University of Zurich
Department of Informatics

Diploma Thesis November 29, 2006

The Fundamentals of iSPARQL

Markus Stocker
of Ettiswil LU, Switzerland

Student-ID: 99-905-788
markus.stocker@gmail.com

Advisor: **Christoph Kiefer**

Prof. Abraham Bernstein, PhD
Department of Informatics
University of Zurich
http://www.ifi.unizh.ch/ddis

Acknowledgements

The author is grateful to Dr. Andy Seaborne and Dr. Christopher James Dollin from HP Labs Bristol (UK), PD Dr. Mosi Mresse, Dipl. Inf. Bastian Quilitz, and Dr. Boris Motik for numerous discussions and support during the last months. Many thanks also to the communities around mailing lists, forums and IRC channels for helpful and quick answers.

The comprehension and English grammar of this thesis was improved in quality thanks to an accurate proofreading by Magdalena Gwozdz and Jakub Kalla. Without their help, reading the thesis would be far from a pleasant task.

Special thanks to my advisor Christoph Kiefer and Prof. Abraham Bernstein, Ph.D., for giving me the opportunity to write this thesis and for supporting me at each stage with groundbreaking discussions. I very much appreciated the good team work and friendly ambiance, as well as the willingness to always find some time to review the work. Additionally, the always challenging discussions constantly motivated me to go further into the topics.

I would like to thank all my friends for tearing me away once in a while from the pixel screen which was tracking me all the time, and for conversations which help the brain to rest from daily issues related to the thesis. Thanks also to the fellow students I got to know during the studies for numerous funny events which sweetened the sometimes dry university classes.

Last but not least, thanks to my parents for supporting me over my entire life. It is also their merit, that I could terminate my studies writing this thesis.

Abstract

The growing amount of semantically annotated data and published ontologies opens an interesting and challenging application for similarity measures. In face of limited knowledge about the distributed data on the Semantic Web, similarity measures allow a retrieval which is expected to improve the performance compared to exact querying. This thesis presents iSPARQL, an extension to SPARQL which allows querying for similar resources in both RDF/RDFS and OWL ontologies, and supports the development of strategies to compute the similarity of ontological resources.

Huge data volume forced the development of query optimization techniques for relational database systems. However, query engines for ontological data based on graph models, mostly execute user queries without considering any optimization. Especially for large ontologies, optimization techniques are required to ensure that query results can be delivered within reasonable time. OptARQ is a first prototype for iSPARQL query optimization based on the concept of triple pattern selectivity estimation. The evaluation we conduct demonstrates how triple pattern reordering according to their selectivity affects the query execution performance.

Zusammenfassung

Die wachsende Menge semantisch annotierter Daten und verfügbaren Ontologien schaffen eine Basis für interessante und herausfordende Anwendungen für Ähnlichkeitsmasse. Bei fehlendem Wissen über die auf dem Semantic Web verteilten Informationen, ermöglichen Ähnlichkeitsmasse eine Suche, die erwartungsgemäss leistungsstärker sein dürfte, als die für exakte Suchsysteme. Diese Diplomarbeit stellt iSPARQL vor, eine SPARQL Erweiterung, die es ermöglicht, nach ähnlichen Ressourcen sowohl in RDF/RDFS als auch in OWL Ontologien zu suchen. Weiter unterstützt iSPARQL die Entwicklung von Strategien, um die Ähnlichkeit zwischen Ressourcen in Ontologien zu berechnen.

Grosse Datenmengen führten bei relationalen Datenbanksystemen schon seit längerem zur Entwicklung verschiedener Optimierungstechniken. Systeme für Datenmodelle, die auf Graphen basieren, führen die Anfrage hingegen meist ohne Optimierungen durch. Speziell für grosse Ontologien sind Optimierungstechniken entscheidend, um die Resultate in nützlicher Zeit zu ermitteln. OptARQ ist ein erster Prototyp für iSPARQL Optimierung, der die Selektivität von *Triple Patterns* berücksichtigt. Die durchgeführte Evaluation zeigt, wie eine Ordnung der *Triple Patterns* gemäss deren Selektivität die Geschwindigkeit der Anfragen massiv verbesseren kann.

Table of Contents

1 Introduction **1**
1.1 Motivation . . . 2
1.2 Related Work . . . 2
1.3 Structure . . . 3

2 Similarity **5**
2.1 Similarity as Function of Features . . . 5
2.2 Similarity Measures . . . 6
2.3 Similarity Strategies . . . 7

3 The Imprecise Extension **9**
3.1 Characteristics . . . 9
3.2 Software Components . . . 9
3.2.1 Jena and ARQ . . . 10
3.2.2 SimPack . . . 10
3.3 Architectural Design . . . 10
3.4 Imprecise Query Language . . . 11
3.4.1 Grammar . . . 11
3.5 Framework Usage . . . 13
3.6 Framework Implementation . . . 16
3.6.1 Measures . . . 16
3.6.2 Strategies . . . 16
3.7 Extending the Framework . . . 17

4 Query Optimization **19**
4.1 Optimization Framework . . . 19
4.2 Selectivity . . . 20
4.3 Selectivity Estimation . . . 20
4.4 Selectivity Cost Function . . . 20
4.4.1 Subject Cost Estimation . . . 21
4.4.2 Predicate Cost Estimation . . . 21
4.4.3 Object Cost Estimation . . . 21
4.5 Examples . . . 22
4.6 Statistical Model . . . 22
4.7 SPARQL Optimization . . . 25
4.7.1 Remove Dispensable Pattern . . . 25
4.7.2 Reorder Imprecise . . . 27

4.7.3 Rewrite Filter Variables 27
4.7.4 Move Up Filter 28
4.7.5 Reorder by Selectivity 29
4.8 Imprecise SPARQL Optimization 29
4.8.1 Reorder Imprecise 29
4.8.2 Similarity Index 29
4.8.3 Avoid Execution of Complex Measures 30
4.8.4 Aggregation Optimization 30
4.9 Some Final Thoughts: Pattern Dependency 30
4.9.1 S/P/O Dependency 31
4.9.2 Triple Pattern Dependency 32

5 Evaluation **33**
5.1 Quantitative Query Performance Evaluation 33
5.1.1 Query Engines 33
5.1.2 Dataset 34
5.1.3 Retrieval Tasks 35
5.1.4 Optimizations 36
5.1.5 Results 38
5.1.6 Similarity Index 49
5.1.7 Final Example: Putting It All Together 51
5.2 Qualitative Query Retrieval Evaluation 53
5.2.1 Datasets 53
5.2.2 Evaluations 54

6 Conclusions **59**
6.1 Limitations 59
6.2 Future Work 60

A Appendix A **63**

B Appendix B **65**

Bibliography **69**

List of Figures

3.1 RDF/RDFS Model of iSPARQL . . . 12

4.1 Example of Histogram Classes . . . 24
4.2 RDF/RDFS Model of Statistics Ontology . . . 26

5.1 Retrieval Task A: Absolute Values . . . 39
5.2 Retrieval Task A: OptARQ Absolute Values . . . 39
5.3 Retrieval Task A: Sesame SeRQL Absolute Values . . . 39
5.4 Retrieval Task A: Absolute Values (logarithmic scale) . . . 40
5.5 Retrieval Task A: OptARQ Normalized Values . . . 40
5.6 Retrieval Task A: Sesame SeRQL OptARQ Normalized Values . . . 41
5.7 Retrieval Task A: Model Load Time . . . 41
5.8 Retrieval Task A: Model Memory Consumption . . . 42
5.9 Retrieval Task A: Optimized . . . 43
5.10 Retrieval Task A: Improvements . . . 44
5.11 Retrieval Task B: Absolute Values . . . 44
5.12 Retrieval Task B: OptARQ and Sesame SeRQL Absolute Values . . . 45
5.13 Retrieval Task B: Absolute Values (logarithmic scale) . . . 45
5.14 Retrieval Task B: Normalized by OptARQ (logarithmic scale) . . . 46
5.15 Retrieval Task B: Optimized . . . 46
5.16 Retrieval Task B: Improvements . . . 47
5.17 Retrieval Task B: Model Load Time . . . 47
5.18 Retrieval Task B: Model Memory Consumption . . . 47
5.19 Experiment A and B . . . 49
5.20 Experiment A and B: Trend Function . . . 49
5.21 Similarity Index Off . . . 50
5.22 Similarity Index On . . . 51
5.23 Final Example: Performance Rule-by-Rule including Similarity Index . . . 52
5.24 Precision, Recall, F-Measure: TFIDF Strategy A . . . 54
5.25 Precision, Recall, F-Measure: Levenshtein of Levenshtein Strategy B . . . 55
5.26 Precision, Recall, F-Measure: Average for Strategy A . . . 55
5.27 Precision, Recall, F-Measure: Average for Strategy B . . . 56
5.28 Gold Standard vs. Levenshtein Strategy . . . 56
5.29 Gold Standard vs. Tree Edit Distance Strategy . . . 57
5.30 Gold Standard vs. Levenshtein of Levenshtein Strategy . . . 57

List of Tables

4.1 Triple Pattern Cost Estimation . . . 22

5.1 SwetoDblp Samples: Sample Size, Number of Triples, Number of Resources and Result Set Size for both Retrieval Tasks A and B . . . 35
5.2 Retrieval Task A: Absolute Values . . . 41
5.3 Retrieval Task A: Model Load Time . . . 42
5.4 Retrieval Task A: Model Memory Consumption . . . 42
5.5 Retrieval Task A: Optimized . . . 43
5.6 Retrieval Task B: Absolute Values . . . 45

5.7 Retrieval Task B: Optimized . . . 46
5.8 Absolute Values for Rule-by-Rule Evaluation including Similarity Index . . . 53
5.9 Strategy Deviation Ranking . . . 58

List of Listings

3.1 iSPARQL PREFIX . . . 13
3.2 iSPARQL Example 1 . . . 14
3.3 iSPARQL Example 2 . . . 14
3.4 iSPARQL Example 3 . . . 15
3.5 iSPARQL Example 4 . . . 16
3.6 iSPARQL Example 5 . . . 17
4.1 Statistical Ontology Model: Average Number of Predicates and Number of Triples 23
4.2 Statistics: Predicate Resource . . . 24
4.3 Statistics: Histogram Representation . . . 25
4.4 Example: Rewrite Filter Variables . . . 27
4.5 Optimized Example: Rewrite Filter Variables . . . 28
4.6 Optimized Example: Move Up Filter . . . 28
5.1 SwetoDblp Retrieval Task A . . . 35
5.2 SwetoDblp Retrieval Task B . . . 36
5.3 SwetoDblp Retrieval Task A (optimized query) . . . 36
5.4 SwetoDblp Retrieval Task A (Sesame SeRQL) . . . 37
5.5 SwetoDblp Retrieval Task B (optimized query) . . . 38
5.6 SwetoDblp Experiment A . . . 48
5.7 SwetoDblp Experiment B . . . 48
5.8 Similarity Index Retrieval Task . . . 50
A.1 iSPARQL Extended Grammar . . . 63
B.1 Final Example: Query . . . 65
B.2 Final Example: Optimized Query . . . 66

1

Introduction

As the acronym iSPARQL indicates, this thesis is about an extension to SPARQL Protocol And RDF Query Language, SPARQL [Prud'hommeaux and Seaborne, 2006].

The denotation of iSPARQL is: imprecise SPARQL. Thus, we present an extension to SPARQL which provides the ability to query for similar resources in ontologies. Ontologies as a 'formal and explicit specification of a shared conceptualization' [Gruber, 1993] represent a fundamental layer of the Semantic Web, which is described by Fensel as an 'extended Web of machine-readable information' [Fensel, 2004], originally referred by Tim Berners-Lee as the future of the current World Wide Web[1].

Primarily a topic in philosophy, ontologies were first introduced in computer science by artificial intelligence research communities to facilitate knowledge sharing and reuse [Fensel, 2004]. Recently the popularity of ontologies has grown because of their adoption in multiple fields such as enterprise information integration, electronic commerce, and knowledge management [Fensel, 2004].

The architecture of the Semantic Web[2] adopt ontologies in two different layers: ontologies are first introduced on the RDF and RDF-Schema layer, the fundamental layer on which the Semantic Web is based. RDF-Schema (RDFS) allows the definition of mainly hierarchic ontologies by means of class and property inheritance with domain and range restrictions. The ontology vocabulary layer (OWL) provides a language which supports the development of more expressive ontologies allowing the definition of disjointness and boolean combinations of classes, cardinality restrictions, special characteristics of properties, and range restrictions that apply to some classes only [Antoniou and van Harmelen, 2004]. Our iSPARQL framework allows querying for similar resources by means of general or specific similarity strategies in both RDF/RDFS and ontology vocabulary ontologies.

The second discussed subject area are (i)SPARQL optimization techniques. In relational database systems query optimization is a highly discussed topic since the advent of IBM System R [Chamberlin et al., 1981]. This thesis investigates the importance of query optimization, focusing on a widely used SPARQL query engine (ARQ). We present a prototype optimization framework, used to evaluate the performance improvement (OptARQ).

---

[1] http://www.xml.com/pub/a/2000/12/xml2000/timbl.html
[2] http://www.w3.org/2000/Talks/1206-xml2k-tbl/slide10-0.html

1.1 Motivation

Similarity plays an important role in human mind. It allows people to make educated guesses in the face of limited knowledge [Medin et al., 1993]. If we do not know something because we miss the experience of it, we try to make a guess based on similarity. To give a simple example, think about an unfamiliar type of tree. Although we may never have seen the tree before and perhaps we may never have experienced the tree family neither, by means of similarity it is possible to guess that the object we are looking at must be a kind of tree.

Similarity is a type of comparison [Medin et al., 1993]. Similar things tend to be grouped together. Thus, we use similarity to categorize things. When we walk through a forest, we may compare features of trees to guess that trees might belong to the same family, even if we don't know the family name. Just because of common or different features, as the tree size or the leaves color and structure, we are able to figure out similarities (or dissimilarities) and use them to categorize.

Similarity enables an 'interpretative flexibility' which is required in a truly distributed Semantic Web where ontologies are integrated from multiple sources. Exact querying requires that information is globally modeled and equal things are also described equally. Similarity querying allows a retrieval which is expected to improve the performance compared to exact querying. Thus, similarity plays an important role in information retrieval, but also for data integration and data mining.

1.2 Related Work

Imprecise RDQL, iRDQL[3], is an extension to traditional RDF Data Query Language (RDQL) that enables querying for similar resources [Bernstein and Kiefer, 2006]. iRDQL was developed by Dynamic and Distributed Information Systems Group, University of Zurich, Switzerland and is a predecessor of iSPARQL. Although the approach used by iRDQL and iSPARQL is different, the aim of a new ranking by similarity is the same. As a predecessor of iSPARQL, iRDQL inspired many concepts and is a headstone on which the approach implemented for iSPARQL is based on. The main difference is the way how similarity strategies are configured. While iRDQL introduces new keywords, such as `SIMMEASURE`, `IMPRECISE`, and `OPTIONS`, for iSPARQL we introduce *virtual triples* as imprecise statements to configure similarity strategies. Thus, the approach used for iRDQL breaks the standard RDQL language grammar, while iSPARQL does not affect the SPARQL grammar.

The Semantic Similarity Retrieval Model[4] (SSRM) [Hliaoutakis et al., 2006] developed at the Technical University of Crete (TUC), Greece, is an information retrieval method based on semantic similarity which allows querying for documents containing conceptually similar terms. SSRM suggests query expansion by adding semantically similar terms to those already in the query. The similarity between document terms is calculated using the lexical database for the English language WordNet[5] and MeSH[6] (Medical Subject Headings), an ontology of medical and biological terms.

In their paper, Siberski *et al.* [Siberski et al., 2006] present a SPARQL extension that allows the user to query ontologies with preferences. They introduce new keywords to SPARQL, such as `PREFERRING`, to allow expressing ranking relevance by adding soft constraints to the query

[3]http://www.ifi.unizh.ch/ddis/?id=333
[4]http://www.intelligence.tuc.gr/similarity/
[5]http://wordnet.princeton.edu
[6]http://www.nlm.nih.gov/mesh/

according to the user preference. The extension covers two features. First, it returns the solutions that are not dominated by any other solutions. Further, preferably only the solutions that satisfy all the constraints are returned (otherwise, constraints are relaxed). Our approach allows querying for objects which shows the highest similarity to a reference object, using a multitude of similarity strategies (focusing on different object features). iSPARQL employs weighting schemes to enable features preference.

Hurtado *et al.* [Hurtardo et al., 2006] present an approach to make queries more flexible by logical relaxation of their conditions. The approach focuses (but is not limited) to RDFS ontologies. The authors argue, that the `OPTIONAL` clause introduced by SeRQL and SPARQL relaxes query conditions only to a limited extent. They introduce a `RELAX` clause as a generalization of the `OPTIONAL` clause. Mainly, the RDFS hierarchy is exploited to rewrite conditions with a more general concept. Thus, the approach saves the user effort of inspecting the ontology, since the system automatically returns more relaxed answers for the same original query. Furthermore, they introduce a notion of ranking which orders relaxed versions of a query from less to more general.

Perz *et al.* [Perez et al., 2006] conduct an extensive analysis of the semantics and complexity of SPARQL, focusing, as argued by the authors, on the two most complicated operators in SPARQL, `UNION` and `OPTIONAL`. This work may be a starting point for discussions about iSPARQL optimizations especially for future optimization rules, since we do currently not consider query optimization for queries including SPARQL `OPTIONAL` or `UNION` keywords.

Sirin [Sirin et al., 2006] presents optimization techniques for OWL-DL ontologies focusing on knowledge bases containing large number of individuals. Aduna Software[7], developer and maintainer of Sesame open source RDF framework[8], introduced some general query optimization techniques based on query rewriting rules for Sesame RDF Query Language (SeRQL).

KAON2[9], an infrastructure for managing OWL-DL ontologies, introduces algorithms which allow optimization of DL reasoning by applying deductive database techniques. According to [Motik and Sattler, 2006] such algorithms yield to significant performance improvement compared to other available DL reasoners.

1.3 Structure

We introduced and motivated the importance of similarity in computer science. In the following chapters, we focus on our framework to query for similar resources in ontologies. Furthermore, we discuss some general query optimization rules and we present our iSPARQL framework. Finally, we conduct an extensive evaluation and we present our findings.

The chapters are organized as follows. Chapter 2, introduces the concept of similarity theory which mainly influenced the development of iSPARQL, stressing on the difference of similarity measures and strategies. Chapter 3, describes in more details our imprecise framework developed as extension to SPARQL. In Chapter 4, we present our (i)SPARQL optimization techniques mainly based on query rewriting rules which are implemented as an ARQ optimization module, called OptARQ. In Chapter 5, we conduct an extensive quantitative and qualitative evaluation of both query and retrieval performance for iSPARQL queries and we present our findings. We conclude our discussion in Chapter 6 with some final thoughts.

[7] http://www.aduna-software.com/
[8] http://www.openrdf.org/
[9] http://kaon2.semanticweb.org/

2
Similarity

The concept of similarity is ubiquitous in psychological theories of perception, learning and judgment. While geometric models dominated the theoretical analysis of similarity relations, Tversky, 'one of the most influential similarity theorists' [Medin et al., 1993], argued in his *Features of Similarity* [Tversky, 1977], that models which define similarity as a metric distance function are appropriate only for certain stimuli (e.g., colors). Tversky suggested that the similarity between stimuli may be better described as a comparison of features, rather than as a computation of metric distance between points.

This chapter outlines the similarity theory defined by Tversky and how it influences our framework to query for similar resources. We discuss the difference between a similarity strategy and a similarity measure, which is fundamental in our framework. Furthermore, we present the most relevant similarity measures used in our framework.

2.1 Similarity as Function of Features

Tversky argued in [Tversky, 1977], that the similarity $s(a, b)$ of two objects a and b, can be expressed as a function of their common and distinctive features. According to this, the similarity function takes three arguments [Tversky and Gati, 1978]:

- $A \cap B$, the features shared by objects a and b
- $A - B$, the features of a that are not shared by b
- $B - A$, the features of b that are not shared by a

Thus, the similarity between two objects behaves accordingly to the number of shared features. Two objects show the highest similarity if $A \cap B = A \cup B$, where $A \cup B$ is the features union of both objects a and b. As we will describe later on in this chapter, our framework supports the development of strategies to quantify the similarity between resources in ontologies. Basically, strategies are functions which map a set of features to a similarity value. They consider the similarity (or dissimilarity) of features and allow feature weighting.

2.2 Similarity Measures

Our iSPARQL framework uses similarity measures as atomic functions to calculate the similarity of a specific feature shared by two objects. We strictly distinguish measures from strategies (Section 2.3). While measures are used to capture the similarity of a specific feature, strategies depict the overall similarity of objects and implement a specific logic required to map features to a similarity value.

A taxonomy for similarity measures with a detailed description about their properties can be found in [Bernstein et al., 2005]. In the following, we briefly discuss the similarity measures which are relevant for this thesis.

Both Levenshtein and Levenshtein Level2 (also called Levenshtein of Levenshtein) similarity measures use the Levenshtein string edit distance to characterize the similarity of strings. The edit distance measures the relatedness in terms of the number of insert, remove, and replacement operations to turn one string into another [Levenshtein, 1966]. The Levenshtein similarity between two strings str_1 and str_2 is calculated by

$$sim_{lev}(str_1, str_2) = 1 - \frac{xform(str_1, str_2)}{xform_{wc}(str_1, str_2)}$$

where $xform(str_1, str_2)$ defines the edit distance which is normalized by the worst case transformation cost $xform_{wc}(str_1, str_2)$. We get a similarity value by subtracting the normalized edit distance from 1.

Resnik [Resnik, 1995] and Lin [Lin, 1998] compute the similarity of objects in a taxonomy in terms of information-theoretic entropy. The entropy of an object corresponds to the negative logarithm of the probability of encountering that object (or those which are subsumed by it). More specific objects in a taxonomy show a lower probability to be encountered and consequently a higher entropy. Both measures can be used to calculate the semantic similarity of concepts in ontologies. According to Resnik, the similarity is computer as

$$sim_{resnik}(o_1, o_2) = max_{o3 \in S(o_1, o_2)}[-log_2 p(o_3)]$$

where $S(o_1, o_2)$ is the set of common ancestors of o_1 and o_2 (i.e., concepts that subsume both objects) and $p(o_3)$ is the probability of encountering a concept of type o_3 (i.e., subsumed by both o_1 and o_2). Lin defined the similarity of two concepts as

$$sim_{lin}(o_1, o_2) = \frac{2 * log_2 p(MRCA(o_1, o_2))}{log_2 p(o_1) + log_2 p(o_2)}$$

where $p(MRCA(o_1, o_2))$ is the probability of the most recent common ancestor of two concepts o_1 and o_2 [Bernstein et al., 2005], i.e., the probability of the most specific class that subsumes both o_1 and o_2.

The Dice [Ganesan et al., 2003] and the Jaccard [Cohen et al., 2003] similarity measure determine the relatedness of objects in terms of common and distinctive attributes. Strings are considered as bags of tokens and the similarity is computed as a function of the common ($|o_1 \cap o_2|$) and distinctive ($|o_1|$ resp. $|o_2|$) attributes. The Dice coefficient is defined as

$$sim_{dice}(o_1, o_2) = \frac{2 * |o_1 \cap o_2|}{|o_1| + |o_2|}$$

whereas Jaccard is defined as

$$sim_{jaccard}(o_1, o_2) = \frac{|o_1 \cap o_2|}{|o_1| + |o_2| - |o_1 \cap o_2|}$$

The Cosine string similarity [Baeza-Yates and Ribeiro-Neto, 1999] maps strings to binary vectors. The similarity is computed as the cosine of the angle between the vectors. For two vectors v_1 and v_2, the measure is defined as

$$sim_{cosine}(v_1, v_2) = \frac{v_1 \bullet v_2}{\|v_1\|_2 \times \|v_2\|_2}$$

where $\|v\|_2 = \sqrt{\sum_{i=1}^{n} |v_i|^2}$, i.e., the L^2-norm.

The TFIDF measure [Baeza-Yates and Ribeiro-Neto, 1999] computes the similarity of strings (generally speaking documents) as the cosine of the angle between the vectors which represent the documents. The measure associates a weight $w_{t_i,d}$ to each term t_i in a document d, which is computed as

$$w_{t_i,d} = tf_{t_i,d} \times idf_{t_i} = tf_{t_i,d} * log(\frac{N}{d_{t_i}})$$

where $tf_{t_i,d}$ is the term frequency of t_i in document d, N the total number of documents in the corpus, and d_{t_i} the number of documents where the term t_i appears. The higher the term frequency and lower the inverse document frequency, the higher is the resulting term weight.

Shortest path and tree edit distance similarity measures consider the ontological graph structure while computing the similarity of objects. Both can be used to calculate the semantic similarity of concepts in ontologies. The shortest path measure is defined as

$$sim_{edge}(o_1, o_2) = \frac{2 * MAX - len(o_1, o_2)}{2 * MAX}$$

where MAX is the length of the longest path from the root of the ontology to any of its leaf concepts and $len(o_1, o_2)$ is the length of the shortest path from o_1 to o_2. Basically, this is a edge counting distance measure which is converted to a similarity measure.

As for Levenshtein, the tree edit distance is a function of the operations required to turn the tree T_1 into the tree T_2 by applying insertion, substitution, and deletion of nodes [Valiente, 2002]. We turn the normalized least-cost transformation of T_1 to T_2 into a similarity value by subtraction from 1. Thus, the similarity based on tree edit distance is defined as

$$sim_{tree}(o_1, o_2) = 1 - \frac{TreeDist(T_1, T_2)}{|T_1| + |T_2|}$$

where $|T|$ is the sum of nodes for T.

2.3 Similarity Strategies

Our framework to query for similar resources in ontologies, iSPARQL, allows the implementation and reuse of similarity strategies. Basically, it is a container for strategies where a strategy implements the required logic to capture the similarity of things. In ontologies, things may be either URI-references or literals and we generally call them objects. In our framework, we assign

ontological objects to strategies. A strategy may exploit not only the corresponding object features, but extract more related features from the underlying ontology in order to get an optimal similarity approximation.

The fundamental questions are (1) how the similarity of objects is best approximated and (2) against whom or what should the approximation be compared, i.e., a reference similarity value is required. A reasonable reference may be human similarity judgment. As in the experiment performed by Miller and Charles [Miller and Charles, 1991], ultimately similarity algorithms are compared to human similarity judgment. Thus, it is a matter of finding the similarity strategy which minimizes the deviation from human judgment.

Given two ontological objects, we can think of very different ways how to determine the similarity between them. For example, we may calculate an overall similarity based on a comparison of either all features shared by the objects or just a subset. Furthermore, the similarity of an object feature can be typically calculated by a number of different measures and each approximates the similarity differently. Normally, there is a measure that approximates better human similarity judgment for the observed feature than others. Moreover, strategies (i.e., the function which maps object features to a similarity) are highly affected by the pursued similarity goal. As a simple example, consider the task of evaluating the similarity of some curriculum vitae. A strategy which captures the similarity on experience and skills may consider different features than a strategy which targets at capturing the similarity in education.

Object features are of different types. We distinguish two classes: intrinsic and extrinsic. Intrinsic features are characteristics that describe an individual object. Object attributes (i.e., predicates) are examples for intrinsic features (e.g., the age of a person). Extrinsic features are characteristics that an object exhibits in reference to other objects. For example, the subtype hierarchy observed for a specific class should be considered as extrinsic feature.

Whether a similarity measure is meaningful for a specific feature depends on the class to which the feature belongs. While the similarity of intrinsic features is usually better captured by some character sequence measure, for extrinsic features, edge counting or information content methods are usually more suitable to approximate human similarity judgment.

Consequently, we are able to measure the similarity of 'things' in a number of different ways, each of them resulting in different and specific strategies. In order to determine the goodness of a strategy, we should evaluate it against a reference value, i.e., human similarity judgment.

3

The Imprecise Extension

Inspired by A. Tversky (Chapter 2), we implement an imprecise extension to SPARQL as a framework for similarity strategies which allows a novel ranking of resources based on similarity.

In the following sections we describe our approach and proposed framework to query for similar resources in ontologies in more details.

3.1 Characteristics

Our approach fulfills two fundamental characteristics. First, iSPARQL queries are fully compatible SPARQL queries. As we will see later in this chapter, imprecise statements are embedded in SPARQL queries and are SPARQL compatible. The approach does not require additional query language constructs and allows defining similarity strategies in SPARQL syntax. Secondly, the imprecise framework can be integrated as extension to the query engine used for the implementation (ARQ).

The combination of both characteristics yields an optimal and seamless integration of our imprecise framework into the query engine. No customization is required for the query engine in order to enable the imprecise framework.

3.2 Software Components

The imprecise framework requires a number of software components. As a SPARQL extension, it requires a SPARQL query engine. Thus, we build the imprecise extension on top of a SPARQL query engine, in order to reuse all the functionality required to execute SPARQL queries. This includes dataset loading, query parsing and syntax checking, query execution planning, query execution and result set consumption.

The imprecise framework fits into the result set consumption stage, where matching resources are compared against their similarity, and it returns a similarity value, which reflects the corresponding strategy. The similarity value may further be reused in (i)SPARQL, e.g., for a similarity ranking.

3.2.1 Jena and ARQ

Jena[1] is an open source Java framework for building Semantic Web applications developed within the HP Labs Semantic Web Programme[2]. It includes a RDF[3] and OWL[4] API, reading and writing RDF in RDF/XML[5], N3[6] and N-Triples[7] and in-memory or persistent dataset storage as well. Jena includes a SPARQL query engine called ARQ[8] - a SPARQL Processor for Jena. ARQ supports multiple query languages like SPARQL, RDQL and ARQ (the engine's own language - a superset of SPARQL) and multiple query engines.

3.2.2 SimPack

SimPack[9] is a generic Java library of similarity measures for the use in ontologies and other application domains activated by the Dynamic and Distributed Information Systems Group[10] of the University of Zurich, Switzerland.

Because of the generic design character used in SimPack, the library is optimally integrated into the imprecise framework. SimPack allows a comfortable usage of similarity measures within the framework. Thus, our imprecise framework delegates the similarity computation to SimPack and acts as a layer between the ARQ query engine and the SimPack library, where it adds the concept of a similarity strategy which allows features mapping of complex ontological resources to basic similarity measures in order to characterize the similarity between ontology resources.

3.3 Architectural Design

As we described above, the imprecise framework is invoked by ARQ during the result set consumption stage of query processing. The framework exploits a special feature of ARQ which allows to wrap Java classes behind predicates of triple pattern statements in SPARQL queries.

Similarity strategies are defined in SPARQL queries as a block of imprecise statements. During result set consumption ARQ processes each statement defined in query and assigns the strategy attributes to the imprecise framework. Once the strategy attributes are set, the corresponding similarity strategy is executed by the framework, which returns a single similarity value as result of this process.

The interface of similarity strategies is kept very small. Every imprecise strategy is obliged to implement a similarity logic that returns an adequate similarity value which is finally bound to a SPARQL variable. Because of the binding, we are able to reference similarity values in SPARQL queries or in other similarity strategies. For example, ARQ may use the similarity value to rank the resulting tuples, or a subsequent similarity strategy may reuse a similarity value as a threshold, or aggregate multiple similarity values.

The imprecise framework supports a number of attributes which may be used to configure similarity strategies. Setting a strategy threshold may be an example for such an attribute. Furthermore, it includes a number of similarity measures which can be used in similarity strategies.

[1] http://jena.sourceforge.net
[2] http://www.hpl.hp.com/semweb/
[3] http://www.w3.org/RDF/
[4] http://www.w3.org/2004/OWL/
[5] http://www.w3.org/TR/rdf-syntax-grammar/
[6] http://www.w3.org/DesignIssues/Notation3
[7] http://www.w3.org/2001/sw/RDFCore/ntriples/
[8] http://jena.sourceforge.net/ARQ/
[9] http://www.ifi.unizh.ch/ddis/research/semweb/simpack/
[10] http://www.ifi.unizh.ch/ddis/

Framework measures are the interface to SimPack and invoke the corresponding SimPack measures from the library. Thus, the framework provides measures as wrappers around SimPack similarity measures.

Figure 3.1 illustrates the RDFS for an iSPARQL similarity strategy. Moreover, we include an individual which specifies a set of imprecise statements to show the usage of the schema for a typical strategy.

3.4 Imprecise Query Language

The imprecise query language, iSPARQL, extends traditional SPARQL by introducing specific imprecise statements (i.e., virtual triples). SPARQL is a declarative language to query RDF graphs. It allows querying by triple patterns and supports the definition of conjunctive, disjunctive as well as optional patterns [Manola and Miller, 2004]. Standard features known from SQL, e.g., order and limit are supported too. In this section we take a closer look to the specific imprecise statements. We will start by defining the extended iSPARQL grammar.

3.4.1 Grammar

The extended imprecise query language grammar is based on traditional SPARQL grammar [Prud'hommeaux and Seaborne, 2006]. As we stated previously in this chapter, iSPARQL does not introduce new language constructs.

Instead, we tie up to the expression of `FilteredBasicGraphPattern` defined in traditional SPARQL grammar, by adding our `ImpreciseBlockOfTriples` symbol. Although the structure of an `ImpreciseBlockOfTriples` is very similar to the one of a `BlockOfTriple`, the behavior of imprecise statements is completely different compared to usual triple patterns, though they may look alike. In fact, instead of matching a pattern in some graph model, imprecise statements are not matched against a graph. The goal of imprecise statements, is to create a bridge between ARQ and the imprecise framework, in order that the strategy attributes specified in some iSPARQL query are assigned to the imprecise framework which executes similarity strategies. To reflect this behavioral difference of SPARQL triple patterns and iSPARQL imprecise statements, we found it necessary to extend the SPARQL grammar. The grammar extension is displayed in Appendix A.

An `ImpreciseBlockOfTriples` is a similarity strategy query definition, which is a sequence of imprecise statements. Our imprecise framework implements two different classes of strategies. The first class includes strategies that calculate a similarity value out of ontology resources. The second class, however, covers strategies that calculate a similarity out of multiple similarity scores. We call them aggregation strategies because they aggregate similarity scores to an overall similarity value.

An `ImpreciseBlockOfTriples` requires a `NameStatement`. In addition, depending on whether or not the `ImpreciseBlockOfTriples` is a aggregation strategy, a `ScoresStatement` or `ArgumentsStatement` is required too. Moreover, a `SimilarityStatement` is required to bind the strategy similarity value to a SPARQL variable.

A `NameStatement` is used to specify the similarity strategy designated to calculate the similarity value. Further, the statement initializes a new strategy inside of the imprecise framework. This statement is mandatory for every strategy, regardless to whether or not it is an aggregation strategy.

An `ArgumentsStatement` assigns two ontological objects to the framework. They will be used while executing the similarity strategy as the objects on which the similarity is calculated.

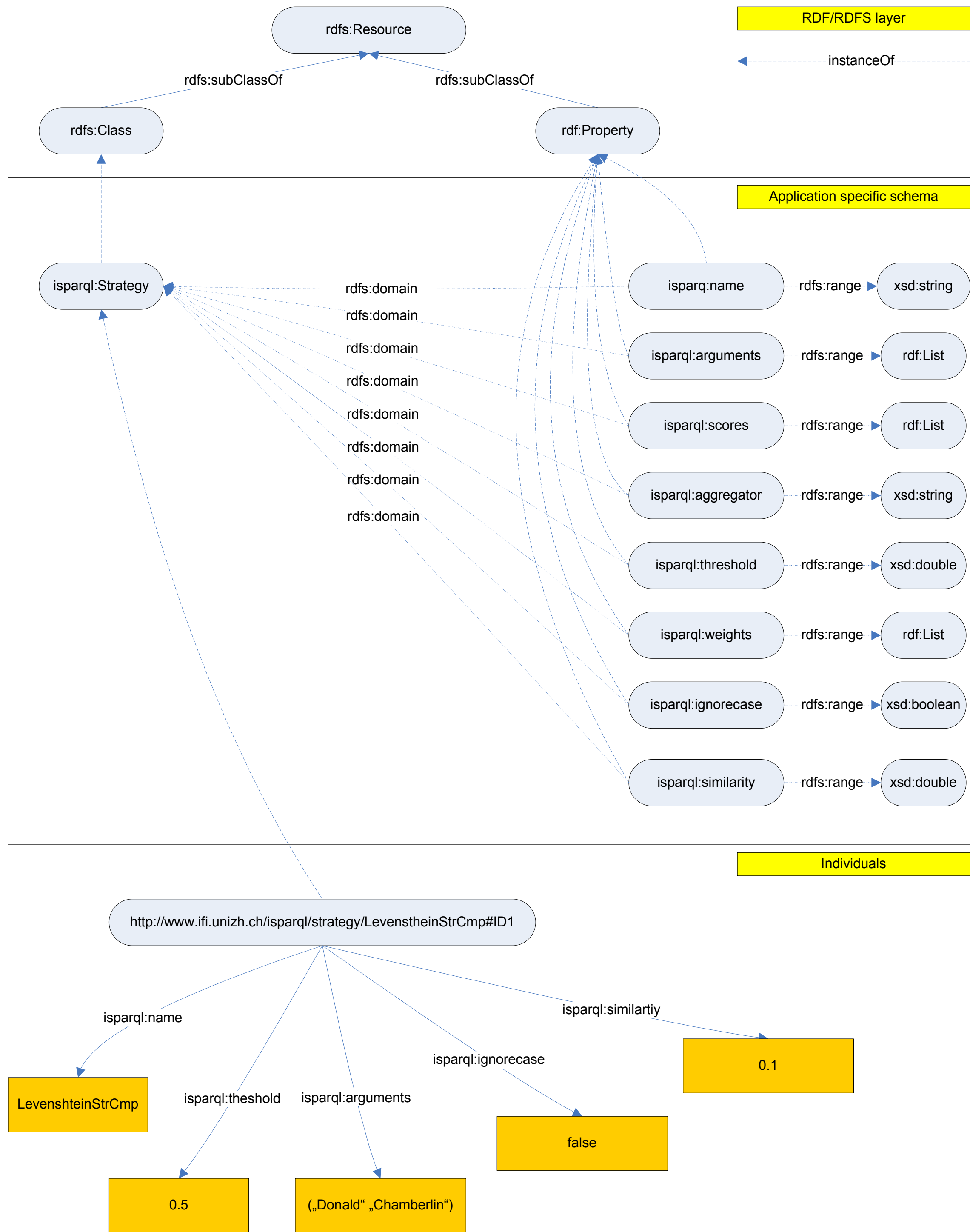

Figure 3.1: RDF-Schema for iSPARQL Imprecise Statements with an Individual.

The statement takes a list of variables, URI-references or literals as parameters. Please note that variables must be bound by ARQ in a previous stage. This statement is mandatory for every strategy except for aggregation strategies.

A `ScoresStatement` takes a list of one or more similarity variables or values as parameters. Variables must be previously bound in query by some other similarity strategies. This statement is used for aggregation strategies only.

A `WeightsStatement` is used in combination with a `ScoresStatement`. Thus, it is inherent to aggregation strategies but not mandatory. The goal is to provide a weighting method for similarity scores. This allows specifying strategy relevance directly in iSPARQL queries by weighting their similarity values.

An `AggregatorStatement` allows setting an aggregation method which is used to aggregate multiple similarity values. An aggregation method is required for an aggregation strategy. Also, strategies where ontology resources are compared by a complex logic usually aggregate multiple similarity scores. The statement is not mandatory and can be used in both strategy classes.

A `ThresholdStatement` allows setting a strategy threshold. This statement is not mandatory and may be used in both strategy classes. If a threshold is set, the similarity value calculated by the strategy has to satisfy the threshold in order to be considered a result. If the threshold is not satisfied, the pattern match is not considered further and the binding is dropped.

An `IgnorecaseStatement` is used in combination with strategies, where the case of character sequences may matter. Thus, it depends on the selected strategy, whether or not this statement is useful.

Finally, a `SimilarityStatement` executes the similarity strategy according to its configuration and implementation. The statement executes the similarity logic implemented by the strategy specified in `NameStatement` and requires a SPARQL variable as parameter. The variable is used to bind the resulting similarity value. The `SimilarityStatement` is mandatory in both similarity strategy classes.

3.5 Framework Usage

This section sets the focus on framework usage which is demonstrated by short commented examples. In order to enable imprecise querying in SPARQL queries, we need to add an additional PREFIX to the query. This special PREFIX imports the Java namespace of iSPARQL strategy attributes. Thus, we first add the PREFIX listed in 3.1 to any iSPARQL query.

Listing 3.1: iSPARQL PREFIX

```
PREFIX isparql: <java:ch.unizh.ifi.isparql.query.property.>
```

The PREFIX enables the defining of iSPARQL statements and may be named arbitrarily. We use ‘isparql’ as a convention in order to better differentiate SPARQL from iSPARQL statements.

The first iSPARQL query example (Listing 3.2) shows a retrieval of resource predicates based on a RDF dataset of individuals. The query returns the similarity value between the first name and last name of each person. The similarity is calculated by means of the `LevenshteinStrCmp` strategy which allows calculating the similarity of two strings using Levenshtein. The query shows a minimal configuration for a valid iSPARQL query. As explained previously in this chapter, we need to specify a strategy, set the similarity arguments, and provide a variable to which the similarity value is bound.

Listing 3.2: iSPARQL Example 1

```
PREFIX person: <http://person/>
PREFIX isparql: <java:ch.unizh.ifi.isparql.query.property.>

SELECT ?firstname ?lastname ?similarity
WHERE {
  ?person person:firstname ?firstname .
  ?person person:lastname ?lastname .

  ?strategy isparql:name "LevenshteinStrCmp" .
  ?strategy isparql:arguments (?firstname ?lastname) .
  ?strategy isparql:similarity ?similarity
}
```

Listing 3.3: iSPARQL Example 2

```
PREFIX person: <http://person/>
PREFIX isparql: <java:ch.unizh.ifi.isparql.query.property.>

SELECT ?firstname ?similarity
WHERE {
  ?person1 person:firstname "Markus" .
  ?person2 person:firstname ?firstname .

  ?strategy isparql:name "LevenshteinResCmp" .
  ?strategy isparql:arguments (?person1 ?person2) .
  ?strategy isparql:threshold 0.7 .
  ?strategy isparql:aggregator "median" .
  ?strategy isparql:ignorecase true .
  ?strategy isparql:similarity ?similarity
}
```

In Listing 3.3 we define a sensitive more complex strategy compared to our first example (Listing 3.2). This second example adds three more imprecise statements: a threshold, a specific aggregation method, and an 'ignore case' for string similarity comparison. Furthermore, we specify another similarity strategy.

The strategy used in this second example is more complex compared to the first one, but it is again a general purpose similarity strategy. Thus, it is not specific to any ontology. Instead of simply calculating the Levenshtein similarity of strings, the strategy iterates over each resource predicate and accumulates similarity scores to an overall similarity. Resource objects are compared by Levenshtein. Objects with a URI-reference are recursively resolved. Please note that the resulting similarity strongly depends on the selected aggregation method.

Our next example (Listing 3.4), employs both strategy classes included in iSPARQL. This is the first example of an aggregation strategy, called `ScoreAggregator`. An aggregation strategy is slightly different from other similarity strategies because there is no similarity measure involved. Nevertheless, it is still a strategy since a similarity strategy is a function that generates a similarity value. As we explained earlier in this chapter, the goal of an aggregation strategy is to aggregate similarity scores previously calculated by other strategies.

Listing 3.4: iSPARQL Example 3

```
PREFIX person: <http://person/>
PREFIX isparql: <java:ch.unizh.ifi.isparql.query.property.>


SELECT ?person1 ?person2 ?aggregated
WHERE {
  ?person1 person:firstname ?firstname1 .
  ?person1 person:lastname ?lastname1 .
  ?person2 person:firstname ?firstname2 .
  ?person2 person:lastname ?lastname2 .

  ?strategy1 isparql:name "LevenshteinStrCmp" .
  ?strategy1 isparql:arguments (?firstname1 ?firstname2) .
  ?strategy1 isparql:similarity ?sim1.

  ?strategy2 isparql:name "LevenshteinStrCmp" .
  ?strategy2 isparql:arguments (?lastname1 ?lastname2) .
  ?strategy2 isparql:similarity ?sim2 .

  ?strategy3 isparql:name "ScoreAggregator" .
  ?strategy3 isparql:scores (?sim1 ?sim2) .
  ?strategy3 isparql:weights (0.8 0.2) .
  ?strategy3 isparql:aggregator "sum" .
  ?strategy3 isparql:similarity ?aggregated
}

ORDER BY DESC(?aggregated)
```

The example query in Listing 3.5 shows the employment of a specific strategy. The similarity strategies used in previous examples were all general, i.e., ontology independent. The strategy involved in this example is intended to better characterize the similarity of resources in MIT Process Handbook Ontology [Malone et al., 2003].

The MIT Process Handbook is a knowledge base including over 5000 business activities. Each activity is described using two dimensions: the parts-uses and the specializations-generalizations. This characteristic allows defining a wide range of strategies.

The strategy used in our query (Listing 3.5) considers the sub processes of two resources. Although it may look very similar to the one used in our previous example (Listing 3.3), this strategy is specific to the MIT Process Handbook ontology. This is because we explicitly compare only the sub processes of business activities (i.e., parts). Thus, we consider only a sub set of the predicates defined for a MIT Process Handbook resource, whereas in our previous example (Listing 3.3) every predicate was considered.

The query defined in Listing 3.6 shows the usage of another specific strategy. Once again, the underlying ontology is MIT Process Handbook. The similarity strategy is specific because it analyzes only a sub set of the resource predicates, namely the sub processes. The involved similarity measure for this example makes the major difference to the strategy used in Listing 3.5. In fact, instead of Levenshtein, the strategy considers the sub process trees generated by resources and the similarity is calculated as the edit distance of trees.

Listing 3.5: iSPARQL Example 4

```
PREFIX ph:      <http://www.ifi.unizh.ch/ddis/ph/2006/08/ProcessHandbook.owl#>
PREFIX isparql:    <java:ch.unizh.ifi.isparql.property.>

SELECT ?name ?similarity

WHERE {
  ?process1 ph:name "Sell software"@en .
  ?process2 ph:name ?name .

  ?strategy isparql:name "MITPHResCmp" .
  ?strategy isparql:arguments (?process1 ?process2) .
  ?strategy isparql:similarity ?similarity

}

ORDER BY DESC(?similarity)

LIMIT 20
```

3.6 Framework Implementation

This section uncovers some of the most important framework implementation details. The design meets a typical framework architecture. Typically a framework architecture allows the organization and implementation of new software projects. Indeed, our imprecise framework is designed to allow the development of new components, especially similarity strategies.

3.6.1 Measures

The imprecise framework is an infrastructure which organizes and implements multiple similarity measures. The measure classes in our framework are not intended to implement any logic to calculate the similarity. Instead, our similarity measures are wrappers [Gamma et al., 1995] for measures implemented in the SimPack library.

Similarity measures are organized in a specific Java package[11] and are invoked in similarity strategies to calculate the similarity of resources. For complex similarity logics a strategy usually deals with two URI-references that correspond to some ontology resource. URI-references are not intended to be compared for similarity. In fact, similarity strategies and measure wrappers need to format the information about resources into data which is subsequently used for similarity computation by SimPack.

3.6.2 Strategies

Similarity strategies extend the framework abstract strategy and implement the strategy interface. The strategy interface consists of a single method which returns a double value and does not require any parameters. Thus, the interface required for a similarity strategy is very small.

[11] ch.unizh.ifi.isparql.query.measure

Listing 3.6: iSPARQL Example 5

```
PREFIX ph:     <http://www.ifi.unizh.ch/ddis/ph/2006/08/ProcessHandbook.owl#>
PREFIX isparql:    <java:ch.unizh.ifi.isparql.query.property.>

SELECT ?similarity

WHERE {
  ?process1 ph:name "Sell software"@en .
  ?process2 ph:name "Sell"@en .

  ?strategy isparql:name "MITPHTreeEditDistanceResCmp" .
  ?strategy isparql:arguments (?process1 ?process2) .
  ?strategy isparql:similarity ?similarity
}
```

The imprecise framework provides every strategy all the required context information about the iSPARQL query, the imprecise attributes, and the underlying graph. Thus, each strategy is able to use context information which allows building complex similarity strategies.

Strategy extensions are organized in a specific Java package[12] and are invoked by a strategy factory [Gamma et al., 1995]. Similarity strategies are identified by a unique URI-reference [Berners-Lee et al., 1998]. This URI-reference is bound to the corresponding SPARQL variable while processing the first imprecise statement (i.e., the `NameStatement`). Each time ARQ encounters an imprecise statement during query processing, the URI-reference bound to the imprecise statement subject is extracted and used as a key to identify the corresponding strategy object inside the imprecise framework. This procedure ensures that imprecise attributes defined in iSPARQL queries are assigned to the corresponding strategy represented in the framework.

3.7 Extending the Framework

The imprecise framework is intended for the implementation of strategies, measures, attributes and aggregation methods. Since we believe that specific similarity strategies result in better approximation of human similarity judgment, extending the framework is expected to be a permanent activity.

Achieving good similarity approximations is anything but a trivial task. A reference similarity judgment, i.e., gold standard and a deep study of the underlying ontology are required. We need to investigate the structure of resources, identify relevant predicates, consider possible subsumption structures. Ultimately, research and experience may lead to a good similarity strategy. Once a strategy is defined, we deliberate how to implement it. The framework provides multiple features that can be reused and allows extending the not yet implemented features required to run a strategy. Wrappers for SimPack measures, imprecise attributes, similarity strategies, or aggregation methods are examples of packages where the framework may be extended while implementing new strategies.

As we explained earlier in this chapter, every similarity strategy extends the framework abstract strategy and is organized in a specific Java package. A new strategy needs to implement the similarity method that returns a double value. Aside from this interface, we are free to implement

[12] ch.unizh.ifi.isparql.query.strategy

any complex logic required to characterize the similarity of resources. The imprecise framework provides all required information about the imprecise attributes set in query and the underlying graph. Thus, we may access this data and gather more information concerning the resources which are usually referenced by a URI-reference.

Sometimes it is required to add a new imprecise attribute perhaps because of a newly added strategy. Imprecise attributes are Java classes that extend ARQ classes[13]. While measure wrappers depend on SimPack interface, imprecise attribute classes depend on ARQ interfaces. In fact, imprecise attributes with a single literal or a variable specified as object (e.g., imprecise threshold statement) extend the ARQ `PFuncSimple` class, whereas imprecise attributes with a list of literals or variables specified as object (e.g., imprecise arguments statement) extend the ARQ `PropertyFunctionBase` class. Extending those ARQ classes enables access to ARQ bindings, the subject, predicate, and object of imprecise attributes, and the ARQ query execution context.

[13]com.hp.hpl.jena.query.pfunction

4

Query Optimization

Since the advent of System R [Selinger et al., 1979], query optimization has always been a research topic. The techniques introduced by D. Chamberlin and his IBM research group in San José, California, in 1979 are still used in commercial database systems [Oommen and Rueda, 2001]. The concept of selectivity factor described in [Selinger et al., 1979] is still a crucial one. Clearly, selectivity estimation methods evolved from simply formulas to more complex statistical techniques, but the foundations are yet applicable.

This chapter outlines our proposed iSPARQL optimization framework. We delineate the optimization techniques applicable to both SPARQL and iSPARQL queries. Moreover, we describe the fundamental concept of triple pattern selectivity estimation on which our query optimization approach is based. Finally, we discuss the optimization prototype implementation and we present the cost function together with the statistical model which is required for selectivity estimation.

4.1 Optimization Framework

The proposed optimization framework for (i)SPARQL queries mainly focuses on static optimization techniques. By static optimization we mean general rules which may be applied in order to get an optimal query execution plan. Thus, static optimization is a query rewriting process usually executed after query parsing and syntax checking.

Our optimization framework which is implemented for ARQ, consists of a number of query rewriting rules. Some of them are trivial since they simply remove dispensable triple patterns or rewrite FILTER expressions by executing them as early as possible in a query. However, as we will describe later on in this chapter, one slightly more complex rule builds on our statistical triple pattern selectivity estimation approach.

The fundamental aim of the proposed optimization framework is to reduce intermediate result sets of triple patterns and imprecise strategies. Basically, each rule is intended to fulfill this goal. Our statistical approach allows a selectivity estimation of triple patterns, whereas a triple pattern execution cost function enables a pattern ranking by (expected) intermediate result set sizes.

Furthermore, we consider the employment of similarity indexes in order to allow a cached lookup for similarity values during query execution instead of computing them. Because of the potentially very large number of resource pairings and similarity comparisons in ontologies this technique my be useful only for small ontologies and highly complex strategies.

Finally, we discuss some best practices that may be considered while developing similarity strategies. For instance, we may avoid to execute some expensive similarity measure by evaluating the result of a faster measure or by testing resource URI-references for equality.

4.2 Selectivity

Selectivity is the most crucial concept on which our optimization model is based. Piatetsky defined the selectivity in [Piatetsky-Shapiro et al., 1984] as follows.

Definition 1 *Selectivity of a condition E, denoted SEL(E), is the fraction of tuples satisfying this condition.*

For example, SEL(number = 6) is about 0.16 for a typical dice. This definition can be adapted with some small modification to ontological data models. We may modify the definition for iSPARQL queries, to

Definition 2 *Selectivity of a triple pattern T, denoted SEL(T), is the fraction of triples satisfying the pattern.*

Selectivity is fundamental because it quantifies the size of intermediate result sets, which themselves are highly relevant in (i)SPARQL, especially when triple patterns are joined. The smaller intermediate result sets are the faster joins are processed. Thus, the key solution is to minimize the triple pattern intermediate result set size which means to execute first triple patterns with small selectivity.

The optimization we focus on is based on query rewriting rules. We found some general rules that can by applied and introduced a histogram based statistical model in order to estimate the selectivity of a triple pattern. Thus, the second fundamental topic is how to estimate the selectivity in an efficient way during query execution.

4.3 Selectivity Estimation

Selectivity can be calculated by an exact formula or an estimation which is based on statistics about the resources contained in the underlying ontology. In our research, we first started to think about an index which enables extracting exact selectivity for each triple pattern. Because an exact triple pattern selectivity computation basically requires the pattern to be executed we moved from considering exact models to statistical models which allowed us to get an estimation of the selectivity.

Later on in the evaluation (Chapter 5), the estimation approach based on statistics showed to be precise enough to optimize our retrieval tasks. We believe, the estimations are precise enough to allow a correct ranking of triple pattern based on the selectivity. One of the most important benefit of using a statistical representation of the ontology resources to get a selectivity estimation is the size of the resulting statistical model. The overhead size of data required for our selectivity cost model is marginal and can be heightened or scaled down, resulting in more or less precise information about the selectivity.

4.4 Selectivity Cost Function

We define a cost function that reflects the selectivity estimation and is used to rank triple patterns in increasing order of selectivity, i.e., increasing order of expected execution cost. The cost function returns a value between 0 and 1, thus, it is basically a normalization to [0,1] of the estimated selectivity.

We model the overall cost for a triple pattern as follows

$$c(t) = c(s) * c(p) * c(o)$$

where $c(t)$ is the overall cost for a triple pattern t and s, p, o are respectively the subject, predicate and object of t. Thus, the expected execution cost for t, $c(t)$, corresponds to the multiplication of the expected cost for the subject $c(s)$, predicate $c(p)$, and object $c(o)$.

4.4.1 Subject Cost Estimation

The subject of a triple pattern can be either a variable or a URI-reference. In the case of a variable, we assign the cost 1.0 since we miss information to make a more precise cost estimation. In the case of a URI-reference, it matches a resource in the model. Thus, the exact number of triples returned by a pattern where the subject is specified by a URI-reference corresponds to the number of predicates defined for the referenced resource. This information would require an index for each resource. Instead, our statistical model estimates the cost of a pattern where the subject is specified with a URI-reference by

$$c(s) = \frac{1}{|R|}$$

where $|R|$ is the total number of resources in our ontology. This results in a constant for the selectivity of subjects in the queried ontology.

4.4.2 Predicate Cost Estimation

The predicate of a triple pattern can be either a variable or a URI-reference. In the case of a variable, we again assign the cost 1.0 as for the subject since we miss information to make a more precise estimation. In the case of a URI-reference, it matches each triple which features the URI as predicate. We estimate the cost for predicate p by

$$c(p) = \frac{|T_p|}{|T|}$$

where $|T_p|$ corresponds to the (exact) number of triples matching predicate p and $|T|$ is the total number of triples. This is the fraction of triples which matches predicate p. Thus, the predicate cost estimation given a URI-reference is exact.

4.4.3 Object Cost Estimation

The triple pattern object can be either a variable or a graph term. In the case of a variable, we assign the cost 1.0 as for subject and predicate. In the case of a term we extract the estimated selectivity from a histogram. The object values domain for predicates is represented by histograms, more precisely equal-width histograms [Piatetsky-Shapiro et al., 1984]. As we will depict later on in this chapter, the range of object values is divided into B equal-width histogram classes where the class height corresponds to the number of objects given a predicate that fall into the class. Hence, for each predicate a histogram of the corresponding object value domain is created. We estimate the object cost $c(o)$ by

$$c(o) = \begin{cases} c(p, o_c), & \text{if p is bound;} \\ \sum_{p_i \in \mathbb{P}} c(p_i, o_c), & \text{otherwise.} \end{cases}$$

where $c(p, o_c) = \frac{h_c(p,o_c)}{|T_p|}$, i.e., the frequency of o_c normalized by the number of triple matching p. In the case a predicate is not bound, the histogram of each predicate in the model is considered for the object cost estimation. Histograms represent the object values domain of a specific predicate. Thus, to address a histogram the predicate URI-reference is required. Please refer to Section 4.9 for a discussion about a more general case where only the object is specified.

4.5 Examples

In order to illustrate the cost model described in previous section, we illustrate some examples for which we calculate the estimated cost. The examples are based on a sample ontology O. We perform an index over O to gather the required statistics. O contains 1'317 triples and an average of 11.52 predicates for each resource. Moreover, the predicate RDFS:label[1] appears in 114 triples. In O, RDFS:label is used to describe the title of publications. The title 'XQuery: A Query Language for XML' [Chamberlin et al., 2001] falls into a histogram class of height 17.

Based on these statistics, we are now able to estimate the cost of triple patterns. Table 4.1 illustrates some examples.

	t	c(s)	c(p)	c(o)	c(t)
1	?s ?p ?o	1.0	1.0	1.0	1.0
2	:s ?p ?o	0.008747	1.0	1.0	0.008747
3	?s rdfs:label ?o	1.0	0.0865604	1.0	0.0865604
4	:s rdfs:label ?o	0.008747	0.0865604	1.0	0.0007571
5	?s rdfs:label "XQuery: A Query ..."	1.0	0.0865604	0.0129081	0.0011173
6	:s rdfs:label "XQuery: A Query ..."	0.008747	0.0865604	0.0129081	0.0000097

Table 4.1: Triple Pattern Cost Estimation

The examples show the execution cost variation according to a variable or specific subject, predicate and object. At this point, an interesting question may be whether the cost estimation behaves naturally in respect to the ontology. Consider the examples listed in Table 4.1. It is straightforward that the triple pattern (5) potentially matches more triples than the pattern (6). Thus, the smaller cost for the pattern (6) is justified, but perhaps the cost function does not behave naturally in respect to the ontology. For example, in case the ontology contains only a single publication entitled 'XQuery: A Query Language for XML', the effective costs for the triple pattern (5) and (6) are the same. Our cost function returns a different estimation for the triple pattern (5) and (6), but this is admissible, since the selectivity of the triple patter (6) is a lower bound for the selectivity of the pattern (5).

4.6 Statistical Model

The required statistical information about the underlying graph model is (usually) previously extracted by an indexing process. In a special case the statistics may be extracted during query execution, namely when the unoptimized query execution is expected to be longer than the indexing process required to gather the statistical information. During our evaluation we experienced that this special case is more frequent than expected.

[1] http://www.w3.org/2000/01/rdf-schema#label

Listing 4.1: Statistical Ontology Model: Average Number of Predicates and Number of Triples

```
<?xml version="1.0"?>

<rdf:RDF
    xmlns:j.0="http://localhost/"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">

<rdf:Description rdf:about="http://localhost/statistics">
    <j.0:avgNrOfPredicates>3.3333333333333335</j.0:avgNrOfPredicates>
    <j.0:nrOfTriples>20</j.0:nrOfTriples>
    <j.0:predicates rdf:resource="http://localhost/predicates/#P1"/>
</rdf:Description>

<rdf:Description rdf:about="http://localhost/predicates/#P1">
    <rdf:_7 rdf:resource="http://localhost/person/lastname"/>
    <rdf:_6 rdf:resource="http://localhost/person/firstname"/>
    <rdf:_5 rdf:resource="http://localhost/address/city"/>
    <rdf:_4 rdf:resource="http://localhost/person/age"/>
    <rdf:_3 rdf:resource="http://localhost/address/zip"/>
    <rdf:_2 rdf:resource="http://localhost/person/address"/>
    <rdf:_1 rdf:resource="http://localhost/address/street"/>
    <rdf:type rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Seq"/>
</rdf:Description>

</rdf:RDF>
```

In either case, the implementation of our ARQ optimizer requires a statistical model for the triple pattern cost estimation. We implement a Jena indexer which creates an ontology model containing all required statistical information. The model is serialized to a RDF/XML representation which can be loaded into a query execution environment[2].

Basically the statistical model is represented as a graph (Listings 4.1, 4.2 and 4.3). It contains a `statistics` resource[3] with both predicates `avgNrOfPredicates`[4] and `nrOfTriples`[5]. Furthermore, the model contains a `predicates` resources[6] with a RDF sequence[7] which lists all distinct predicates contained in the indexed ontology. Each predicate in the list is described on its part by a RDF resource with a predicate `frequency`[8] for the absolute number of occurrences the predicate appears in triples and a predicate `histogram`[9] which is a reference to the histogram representation for the object values of the corresponding predicate (Listing 4.2). A histogram representation is again a resource which contains a RDF sequence of histogram classes (Listing 4.3). The histogram classes are URI-references to a RDF resource (Listing 4.3) with a predicate `label`[10]

[2]Please note that for performance reasons it is advisable to load the statistical model only once and to keep it in memory. This may be achieved by some query execution environment, either an own application or for example Joseki, http://www.joseki.org/

[3]http://localhost/statistics

[4]http://localhost/avgNrOfPredicates

[5]http://localhost/nrOfTriples

[6]http://localhost/predicates

[7]http://www.w3.org/TR/rdf-primer/

[8]http://localhost/frequency

[9]http://localhost/histogram

[10]http://localhost/histogram/class/label

Listing 4.2: Statistics: Predicate Resource

```
<?xml version="1.0"?>

<rdf:RDF
    xmlns:j.0="http://localhost/"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">

<rdf:Description rdf:about="http://localhost/person/lastname">
    <j.0:histogram rdf:resource="http://localhost/histogram/#H7"/>
    <j.0:frequency>3</j.0:frequency>
</rdf:Description>

</rdf:RDF>
```

which specifies the histogram class lower bound and a predicate `items`[11] which describes the histogram class height, i.e., the number of elements falling into the class.

To give an example, we show some RDF/XML extracts from a statistical model. Listing 4.1 displays the `statistics` resource. The average number of predicates corresponds to 3.33, whereas the total number of triples found in the ontology is 20. In addition, the `predicates` resource is displayed too. This resource features a RDF list of all distinct resources that are identified during indexing.

Listing 4.2 displays an example `predicate` resource which describes the occurrences of the predicate within the ontology (3) and the URI-reference for the corresponding histogram. Listing 4.3 shows a histogram representation which is a RDF sequence of classes for the histogram (H7). In Listing 4.3 both `histogram` and `class` resources are described[12]. Figure 4.1 displays a histogram example used to model the object domain values for a specific predicate.

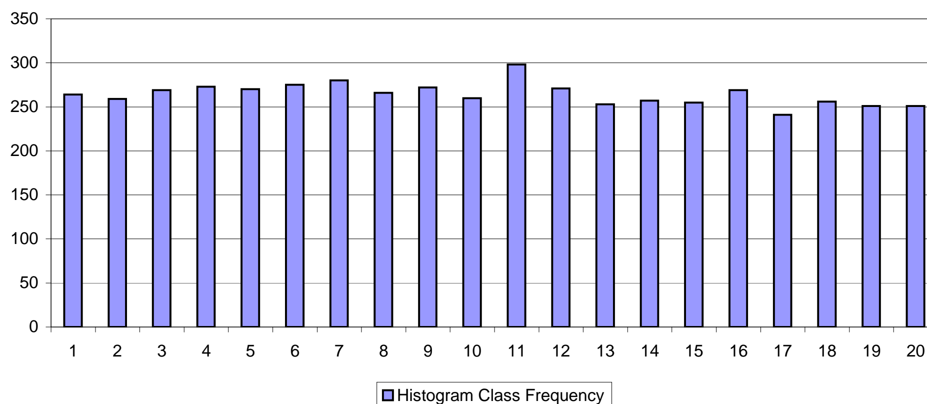

Figure 4.1: Example of Histogram Classes

Figure 4.2 displays the RDFS which defines the statistical ontology used to describe the statistical information required for triple pattern selectivity estimation. In addition, the example individual listed in 4.1, 4.2 and 4.3 is displayed as instance data.

[11] http://localhost/histogram/class/items

[12] Please remark the label. In order to abstract from the specific object data type, we use the object hash code, to classify the item inside of the histogram.

Listing 4.3: Statistics: Histogram Representation

```
<?xml version="1.0"?>

<rdf:RDF
    xmlns:j.1="http://localhost/histogram/class/"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">

<rdf:Description rdf:about="http://localhost/histogram/#H7">
    <rdf:_6 rdf:resource="http://localhost/histogram/class/#C39"/>
    <rdf:_5 rdf:resource="http://localhost/histogram/class/#C38"/>
    <rdf:_4 rdf:resource="http://localhost/histogram/class/#C37"/>
    <rdf:_3 rdf:resource="http://localhost/histogram/class/#C36"/>
    <rdf:_2 rdf:resource="http://localhost/histogram/class/#C35"/>
    <rdf:_1 rdf:resource="http://localhost/histogram/class/#C34"/>
    <rdf:type rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Seq"/>
</rdf:Description>

<rdf:Description rdf:about="http://localhost/histogram/class/#C34">
    <j.1:label>4695060.980000019</j.1:label>
    <j.1:items>6</j.1:items>
</rdf:Description>

</rdf:RDF>
```

4.7 SPARQL Optimization

This section focuses on SPARQL optimization. We describe our general optimization rules considered in the framework. As for the imprecise extension iSPARQL (Chapter 3), our proposed optimization architecture may be viewed as a framework too, i.e., a general infrastructure for the implementation of optimization rules.

Rules are organized in specific packages and embody two stages: a prepare and a transform stage. During a prepare stage we usually gather information about the query required during a transform stage in order to rewrite the query according to a specific rule. For example, during a prepare stage we may create statistics about the variables used in query to assure a correct rewriting of variables while processing FILTER rewriting rule.

The execution order of rules is relevant for multiple reasons. First, the triple pattern execution cost may be different when some variable is substituted during FILTER rewriting. Moreover, the execution performance of some transformation rules may be enhanced when dispensable elements in a query are removed first. Finally, for iSPARQL queries we need to identify imprecise statements and reorder them in a correct manner. We recommend a rule execution ordering that reflects the sequence of the following sections.

4.7.1 Remove Dispensable Pattern

The purpose of this rule is to check whether the query contains triple patterns that are not expected to be meaningful. A triple pattern may not be meaningful, if the variables for subject, predicate, and object are never read in the query. Yet, this is a little tricky. In fact, although variables of some triple pattern may not be used in the query, it may modify the result set. Thus,

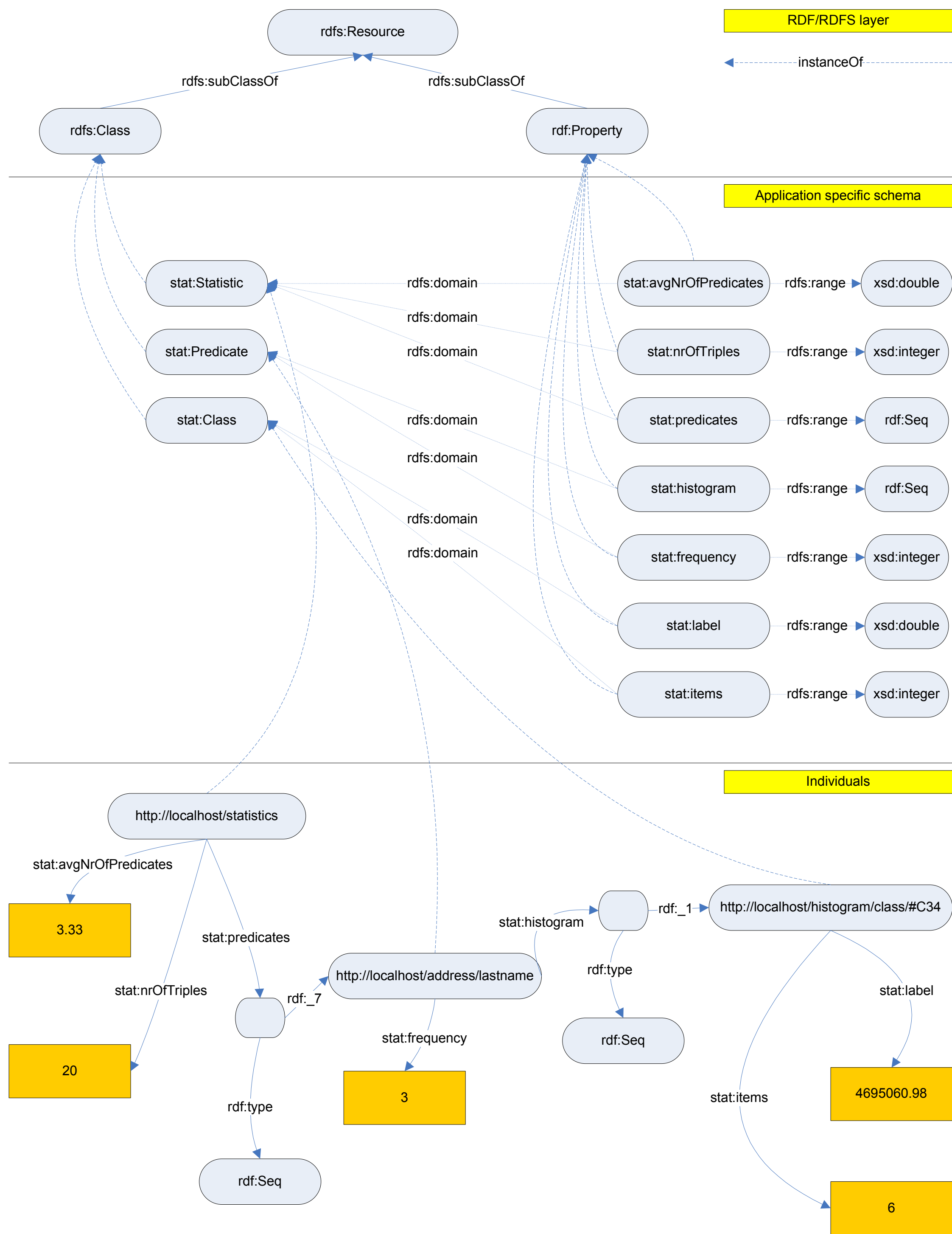

Figure 4.2: RDF/RDFS Model of Statistics Ontology

Listing 4.4: Example: Rewrite Filter Variables

```
PREFIX  person:   <http://person/>

SELECT ?person ?firstname
WHERE {
  ?person person:firstname ?firstname .
  ?person person:lastname ?lastname .
  ?person person:age ?age .

  FILTER (?firstname = "Donald" && ?lastname = "Chamberlin" && ?age > 30)
}
```

although at first glance the elimination of a triple pattern may not affect the query semantics, this assumption may be wrong. Even in the case we define a triple pattern that matches the entire graph, i.e., ?s ?o ?p with variables that are not read somewhere in query, the elimination of this pattern modifies the result set. Even though the pattern does not constrain the results and none of the variables is read in query, the triple pattern does affect the result set size. Thus, if a query should always return the correct result set, regardless if they are meaningful or not, we may consider avoiding the execution of this rule.

4.7.2 Reorder Imprecise

Although this is not properly an optimization rule, it alleviates a drawback of our iSPARQL framework and is required for a correct execution of subsequent rules, even though it is only relevant for the optimization of iSPARQL queries.

As explained in Chapter 3, a major problem of iSPARQL is the required ordering of imprecise statements. In order to alleviate this drawback which is contrary to the logic of SPARQL where ordering is not relevant for a correct execution, this rule allows to specify a casual order of imprecise statements in an iSPARQL query. The rule rewrites the query in order that it can be executed by ARQ.

In fact, the rule does have an optimization function, but it is not inherent to SPARQL optimization. We discuss the optimization achieved by this rule in Section 4.8.

4.7.3 Rewrite Filter Variables

The purpose of this rule is to inspect whether FILTER expressions can be decomposed and variables included in expressions eliminated by substituting the value directly in some triple pattern. There are a couple of issues to consider. First, we can decompose only FILTER expressions that are connected by an AND logical operator. SPARQL triple patterns are joined by a logical AND. Thus, if we substitute a triple pattern variable, we need to make sure that the variables in FILTER expressions are connected by AND. The substitution of variables connected by OR would lead to a different semantic and to a wrong result set. Another important remark is that a variable in some triple pattern cannot be simply substituted, if it is read (referenced) somewhere else in the query. Thus, we need to make sure that substituted variables occur only once in query. Finally, we need to consider the operator used in FILTER expressions for a variable and his value. There is obviously only one case that a variable can be substituted, namely for an equal operator (=).

Listing 4.5: Optimized Example: Rewrite Filter Variables

```
PREFIX  person:  <http://person/>

SELECT ?person ?firstname
WHERE {
  ?person person:firstname "Donald" .
  ?person person:lastname "Chamberlin" .
  ?person person:age ?age .
  ?person person:firstname ?firstname .

  FILTER (?age > 30)
}
```

Listing 4.6: Optimized Example: Move Up Filter

```
PREFIX  person:  <http://person/>

SELECT ?person ?firstname
WHERE {
  ?person person:firstname "Donald" .
  ?person person:lastname "Chamberlin" .
  ?person person:age ?age .
  FILTER (?age > 30)
  ?person person:firstname ?firstname .
}
```

Listing 4.4 shows an example where the optimization rule is applied. Please mark that some of the issues described above need to be considered. The variables defined in FILTER expression are connected by a logical AND. Hence, they may be decomposed. Furthermore, the variable `?firstname` is listed in projection variables. Thus, we cannot simply rewrite the first triple pattern (Listing 4.4).

We have two options at this point. We could leave the triple pattern with the variable and filter the variable in FILTER expression. A more optimized alternative is to rewrite the `?firstname` variable for the first triple pattern (Listing 4.4) and to add the same triple pattern holding the variable as object at the bottom of the query. This way, we very early constrain the intermediate result set by matching only resources with first name 'Donald'. Because the variable `?firstname` is specified in projection we need to add a triple pattern containing the variable as object. Further, we need to pay attention to the variable `?age` defined in FILTER expression. Age is of type Integer and the operator used is '>'. Thus, we cannot rewrite the `?age` variable specified in the pattern.

The optimized query is displayed in Listing 4.5. Please note that `?lastname` is the only variable which could be rewritten without any further modification.

4.7.4 Move Up Filter

The purpose of this rule is to decompose FILTER expressions and execute them as early as possible in query, i.e., after the first variable encounter in the query. We need to make similar considerations as for the rule described above (Rewrite Filter Variables). In fact, we can decompose FILTER expressions only if the FILTER elements are connected by a logical AND.

As an example, we take the query specified in Listing 4.5. There is a FILTER expression defined at the bottom of the query. We can move the FILTER expression closer to the `?age` variable defined as object in the pattern. The optimized version of the query is listed in 4.6.

4.7.5 Reorder by Selectivity

After defining some trivial transformations this last rule is a bit more sophisticated. While the other rules may also be manually accomplished by someone that is sensitized to optimization issues, this last rule requires statistical information about the underlying ontology. Thus, it can hardly be executed manually.

The purpose of this rule is to reorder triple patterns according to their selectivity. Please refer to the explications of triple pattern selectivity estimation used in our optimization approach in Section 4.3.

The rule computes the expected execution cost for each triple pattern using our cost function (Section 4.3). This is done in the prepare stage of the rule. During the transform stage triple patterns are reordered according to their costs in increasing order (i.e., increasing selectivity). Thus, triple patterns that potentially yield smaller intermediate result sets are executed first. As we show in Chapter 5 this transformation is very fundamental and yields significant iSPARQL optimization.

4.8 Imprecise SPARQL Optimization

Our proposed optimization framework considers besides specific SPARQL optimization techniques iSPARQL optimization too. Of course, SPARQL optimization rules are fundamental for iSPARQL. Thus, the rules discussed in Section 4.7 are valid and relevant for iSPARQL too. Moreover, for iSPARQL we discuss some rules and techniques that may result in further optimization. This section addresses specific iSPARQL optimization techniques.

4.8.1 Reorder Imprecise

We discussed this rule previously in Section 4.7 where the focus was set on a correct reordering of imprecise statements. Moreover, this rule executes a specific iSPARQL optimization.

In fact, the rule can be applied for iSPARQL strategies which define a threshold. They potentially constrain the values in the final result set, because of the threshold. We apply our general optimization idea of executing first operations which result in smaller intermediate sets also for this rule. The more the results are constrained by a high threshold, the less matches potentially lead to final results. Thus, we rewrite the query in order to execute iSPARQL strategies ordered by decreasing threshold.

4.8.2 Similarity Index

A similarity index is basically a cache for strategy similarity values. It allows a lookup for a similarity value during iSPARQL query execution. The cache may be a table holding combinations of resource URI-references and the corresponding similarity value which is calculated for each similarity strategy. The high maintenance effort required for a similarity index is the major problem of this optimization technique. Yet, the achieved performance optimization is considerably (Chapter 5). This is true especially for complex strategies, i.e., strategies based on information content or

edge counting similarity measures. Anyway, the time required to calculate the complete index makes it difficult to provide a useful index, especially for massive ontologies.

Nevertheless, for stable ontologies where resources are not modified, deleted or added, a similarity index may be a very useful optimization technique. Unstable ontologies are more problematic. Because modifications to a single ontology resource (especially for OWL or RDFS ontologies where concepts are related in a taxonomy or specific tree structure) may affect other resources too, simply re-calculate the similarities of the modified resource is not sufficient. The similarities of affected resources have to be re-calculated too. Not only it is algorithmically challenging to identify resources that are affected by some modification, but the time required to compute the similarities of affected resources obviously grows.

4.8.3 Avoid Execution of Complex Measures

This optimization technique is a recommendation which is not globally implemented in our optimization framework. It is applicable while developing similarity strategies or designing iSPARQL queries. The aim is to avoid executing complex similarity measures when the result can be predicted by the similarity computation of a more efficient measure. For example, we may consider to execute a fast Levenshtein measure over URI-references to first verify if we face the same resource. For equal resources we can avoid to calculate the complex Lin similarity because the result of Levenshtein is already sufficient[13].

Constraining the resources that need to be compared for similarity using a threshold is another practice. For example, we may consider to add a Levenshtein strategy which constrains the resulting resources by setting a high threshold. Although we may be interested in a semantic similarity of concepts, we can constrain with Levenshtein the number of resources that need to be compared for semantic similarity (e.g., using Lin measure). Thus, we filter the subset of resources we are interested in.

4.8.4 Aggregation Optimization

Some aggregation methods fulfill properties that can be exploited while similarity values are sequentially computed and subsequently aggregated. For example, if similarity values are expected to be in $[0, 1]$ and a max or a noisy-or aggregation method is used we may stop computing the similarity as soon as we get a 1.0 similarity value since we are able to return 1.0 as final similarity. Similarly, for a min, noisy-and or geometric aggregation method we may stop as soon as we get a 0.0 similarity value because we can return 0.0 as final similarity. This optimization should be considered while developing similarity strategies.

4.9 Some Final Thoughts: Pattern Dependency

Pattern dependency is an important consideration. In fact, triple patterns and their elements are dependent entities. First, there is a dependency between a subject, predicate, and object, thus, a triple pattern internal dependency. Further, there is a dependency between triple patterns, thus, a triple pattern external dependency. While for simple queries with a few patterns such deliberations may be less important (or in practice even irrelevant) for more complex queries pattern dependency becomes relevant.

[13]By the way, the Java equals method for strings leads to the same result and is even faster.

4.9.1 S/P/O Dependency

Subject, predicate, and object (S/P/O) dependency corresponds to triple pattern internal dependency, i.e., the interdependency of subject, predicate and object. In this section, we describe the interdependency by considering the combination of triple pattern states. Each triple pattern element (S/P/O) may be either bound or unbound. A triple pattern element is unbound only in the case of an unbound variable. However, a triple pattern element is bound in the case of a bound variable or an explicitly specified value. A variable v is bound if v appears previously in query.

Consider $b(e)$ to be a bound element and $u(e)$ an unbound element. For example, $b(s)$ corresponds to a bound subject. This leads to 8 different triple pattern states, namely the following:

- $u(s)$, $u(p)$, $u(o)$: This triple pattern matches every triple in the underlying graph model. We know nothing about the elements of the pattern. Thus, it is reasonable to assign the highest selectivity to the pattern.
- $b(s)$, $u(p)$, $u(o)$: The subject is bound either by a specified URI-reference or a previously bound variable. In both cases, the selectivity of the pattern is only constrained by the subject and is quantified by the number of predicates for the resource referenced by the URI.
- $u(s)$, $b(p)$, $u(o)$: The predicate is bound either by a URI-reference or a previously bound variable. The selectivity is only constrained by the predicate and is quantified by the number of occurrences returned for the predicate referenced by the URI.
- $u(s)$, $u(p)$, $b(o)$: Because of the potentially unlimited domain of values for objects, it is useful to describe the domain values as statistical distribution. Object domain values are characterized in respect to a specific predicate. If the predicate is unbound the object selectivity is approximated by aggregating the estimations of multiple distributions, i.e., histograms.
- $b(s)$, $b(p)$, $u(o)$ and $b(s)$, $u(p)$, $b(o)$ and $u(s)$, $b(p)$, $b(o)$ and $b(s)$, $b(p)$, $b(o)$: This are combinations of basic cases and we believe that multiplying the single pattern selectivity is a reasonable approximation to get a correct ordering for the triple pattern selectivity.

There is another case which should be considered too. In SPARQL we can specify FILTER expressions for object values which constrain the values by a relation, e.g., $>$, $<$, $<=$ or $>=$. The selectivity of a triple pattern is affected by a possible FILTER constrain. In other words, it is reasonable to consider such FILTER expressions for the pattern selectivity estimation. For example, the following triple pattern

?s :p ?o FILTER (?o = 30)

features potentially a smaller selectivity than

?s :p ?o FILTER (?o >= 30)

Generally we may state that the selectivity of a pattern is affected to a great extent by FILTER expressions. Thus, they should be considered on triple pattern selectivity estimation. The selectivity of the first pattern above is estimated correctly because our framework rewrites FILTER expressions for elements constrained by an equal operator ($=$). However, our framework does not consider the FILTER expression of the second pattern above although the statistical model supports this type of operator in a straightforward manner, i.e., by considering all histogram classes of elements $>= 30$. The selectivity evaluation of special operators such as regular expressions is another interesting case which is not further discussed in this thesis.

4.9.2 Triple Pattern Dependency

Triple pattern dependency considers the interdependency of triple patterns. Triple patterns can be joined by a shared variable. This dependency of two or more patterns affects the selectivity of the involved patterns. Again, we list the possible join variable combinations of two patterns to describe the selectivity behavior. Potentially, we have a total of 9 possible S/P/O joins for two patterns. We believe that a subset of them is more relevant in practice, namely joins of both subjects and joins of subject and object (or object and subject).

- Join of subjects: The selectivity of the joined triple pattern (i.e., the pattern which is joined) is directly affected by the selectivity of the joining triple pattern (i.e., the pattern which joins). The selectivity of predicate and object determines which pattern should be executed first. However, the selectivity of the joined pattern should be considered in dependency to the selectivity of the joining pattern. Consider the example below. We assign the highest cost (1.0) to the subject variable `?s` of the first triple pattern (tp1). Instead of assigning the same cost to the subject variable `?s` of the second triple pattern (tp2), which would be a too rough estimation, we need to consider the cost in dependency to the first triple pattern. This may be modeled by assigning the cost for the first pattern, c(tp1), to the cost of the subject variable `?s` of the second triple pattern. Thus,

  tp1 := ?s :p1 ″o1″
  tp2 := ?s :p2 ″o2″
  c(tp1) = 1.0 * c(:p1) * c(:p1, ″o1″)
  c(tp2) = c(tp1) * 1.0 * c(:p2) * c(:p2, ″o2″) = c(tp1) * c(:p2) * c(:p2, ″o2″)

- Join of subject and object (vice versa): The considerations we made above for joining subjects may be applied as well to this case. Thus, we adopt the same strategy of multiplying the cost of the joining triple pattern to the join variable. Please note that this model has a cascading behavior. If a third pattern is joined we simply assign the cost of the second joining triple pattern to the third variable.

  tp1 := ?s1 :p1 ″o1″
  tp2 := :s2 :p2 ?s1
  c(tp1) = 1.0 * c(:p1) * c(:p1, ″o1″)
  c(tp2) = c(:s2) * c(:p2) * c(tp1)

  tp1 := :s1 :p1 ?o1
  tp2 := ?o1 :p2 ″o2″
  tp3 := ?o1 :p3 ″o3″
  c(tp1) = c(:s1) * c(:p1) * 1.0
  c(tp2) = c(tp1) * c(:p2) * c(:p2, ″o2″)
  c(tp3) = c(tp2) * c(:p3) * c(:p3, ″o3″)

- Join of predicates, join of objects, and other combinations: The model described for the previous cases may be adopted for other combinations too.

We can state that the selectivity for the joined triple pattern is constrained as a function of the selectivity for the joining pattern. A general way how to model this in our framework is to multiply the variable cost (i.e., 1.0) with the estimated cost of the joining triple pattern.

5 Evaluation

In order to evaluate and validate both frameworks, i.e., the imprecise extension described in Chapter 3 and the optimization framework described in Chapter 4, we build an evaluation environment which allows to evaluate several retrieval tasks for multiple query engines.

This chapter illustrates the selected evaluation method and the results in more details. We first describe the quantitative evaluation approach used to evaluate the SPARQL query execution performance. Afterward, we describe the qualitative evaluation approach used to evaluate the iSPARQL retrieval performance. For each approach we illustrate the methods, datasets, query engines, and retrieval tasks used and we present our findings.

5.1 Quantitative Query Performance Evaluation

The quantitative evaluation focuses on execution performance of SPARQL queries on a sampled dataset. We show how the performance of retrieval tasks scales for multiple query engines. The evaluation is based on a dataset which fits into main memory. Thus, the results presented in this chapter focuses on execution performance evaluation of SPARQL query engines with in-memory models. Although the considerations we made in Chapter 4 about query optimization and the evaluations presented in this chapter are valid also for triple stores, the results and charts presented here are valid for in-memory models only.

We conduct all our experiments on a two processor dual core AMD Opteron 270 2.0 GHz server with 4 GB main memory and two 150 GB 7200rpm disks with a 32 bit version of Fedora Core 5 as operating system.

5.1.1 Query Engines

We evaluate the SPARQL query performance on different query engines, namely ARQ[1], Sesame[2] and KAON2[3]. Our ARQ optimization framework is used as reference for comparison to other engines. To distinguish the optimization framework from ARQ we name the optimized ARQ engine *OptARQ*. ARQ is a query engine for Jena (Chapter 3). Sesame is a RDF database with support for RDF-Schema inferencing and querying. Sesame was originally developed by Aduna[4]

[1]http://jena.sourceforge.net/ARQ/
[2]http://www.openrdf.org/
[3]http://kaon2.semanticweb.org/
[4]http://www.aduna-software.com/

for an EU research project and is still maintained by Aduna in collaboration with the community and NLnet Foundation[5]. Sesame supports an own query language called SeRQL[6] and provides a SPARQL engine which is developed third party by Ryan Levering[7]. KAON2[8] is an infrastructure for managing OWL-DL, SWRL, and F-Logic ontologies.

5.1.2 Dataset

In order to investigate the scale performance of retrieval tasks for different query engines, we conduct a sampling of the dataset (SwetoDblp). The sampling procedure is described later on in this section. In order to get useful results we consider two important characteristics the samples must accomplish. First, the samples should grow linearly. Secondly, the resulting set size for a retrieval task needs to grow linearly too. We must assure that the results for a specific retrieval task are not all contained in a couple samples since this would lead to erroneous results.

SwetoDblp

SwetoDblp[9] is a RDF representation of the DBLP[10] publication database and is published by the Large Scale Distributed Information Systems (LSDIS) lab, University of Georgia, USA. SwetoDblp is a spin-off of the Semantic Web Technology Evaluation Ontology (SWETO)[11] and is intended as an infrastructure for testing the scalability of new software. The schema-vocabulary of SwetoDblp aggregates concepts from FOAF[12], Dublin Core[13] and OPUS (specific to the LSDIS library). SwetoDblp contains approximatively 1.3 million resources with a size of 787 MB (November 2006). The considerable size of the ontology allows an extensive query execution performance evaluation.

SwetoDblp Sampling

In order to evaluate the scale performance of multiple retrieval tasks for different query engines, we create a set of samples of the full SwetoDblp ontology. The sampling growth is set to approximatively 10%. Thus, we sample the complete ontology in 10 samples. Table 5.1 illustrates the sample sizes in mega bytes including the number of triples and resources. Moreover we list the resulting set size in number of resources for both retrieval tasks (RT A and RT B). Please refer to Section 5.1.3 for more details about the retrieval tasks.

The sample development is challenging for a couple of reasons. The native RDF SwetoDblp ontology in not compatible with KAON2, since KAON2 is an infrastructure for managing OWL-DL ontologies. To turn SwetoDblp into a compatible ontology we need to apply some transformation. First, we must explicitly state the ontology type by adding the owl:Ontology tag[14]. Secondly, RDF sequences (rdf:Seq) need to be rewritten since they are undefined for OWL-DL ontologies.

[5] http://www.nlnet.nl/
[6] http://www.openrdf.org/doc/sesame/users/ch06.html
[7] http://ryan.levering.name
[8] http://kaon2.semanticweb.org
[9] http://lsdis.cs.uga.edu/projects/semdis/swetodblp/
[10] http://dblp.uni-trier.de/
[11] http://lsdis.cs.uga.edu/projects/semdis/sweto/
[12] http://xmlns.com/foaf/0.1/
[13] http://dublincore.org/
[14] Please note that this tag is required for KAON2. Removing this tag, leads to a weird behavior. KAON2 manages a buffer of exact 1 million bytes, used to figure out the ontology type. The buffer is filled until it is full or the ontology type is found. If the ontology type could not be detected after reading the first 1 million bytes of the ontology, KAON2 throws an I/O exception.

Sample (%)	Size (MB)	Triples	Resources	RT A	RT B
10	78.7	893'965	79'733	2	4
20	157.3	1'787'629	159'467	4	8
30	235.8	2'681'237	239'203	6	12
40	314.6	3'573'684	318'934	8	16
50	393.4	4'468'666	398'665	10	20
60	472.0	5'360'604	478'401	12	24
70	550.6	6'253'930	558'112	14	28
80	629.4	7'145'344	637'852	16	32
90	708.1	8'040'767	717'574	18	36
100	786.5	8'933'272	797'278	20	40

Table 5.1: SwetoDblp Samples: Sample Size, Number of Triples, Number of Resources and Result Set Size for both Retrieval Tasks A and B

Further, the amount of character encoded data of the full SwetoDblp is another challenge. Since we need to redistribute the matching resources for both retrieval tasks linearly over the samples, a careful selection of the sampling method is required. In order to manage the amount of data and to allow flagging matching resources that need to be linearly distributed over the samples, we first import the SwetoDblp resources into a relational database. Once in database we create the required samples.

5.1.3 Retrieval Tasks

Listing 5.1: SwetoDblp Retrieval Task A

```
PREFIX  opus: <http://lsdis.cs.uga.edu/propono#>
PREFIX  rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?label ?author ?volume ?pages ?number
WHERE {
  ?article opus:year ?year .
  ?article opus:publication_authored_by ?author .
  ?article rdfs:label ?label .
  ?article opus:volume ?volume .
  ?article opus:pages ?pages .
  ?article opus:number ?number .
  ?article opus:journal_name ?journal_name .
  FILTER (?year = 2004 && ?journal_name = "VLDB J.")
}
```

We specify two retrieval tasks for SwetoDblp which reflect a common usage of the ontology. The first task (Listing 5.1) focuses on articles published by a journal during a specific year. It extracts all articles published in 2004 by VLDB journal[15]. An article is described by its title, the researchers that authored the publication, the journal volume and number, the publication year

[15]http://www.informatik.uni-trier.de/ ley/db/journals/vldb/index.html

Listing 5.2: SwetoDblp Retrieval Task B

```
PREFIX  opus: <http://lsdis.cs.uga.edu/propono#>
PREFIX  rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?label ?year
WHERE {
  ?author rdfs:label ?name .
  ?article opus:publication_authored_by ?author .
  ?article opus:year ?year .
  ?article rdfs:label ?label .

  FILTER (?name = "Donald D. Chamberlin")
}
```

and the number of pages. The second retrieval task (Listing 5.2) focuses on a specific researcher and extracts all articles authored by him quoting the year and title of the publication.

5.1.4 Optimizations

In order to better understand the evaluation, we first describe how our optimization approach rewrites the input query for both retrieval tasks. The optimized query returned is expected to be optimal according to the statistical selectivity estimation. In addition we shortly describe the optimizations executed by Sesame for the SeRQL query language. SeRQL is a RDF/RDFS query language developed by Aduna[16] as a part of Sesame[17].

Listing 5.3: SwetoDblp Retrieval Task A (optimized query)

```
1  PREFIX  opus: <http://lsdis.cs.uga.edu/propono#>
2  PREFIX  rdfs: <http://www.w3.org/2000/01/rdf-schema#>
3
4  SELECT ?label ?author ?volume ?pages ?number
5  WHERE {
6    ?article opus:journal_name "VLDB J." .
7    ?article opus:year 2004 .
8    ?article opus:number ?number .
9    ?article opus:pages ?pages .
10   ?article opus:volume ?volume .
11   ?article rdfs:label ?label .
12   ?article opus:publication_authored_by ?author .
13 }
```

The query of retrieval task A (Listing 5.1) is optimized by two different optimization rules. First, we rewrite the FILTER expression. During the prepare stage, the optimizer checks for FILTER expressions and whether they can be rewritten. We may rewrite a FILTER expression when its elements are connected by a boolean AND operator and they are compared by equal (=). Since both variables `?year` and `?journal_name` are not read elsewhere in the query, we can just

[16]http://www.aduna-software.com/

[17]http://www.openrdf.org/doc/sesame/users/ch06.html

Listing 5.4: SwetoDblp Retrieval Task A (Sesame SeRQL)

```
SELECT
  Title, Author, Volume, Pages, Number
FROM
  {Article} opus:year {2004};
            opus:journal_name {"VLDB J."};
            opus:publication_authored_by {Author};
            rdfs:label {Title};
            opus:volume {Volume};
            opus:pages {Pages};
            opus:number {Number};
USING NAMESPACE
  opus = <http://lsdis.cs.uga.edu/propono#>,
  rdfs = <http://www.w3.org/2000/01/rdf-schema#>
```

overwrite both object variables in the corresponding triple patterns. Further, we apply the rule 'Reorder by Selectivity'. During prepare stage of the rule, the optimizer calculates the estimated triple pattern execution costs as a function of the estimated selectivity (please refer to Chapter 4 for further details about the optimization technique). We get a set of [0,1]-values which is used in transformation stage to reorder the triple patterns. The optimized query is listed in 5.3. After rewriting the FILTER expression the first triple pattern (line 6) has the smallest selectivity, i.e., lowest cost and is thus placed first. This is reasonable compared to the second triple pattern (line 7) since the articles published by VLDB journal are expected to be less than the articles published in 2004. The following three patterns (lines 8 - 10) are more difficult to anticipate, but it is reasonable that there are more triples matching the predicate `opus:number` than triples matching the second triple pattern (line 7) since not only the articles published in 2004 but every article will match the third pattern (line 8). Because the ontology not only contains articles but other resources too (e.g., master thesis) it is straightforward that the sixth pattern (line 11) is placed after the pattern with the `opus:volume` predicate (line 10). Only article resources match the `opus:number` predicate whereas each resource matches the `rdfs:label` predicate (i.e., the resource title). Last but not least, we may explain why the seventh pattern (line 12) is placed last. Since an article features only one title but it is often written by one or more authors, the last pattern potentially matches more triples. Thus, it is placed to the bottom of the query.

Listing 5.4 presents the optimization executed by Sesame for the SeRQL query language. Sesame applies some general optimization rules. In fact, it rewrites expressions in SeRQL WHERE clause as our optimization framework for FILTER expressions. Further, Sesame reorders SeRQL triple pattern defined in FROM clause according to the number of triple pattern variables. Patterns with more variables are considered to be less specific. Thus, they are executed later in query. This is a more naive approach compared to ours, but it is based on the same idea of reducing the intermediate result set sizes.

By looking at the query (Listing 5.4), we may notice that the triple pattern containing the predicate `opus:journal_name` should be executed first, since the number of articles published by VLDB journal are less than those published in 2004. Another problem of the approach used by Sesame emerges for the following triple patterns where just the predicate is specified (Listing 5.4). Since an article has only one title but mostly several authors, the ordering of the patterns is obviously wrong. This behavior arises because of unavailable statistics about the selectivity of triple patterns. Thus, Sesame is missing fundamental information to optimize the query with a more precise ordering of triple patterns. As we will see later in this section, the more precise

Listing 5.5: SwetoDblp Retrieval Task B (optimized query)

```
1 PREFIX  opus: <http://lsdis.cs.uga.edu/propono#>
2 PREFIX  rdfs: <http://www.w3.org/2000/01/rdf-schema#>
3
4 SELECT  ?label ?year
5 WHERE {
6   ?author rdfs:label "Donald D. Chamberlin" .
7   ?article opus:publication_authored_by ?author .
8   ?article opus:year ?year .
9   ?article rdfs:label ?label .
10 }
```

reordering of our approach makes a significant difference. Yet, the optimization achieved through FILTER rewriting is even more important.

Now we take a look closer to the optimization that can be executed for the query used in retrieval task B (Listing 5.2). The optimized query is listed in 5.5. Again, the FILTER expression can be rewritten. This is the only optimization rule that can be applied. The second triple pattern (line 7) is constrained by the joined variable `?author` which is defined as subject in the first pattern (line 6). It is trivial to state that the first triple pattern (line 6) potentially features the smallest selectivity. For our ontology, the `?author` variable is bound to just one URI-reference, namely the one for Mr. Chamberlin. This constrains the joined triple pattern (line 7). Although the object of the pattern at line 7 is a variable (i.e., `?author`) the selectivity of the object should not be estimated as usual for variables by 1.0. Please refer to Section 4.9 for a detailed description about the behavior of joined variables. The selectivity of articles where Mr. Chamberlin is listed as author is expected to be smaller than the selectivity of articles featuring both predicates `opus:year` and `rdfs:label`. Thus, the pattern with predicate `opus:publication_authored_by` takes the second place (line 7). Similar considerations are applicable to the following patterns (line 8 and 9). Resources featuring the `opus:year` predicate are expected to be less than those with a title (predicate `rdfs:label`). This explains the selected ordering by our optimizer. An interesting variation of retrieval task B (Listing 5.2) is discussed in Section 5.1.5. There, we evaluate the performance when the second triple pattern (line 7, Listing 5.5) is replaced by the third pattern (line 8, Listing 5.5).

The optimizations executed by Sesame for retrieval task B (Listing 5.2) are the same as those executed by our optimization framework. Thus, we expect that the Sesame SeRQL performance for retrieval task B is very similar or even better than OptARQ.

5.1.5 Results

After describing the optimizations that can be performed for both retrieval tasks we now present our evaluation results for both retrieval tasks A (Listing 5.1) and B (Listing 5.2).

Retrieval Task A

Figure 5.1 shows the absolute values for retrieval task A (Listing 5.1). We evaluate the query for ARQ, KAON2 and Sesame, where Sesame is evaluated for both SPARQL and SeRQL query languages. The values are measured for each sample. The approximatively linear behavior (Figure 5.1) is expected because of the linear distribution for the resources which matches the retrieval

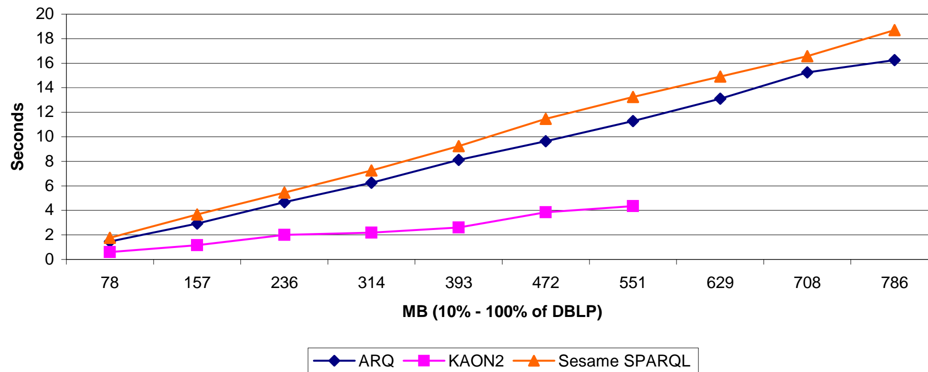

Figure 5.1: Retrieval Task A: Absolute Values

task. Because of main memory limitations of our test server, the measurements for KAON2 stop at 70%. For KAON2, the sample with 551 MB requires over 2.5 GB, which is the maximum amount of memory we can allocate for the Java virtual machine.

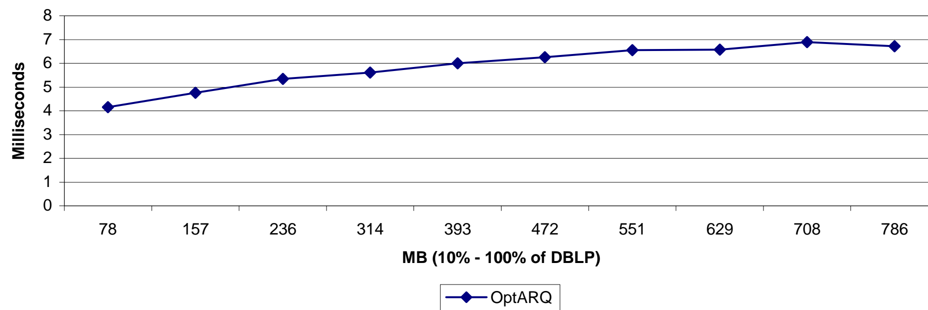

Figure 5.2: Retrieval Task A: OptARQ Absolute Values

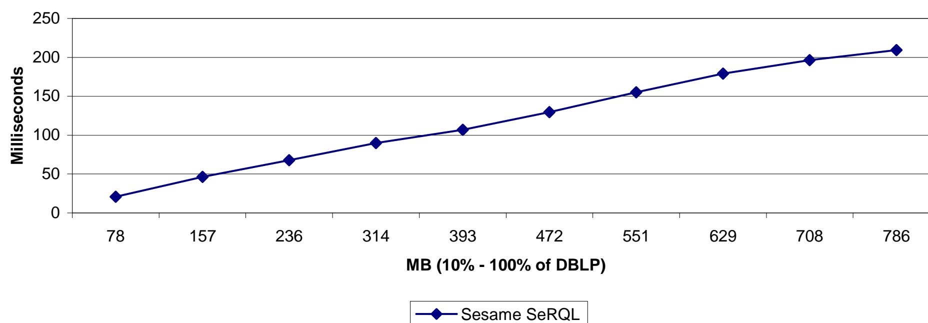

Figure 5.3: Retrieval Task A: Sesame SeRQL Absolute Values

Since the behavior for OptARQ and Sesame SeRQL can be hardly extracted from Figure 5.1, we create two separate charts for both engines. The absolute values for OptARQ are displayed in Figure 5.2 those for Sesame SeRQL in Figure 5.3. The difference between OptARQ and Sesame SeRQL is considerable: at 100% OptARQ is 31.22 times faster than Sesame SeRQL (Figure 5.6).

As we remarked previously in this section, Sesame SeRQL perfoms some optimizations. This explains the difference between Sesame SeRQL and other engines.

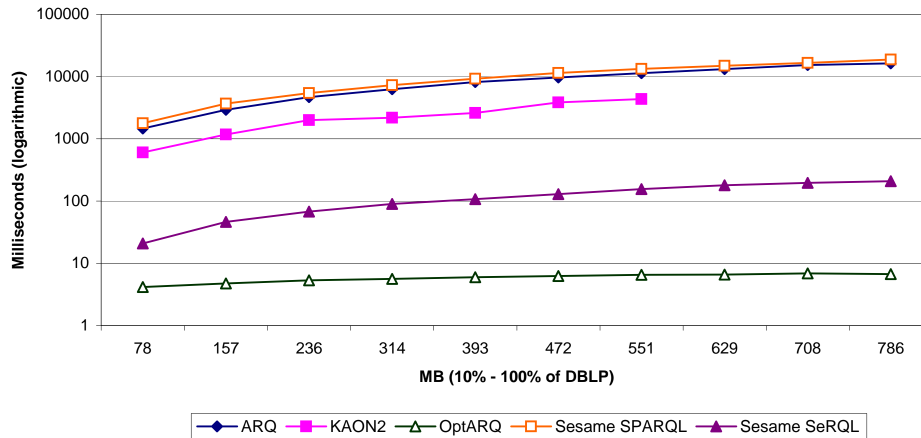

Figure 5.4: Retrieval Task A: Absolute Values (logarithmic scale)

The main difference in optimization between OptARQ and Sesame SeRQL consists in the selected approach for triple pattern reordering. While Sesame SeRQL is using a more simple and general approach, the model based on statistical information about the underlying ontology used in our optimization framework yields a more accurate reordering. Yet, the performance improvement obtained because of FILTER rewriting is considerably more important than the one obtained through a statistical reordering of triple patterns. Nevertheless, our approach is useful since FILTER rewriting is a trivial optimization that can be done manually, whereas our approach for triple pattern reordering can hardly be done manually. Figure 5.4 resumes the evaluated engines on a logarithmic scale.

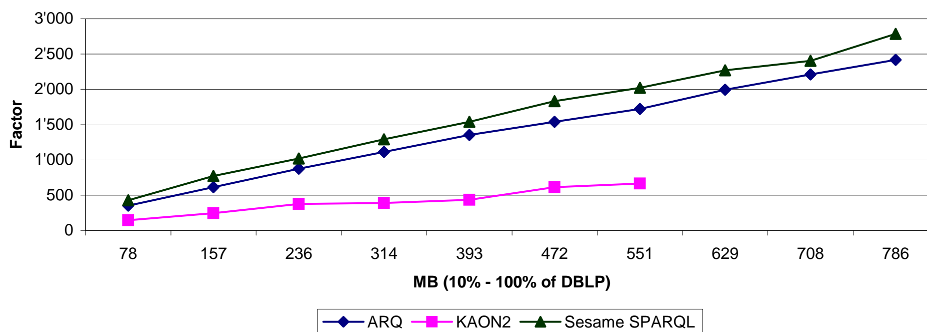

Figure 5.5: Retrieval Task A: OptARQ Normalized Values

In order to quantify the performance improvement for OptARQ compared to the other engines, we create a chart which shows how many times OptARQ is faster compared to the other engines. We call the charts 'OptARQ normalized'. Figure 5.5 shows the factor for ARQ, KAON2 and Sesame SPARQL. Because of a smaller scale, we separately show in Figure 5.6 the factor between OptARQ and Sesame SeRQL. Table 5.2 shows the measured values for each engine. All values are listed in milliseconds and approximated to the second decimal place.

Next we evaluate the time required to load the underlying model in main memory and the

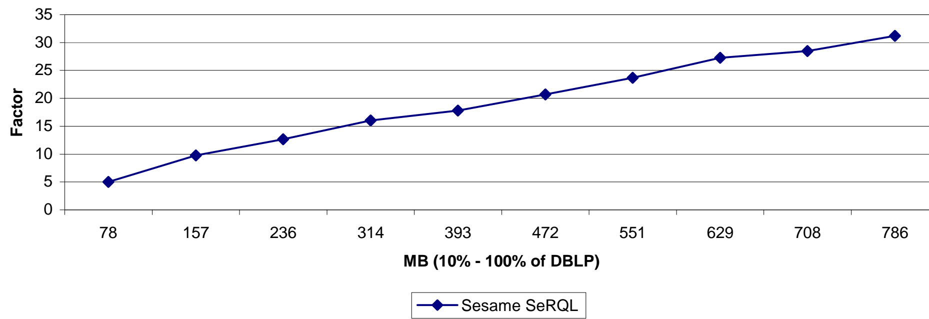

Figure 5.6: Retrieval Task A: Sesame SeRQL OptARQ Normalized Values

Sample (MB)	OptARQ	Sesame SeRQL	KAON2	ARQ	Sesame SPARQL
78	4.16	20.87	606.20	1'459.59	1'771.45
157	4.76	46.39	1'169.09	2'920.37	3'669.46
236	5.34	67.77	1'997.22	4'668.49	5'444.43
314	5.61	89.90	2'182.24	6'252.73	7'251.89
393	6.00	106.87	2'611.55	8'122.98	9'239.51
472	6.26	129.58	3'841.20	9'641.68	11'454.73
551	6.55	155.05	4'355.03	11'291.75	13'241.42
629	6.57	179.12		13'110.01	14'915.84
708	6.89	196.30		15'247.22	16'575.31
786	6.71	209.46		16'243.94	18'697.85

Table 5.2: Retrieval Task A: Absolute Values

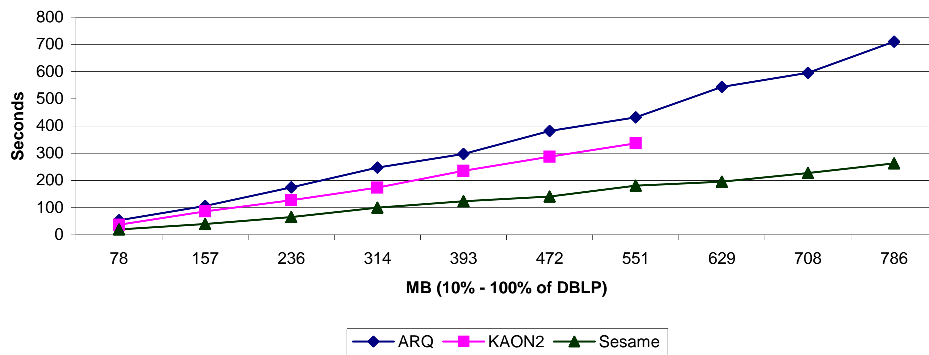

Figure 5.7: Retrieval Task A: Model Load Time

consumption of main memory required to load the model. Figure 5.7 shows the time required to load the model in main memory for each evaluated engine. Sesame definitively outperforms the other engines. Figure 5.8 displays the amount of main memory consumed after loading the underlying model. Again, Sesame outperforms the other engines. Further, the figure shows why our evaluations for KAON2 always stop after sample 70%. KAON2 requires for the 551 MB sample the complete memory that can be allocated to the Java virtual machine on our test server. Table 5.3 lists the absolute values of the time required to load the model whereas table 5.4 shows the memory consumption for each engine.

Sample (MB)	Sesame	KAON2	ARQ
78	19'799	36'857	53'702
157	39'676	85'954	106'369
236	65'477	127'508	174'253
314	99'935	173'339	247'007
393	123'198	235'027	297'636
472	141'119	287'628	382'143
551	180'575	336'154	431'507
629	195'761		543'878
708	226'848		595'898
786	262'796		709'669

Table 5.3: Retrieval Task A: Model Load Time (milliseconds)

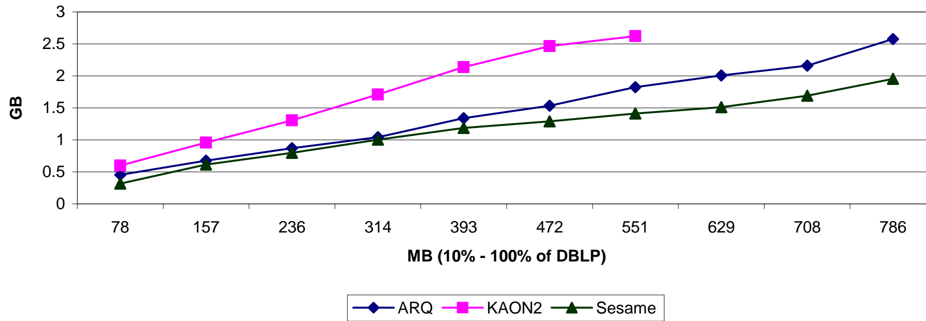

Figure 5.8: Retrieval Task A: Model Memory Consumption

Sample (MB)	Sesame	ARQ	KAON2
78	316'276'736	454'885'376	598'278'144
157	615'907'328	673'710'080	958'726'144
236	798'162'944	869'662'720	1'305'018'368
314	1'003'552'768	1'041'432'576	1'710'555'136
393	1'185'153'024	1'340'604'416	2'137'391'104
472	1'290'141'696	1'535'049'728	2'466'578'432
551	1'413'677'056	1'823'604'736	2'620'260'352
629	1'509'556'224	2'006'253'568	
708	1'691'484'160	2'159'804'416	
786	1'952'448'512	2'574'909'440	

Table 5.4: Retrieval Task A: Model Memory Consumption (bytes)

Finally, we evaluate the performance for each engine when the optimized query is used as input query. Thus, instead to execute the query defined in Listing 5.1 we execute the optimized query defined in Listing 5.3. Figure 5.9 shows the evaluation for the optimized query. Sesame SeRQL outperforms the other engines. The SPARQL implementation in Sesame resulted to be less efficient. Further, the figure shows OptARQ less efficient than ARQ which is straightforward because of the overhead due to the optimizer (which is executed although the query is already optimized). Table 5.5 lists the absolute values measured for this evaluation including those for

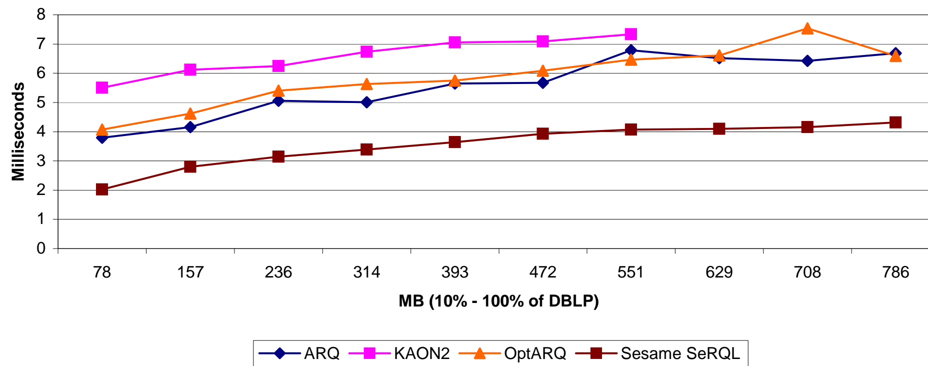

Figure 5.9: Retrieval Task A: Optimized

Sesame SPARQL[18]. All values are listed in milliseconds and approximated to the second decimal place.

Sample (MB)	OptARQ	ARQ	Sesame SeRQL	KAON2	Sesame SPARQL
78	4.07	3.80	2.02	5.50	350.41
157	4.62	4.15	2.79	6.12	1'073.65
236	5.40	5.06	3.15	6.25	1'961.03
314	5.63	5.01	3.39	6.73	2'431.63
393	5.75	5.65	3.64	7.05	3'211.96
472	6.09	5.67	3.93	7.09	4'369.13
551	6.46	6.79	4.07	7.34	4'812.85
629	6.61	6.52	4.10		6'095.34
708	7.53	6.42	4.16		6'956.67
786	6.59	6.68	4.31		8'303.01

Table 5.5: Retrieval Task A: Optimized

To the best of my understanding, I believe that the engine evaluation with the optimized query demonstrates the correctness of our approach and the optimization techniques discussed in this thesis. Since the engines behaves very similar when the optimized query is used, I believe that the performance improvement which is achieved by the optimization techniques is realistic.

Figure 5.10 shows the improvements between the original query for retrieval task A (Listing 5.1) and the corresponding optimized query (Listing 5.3) for each engine on a logarithmic scale. Please note the performance of Sesame SPARQL. Although the optimized query is also performing better for Sesame SPARQL, the performance improvement becomes smaller when the sample size grows.

[18]I believe, Sesame SPARQL performs not better even if the optimized query is used because it does not drop potential results as soon as they don't satisfy the query pattern. I conclude this because of a separate evaluation performed for Sesame SPARQL where the optimized query for retrieval task A listed in 5.3 was executed triple pattern by triple pattern. The evaluation showed that Sesame SPARQL performed similar to KAON2 and better than ARQ and OptARQ for the query containing only the first triple pattern. For the query containing the first two patterns Sesame SPARQL performed already 3.5 times inferior to OptARQ at 10%. With three triple patterns the factor is 12.6 at 10%.

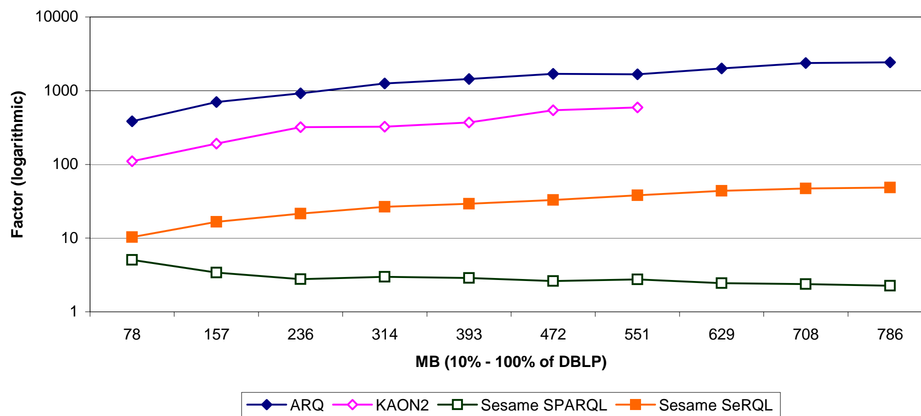

Figure 5.10: Retrieval Task A: Improvements

Retrieval Task B

The evaluation method for retrieval task B is very similar to the one used for retrieval task A. Because of a different query (Listing 5.2) the optimizations performed are different (for a detailed description refer to Section 5.1.4). Thus, the results are expected to be different but the general trend should be comparable.

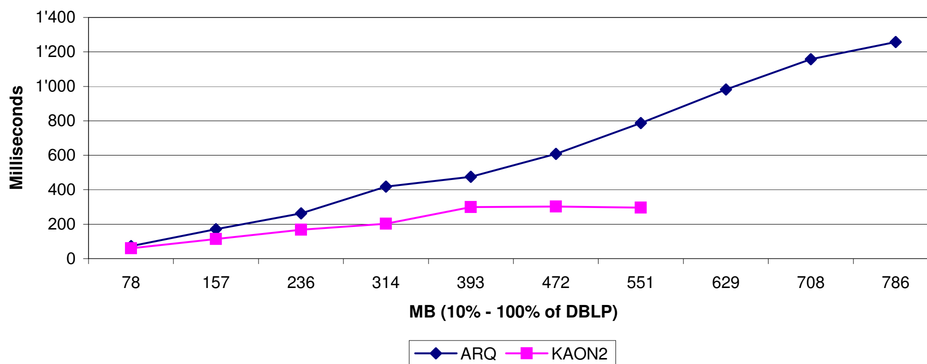

Figure 5.11: Retrieval Task B: Absolute Values

Figure 5.11 displays the absolute values measured for both ARQ and KAON2 engines. Compared to retrieval task A (Figure 5.1) we may mark the different scale (milliseconds toward seconds). Although this difference, the trend for both engines is comparable. Figure 5.12 shows the absolute values for OptARQ and Sesame executed with the SeRQL query language. Compared to retrieval task A (Figures 5.2 and 5.3) we may mark a very similar behavior for OptARQ. However, the difference between OptARQ and Sesame SeRQL in both retrieval tasks is considerable. In retrieval task B the performance of Sesame SeRQL is better compared to OptARQ. However, in retrieval task A, OptARQ was much better than Sesame SeRQL. This is because the optimizations performed by Sesame for SeRQL are alike to those performed for OptARQ. Although our statistical model is more precise in triple pattern selectivity estimation, retrieval task B is an example query where the statistical information does not yield a more optimized query because the input query is already optimal according to the statistical model (after FILTER rewriting).

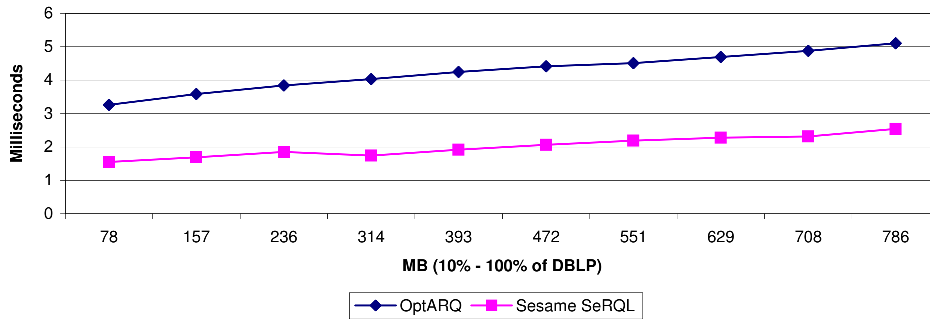

Figure 5.12: Retrieval Task B: OptARQ and Sesame SeRQL Absolute Values

Sample (MB)	OptARQ	ARQ	Sesame SeRQL	KAON2	Sesame SPARQL
78	3.26	72.71	1.55	60.16	434.37
157	3.56	170.79	1.69	114.41	1'195.66
236	3.84	262.87	1.85	167.53	1'710.13
314	4.03	417.59	1.74	202.62	2'710.30
393	4.23	475.57	1.92	298.90	3'961.51
472	4.41	607.71	2.07	302.50	4'504.60
551	4.51	787.42	2.19	306.79	6'050.07
629	4.69	981.24	2.28		6'967.85
708	4.88	1'158.09	2.31		7'855.22
786	5.10	1'257.53	2.54		9'178.89

Table 5.6: Retrieval Task B: Absolute Values

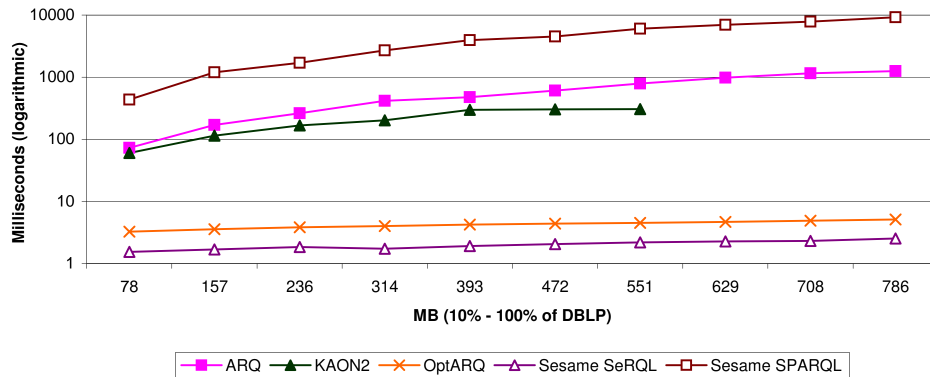

Figure 5.13: Retrieval Task B: Absolute Values (logarithmic scale)

Table 5.6 shows the absolute values measured for retrieval task B. All values are listed in milliseconds and are approximated to the second decimal place. Figure 5.13 resumes the performance for each engine on a logarithmic scale. Figure 5.14 shows the performance for each engine normalized by OptARQ on a logarithmic scale.

Figure 5.15 displays the behavior for each engine when the optimized query (Listing 5.5) is used as input query. Again, the behavior is very similar to the one resulting for retrieval task A (Figure 5.9). Figure 5.16 shows the improvements between the original query for retrieval task B

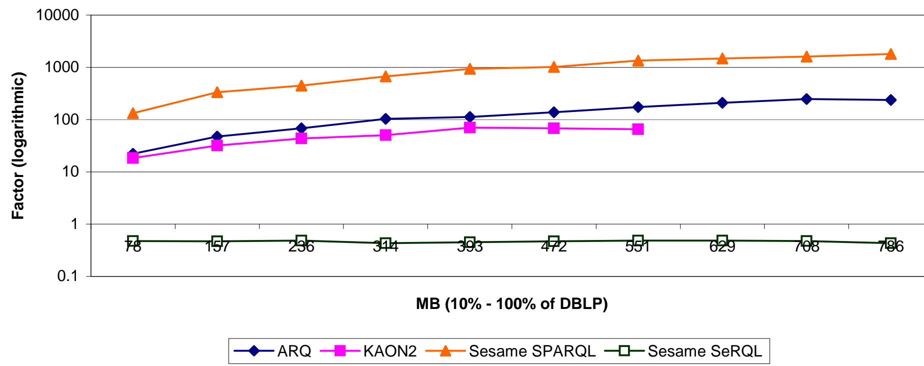

Figure 5.14: Retrieval Task B: Normalized by OptARQ (logarithmic scale)

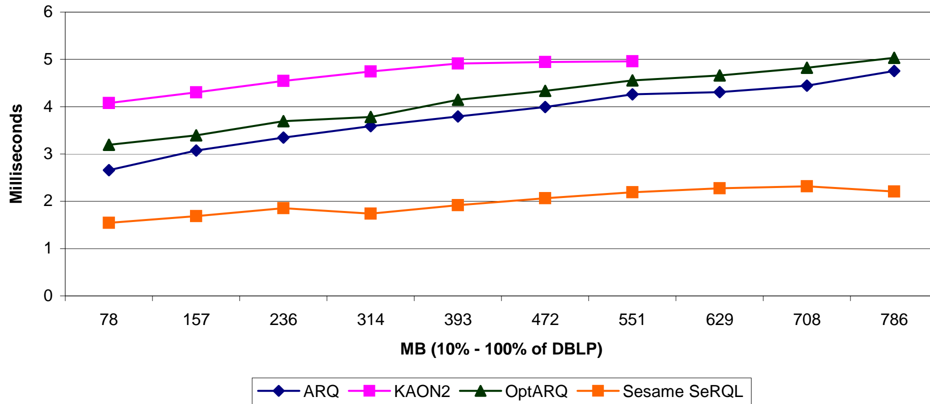

Figure 5.15: Retrieval Task B: Optimized

Sample (MB)	OptARQ	ARQ	Sesame SeRQL	KAON2	Sesame SPARQL
78	3.20	2.66	1.55	4.07	357.96
157	3.39	3.07	1.69	4.30	1'124.64
236	3.69	3.35	1.85	4.54	1'708.89
314	3.78	3.59	1.74	4.74	2'594.69
393	4.14	3.79	1.92	4.91	3'094.30
472	4.33	3.99	2.07	4.95	4'311.97
551	4.55	4.26	2.19	4.98	4'957.93
629	4.66	4.31	2.28		6'035.04
708	4.82	4.44	2.31		6'747.16
786	5.03	4.75	2.35		7'530.76

Table 5.7: Retrieval Task B: Optimized

(Listing 5.2) and the corresponding optimized query (Listing 5.5) for each engine on a logarithmic scale.

Finally, we show both charts for the required time to load the underlying graph model in main memory (Figure 5.17) and the consumed main memory after loading the model (Figure 5.18). As we could expect, the behavior is very similar to retrieval task A (Figure 5.7 and 5.8).

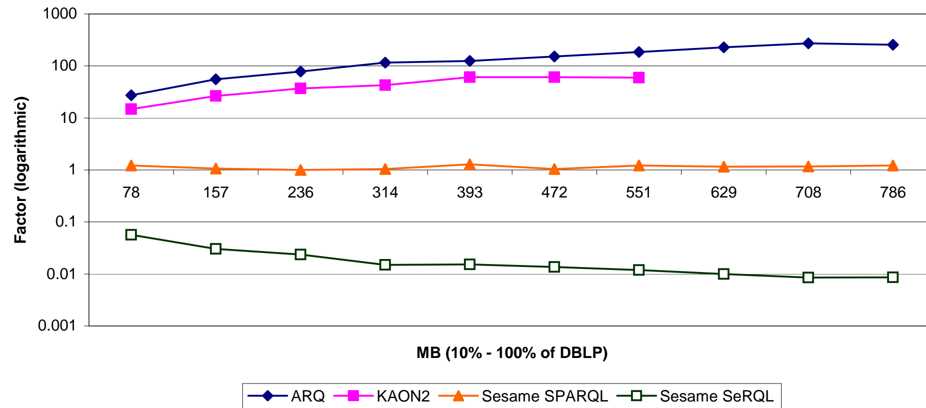

Figure 5.16: Retrieval Task B: Improvements

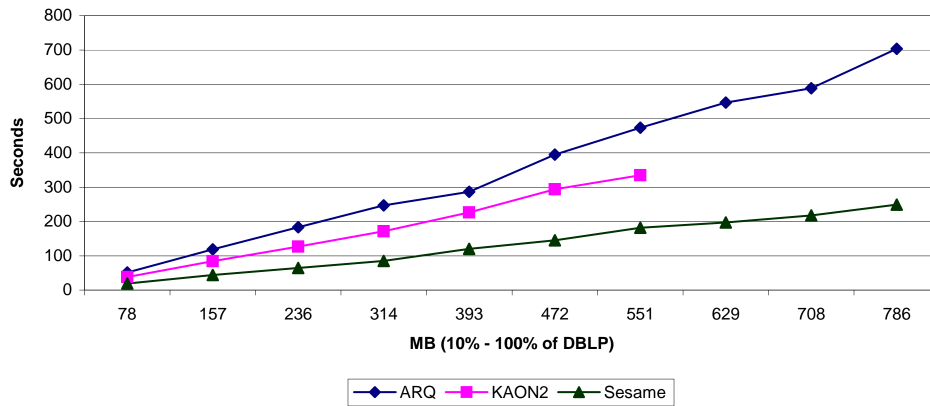

Figure 5.17: Retrieval Task B: Model Load Time

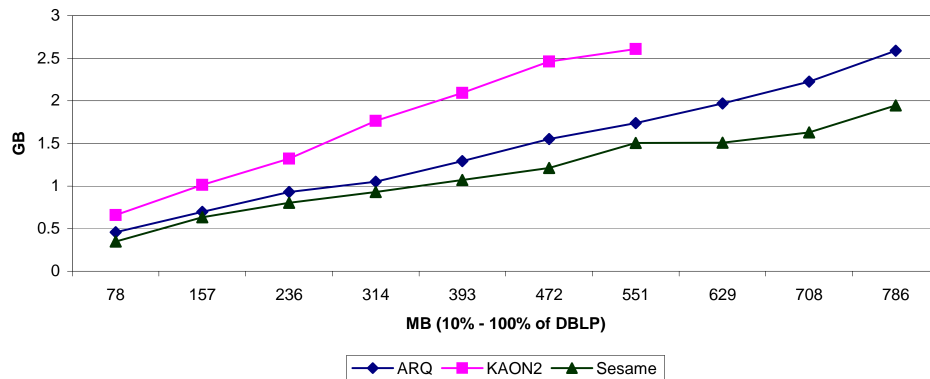

Figure 5.18: Retrieval Task B: Model Memory Consumption

One More Experiment

This experiment is executed only for ARQ and with a different sampling size using retrieval task B (Listing 5.2). Instead to sample SwetoDblp with 10 samples holding 10% - 100% of the data we

create 10 samples holding just 1% - 10%. As we will see, the performance downgrades consistently by simply replacing the joined triple pattern by one or two patterns lower. The relevant queries for this experiment are respectively listed in 5.6 and 5.7

Listing 5.6: SwetoDblp Experiment A

```
PREFIX  opus: <http://lsdis.cs.uga.edu/propono#>
PREFIX  rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?label ?year
WHERE {
  ?author rdfs:label ?name .
  ?article opus:year ?year .
  ?article opus:publication_authored_by ?author .
  ?article rdfs:label ?label .

  FILTER (?name = "Donald D. Chamberlin")
}
```

Listing 5.7: SwetoDblp Experiment B

```
PREFIX  opus: <http://lsdis.cs.uga.edu/propono#>
PREFIX  rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?label ?year
WHERE {
  ?author rdfs:label ?name .
  ?article opus:year ?year .
  ?article rdfs:label ?label .
  ?article opus:publication_authored_by ?author .

  FILTER (?name = "Donald D. Chamberlin")
}
```

Please refer to Listing 5.2 and remark the difference between the queries. As we will demonstrate later in this section, the performance over 10 samples can be approximated by a parabolic function. By means of a trend analysis, we estimate the execution time required to execute both queries on 100% of SwetoDblp.

Figure 5.19 shows the ARQ query execution performance for both queries of this experiment. The curve 'ARQ a' characterizes the performance for the query listed in 5.6 for the samples containing 1% - 10% of the SwetoDblp data. The curve 'ARQ b' describes the performance for the query listed in 5.7. The equations for both trend functions are displayed in Figure 5.19. A parabolic approximation of the measured values seems to be reasonable.

Based on both trend functions extracted from the measurements, we show in Figure 5.20 the expected behavior for the full SwetoDblp ontology, i.e., the performance between 1% and 100% of the dataset. For query 5.7 the expected execution time is approximatively 4 million seconds which corresponds to 46.3 days.

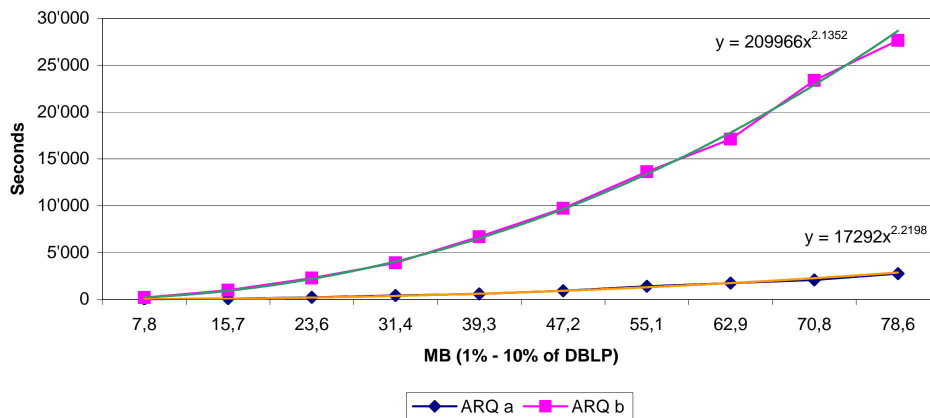

Figure 5.19: Experiment A and B

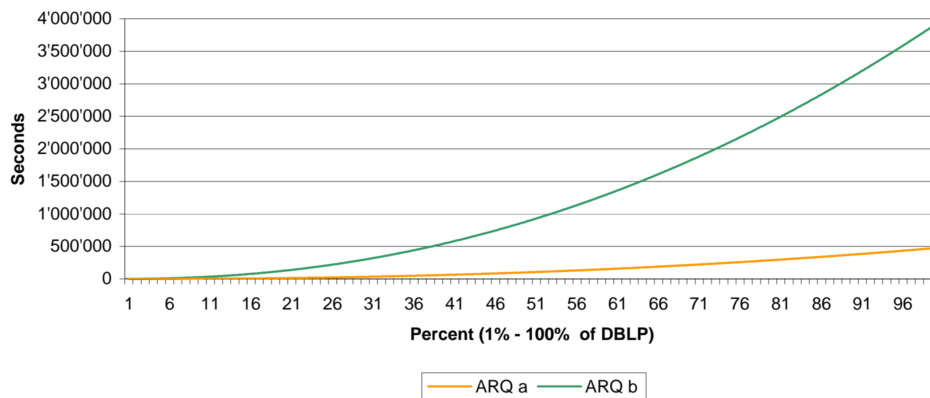

Figure 5.20: Experiment A and B: Trend Function

5.1.6 Similarity Index

Because of the usually very high computational complexity of similarity measures and the consequent poor execution performance, we investigate the usage of a precomputed similarity index which allows a similarity lookup of resources according to a specific similarity strategy. We evaluate the usefulness of a similarity index using a semantic similarity measure, more precisely a similarity strategy based on Lin [Lin, 1998] similarity measure with a retrieval task defined for the Suggested Upper Merged Ontology (SUMO)[19].

In the following, we specify the retrieval task used for the evaluation and the results. The selected task can be formulated as a retrieval of the 20 most similar concepts to 'Wine' in the SUMO. The retrieval task is specified in Listing 5.8 as iSPARQL query. First, each concept defined as OWL class is matched in the underlying ontology. The resources returned are then compared by means of a strategy based on Lin similarity measure. The similarity between each concept and the 'Wine' concept is calculated and ranked in decreasing order of similarity.

Figure 5.21 shows the performance of our retrieval task when the similarity index is deactivated, i.e., the Lin similarity is calculated during query execution. However, Figure 5.22 shows

[19] http://ontology.teknowledge.com/

Listing 5.8: Similarity Index Retrieval Task

```
PREFIX  rdf:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX  owl:    <http://www.w3.org/2002/07/owl#>
PREFIX  sumo:   <http://127.0.0.1/ontology/simplified_sumo.owl#>
PREFIX  isparql:  <java:ch.unizh.ifi.isparql.query.property.>

SELECT ?resource ?similarity
WHERE {
  ?resource rdf:type owl:Class .

  ?strategy isparql:name "SUMOLinResCmp" .
  ?strategy isparql:arguments (sumo:Wine ?resource) .
  ?strategy isparql:similarity ?similarity
}

ORDER BY DESC(?similarity)

LIMIT 20
```

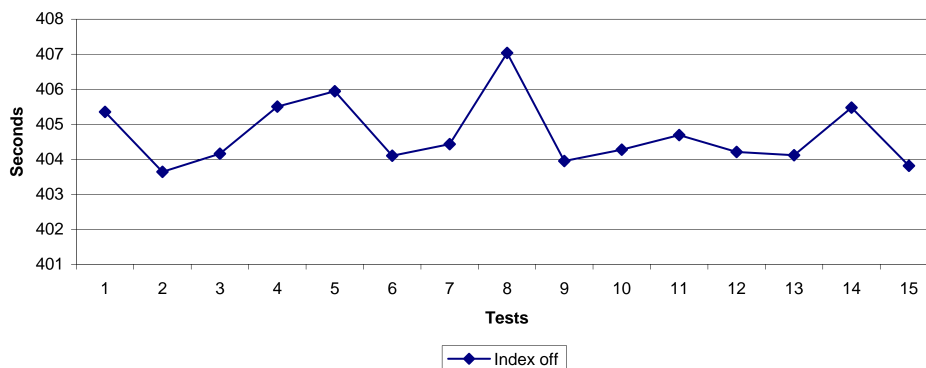

Figure 5.21: Similarity Index Off

the performance when the similarity index is activated. Thus, instead to calculate the similarity during query execution the framework performs an index lookup which results in a remarkable performance improvement.

We implement the similarity index as a MySQL table where resource pairs specified by URI-references are stored together with the corresponding similarity calculated by a specific strategy. The SUMO contains 121 concepts. In order to evaluate our retrieval task, we create an index of similarities between the 'Wine' concept and all other concepts. Thus, the index contains 121 entries. The index computation for our retrieval task takes 422 seconds (around 7 minutes) on our test server.

In a more realistic environment, the maintenance of a similarity index requires more effort. First, each time a new concept is added to the ontology the index should be updated by adding the new similarity pairings. Since the addition of a new concept to an ontology may affect the similarity of existing resources (e.g., type hierarchy alteration) an incrementally calculated index should consider to compute similarity values not only for new pairings but also for resources that are affected by some modification. Further, it is typically not enough to index the similarity of a

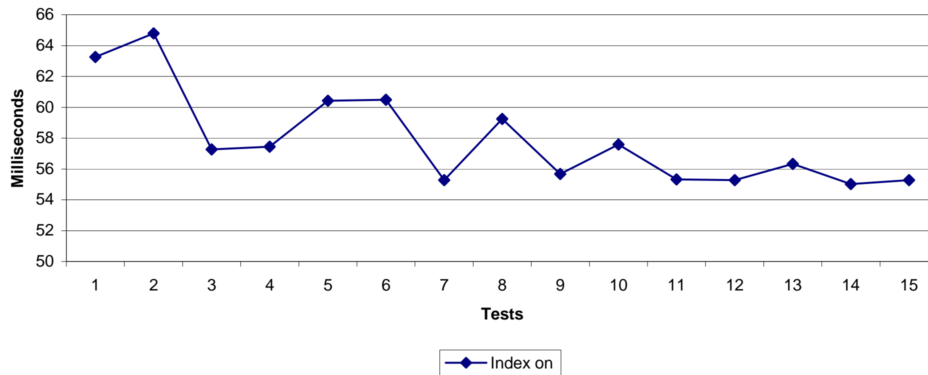

Figure 5.22: Similarity Index On

concept and each other concept in the ontology (1:n) but we need to index the similarity of each possible concept pair (n:n). Since usually we can assume that

$$sim(A, B) = sim(B, A)$$

the number of similarity values that need to be indexed can be specified by

$$I = \binom{m+1}{2} = \frac{(m+1)!}{2!(m-1)!} = \frac{m(m+1)}{2}$$

for an ontology holding m concepts. Thus, in order to create a complete index of the SUMO we need to calculate

$$I = \frac{(121+1)!}{2!(121-1)!} = 7'381$$

pairings of two concepts, where the order does not matter and a concept can be chosen more than once (combination with repetition). We may estimate the time required to create the similarity index for the SUMO to 25'742 seconds which is around 7 hours. Please note the required effort and mark that the number of concepts contained in the SUMO is relatively small and the similarity strategy used could be even more complex.

5.1.7 Final Example: Putting It All Together

To close this first evaluation, we perform a final retrieval task on another ontology and with a new purpose. This final evaluation shows two things. First, how similarity may be used to join data of different ontologies by means of similarity strategies executed on a key attribute shared by both ontologies. Secondly, the performance improvement rule-by-rule including the similarity index.

For this purpose, we take a subset of the SwetoDblp ontology and another publication ontology maintained by the Institute for *Angewandte Informatik und Formale Beschreibungsverfahren* (AIFB), University of Karlsruhe, Germany[20]. The resource schema in both ontologies is different not only because of unlike URI-references but also because of a different set of predicates.

[20] http://www.aifb.uni-karlsruhe.de/

Our purpose is to extract information from both ontologies by joining a shared resource predicate. A similarity strategy specified for this predicate allows a ranking of the most similar resources to the referenced resource (i.e., the AIFB publication entitled 'Semantic Methods for P2P Query Routing'). The top result is expected to be the one which joins the same resource (i.e., publication) in both ontologies. In Appendix B we list the query used for the evaluation.

The second purpose is to show the performance improvement achieved rule-by-rule. Figure 5.23 displays for each rule level the seconds required to execute the query. A rule level executes not only the newly added rule but all the previous rules too. As showed in Figure 5.23, each rule level improves the performance. The improvement ratios between the rules highly depends on the query and ontology. For example, the reorder imprecise rule may not yield significant performance improvement if Levenshtein is used for both strategies defined in query (Appendix B). Although some rules may affect more than others, it is the total improvement achieved by all rules that matters.

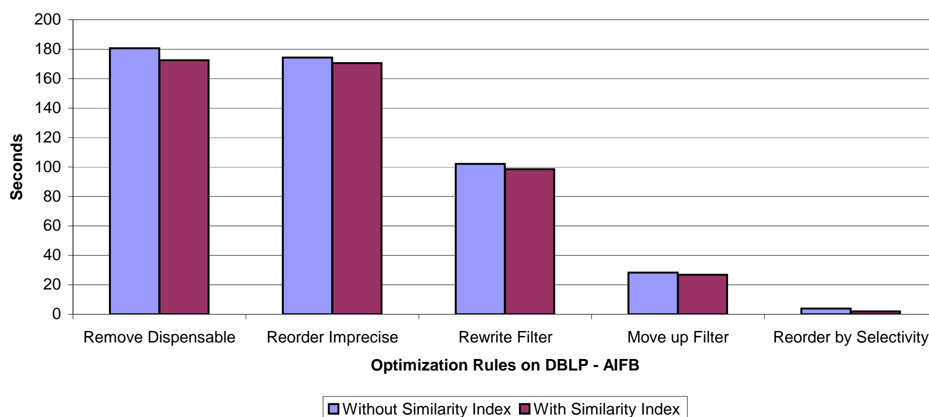

Figure 5.23: Final Example: Performance Rule-by-Rule including Similarity Index

Since the measures used in both similarity strategies (DiceStrCmp and LOLStrCmp) are very efficient in similarity computation, the similarity index method used in the previous Section 5.1.6 based on a MySQL table would not lead to optimization because of the JDBC[21] overhead. Thus, we implement an in-memory similarity index based on a HashMap[22] which allows a fast lookup for similarities. As illustrated in Figure 5.23, at each optimization level the similarity index based on a HashMap performs faster. The difference in optimization achieved by the similarity index for both 'Remove Dispensable' and 'Reorder Imprecise' is notable. In fact, because of the 'Reorder Imprecise' rule, the amount of similarity comparisons is almost reduced to the half. This behavior can be noticed in Figure 5.23.

Please note that because of the dispensable triple pattern specified in query, we could not measure the time required to execute the query without any activated rule. This pattern yields a tremendous intermediate result set because it matches each resource specified in the ontology.

In Appendix B we list both original and optimized queries used to perform this evaluation. Table 5.8 lists the absolute values measured for this evaluation.

[21]http://java.sun.com/javase/technologies/database/index.jsp
[22]http://java.sun.com/j2se/1.4.2/docs/api/java/util/HashMap.html

Rule Level	Without Similarity Index	With Similarity Index
Remove Dispensable	180’618.49	172’582.19
Reorder Imprecise	174’388.29	170’506.82
Rewrite Filter	102’036.89	98’501.38
Move up Filter	28’274.48	26’849.87
Reorder by Selectivity	3’980.61	1’899.09

Table 5.8: Absolute Values for Rule-by-Rule Evaluation including Similarity Index

5.2 Qualitative Query Retrieval Evaluation

The qualitative evaluation focuses on information retrieval performance of iSPARQL queries using different similarity strategies. Mainly, we perform standard precision, recall, and f-measure [Baeza-Yates and Ribeiro-Neto, 1999] evaluations over test collections in order to qualify the performance of our imprecise framework. Moreover, we evaluate the quality of similarity strategies according to a human similarity judgment (i.e., gold standard).

5.2.1 Datasets

For iSPARQL precision and recall evaluation we use OWLS-TC[23]. OWLS-TC is a retrieval test collection to support the evaluation of the performance of Semantic Web service matchmaking algorithms. The test collection contains 28 queries specified for different domains (e.g., communication, economy, travel, medical). For each query a corresponding relevance set of services is given. OWLS-TC services are characterized by a service name and description. Moreover, each service may have one or more service inputs and outputs. Both inputs and outputs are concepts defined in 43 OWL ontologies which are included in OWLS-TC. For example, a traveling service may require some information about which sports we like to do (e.g., hiking and surfing) which are used as inputs and returns (output) destinations where we are able to play them.

Further, we use the MIT Process Handbook[24] [Malone et al., 2003] for an interesting evaluation about the deviation of similarity strategies to human similarity judgment. The MIT Process Handbook is a rich ontology of business activities (e.g., sell, make, buy) which are classified according to multiple hierarchical dimensions. Activities are described by a name and description. Descriptions may be multiple paragraphs long and can include any kind of information. Moreover, activities are organized against both generalization-specialization and parts-uses dimensions. Generalizations are related activities, i.e., the more general activities (e.g., ‘Exchange’ and ‘Provide’ are general activities to ‘Sell’). Specializations are other ways an activity can be done, i.e., the different types (e.g., ‘Sell via store’ or ‘Sell via other direct marketing’ are other ways ‘Sell’ can be done). The part attributes specify sub activities (e.g., ‘Obtain order’ and ‘Receive payments’ are sub activities of ‘Sell’) whereas the uses attributes are lists of activities that use an activity (e.g., the activity ‘Produce as a Distributor’ uses the ‘Sell’ activity). The broad specification of activities allows the usage and definition of very different similarity strategies which includes different similarity measures and allows the consideration of miscellaneous features (e.g., activity name, description, parts, specializations, etc.)

[23] Version 2, Release 1, Date 11/15/2005, http://projects.semwebcentral.org/projects/owls-tc/

[24] http://ccs.mit.edu/ph/

5.2.2 Evaluations

We start evaluating iSPARQL precision and recall on OWLS-TC dataset. We show how the precision and recall changes depending on the selected similarity strategy and we calculate the f-measure performance improvement for strategies. The evaluations discussed in this section clearly illustrate how a similarity strategy affects the retrieval performance. Thus, our evaluation supports the theory depicted in Chapter 3 which states that specific similarity strategies improve retrieval performance.

iSPARQL: Precision and Recall

We evaluate precision, recall, and f-measure of OWLS-TC services by using the relevance sets of services provided by the test collection. A similar approach is used and described for the evaluation of iRDQL [Bernstein and Kiefer, 2006]. The goal is to perform a qualitative evaluation of different strategies investigating their precision and recall. Further, we compute the average precision, recall, and f-measure of 10 services for different domains and strategies and we measure the average f-measure performance improvement for the selected strategies.

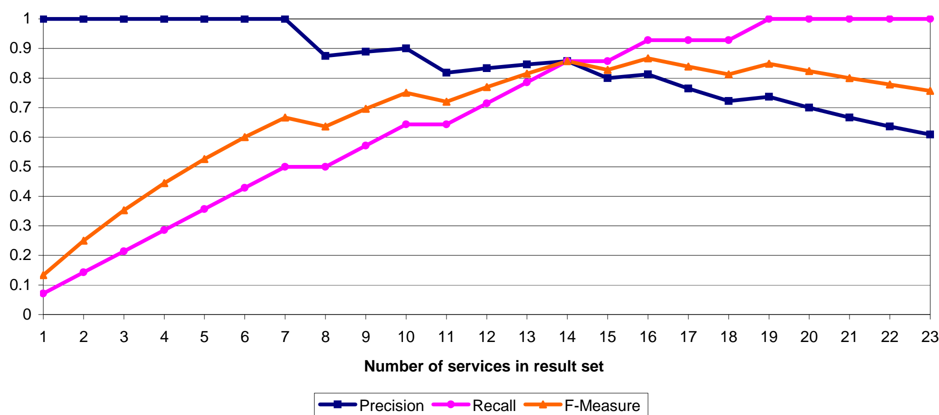

Figure 5.24: Precision, Recall, F-Measure: TFIDF Strategy

We chose two slightly different similarity strategies. Both consider the service name, description and inputs/outputs of services. The difference consists in the selected similarity measure used to calculate the similarity of descriptions. The first similarity strategy uses TFIDF whereas the second uses Levenshtein of Levenshtein to compute the similarity of descriptions. The name predicate is compared by means of Levenshtein whereas the Jaccard is used to compare the sets of service inputs and outputs.

Our strategy based on TFIDF (strategy A) performs generally better compared to the one based on Levenshtein of Levenshtein (strategy B). Figures 5.24 and 5.25 show this behavior for a specific OWLS-TC query service (i.e., the DVD and MP3 player price service). For strategy A the top 7 retrieved services are contained in the relevance set (given by OWLS-TC test collection). For strategy B only the first 5 retrieved services. Looking at recall, we mark that strategy A identifies each service contained in relevance set after 19 retrieved services whereas strategy B fulfills only after 93 retrieved services the complete relevance set.

Figures 5.26 and 5.27 resume the average precision, recall, and f-measure of 10 queries performed over OWLS-TC for both strategies A and B using different OWLS-TC service domains. As expected, the strategy based on TFIDF (strategy A) performs slightly better than strategy B

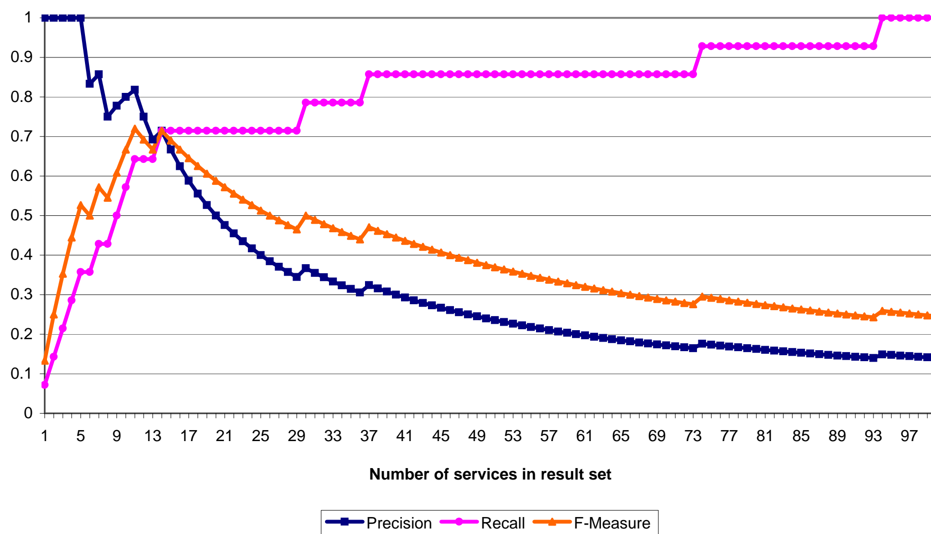

Figure 5.25: Precision, Recall, F-Measure: Levenshtein of Levenshtein Strategy

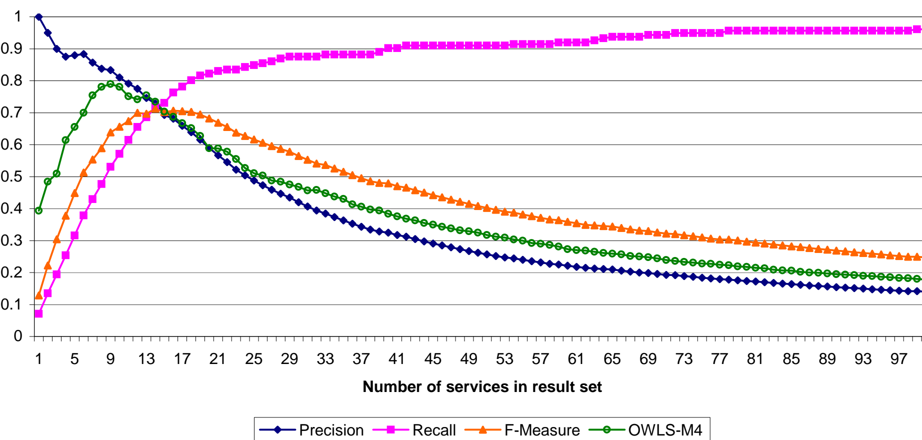

Figure 5.26: Precision, Recall, F-Measure: Average for Strategy A

based on Levenshtein of Levenshtein. The average f-measure improvement of strategy A compared to strategy B is 7.39%. Further, the figures includes the average f-measure for OWLS-M4 which is the best performing matchmaking algorithm of the OWLS-MX hybrid Semantic Web service matchmaker ([Bernstein and Kiefer, 2006] and [Klusch et al., 2006]).

iSPARQL: Gold Standard Deviation

In order to evaluate the quality of iSPARQL strategies, we conduct an evaluation using a gold standard based on human similarity judgment. The gold standard consists of 21 process pairs extracted from the MIT Process Handbook. The similarity of each pair is judged by a number of people which estimated the similarity of a process pair between 1 (lowest similarity) and 5 (highest similarity). For each process pair we calculate the average human similarity judgment

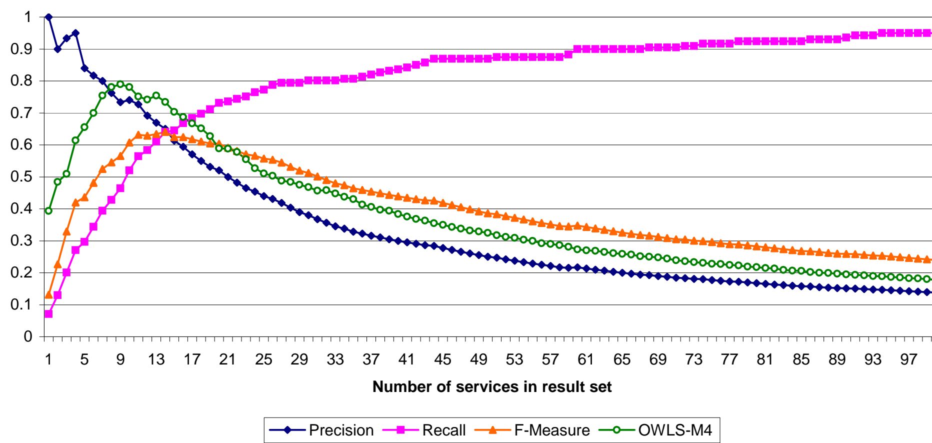

Figure 5.27: Precision, Recall, F-Measure: Average for Strategy B

value which is then normalized to [0,1]. We use the values resulting from human judgment as gold standard to evaluate the goodness of similarity strategies implemented in our imprecise framework.

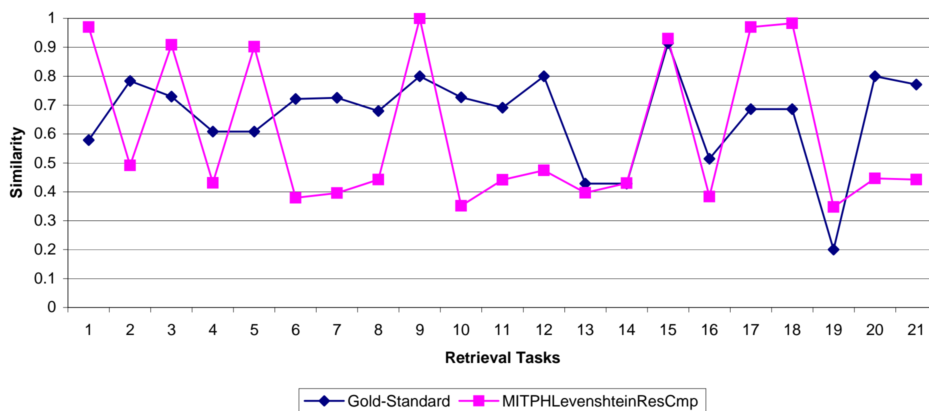

Figure 5.28: Gold Standard vs. Levenshtein Strategy

In the following we show the results for three different similarity strategies used to evaluate the similarity between processes of the MIT Process Handbook. The first strategy is based on a Levenshtein resource comparison of both process parts (i.e., sub activities). The similarity of two processes is calculated considering the structure of parts (i.e., the number of parts) and the Levenshtein similarity of the corresponding part names. The second strategy is based on the tree edit distance. Hence, the similarity is calculated comparing the trees of process parts.

Figure 5.28 shows the gold standard compared to the Levenshtein based strategy whereas Figure 5.29 shows the gold standard compared to the tree edit distance strategy. We may point out a smaller deviation for the strategy based on tree edit distance. Thus, we can state that the similarity strategy based on tree edit distance approximates better human similarity judgment of MIT Process Handbook processes (assuming a qualitative human judgment). Figure 5.30 shows the gold standard compared to a similarity strategy based on the Levenshtein of Levenshtein similarity measure which considers the name, description, and parts of processes. This is the

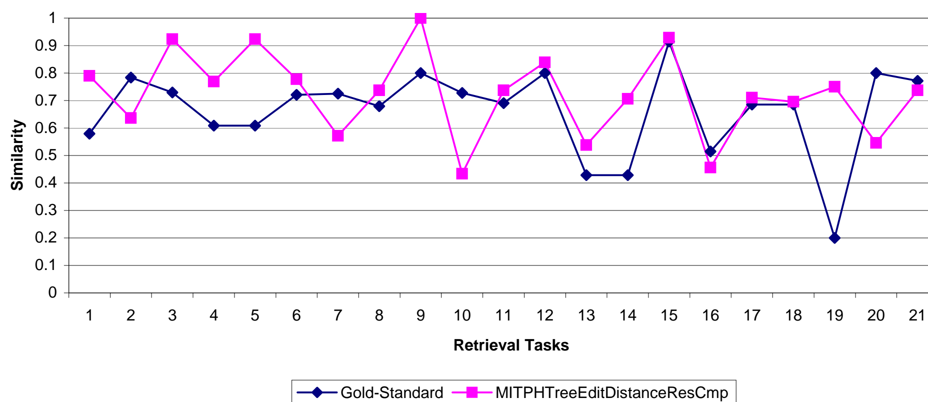

Figure 5.29: Gold Standard vs. Tree Edit Distance Strategy

strategy which best approximates human similarity judgment in our evaluation.

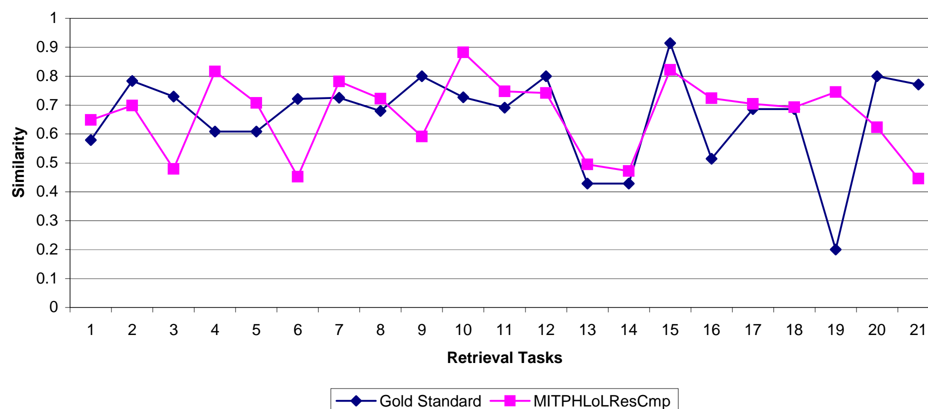

Figure 5.30: Gold Standard vs. Levenshtein of Levenshtein Strategy

Related to this evaluation we present in the following another experiment. The aim is to evaluate the average deviation from our gold standard for multiple similarity strategies using different process features (i.e., predicates) which are compared by various similarity measures. The result is an ordering of similarity strategies ranked by increasing deviation from the gold standard. The deviation is measured as an average value of the absolute differences between the gold standard and the similarity measured for a specific strategy.

Some strategies consider only a single predicate of business activities defined in the MIT Process Handbook (e.g., name or description) while other consider multiple predicates (e.g., name and description). Further, attributes are compared by means of Levenshtein, tree edit distance, Levenshtein of Levenshtein or TFIDF, depending whether the measure is meaningful or not for the corresponding object value. The evaluation shows that a careful selection of attributes and a corresponding meaningful measure yield to significant performance improvement.

This evaluation supports our original idea of similarity strategies based on the 'Features of Similarity' theory of A. Tversky [Tversky, 1977] described in Chapter 2. The evaluation demonstrates that a careful selection of features (i.e., predicates) of ontological resources and the investigation of meaningful similarity measures is necessary to reach a good approximation of human

similarity judgment. Although a general similarity measure or strategy can be used to compare ontological resources, I believe that in general a specific similarity strategy which considers specific features and compares them using meaningful measures yields to respectable performance improvement.

The definition of strategies is a task which may be affected by the attained similarity goal. As a simple example, in an ontology which describes the curriculum vitae of individuals, a comparison of personal details, e.g., first name or last name, is not meaningful if our aim is to characterize the similarity of people according to their knowledge and work experience. Thus, careful feature selection matters and is related to the attained goal.

Table 5.9 shows the results of our evaluation ranked by the increasing average deviation to the human similarity judgment.

Strategy	Average Deviation
LOL(Name, Description, Parts)	0.144846435
LOL(Parts)	0.15147453
TED(Parts)	0.152694516
LOL(Name, Description); TED(Parts)	0.154761688
LOL(Name); TED(Parts)	0.165089958
Levenshtein(Parts)	0.237248183
LOL(Name, Parts); TFIDF(Description)	0.276288123
LOL(Name); TFIDF(Description); Levenshtein(Parts)	0.291227587
LOL(Name)	0.29972154
LOL(Name); TFIDF(Description)	0.317672939
Levenshtein(Name); TFIDF(Description)	0.387880911
TFIDF(Description)	0.485221486

Table 5.9: Strategy Deviation Ranking

The strategy which performs best in our evaluation implements the Levenshtein of Levenshtein (LOL) similarity measure on name, description, and part predicates of MIT Process Handbook processes. Generally, the Levenshtein of Levenshtein measure approximates well human similarity judgment especially if compared to the Levenshtein or the TFIDF. A measure which compares the structure of sub processes, e.g., the tree edit distance, yields good approximations too. The TFIDF generally yields poor performance. This may be because of bad text quality (HTML tags inside of descriptions). The Levenshtein of Levenshtein does clearly enhance the performance of the TFIDF also compared to the Levenshtein measure. Finally, we can mark that strategies which consider more features or features where more data is compared (process parts) are performing better.

All this considerations are specific to the MIT Process Handbook ontology and may be totally different for other ontologies. This supports our preliminary statement that strategies should be studied for a specific ontology. We may expect that specific similarity strategies perform better compared to general purpose similarity strategies.

6

Conclusions

The seamless integration of our imprecise framework into ARQ and SPARQL is one of the major advantages of the proposed approach. Traditional SPARQL, Jena, and ARQ can be used to run iSPARQL queries. This allows an uncomplicated and fast deployment of iSPARQL on different systems.

I believe, the similarity between complex resources in ontologies may be measured by different logics, each resulting in varying similarity values. In order to evaluate the goodness of a logic, a gold standard is required (i.e., human similarity judgment). Thus, we are able to identify the logic which best approximates the gold standard. Complex ontology resources may not only be measured by different logics, but similarity logics also highly depend on the underlying ontology. The design of our imprecise framework is in line with the nature of similarities for complex ontology resources, since it allows extending strategies that meets best the properties of ontologies and their resources. Similarity strategies allow particular fine tuning and facilitate good approximation of human similarity judgment.

Our proposed iSPARQL optimization framework is a first approach for triple pattern selectivity estimation based on statistical information about the resources contained in the underlying ontology. As the quantitative query performance evaluation in Section 5.1 shows, the approach seems to be reasonable, and I believe this is the way to go. Obviously, more research work is required to get even more accurate estimations. It is remarkable that a few optimization rules which all aims the common goal to reduce the intermediate result set size of triple patterns highly affect query execution performance.

The optimization work discussed in this thesis, mainly focuses on static query reordering in order to get an execution plan which is optimal against the selectivity of triple patterns. Static optimization techniques may be combined with dynamic techniques to achieve optimization also when static techniques do not lead to any effective optimization (e.g., when the query is already optimized according to the selectivity of triple patterns).

6.1 Limitations

Our flexibility for similarity strategies is also a drawback. Extending strategies not only requires a deep study of the underlying ontology and knowledge about similarity measures, but necessitates the implementation of specific classes, written in Java. Hence, implementing a specific strategy with good gold standard approximation is anything but a trivial task and the required effort not

negligible. Nevertheless, our imprecise framework allows the implementation of generic strategies too. Generic strategies are ontology independent and usually they involve a single similarity measure.

iSPARQL queries require a specific statement ordering. This is the major drawback of our approach. On the one hand, the SPARQL variables used as strategy arguments must be previously bound by ARQ. On the other hand, imprecise strategy statements need to conform a special ordering. Nevertheless, our iSPARQL optimizer (Chapter 4) implements a rule which reorders imprecise statements to meet this requirement. Hence, imprecise statements may be written in any ordering, provided that the optimizer is activated.

The proposed iSPARQL optimization framework implements a basic statistical model used for selectivity estimation. More research should be invested to improve the accuracy of estimations. For example, SPARQL variables constrained by an inequality operator (e.g., $>$, $<$) are not considered yet while estimating the selectivity of patterns. It is straightforward, that the selectivity of a variable constrained in a FILTER expression can be estimated more accurately. Moreover, our framework calculates the object selectivity according to a specific predicate. This limitation can be avoided by generalizing the function for the selectivity estimation of triple pattern objects. In fact, when a predicate is unknown, we may compute the selectivity of an object by considering the histograms describing the object domain values for each predicate. Furthermore, the subject selectivity estimation formula used in our framework may be determined more accurately too. In fact, the subject selectivity is constant in our framework. A more precise statistic where the number of predicates are modeled for each resource class inside the ontology may lead to a more accurate subject selectivity estimation. For example, resources of a class `Person` may have more predicates compared to resources of a class `Address`. A statistical representation of the average number of predicates for resources according to the resource class may lead to more accurate and natural subject selectivity estimation. Last but not least, our framework does not consider the behavior of triple pattern selectivity for joined variables (please refer to the discussion about the selectivity propagation in Section 4.9). In fact, assigning the selectivity 1.0 to variables which are previously bound is a too rough approximation. In Section 4.9 we describe our proposed approach to solve this limitation.

6.2 Future Work

Reordering of imprecise statements is assured only if OptARQ is activated. In fact, the required logic is implemented as rule in our optimization framework. To allow reordering of iSPARQL statements even when the optimizer is deactivated, we should split the corresponding rule (Section 4.7) into separate logics.

In order to enable a comfortable deployment of similarity strategies for iSPARQL, a possible extension could be a sort of building set which allows a configuration of similarity strategies. Instead of implementing similarity strategies imperatively, i.e., by giving a sequence of commands the strategy has to execute, we could implement them in a declarative way, i.e., describing *what* the strategy should do.

Some of the limitations described above, especially those for the iSPARQL optimization framework, may be a starting point for future work. We expect that resolving the limitations discussed will lead to more accurate selectivity estimation and thus, enhance the model to a more general one which may be even more robust in a productive field.

During the last decades, different statistical models have been proposed to characterize attribute value distributions. Our model uses an equal-width histogram to represent the object value domains (Chapter 4). Other histogram based approaches have been proposed and are

summarized in [Oommen and Rueda, 2001]. They all aim the cost estimation of query execution plans. It has been shown that some methods are less erroneous on selectivity estimation compared to others. In fact, the equal-width histogram based selectivity estimation used in our optimization framework is a relatively simple approach which may lead to significantly higher estimation errors (i.e., large classes result into inaccurate estimation). Moreover, other selectivity estimation models have been proposed too, e.g., probabilistic selectivity estimation models [Getoor et al., 2001].

We focused mainly on static optimization techniques, e.g., query reordering or similarity indexing. In Section 4.8 we sketched the aggregation optimization which exploits the characteristics of some aggregation method. This optimization technique is performed during query execution. Thus, it belongs to the class of dynamic query optimization. Dynamic optimization involves techniques which consider the state of the environment. For example, a plan which is expected to be optimal before query execution may not be optimal during execution, perhaps because of a modified environment or outdated statistics. Thus, the optimizer generates sub-optimal plans due to invalid assumptions [Markl et al., 2004]. Because of the distributed nature of Semantic Web knowledge bases considering their attributes and environment changes during query execution will be fundamental.

A

Appendix A

Listing A.1: iSPARQL Extended Grammar

```
[21]  FilteredBasicGraphPattern ::=
        BlockOfTriples? | ImpreciseBlockOfTriples?
        ( Constraint '.'? FilteredBasicGraphPattern )?
[94]  ImpreciseBlockOfTriples ::=
        NameStatement '.'
        ArgumentsStatement | ScoresStatement '.'
        ( AggregatorStatement '.' ThresholdStatement '.' )
        ( IgnorecaseStatement | WeightsStatement '.' )
        SimilarityStatement '.'
[95]  NameStatement        ::= Var IRIrefName String '.'
[96]  ArgumentsStatement   ::= Var IRIrefArguments ObjectList '.'
[97]  ScoresStatement      ::= Var IRIrefScores ObjectList '.'
[98]  AggregatorStatement  ::= Var IRIrefAggregator String '.'
[99]  ThresholdStatement   ::= Var IRIrefThreshold String '.'
[100] WeightsStatement     ::= Var IRIrefWeights ObjectList '.'
[101] IgnorecaseStatement  ::= Var IRIrefIgnorecase String '.'
[102] SimilarityStatement  ::= Var IRIrefSimilarity Var '.'
[103] IRIrefName           ::=
        java:ch.unizh.ifi.isparql.query.property.name
[104] IRIrefArguments      ::=
        java:ch.unizh.ifi.isparql.query.property.arguments
[105] IRIrefScores         ::=
        java:ch.unizh.ifi.isparql.query.property.scores
[106] IRIrefAggregator     ::=
        java:ch.unizh.ifi.isparql.query.property.aggregator
[107] IRIrefThreshold      ::=
        java:ch.unizh.ifi.isparql.query.property.threshold
[108] IRIrefWeights        ::=
        java:ch.unizh.ifi.isparql.query.property.weights
[109] IRIrefIgnorecase     ::=
        java:ch.unizh.ifi.isparql.query.property.ignorecase
[110] IRIrefSimilarity     ::=
        java:ch.unizh.ifi.isparql.query.property.similarity
```

B

Appendix B

Listing B.1: Final Example: Query

```
PREFIX  swrc:     <http://swrc.ontoware.org/ontology#>
PREFIX  opus:   <http://lsdis.cs.uga.edu/projects/semdis/opus#>
PREFIX  rdfs:   <http://www.w3.org/2000/01/rdf-schema#>
PREFIX  isparql:  <java:ch.unizh.ifi.isparql.query.property.>

SELECT ?title1 ?proceedings1 ?year1 ?abstract2 ?series2
       ?volume2 ?booktitle2 ?month2 ?pages2 ?similarity3
WHERE
{
  ?resource2 swrc:year ?year2 .
  ?resource1 rdfs:label ?title1 .
  ?resource1 opus:book_title ?booktitle1 .
  ?resource2 swrc:title ?title2 .
  ?resource2 swrc:abstract ?abstract2 .
  ?resource2 swrc:series ?series2 .
  ?resource2 swrc:volume ?volume2 .
  ?resource2 swrc:booktitle ?booktitle2 .
  ?resource2 swrc:month ?month2 .
  ?resource2 swrc:pages ?pages2 .
  ?resource1 opus:year ?year1 .
  ?resource1 opus:contained_in_proceedings ?proceedings1 .
  ?resource3 rdfs:label ?title3 .
  FILTER (?year2 >= 2005)
  FILTER (?title2 = "Semantic Methods for P2P Query Routing")
  ?strategy1 isparql:name "LOLStrCmp" .
  ?strategy1 isparql:arguments (?booktitle1 ?booktitle2) .
  ?strategy1 isparql:similarity ?similarity1 .
  ?strategy2 isparql:name "DiceStrCmp" .
  ?strategy2 isparql:threshold 0.5 .
  ?strategy2 isparql:arguments (?title1 ?title2) .
  ?strategy2 isparql:similarity ?similarity2 .
  ?strategy3 isparql:name "ScoreAggregator" .
  ?strategy3 isparql:scores (?similarity1 ?similarity2) .
  ?strategy3 isparql:similarity ?similarity3 .
}
```

Listing B.2: Final Example: Optimized Query

```
PREFIX swrc:    <http://swrc.ontoware.org/ontology#>
PREFIX opus:   <http://lsdis.cs.uga.edu/projects/semdis/opus#>
PREFIX rdfs:   <http://www.w3.org/2000/01/rdf-schema#>
PREFIX isparql: <java:ch.unizh.ifi.isparql.query.property.>

SELECT ?title1 ?proceedings1 ?year1 ?abstract2 ?series2
       ?volume2 ?booktitle2 ?month2 ?pages2 ?similarity3
WHERE
{
  ?resource2 swrc:title "Semantic Methods for P2P Query Routing" .
  ?resource2 swrc:volume ?volume2 .
  ?resource2 swrc:series ?series2 .
  ?resource2 swrc:pages ?pages2 .
  ?resource2 swrc:abstract ?abstract2 .
  ?resource2 swrc:month ?month2 .
  ?resource2 swrc:booktitle ?booktitle2 .
  ?resource2 swrc:year ?year2 . FILTER (?year2 >= 2005)

  ?resource1 opus:contained_in_proceedings ?proceedings1 .
  ?resource1 opus:book_title ?booktitle1 .
  ?resource1 rdfs:label ?title1 .
  ?resource1 opus:year ?year1 .

  ?strategy2 isparql:name "DiceStrCmp" .
  ?strategy2 isparql:threshold 0.5 .
  ?strategy2 isparql:argument1 ?title1 .
  ?strategy2 isparql:argument2 "Semantic Methods for P2P Query Routing" .
  ?strategy2 isparql:similarity ?similarity2 .

  ?strategy1 isparql:name "LOLStrCmp" .
  ?strategy1 isparql:arguments (?booktitle1 ?booktitle2) .
  ?strategy1 isparql:similarity ?similarity1 .

  ?strategy3 isparql:name "ScoreAggregator" .
  ?strategy3 isparql:scores (?similarity1 ?similarity2) .
  ?strategy3 isparql:similarity ?similarity3 .
}
```

Bibliography

[Antoniou and van Harmelen, 2004] Antoniou, G. and van Harmelen, F. (2004). *A Semantic Web Primer*. The MIT Press.

[Baeza-Yates and Ribeiro-Neto, 1999] Baeza-Yates, R. and Ribeiro-Neto, B. (1999). *Modern Information Retrieval*. Addison Wesley Longman Publishing Co. Inc.

[Berners-Lee et al., 1998] Berners-Lee, T., Fielding, R., and Masinter, L. (1998). Uniform Resource Identifiers (URI): Generic Syntax. RFC 2396.

[Bernstein et al., 2005] Bernstein, A., Kaufmann, E., Kiefer, C., and Bürki, C. (2005). Simpack: A Generic Java Library for Similiarity Measures in Ontologies.

[Bernstein and Kiefer, 2006] Bernstein, A. and Kiefer, C. (2006). Imprecise RDQL: Towards Generic Retrieval in Ontologies Using Similarity Joins. In *21th Annual ACM Symposium on Applied Computing (ACM SAC 2006)*, New York, NY, USA. University of Zurich, Department of Informatics, ACM Press.

[Chamberlin et al., 1981] Chamberlin, D. D., Astrahan, M. M., Blasgen, M. W., Gray, J. N., King, W. F., Lindsay, B. G., Lorie, R., Mehl, J. W., Price, T. G., Putzolu, F., Selinger, P. G., Schkolnick, M., Slutz, D. R., Traiger, I. L., Wade, B. W., and Yost, R. A. (1981). A History and Evaluation of System R. *Commun. ACM*, 24(10):632–646.

[Chamberlin et al., 2001] Chamberlin, D. D., Florescu, D., Robie, J., Siméon, J., and Stefanescu, M. (2001). XQuery: A Query Language for XML. Technical report.

[Cohen et al., 2003] Cohen, W., Ravikumar, P., and Fienberg, S. (2003). A Comparison of String Distance Metrics for Name-Matching Tasks.

[Fensel, 2004] Fensel, D. (2004). *Ontologies: A Silver Bullet for Knowledge Management and Electronic Commerce*. Springer.

[Gamma et al., 1995] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional; 1st edition.

[Ganesan et al., 2003] Ganesan, P., Garcia-Molina, H., and Widom, J. (2003). Exploiting Hierarchical Domain Structure to Compute Similarity. *ACM Trans. Inf. Syst.*, 21(1):64–93.

[Getoor et al., 2001] Getoor, L., Taskar, B., and Koller, D. (2001). Selectivity Estimation using Probabilistic Models. In *SIGMOD '01: Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, pages 461–472, New York, NY, USA. ACM Press.

[Gruber, 1993] Gruber, T. R. (1993). Towards Principles for the Design of Ontologies Used for Knowledge Sharing. In Guarino, N. and Poli, R., editors, *Formal Ontology in Conceptual Analysis and Knowledge Representation*, Deventer, The Netherlands. Kluwer Academic Publishers.

[Hliaoutakis et al., 2006] Hliaoutakis, A., Varelas, G., Voutsakis, E., Ptrakis, E., and Milios, E. (2006). Information Retrieval by Semantic Similarity. *International Journal on Semantic Web and Information Systems*, 3(3):55–72.

[Hurtardo et al., 2006] Hurtardo, C. A., Pulovassilis, A., and Wood, P. T. (2006). A Relaxed Approach to RDF Querying.

[Klusch et al., 2006] Klusch, M., Fries, B., and Sycara, K. (2006). Automated Semantic Web Service Discovery with OWLS-MX. In *Proceedings of 5th International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*. ACM Press.

[Levenshtein, 1966] Levenshtein, V. I. (1966). Binary Codes Capable of Correcting Deletions, Insertions, and Reversals. *Soviet Physics Doklady*, 10(8):707–710.

[Lin, 1998] Lin, D. (1998). An Information-Theoretic Definition of Similarity. In *Proc. 15th International Conference on Machine Learning*, pages 296–304. Morgan Kaufmann, San Francisco, CA.

[Malone et al., 2003] Malone, T. W., Crowston, K., and Herman, G. A. (2003). *Organizing Business Knowledge: The MIT Process Handbook*. MIT Press.

[Manola and Miller, 2004] Manola, F. and Miller, E. (2004). RDF Primer. Technical report, W3C.

[Markl et al., 2004] Markl, V., Raman, V., Simmen, D., Lohman, G., Pirahesh, H., and Cilimdzic, M. (2004). Robust Query Processing through Progressive Optimization. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, pages 659–670, New York, NY, USA. ACM Press.

[Medin et al., 1993] Medin, D. L., Goldstone, R. L., and Gentner, D. (1993). Respects for Similarity. *Psychological Review*, 100(2):254–278.

[Miller and Charles, 1991] Miller, G. and Charles, W. (1991). Contextual Correlates of Semantic Similarity. *Language and Cognitive Processes*.

[Motik and Sattler, 2006] Motik, B. and Sattler, U. (2006). A Comparison of Reasoning Techniques for Querying Large Description Logic ABoxes.

[Oommen and Rueda, 2001] Oommen, B. and Rueda, L. (2001). The Efficiency of Modern-day Histogram-like Techniques for Query Optimization.

[Perez et al., 2006] Perez, J., Arenas, M., and Gutierrez, C. (2006). Semantics and Complexity of SPARQL.

[Piatetsky-Shapiro et al., 1984] Piatetsky-Shapiro, G., Piatetsky-Shapiro, C. C., and Connell, C. (1984). Accurate Estimation of the Number of Tuples Satisfying a Condition. In *SIGMOD '84: Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, pages 256–276, New York, NY, USA. ACM Press.

[Prud'hommeaux and Seaborne, 2006] Prud'hommeaux, E. and Seaborne, A. (2006). SPARQL Query Language for RDF. Technical report, W3C.

[Resnik, 1995] Resnik, P. (1995). Using Information Content to Evaluate Semantic Similarity in a Taxonomy. In *IJCAI*, pages 448–453.

[Selinger et al., 1979] Selinger, P. G., Astrahan, M. M., Chamberlin, D. D., Lorie, R. A., and Price, T. G. (1979). Access Path Selection in a Relational Database Management System. In *SIGMOD '79: Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, pages 23–34, New York, NY, USA. ACM Press.

[Siberski et al., 2006] Siberski, W., Pan, J. Z., and Thaden, U. (2006). Querying the Semantic Web with Preferences. In *Proc. of the 5th International Semantic Web Conference (ISWC 2006)*.

[Sirin et al., 2006] Sirin, E., Grau, B. C., and Parsia, B. (2006). From Wine to Water: Optimizing Description Logic Reasoning for Nominals.

[Tversky, 1977] Tversky, A. (1977). Features of Similarity. *Psychological Review*, (84):327–353.

[Tversky and Gati, 1978] Tversky, A. and Gati, I. (1978). Studies of Similarity. *Cognition and Categorization*, pages 79–98.

[Valiente, 2002] Valiente, G. (2002). *Algorithms on Trees and Graphs*. Springer-Verlag.