.

Extend Query Expressivity The typed, tree-shaped queries that our system supports are able to cover many information needs, but not all, and in the future, we would like to extend our framework to also cover more expressive queries. Based on the queries from the WD query log, the most popular missing features are:

- optionals
- unions
- typeless variables
- multi-typed variables
- non-tree-shaped queries

Each of these features must be considered carefully before they can be introduced, because they may alter some of the rules that we base our extension index on. For example, if queries without types are allowed, then we cannot even use the

navigation graph anymore to decide when a query is legal or not since there is no type corresponding to each of the variables.

Improve Cardinality Estimation The cardinality estimation techniques we presented in Section 8.1 are based on simple assumptions and statistics about the data, and the quality of these estimates can be improved by using better statistics or more advanced estimation techniques. In particular, it would be interesting to see if it is possible to obtain a better estimate of the set intersection in Equation 8.3, based on the tree structure of the pruned queries that generate these sets. Proper analysis and evaluation of these cardinality estimates are also needed, to indicate how good they really are.

Improve Search Methods The configuration generation search methods we presented in Section 8.2 are relatively simple, and we believe that there exist more sophisticated search methods that would lead to either better configurations, faster generation, or both. One possible improvement is to incorporate the cost and maximum cost M when deciding which direction the search should go. Another idea is to let the search methods remember how promising certain properties are, and reuse this knowledge to make the search more efficient in future steps. For example, if adding the height of a person leads to no precision increase initially, then it will likely not be a good choice later either. A third approach is to start the search from another configuration than the empty one, and possibly allow the search to remove variables from a configuration set.

Use Query Log Statistics in Evaluation The way we calculate the precision of a given configuration query with respect to a query log is relatively slow, because we have to consider every single query in the whole log, and all possible ways to make it into a rooted query (see Section 5.4.2). The efficiency of the evaluation can be improved by making a summary of the query log, and then estimate the precision based on this summary and not the whole query log. This may allow the system to explore a larger part of the search space for configurations, to find a better result. But, at the same time, this would introduce yet another source for estimation errors, so the quality of the resulting estimates would again have to be evaluated carefully.

Incorporate Ontology Constraints If the ontology contains constraints that limit which data values a particular property can take, they are today just ignored by the VQS, because the navigation graph we use does not support such constraints. In the future, we would like to use this information to improve the suggestion of extension values.

User Studies Throughout the whole project, we have been using precision and recall of the productive data values as the main evaluation measure. But high precision and recall do not necessarily correspond to high usability, so a natural next step would be to perform a user study of the system. Such a user study

should compare multiple copies of the system, where different configurations based on different maximum cost thresholds are used. It should also be compared to the system based on $\mathcal{W}_r$ and $\mathcal{W}_l$, and a possibly very slow system where perfect dead-end detection is done without an index.

Bibliography

[1] Arenas, Marcelo et al. "Faceted Search over Ontology-Enhanced RDF Data". In: *CIKM 2014 – Proceedings of the 2014 ACM International Conference on Information and Knowledge Management* (Nov. 2014), pp. 939–948.

[2] Arenas, Marcelo et al. "Faceted search over RDF-based knowledge graphs". In: *Journal of Web Semantics* vol. 37–38 (2016), pp. 55–74.

[3] Arenas, Marcelo et al. "SemFacet: Semantic Faceted Search over Yago". In: *Proceedings of the 23rd International Conference on World Wide Web*. New York, NY, USA: Association for Computing Machinery, 2014, pp. 123–126.

[4] Arenas, Marcelo et al. "Towards Semantic Faceted Search". In: *Proceedings of the 23rd International Conference on World Wide Web*. New York, NY, USA: Association for Computing Machinery, 2014, pp. 219–220.

[5] Berners-Lee, Tim, Hendler, James, and Lassila, Ora. "The Semantic Web". In: *Scientific American* vol. 284, no. 5 (May 2001), pp. 34–43.

[6] Berners-Lee, Tim et al. "Tabulator: Exploring and Analyzing linked data on the Semantic Web". In: *Proceedings of the 3rd International Semantic Web User Interaction*. 2006.

[7] Brooke, John. "SUS: a quick and dirty usability scale". In: CRC press, 1996, p. 189.

[8] Brunetti, Josep Maria, Auer, Sören, and García, Roberto. "The Linked Data Visualization Model". In: *Proceedings of ISWC 2012, Posters & Demonstrations Track*. Vol. 914. CEUR, 2012, pp. 5–8.

[9] Brunetti, Josep, García, Roberto, and Auer, Sören. "From Overview to Facets and Pivoting for Interactive Exploration of Semantic Web Data". In: *International Journal on Semantic Web and Information Systems* vol. 9 (Apr. 2013), pp. 1–20.

[10] Calvanese, Diego et al. "Ontop: Answering SPARQL queries over relational databases". In: *Semantic Web* vol. 8 (Feb. 2016).

[11] Catarci, Tiziana et al. "Visual Query Systems for Databases: A Survey". In: *Journal of Visual Languages & Computing* vol. 8, no. 2 (1997), pp. 215–260.

[12] Chaudhuri, Surajit. "An overview of query optimization in relational systems". In: *Proceedings of the 17th ACM SIGACT-SIGMOD-SIGART symposium on Principles of Database Systems*. 1998, pp. 34–43.

[13] Christoffersen, Tom Fredrik. "SPARQL Extension Ranking – Collaborative filtering for OptiqueVQS-queries". Department of Informatics, University of Oslo, Norway, May 2020.

[14] Coulom, Rémi. "Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search". In: *Computers and Games*. Ed. by Herik, H. Jaap van den, Ciancarini, Paolo, and Donkers, H. H. L. M. (Jeroen). Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 72–83.

[15] Dudás, Marek et al. "Ontology visualization methods and tools: a survey of the state of the art". In: *Knowledge Eng. Review* vol. 33 (2018), e10.

[16] Giese, M. et al. "Optique: Zooming in on Big Data". In: *Computer* vol. 48, no. 3 (Mar. 2015), pp. 60–67.

[17] Guarino, Nicola, Oberle, Daniel, and Staab, Steffen. "What is an ontology?" In: *Handbook on ontologies*. Springer, 2009, pp. 1–17.

[18] Hitzler, Pascal, Krtzsch, Markus, and Rudolph, Sebastian. *Foundations of Semantic Web Technologies*. 1st. Chapman & Hall/CRC, 2009.

[19] Hoy, Matthew. "Alexa, Siri, Cortana, and More: An Introduction to Voice Assistants". In: *Medical Reference Services Quarterly* vol. 37 (Jan. 2018), pp. 81–88.

[20] Ioannidis, Yannis E. and Poosala, Viswanath. "Balancing Histogram Optimality and Practicality for Query Result Size Estimation". In: *SIGMOD Rec.* Vol. 24, no. 2 (May 1995), pp. 233–244.

[21] Ioannidis, Yannis E. and Poosala, Viswanath. "Histogram-Based Solutions to Diverse Database Estimation Problems". In: *IEEE Data Eng. Bull.* Vol. 18 (1995), pp. 10–18.

[22] Kharlamov, Evgeny et al. "How Semantic Technologies Can Enhance Data Access at Siemens Energy". In: *The Semantic Web – ISWC 2014*. Ed. by Mika, Peter et al. Cham: Springer International Publishing, 2014, pp. 601–619.

[23] Kharlamov, Evgeny et al. "Ontology Based Access to Exploration Data at Statoil". In: *The Semantic Web – ISWC 2015*. Ed. by Arenas, Marcelo et al. Cham: Springer International Publishing, 2015, pp. 93–112.

[24] Klungre, Vidar N. *A Faceted Search Index for Graph Queries*. Tech. rep. 469. Department of Informatics, University of Oslo, 2017.

[25] Klungre, Vidar N. and Giese, Martin. "A Faceted Search Index for OptiqueVQS". In: *Proceedings of the ISWC 2017 Posters & Demonstrations and Industry Tracks*. Ed. by Nikitina, Nadeschda et al. Vol. 1963. CEUR Workshop Proceedings. CEUR-WS.org, 2017.

[26] Klungre, Vidar N. and Giese, Martin. "Approximating Faceted Search for Graph Queries". In: *12th Intl. Workshop on Scalable Semantic Web Systems (SWSS)*. Vol. 2179. CEUR-WS, 2018, pp. 61–76.

[27] Klungre, Vidar and Giese, Martin. "Evaluating a Faceted Search Index for Graph Data". In: *Proc. On the Move to Meaningful Internet Systems (OTM 2018)*. Vol. 11230. LNCS. 2018, pp. 573–583.

[28] Klungre, Vidar et al. "On Enhancing Visual Query Building over KGs Using Query Logs". In: *Semantic Technology (JIST 2018)*. Ed. by Ichise, Ryutaro et al. 2018, pp. 77–85.

[29] Klungre, Vidar et al. "Query Extension Suggestions for Visual Query Systems Through Ontology Projection and Indexing". In: *New Generation Computing* (Aug. 2019).

[30] Kogalovsky, Mikhail R. "Ontology-based data access systems". In: *Programming and Computer Software* vol. 38 (2012), pp. 167–182.

[31] Kules, William et al. "From Keyword Search to Exploration: How Result Visualization Aids Discovery on the Web". 2008.

[32] Lehmann, Jens et al. "DBpedia – A Large-scale, Multilingual Knowledge Base Extracted from Wikipedia". In: *Semantic Web Journal* vol. 6 (Jan. 2014).

[33] Leis, Viktor et al. "How good are query optimizers, really?" In: *Proceedings of the VLDB Endowment* vol. 9, no. 3 (2015), pp. 204–215.

[34] Lopez, Vanessa and Motta, Enrico. "PowerAqua: an ontology question answering system for the Semantic Web". In: (2005).

[35] Malyshev, Stanislav et al. "Getting the Most out of Wikidata: Semantic Technology Usage in Wikipedia's Knowledge Graph". In: *Proceedings of the 17th International Semantic Web Conference (ISWC'18)*. Ed. by Vrandečić, Denny et al. Vol. 11137. LNCS. Springer, 2018, pp. 376–394.

[36] Miller, Robert B. "Response Time in Man-Computer Conversational Transactions". In: *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*. AFIPS '68 (Fall, part I). San Francisco, California: Association for Computing Machinery, 1968, pp. 267–277.

[37] Munir, Kamran, Odeh, Mohammed, and Mcclatchey, Richard. "Ontology-Driven Relational Query Formulation Using the Semantic and Assertional Capabilities of OWL-DL". In: *Knowledge-Based Systems* vol. 35 (Nov. 2012).

[38] Muralikrishna, M. and DeWitt, David J. "Equi-Depth Histograms For Estimating Selectivity Factors For Multi-Dimensional Queries". In: *SIGMOD 1988*. 1988.

[39] Musen, Mark A. "The Protégé Project: A Look Back and a Look Forward". In: *AI Matters* vol. 1, no. 4 (June 2015), pp. 4–12.

[40] Nenov, Yavor et al. "RDFox: A Highly-Scalable RDF Store". In: *The Semantic Web – ISWC 2015*. Ed. by Arenas, Marcelo et al. Cham: Springer International Publishing, 2015, pp. 3–20.

[41] Neumann, T. and Moerkotte, G. "Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins". In: *2011 IEEE 27th International Conference on Data Engineering*. 2011, pp. 984–994.

[42] Poosala, Viswanath et al. “Improved Histograms for Selectivity Estimation of Range Predicates”. In: *SIGMOD Rec.* Vol. 25, no. 2 (June 1996), pp. 294–305.

[43] Regalia, Blake, Janowicz, Krzysztof, and Mai, Gengchen. “Phuzzy.link: A SPARQL-powered Client-Sided Extensible Semantic Web Browser”. In: *3rd Intl. Workshop on Visualization and Interaction for Ontologies and Linked Data (VOILA2017).* Vol. 1947. CEUR-WS, Nov. 2017.

[44] Rodriguez-Muro, Mariano and Calvanese, Diego. “Quest, a System for Ontology Based Data Access”. In: *CEUR Workshop Proceedings* vol. 849 (Jan. 2012).

[45] Schmidt, Michael, Meier, Michael, and Lausen, Georg. “Foundations of SPARQL Query Optimization”. In: *Proceedings of the 13th International Conference on Database Theory.* ICDT ’10. Lausanne, Switzerland: Association for Computing Machinery, 2010, pp. 4–33.

[46] Shearer, Rob, Motik, Boris, and Horrocks, Ian. “HermiT: A Highly-Efficient OWL Reasoner.” In: *Owled.* Vol. 432. 2008, p. 91.

[47] Silver, David et al. “Mastering the game of Go with deep neural networks and tree search”. In: *Nature* vol. 529, no. 7587 (2016), pp. 484–489.

[48] Sirin, Evren et al. “Pellet: A practical owl-dl reasoner”. In: *Journal of Web Semantics* vol. 5, no. 2 (2007), pp. 51–53.

[49] Soylu, Ahmet and Giese, Martin. “Qualifying Ontology-Based Visual Query Formulation”. In: *Flexible Query Answering Systems 2015.* Ed. by Andreasen, Troels et al. Cham: Springer International Publishing, 2016, pp. 243–255.

[50] Soylu, Ahmet and Kharlamov, Evgeny. “Navigating OWL 2 Ontologies Through Graph Projection”. In: *Metadata and Semantic Research.* Ed. by Garoufallou, Emmanouel et al. Cham: Springer International Publishing, 2019, pp. 113–119.

[51] Soylu, Ahmet et al. “Experiencing OptiqueVQS: A Multi-paradigm and Ontology-based Visual Query System for End Users”. In: *Universal Access in the Information Society* vol. 15 (Mar. 2016), pp. 129–152.

[52] Soylu, Ahmet et al. “Ontology-based End-user Visual Query Formulation: Why, what, who, how, and which?” In: *Universal Access in the Information Society* (Apr. 2016).

[53] Soylu, Ahmet et al. “Ontology-based Visual Querying with OptiqueVQS: Statoil and Siemens Cases”. In: *Norwegian Big Data Symposium (NOBIDS) 2016, Trondheim, Norway.* Ed. by Gulla, Jon Atle et al. CEUR Workshop Proceedings. CEUR, 2016.

[54] Soylu, Ahmet et al. “OptiqueVQS: a Visual Query System over Ontologies for Industry”. In: *Semantic Web* (Aug. 2017).

[55] Soylu, Ahmet et al. “Towards Exploiting Query History for Adaptive Ontology-Based Visual Query Formulation”. In: *Metadata and Semantics Research*. Ed. by Closs, Sissi et al. Cham: Springer International Publishing, 2014, pp. 107–119.

[56] Stefanoni, Giorgio, Motik, Boris, and Kostylev, Egor V. “Estimating the Cardinality of Conjunctive Queries over RDF Data Using Graph Summarisation”. In: *Proceedings of the 2018 World Wide Web Conference*. WWW ’18. Lyon, France: International World Wide Web Conferences Steering Committee, 2018, pp. 1043–1052.

[57] Störrle, Harald. “VMQL: A visual language for ad-hoc model querying”. In: *J. Vis. Lang. Comput.* Vol. 22 (Feb. 2011), pp. 3–29.

[58] Sutherland, Dougal J. and Carlson, Ryan. *Evaluating Multidimensional Histograms in ProstgreSQL*. 2010.

[59] Tunkelang, Daniel. *Faceted Search*. Synthesis Lectures on Information Concepts, Retrieval, and Services. Morgan & Claypool Publishers, 2009.

[60] Vega-Gorgojo, Guillermo, Giese, Martin, and Slaughter, Laura. “Exploring semantic datasets with RDF Surveyor”. In: *Proceedings of the ISWC 2017 Posters & Demonstrations and Industry Tracks*. Oct. 2017.

[61] Vega-Gorgojo, Guillermo et al. “Linked Data Exploration with RDF Surveyor”. In: *IEEE Access* vol. 7 (Nov. 2019), pp. 1–1.

[62] Vega-Gorgojo, Guillermo et al. “PepeSearch: Semantic Data for the Masses”. In: *PLOS ONE* vol. 11 (Mar. 2016), e0151573.

[63] Xu, Kun et al. “Hybrid Question Answering over Knowledge Base and Free Text”. In: *COLING 2016, 26th International Conference on Computational Linguistics, Proceedings of the Conference: Technical Papers, December 11-16, 2016, Osaka, Japan*. Ed. by Calzolari, Nicoletta, Matsumoto, Yuji, and Prasad, Rashmi. ACL, 2016, pp. 2397–2407.

[64] Zheleznyakov, Dmitriy et al. “KeywDB: A System for Keyword-Driven Ontology-to-RDB Mapping Construction”. In: *Proceedings of the ISWC 2016 Posters & Demonstrations Track co-located with 15th International Semantic Web Conference (ISWC 2016), Kobe, Japan, October 19, 2016.* 2016.

Index

$\mathcal{Q}_e$, 71
S_a, 87
S_d, 73
S_e, 73
S_l, 79
S_o, 73
S_r, 78
$\mathcal{W}_d$, 119
$\mathcal{W}_l$, 121
$\mathcal{W}_l^d$, 122
$\mathcal{W}_m$, 121
$\mathcal{W}_r$, 119
$\mathcal{W}_r^d$, 122
X_o, 71
ans_I, 100
ans_O, 105
ans_E, 108
ans_{PI}, 100
ans_S, 101
bf, 147
C_e, 143
C_{es}, 143
$\widehat{\mathsf{ans}}$, 147, 148
$\widehat{\mathsf{ans}}_O$, 150, 151
$\widehat{\mathsf{ans}}_E$, 151, 152
$\widehat{\mathsf{ans}}_P$, 149
C_{et}, 143
X_e, 65
p_e, 65
t_e, 65
v_e, 65
χ, 100, 107
H_e, 145
ω, 100
$\mathsf{prec}_{\mathrm{C}}$, 74
$\mathsf{prec}_{\mathrm{L}}$, 77
$\mathsf{prec}_{\mathrm{Q}}$, 77
$\mathsf{prec}_{\mathrm{U}}$, 77
prune, 87
setRoot, 65
J, 69

actions, 63
ad-hoc query, 23
adaptivity, 26
advanced value functions, 78
Amazon, 27
ansopt, 105
aspects of data access systems, 21

basic counts, 142
basic graph pattern, 13
branching factor, 146, 147
bucketing, 111, 146, 174

cardinality estimation, 141, 176
casual user, 23
children, 50
class, 50
class count, 142
component set, 51
configuration generation, 139
configuration generation problem, 122, 123
configuration query, 84
configuration set, 117
configuration-based value function, 87
contributions, 3
correct index, 100
cost, 110, 118
cost estimation, 139
coverage, 84

data access system, 21
data browser, 22
data edge, 47
data graph, 53
data property extension, 65

data quality, 25
data retrieval, 25
data values, 50
data variable, 50
data velocity, 25
data volume, 25
database expert, 22
dataset, 53
datatypes, 50
dead-end detection, 21, 37
Dead-end extension, 1
delete, 64
direction-ignorant path, 49
direction-ignorant walk, 49
distribution, 144
domain expert, 22
domain ontology, 13
domain-based value function, 73

eBay, 27
edge, 46
edge count, 143
edge Counts, 142
edge inverse, 47
edge target distribution, 144
edge target distribution function, 145
efficiency, 26, 81
Elastic search, 28
empty value function, 73
entity, 51
errors, 75
Exareme, 31
existence symbol, 100, 107
existential answers, 108
existential object variables, 107
exploration system, 22
Exploratory Method, 156
expressiveness, 23
expressivity, 23
extend, 65
extended query, 71
extension, 63
extension filter set, 67
extension framework, 83
extension function, 67
extension index, 99
extension pair, 69
extension property, 67
extension specification, 67
extension type, 67
extension variable, 65

facet, 27
faceted search, 27
filter, 15, 148
filter-ignorant renaming, 84
focus, 64
focus class, 65
focus variable, 33, 64, 65
function composition, 17
function restriction, 17

Greedy Precision Method, 156
Greedy Query Weight Method, 155

Hermit, 13
histogram, 145
homomorphism, 17, 48

independence assumption, 141
index, 99, 100, 174
index construction, 101
index cost, 110, 118
index efficiency, 109
index table, 110
information retrieval, 25, 35
instances, 50
interactivity, 23
inverse, 47
isomorphism, 18, 48, 49
IT expert, 22

labeled, directed graph, 17
lay user, 23
legal extension, 68
legal extension pair, 69
legal query, 57
linked data browser, 29
literal, 10
local value function, 79

Monte Carlo tree search, 157

navigation graph, 33, 52
NPD Factpages, 93
null symbol, 100
nullary intersection, 88

OBDA, 31
object edge, 47
object property extension, 65
object variable, 50
ontology, 12
ontology constraints, 176
ontology projection, 33
ontology-based data access, 31
Ontop, 31
optionals, 15
Optique Project, 31
OptiqueVQS, 31
OWL, 9, 12

parent, 50
Pareto frontier, 114
Pareto optimality, 113
path, 49
Pellet, 13
PepeSearch, 30
portability, 26
precision, 74, 76
precision estimation, 139, 140
prefix, 10, 14
PriceSpy, 27
probabilistic setting, 35
productive extension, 70
productive query, 38, 60
productive value function, 73
productive values, 71
projected answers, 60
property, 50
Protégé, 13
pruning, 87

QA system, 36
query, 54
query answers, 59
query construction session, 43
query coverage, 84
query expressiveness, 23
query expressivity, 175
query extension, 63
query log, 77
query pruning, 87
query renaming, 59
query repetitiveness, 23
question answering system, 36

Random Method, 156
range-based value function, 78
RDF, 9, 10
RDF graph, 11
RDF Surveyor, 29
RDF triple, 10
RDF-based systems, 28
recall, 74
refocus, 64
repetitiveness, 23
research papers, 3
resource, 10
resource graph, 44
resource tree, 49, 50
Rhizomer, 34
root class, 65
root variable, 65
rooted query, 56
rooted resource tree, 50

search engine, 35
search methods, 176
search space, 123
semantic technologies, 9
semantic web stack, 9
SemFacet, 34
session, 43
set root, 65
simple query, 58
simple value functions, 73
SOLR, 28
SPARQL, 9, 13
star-shaped query, 28
subclasses, 174
subfunction, 104
subgraph, 48
subquery, 58
subquery answers, 101

successor, 119, 153

Turtle, 11
type, 50
type I error, 75
type II error, 75
typing function, 53

uniformity assumption, 141
union, 16
universe, 44
URI, 10
usability, 23
user actions, 63
user intention, 22
user studies, 176

value function, 73
variable, 51
variable-preserving pruning, 87
visual query system, 43
vocabulary, 12

W3C, 9
walk, 49
Wikidata, 127, 128
Wikidata dataset, 130
Wikidata navigation graph, 129
Wikidata query log, 132
World Wide Web Consortium, 9

Zipfian distribution, 141