Managing, Querying and Analyzing Big Data on the Web

Marios Meimaris

Department of Computer Science and Biomedical Informatics

University of Thessaly

A thesis submitted for the degree of

Doctor of Philosophy

April 2018

This thesis is dedicated to
my family and friends

Acknowledgements

First and foremost, I would like to thank my supervisor, Prof. Ioannis Anagnostopoulos, for guiding me, completely trusting in me and granting the required autonomy to pursue the directions studied in this thesis. Furthermore, I would like to thank Dr. George Papastefanatos for what has become a very fruitful and productive collaboration over the course of this thesis. I would also like to thank Prof. Panos Vassiliadis for his mentoring and guidance. Finally, I would like to thank all of my co-authors and colleagues.

Abstract

In this thesis, we study information management problems that arise in the Semantic Web, focusing on the Resource Description Framework (RDF) model and its associated SPARQL query language. To this end, we focus in three directions, namely (i) RDF data evolution, (ii) storage, indexing and query optimization in RDF/SPARQL engines, and (iii) efficient and scalable information retrieval from multidimensional RDF datasets. We present efficient and scalable methods focused on specific problems in the aforementioned directions, with the ultimate aim to propose advancements in the relevant state of the art.

In the first direction (chapters 2 and 3), we study the problem of representing, storing and querying evolving RDF data. To this end, a novel data model and query language are proposed, that address representation of versioning in heterogeneous domains,. Furthermore, in order to assist evaluation of RDF versioning and evolution management engines and frameworks, a novel synthetic dataset generator is introduced.

In the second direction (chapters 4, 5 and 6), we tackle the problem of indexing and query optimization, specifically focusing on heavy query workloads in loosely-structured RDF datasets. To this end, we propose a novel indexing and storage scheme for RDF data that relies on the underlying graph schema of the data, as well as query optimization algorithms that take advantage of the underlying schema in order to accelerate processing of complex SPARQL queries that traditional systems fail to address. Furthermore, we provide a method for logical query optimization by triple pattern reordering, in order to further optimize the query processing tasks commonly adopted by database systems. Finally, we introduce a series of algorithms that aim to efficiently transform and compact the underlying RDF schema in order to optimize both storage and query processing.

Finally, in the third direction (chapter 7), we define several types of relationships for multidimensional RDF data cubes, and we propose a series of computational algorithms that target efficient retrieval of these relationships. Extensive experimental evaluations of our methods indicate significant performance improvements with respect to the state of the art.

Περίληψη

Η παρούσα διδακτορική διατριβή πραγματεύεται θέματα και προβλήματα διαχείρησης δεδομένων που προκύπτουν εντός του Σημασιολογικού Ιστού και εστιάζει στο μοντέλο Resource Description Framework (RDF) και τη γλώσσα επερωτήσεων SPARQL. Σε αυτο το πλαίσιο ακολουθούνται τρεις ερευνητικές κατευθύνσεις, συγκεκριμένα (i) η διαχείρηση εξελισσόμενων RDF δεδομένων, (ii) η αποθήκευση, ευρετηρίαση και βελτιστοποίηση περίπλοκων επερωτήσεων σε συστήματα βάσεων RDF/SPARQL, και (iii) η αποδοτική και κλιμακώσιμη ανάκτηση πληροφορίας από σύνολα πολυδιάστατων RDF δεδομένων. Παρουσιάζονται αποδοτικές και κλιμακώσιμες μέθοδοι, εστιαζόμενες σε συγκεκριμένα προβλήματα των προαναφερθείσων κατευθύνσεων, με τελικό σκοπό να προταθούν προοδευτικές εξελίξεις στην αιχμή της έρευνας.

Στην πρώτη κατεύθυνση, και συγκεκριμε΄να στα κεφάλαια 2 και 3, μελετάται το πρόβλημα της αναπαράστασης, αποθήκευσης και επερώτησης εξελισσόμενων RDF δεδομένων. Υπό αυτό το πρίσμα, προτείνεται ένα νέο μοντέλο δεδομένων και μια νέα γλώσσα επερωτήσεων, στοχεύοντας στην αναπαράσταση της εξέλιξης σε περιστάσεις ετερογενών πεδίων πληροφορίας. Ακολούθως, προτείνεται μία νέα μέθοδος παραγωγής συνθετικών εξελισσόμενων RDF δεδομένων, η οποία στοχεύει στην καλύτερη αξιολόγηση συστημάτων διαχείρησης εκδόσεων (versioning).

Στη δεύτερη κατεύθυνση, και συγκεκριμένα στα κεφάλαια 4, 5 και 6, αντιμετωπίζεται το πρόβλημα της ευρετηρίασης και της αποτίμησης ερωτημάτων, εστιάζοντας συγκεκριμένα σε ερωτήματα βαρέως φόρτου εργασίας σε ημι-δομημένα σύνολα δεδομένων RDF. Υπό αυτό το πρίσμα, προτείνεται μία νέα μέθοδος ευρετηρίασης και αποθήκευσης δεδομένων RDF, η οποία βασίζεται στην ανάκτηση του υποκείμενου σχήματος των δεδομένων, καθώς και νέοι αλγόριθμοι αποτίμησης ερωτημάτων SPARQL που εκμεταλλεύονται το υποκείμενο σχήμα ώστε να βοηθήσουν την αποδοτική και ταχεία αποτίμηση περίπλοκων ερωτημάτων, όπου τα υπάρχοντα συστήματα παρουσιάζουν προβλήματα. Επιπροσθέτως, προτείνεται μια νέα μέθοδος λογικής βελτιστοποίησης βασιζόμενη στην αναπροσαρμογή της σειράς αποτίμησης των τριπλετών (triple pattern reordering). Τέλος, παρουσιάζεται μια σειρά από τεχνικές που στοχεύουν στην σύμπτυξη του υποκείμενου σχήματος με σκοπό την περαιτέρω βελτιστοποίηση της διαδικασίας αποτίμησης.

Τέλος, στην τρίτη κατεύθυνση, και συγκεκριμένα στο κεφάλαιο 7, ορίζεται μια σειρά από τύπους συσχετίσεων μεταξύ δεδομένων σε πολυδιάστατα σύνολα κύβων RDF, και προτείνεται μια σειρά από υπολογιστικές μεθόδους και αλγορίθμους που στοχεύουν στην ταχεία και αποδοτική ανάκτηση αυτών των συσχετίσεων. Η αξιολόγηση των μεθόδων, μέσα από μία εκτεταμένη πειραματική διαδικασία, υποδεικνύει ότι οι προτεινόμενες μέθοδοι προσφέρουν σημαντικά πλεονεκτήματα απόδοσης σε σχέση με την τρέχουσα ερευνητική αιχμή.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Over the recent years, the World Wide Web has been established as a vast source of data from diverse domains, such as biology, statistics, finance, and health, collectively called the *Web of Data*. It consists of an increasing quantity of scientific, corporate, government and crowd-sourced data that are being published and interlinked across disparate sites and sources. A large amount of these data are published in the form of Linked Open Data (LOD). The standard way of modelling LOD is the Resource Description Framework (RDF)[Con14], a recommendation of the World Wide Web Consortium (W3C). In essence, RDF is a graph data model that supports modelling facts about entities in a simple triple format consisting of a *subject*, a *predicate* and an *object*, leading to rich and descriptive directed graphs with semantically labelled edges. In this context, graph nodes represent tangible and intangible entities that are identified uniquely by Uniform Resource Identifiers (URIs), this way defining a common grounds amongst remote agents to publish inherently interlinked datasets. The standard recommendation for querying RDF data is SPARQL[PS+06], a graph query language tailored around the specificities of the RDF model.

Recent advances in data-aware practices, such as data interlinking between heterogeneous sources and data visualization, have a huge potential to create insights and additional value across several sectors. As these data become larger and wider in range, coverage and structural versatility, complex challenges and problems start to emerge. In this thesis, we study the issues and challenges that stem from this eruption in web-scale data, centering on the case of RDF data, and focusing on three main directions, namely (i) *managing*, (ii) *querying*, and (iii) *analysing* large amounts of RDF data. Our main aim is to provide efficient algorithms, methods and techniques that advance the state of the art in a representative set of problems that stems from each of these directions.

The first direction is addressed in chapters 2 and 3 and is concerned with evolving RDF datasets, i.e., datasets that are temporally dynamic and change over time. The Semantic Web relies on the reuse of common resources, vocabularies and ontologies as a shared means of communication between different data sources and software agents. However, little attention has been given to the long-term accessibility and usability of these types of resources in the Data Web. Specifically, linked open datasets are subject to frequent factual or structural changes, which are often performed under no centralized administration. This can eventually lead to inconsistencies, broken links and outdated definitions across interlinked sources. Preservation and sustainable accessibility need to be addressed into a unified framework that by definition takes into account diverse issues such as change representation, provenance tracking and temporal querying. Furthermore, recent pursuits of this direction call for the introduction of appropriate benchmarking models and frameworks, in order to allow interested parties to assess and evaluate the performance of the proposed approaches.

The second direction, addressed in chapters 4, 5 and 6, is concerned with the challenges that arise from the schema-generic nature of RDF, which allows users to define custom, loose relationships between data and does not impose structural (i.e., schema) restrictions. Specifically, efficient SPARQL query answering over semi-structured RDF data becomes an issue when traditional and even state of the art query processing systems are called to process long and complex queries on datasets with loosely defined schemas. This is in part due to the generic nature of the underlying indexing and storage techniques, that do not take into account inherent data characteristics such as the implicit structure. State of the art systems perform efficiently on small and simple query patterns, but lose their competitive edge on more complex queries. Focusing on these limitations and studying novel indexing and querying paradigms can be beneficial in RDF database management systems. Furthermore, query optimization techniques in the SPARQL domain are largely adopted from the relational database setting and fail to account for nuances and specificities found in the graph data model. Thus, the need arises for novel algorithms, methods and techniques for storing, indexing and querying RDF data.

The third and last direction, addressed in chapter 7, is concerned with the efficient and scalable analysis of large amounts of Linked Open Data in order to derive useful information. The increasing adoption of RDF as the de facto standard for publishing open data has led the industrial, government, and academic sectors to publish, re-use and extend proprietary data, a large subset of which is in the form of multidimensional data about policies, demographics, socio-economics and health data among others. In this context, remote datasets exhibit implicit and explicit overlaps, this way creating

relationships between data points that lie in remote sources. The large volume of these datasets imposes a significant overhead in data processing tasks for identification and retrieval of such relationships. Thus, efficient methods are needed in order to tackle these issues.

1.1 Contributions

In this thesis, we study the aforementioned directions and present efficient and scalable methods for managing, querying and analyzing RDF data, with the aim to provide targeted research advancements in the state of the art of the series of problems that are discussed herein. Specifically, in the first direction, we propose a novel data model and query language for managing evolving RDF data, as well as a benchmarking framework for Linked Data evolution management systems. In the second direction, we propose a novel indexing and storage scheme for RDF data that relies on the implicit schema of the data, as well as query optimization algorithms and methods over the defined scheme, with the aim to accelerate processing of complex SPARQL queries that traditional systems fail to address. Furthermore, we provide a method for logical query optimization by triple pattern re-ordering, in order to further optimize the query processing tasks commonly adopted by database systems. Finally, in the third direction, we define several types of instance-level relationships for multidimensional RDF data cubes, and we propose a series of computational algorithms that target efficient retrieval of these relationships. Our contributions include the following:

1. The growing availability of open linked datasets has brought forth significant new challenges regarding their proper preservation and the management of evolving information within them. We study the evolution and preservation challenges related to publishing and preserving evolving linked data across time. We discuss the main problems regarding the modelling and querying of dynamically changing datasets, and provide a conceptual model and a query language for modelling and retrieving evolving data along with changes affecting them. We present in details the syntax of the query language and demonstrate its functionality over a real-world use case of evolving linked dataset from the biological domain. This work addresses the problems of the first of the three aforementioned directions. The methods discussed and the results obtained have been published in [Mei+14; MPP15; Mei+16a].

2. Artificial and synthetic data are widely used for benchmarking and evaluating database, storage and query engines. This is usually performed in static contexts

with no evolution in the data. In the context of evolution management, the community lacks systems and tools for benchmarking versioning and change detection approaches. To tackle this issue, we address generation of synthetic, evolving data represented in the RDF model, and we study the requirements and parameters that drive this process. Furthermore, we discuss query workloads in the context of evolution. To this end, we present *EvoGen*, a generator for evolving RDF data, that offers functionality for instance and schema-based evolution, fine-grained change representation between versions as well as adaptive workload generation. This work addresses the problems of the first of the three aforementioned directions. The methods and results have been published in [Mei16; MP16b].

3. SPARQL query execution in state of the art RDF engines is limited by and inherently dependent on the underlying storage and indexing schemes. Existing systems typically store exhaustive permutations of the standard large three-column triples table, with each column representing the *subject*, *predicate*, and *object* of a triple. However, even though the RDF model can give birth to datasets with loosely defined schemas, it is common for an implicit structure to appear in the data. Based on this emerging structure, we introduce a novel indexing and storage scheme for RDF data, that takes advantage of the inherent structure of triples in order to efficiently index and store data. To this end, we define the *Extended Characteristic Set* (ECS), a schema abstraction that classifies triples based on the properties of their subjects and objects, and we propose methods and algorithms for the identification and extraction of ECSs. We show how these can be used to optimize SPARQL query processing, and we implement *axonDB*, an RDF storage and querying engine based on ECS indexing. We perform an experimental evaluation on real world and synthetic datasets and observe that axonDB outperforms the competition by a few orders of magnitude. This work addresses the problems of the second of the three aforementioned directions. The algorithms, methods and results are published in [Mei+17; MP16a].

4. SPARQL query optimization relies on the design and execution of query plans that involve reordering triple patterns, in the hopes of minimizing cardinality of intermediate results. In practice, this is not always effective, as many existing systems succeed in certain types of query patterns and fail in others. This kind of trade-off is often a derivative of the algorithms behind query planning. To this end, we introduce a novel join reordering approach that translates a query into a multidimensional vector space and performs distance-based optimization by taking into account the relative differences between the triple patterns. Preliminary experiments

on synthetic data show that our algorithm consistently outperforms established methodologies, providing better plans for many different types of query patterns. This work addresses the problems of the second of the three aforementioned directions. The methods discussed and the results obtained have been published in [MP17].

5. Characteristic sets (CS) organize RDF triples based on the set of properties characterizing their subject nodes. This concept is recently used in indexing techniques, as it can capture the implicit schema of RDF data. While most CS-based approaches yield significant improvements in space and query performance, they fail to perform well in the presence of schema heterogeneity, i.e., when the number of CSs becomes very large, resulting in a highly partitioned data organization. This has become evident in the aforementioned work on ECS indexing. In this thesis, we addressed this problem by introducing a novel method for merging CSs based on their underlying hierarchical structure. To this end, we employ a hierarchical lattice to capture the ancestral relationships between CSs, identifying dense CSs in the process and merging dense CSs with their ancestors. The resulting schema is more compact and efficient, both storage-wise and from the perspective of query processing, as the size of the CSs as well as the links between them are reduced. We implemented our approach on top of a relational backbone, where each merged CS is stored in a relational table, and we performed an extensive experimental study to evaluate the performance and impact of merging to the storage and querying of RDF datasets, indicating significant improvements. This work addresses the problems of the second of the three aforementioned directions.

6. The increasing availability of diverse multidimensional data on the web has led to the creation and adoption of common vocabularies and practices that facilitate sharing, aggregating and reusing data from remote origins. One prominent example in the Web of Data is the RDF Data Cube vocabulary, which has recently attracted great attention from the industrial, government and academic sectors as the de facto representational model for publishing open multidimensional data. As a result, different datasets share terms from common code lists and hierarchies, this way creating an implicit relatedness between independent sources. Identifying and analyzing relationships between disparate data sources is a major prerequisite for enabling traditional business analytics at the web scale. However, discovery of instance-level relationships between datasets becomes a computationally costly procedure, as typically all pairs of records must be compared. In this context, we define three types of relationships between multidimensional observations, namely

full containment, *partial containment* and *complementarity*, and we propose four methods for efficient and scalable computation of these relationships. We conduct an extensive experimental evaluation over both real and synthetic datasets, comparing with traditional query-based and inference-based alternatives, and we show how our methods provide efficient and scalable solutions. This work addresses the problems of the third of the three aforementioned directions. The results of this work have been published in [MP14; Mei+16b; Mei+18].

1.2 Thesis Outline

The remainder of this thesis is structured as follows. Chapter 2 presents the data model and query language for managing and preserving evolving data on the web. Chapter 3 discusses the requirements and presents a framework for benchmarking data management systems for evolving RDF. Chapter 4 presents a novel graph indexing and storage scheme, as well as query optimization algorithms and methods for RDF and SPARQL. Chapter 5 proposes a method for logical SPARQL query optimization that is based on triple reordering. Chapter 6 presents novel methods and optimizations for discovering the underlying schema of semi-structured RDF data, and taking advantage of the hierarchical similarities in the discovered schema in order to optimize storage, indexing and query processing. Chapter 7 defines three relationships for multidimensional RDF observations, and proposes several efficient and scalable algorithms for the computation of these relationships. Finally, chapter 8 concludes the thesis.

Chapter 2

A Query Language for Multi-version Data Web Archives

2.1 Introduction

The Data Web consists of an increasing quantity of scientific, corporate, government and crowd-sourced data being published and interlinked across disparate sites on the web, usually in the form of Linked Open Data (LOD). The standard way of modelling LOD is the Resource Description Framework (RDF)[Con14], which is a W3C recommendation. RDF supports the modelling of facts about entities in a simple triple format consisting of a *subject*, a *predicate* and an *object*. Entities are identified by their Uniform Resource Identifiers (URIs), which are also referred to as Internationalized Resource Identifiers (IRIs). Collections of triples form directed labelled graphs of nodes connected to other nodes or literals in semantically meaningful ways. Furthermore, the standard recommendation for querying RDF datasets is SPARQL[PS+06], which is essentially a graph query language. Because RDF is generic enough to enable users to define custom, loose relationships between data, it is not trivial to represent more complex meta-correlations, enable annotations in data at the triple level, assign context, model changes and so on. Data-aware practices, such as data interlinking between heterogeneous sources and data visualization, have a huge potential to create insights and additional value across several sectors, however little attention has been given to the long-term accessibility and usability of open datasets in the Data Web. Linked open datasets are subject to frequent changes in the encoded facts, in their structure, or the data collection process itself. Most changes are performed and managed under no centralized administration, eventually inducing several inconsistencies across interlinked datasets. LOD should be preserved by keeping them constantly accessible and integrated into a well-designed framework for evolving datasets that offers functionality for versioning, provenance tracking, change detection

and quality control while at the same time provides efficient ways for querying the data both statically and across time.

Most of the challenges related to the management of LOD evolution stem from the decentralized nature of the publication, curation and evolution of interdependent datasets, with rich semantics and structural constraints, across multiple disparate sites. Traditional database versioning assumes that data and evolution management take place within well-defined environments where change operations and data dependencies can be monitored and handled. On the other hand, web and digital preservation techniques assume that preservation subjects, such as web pages, are plain digital assets that are collected (usually via a crawling mechanism), time stamped and archived for future reference. In contrast to these two approaches, the Data Web poses new requirements for the management of evolution [Mei+13; Pap13]. Observe Figure 2.1 where an example from the biological domain is presented. The Experimental Factor Ontology (EFO)[Mal+10] is an ontology that combines parts of several life science ontologies, including anatomy, dis-ease and chemical compounds. Its purpose is to enable annotation, analysis and visualization of data related to experiments of the European Bioinformatics Institute[1]. In the figure, a URI that represents a Cell Line class changes between two consecutive versions and becomes obsolete. EFO entities are published in LOD format, enabling other sites to reference and interlink with them. EFO is regularly updated and new versions are published on the web, usually overwriting previous ones. In this context, several interesting problems and challenges arise related to long-term preservation and accessibility of evolving LOD datasets:

Modelling evolving datasets. LOD datasets are evolving entities for which additional constraints may hold related to the way data is published, and evolve as dictated by domain-specific, complex changes. This calls for appropriate modelling methods for preserving across time a multitude of dimensions related to the internal structure of a dataset, its content and semantics as well as the context of its publication. Preservation should exhibit format-independence, data traceability and reproducibility and a common representation for data that originate from different models. Reference schemes and appropriate URIs must be properly assigned such that unique identification and resolution is achieved across different sites, and most importantly across time. Provenance metadata can capture dataset lineage from the dataset level to the record level. Distributed replication of LOD enhanced with temporal and provenance annotations can enable long-term availability and trust.

[1] http://www.ebi.ac.uk

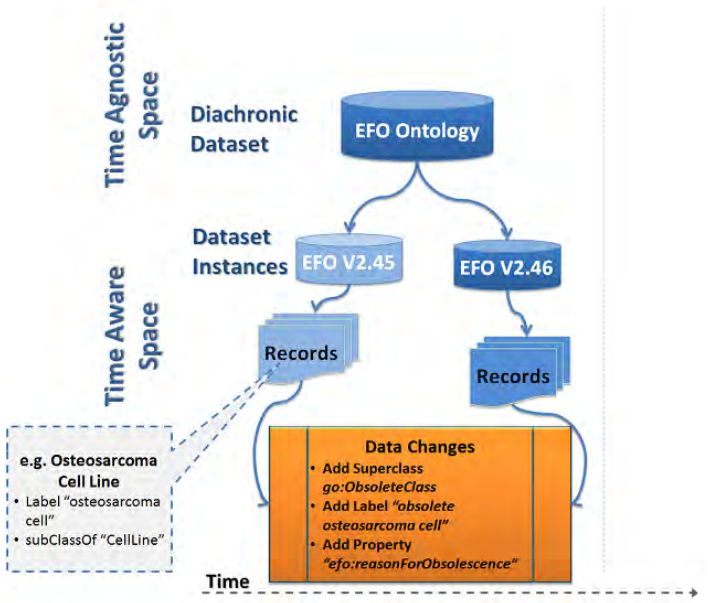


Figure 2.1: Evolution of a Cell Line between versions 2.45 and 2.46 of the Experimental Factor Ontology.

Change management. Changes can occur at different granularity levels. At the dataset level, datasets are added, republished, or even removed, without versioning or preservation control; at the schema level, the structure may change calling for repair and validation on new versions; finally, at the instance level data resources and facts are added, deleted or updated. Discovering changes and representing them as first class citizens with structural, semantic, temporal and provenance information is vital in various tasks such as the synchronization of autonomously developed LOD versions, or visualizing the evolution history of a particular dataset [PSG13]. A unified framework that deals with evolution must be able to allow change management as a dimension of the dataset's evolution.

Longitudinal accessibility and querying. LOD preservation mechanisms must enable the long-term accessibility of datasets and their meaningful exploration over time. Datasets with different time and schema constraints coexist and must be uniformly accessed, retrieved and combined. Longitudinal query capabilities must be offered such that data consumers can answer several types of queries, within a version or across sets of versions. Querying must take place (i) across time, (ii) across datasets and (iii) across different levels of granularity of evolving things. Considering the above, the benefits of managing evolving LOD datasets can be placed into two categories, namely quality control and data analysis. Data evolution provides valuable insights on the dynamics of the data, their domains and the operational aspects of the communities they are found in, while tracking the history of, and maintaining proper metadata of data objects across time enables better interoperability, trust and data quality.

To address these challenges, in this chapter we propose a conceptual model and a query language for evolving Linked Open Datasets, as well as an archiving system that implements these proposals. At the basis of the system lies a conceptual model, called *DIACHRON* model, that captures structural concepts like datasets and their schemas, semantics like web resources, their properties and links between them as well as changes occurring on these concepts in different granularity levels. In the same time, our approach models in a uniform way both time-aware (*evolving*) and time-agnostic (*diachronic*) concepts, representing the interconnections between them. Based on this model, a query language is designed that specifically caters for the model's inherent characteristics and takes advantage of appropriate abstraction levels, thus making the user avoid complicated, implementation-dependent queries. The query language is designed as an extension of SPARQL, specific to the DIACHRON model, that tackles the duality of data (evolving vs. diachronic objects) in order to provide a query mechanism with the ability to correlate source data with changes, annotations at various levels and other kinds of DIACHRON related metadata across time. Finally, we implement these as an archiving framework capable of storing and making available in the long term evolving LOD datasets.

To summarize, this chapter provides the following contributions:

1. We formally define the DIACHRON data model, a conceptual model for the representation of datasets and their evolving aspects, such as their structural, semantic, and metadata evolution. Specifically, we provide entities for modelling data that change through time in multi-version contexts, where their schema, data and metadata exhibit changes in a multitude of levels, from tuples, to collections of datasets.

2. We propose and formally define the *DIACHRON Query Language* as a means to enable retrieval of data and metadata across versions and datasets. The proposed query language enables querying of evolving entities across time, along with the structural elements of the entities (e.g. the reified triples) as well as the changes affecting them.

3. We provide an implementation of an archiving system that uses the DIACHRON model and implements the DIACHRON Query Language as an extension of SPARQL, and we perform experimental evaluation in terms of usability and performance on real-world datasets from the life sciences domain.

The rest of this chapter is outlined as follows. In section 2 we discuss related work, in section 3 we present the DIACHRON data model, in section 4 we present the DIACHRON query language, in section 5 we describe our implementation of an archive that uses

the proposed model and query language, while in section 6 we perform experimental evaluation. Finally, section 7 concludes the chapter.

2.2 Related Work

Managing LOD evolution is a multi-faceted problem that consists of versioning, efficient archiving, change representation, change detection, model abstraction and provenance tracking, among others. Work has been done in most of these fields individually, but few approaches have regarded the issue as a singular problem of many interdependencies, less so in the case of the Data Web, where datasets evolve independently, often in non-centralized ways, while citing and using one another. Versioning for LOD in the context of complete systems or frameworks has been addressed in [AH06; Cic+13; Gra11; HBW15; ILK12; Kei+11; Bro+11; PJS11; Som+10]. However, these approaches address a subset of the problems discussed as will be discussed.

Ontology or schema based approaches have been proposed in [BSP11; Cic+13; Kei+11] with the most prominent example being the PAV ontology [Cic+13], a specialization of the W3C recommended PROV ontology [Leb+13] for modelling provenance. In our work, we consider the representation of provenance as an orthogonal problem, in the sense that any model for representing metadata annotations can be used in conjunction with our work.

As far as querying is concerned, work has been done in extending SPARQL with temporal capabilities [BSP11; KK10; Lop+10; PJS11]. Contrary to our approach, in [PJS11] no explicit data model is proposed, instead temporal information is used to separate triples in named graphs. Incorporation of annotations and provenance on the query side has been approached in [Lop+10] where triple annotations serve as context in the proposed SPARQL extension. This approach however does not differentiate between types of annotations, and is limited to treating annotations as singular tags of triples. In [BSP11] an ontology-based approach is followed where temporal reasoning capabilities are provided to OWL-2.0 and SPARQL is extended to cater for the temporal dimension. While [BSP11] extends an existing RDF query language with temporal reasoning, it limits its functionality in this context and does not deal with evolution of structural concepts such as datasets, tuples, or individual triples. In contrast, our approach aims at providing querying capabilities for both the semantic and the structural elements of an evolving dataset. In [KKK12] a triple store is implemented that incorporates spatiotemporal querying by utilizing the SPARQL extensions proposed in [PJS11]. These approaches are specifically tuned to address temporal or spatiotemporal querying in RDF data, and

do not rely on conceptual models for representing in a uniform way semantically rich evolving datasets, changes, and metadata through time.

In [Som+10] an approach is presented that builds on the Memento framework [Som+09], an extension of HTTP to include a traversable and queryable temporal dimension, adapted for LOD purposes. Non-changing, time-independent URIs are employed for current state identification. Dereferencing past versions of resources is done with temporal content negotiation, an HTTP extension. We draw from this work the notion of time-independent URIs for current state identification, however, we are not interested in providing functionality at the HTTP level; instead we take on a data-centric rather than a document-centric approach for deep archiving and preservation of large datasets.

In [WL02], the authors tackle the problem of version management for XML documents by using deltas to capture differences between sequential versions and use deltas as edit scripts to yield sequential versions. The introduced space redundancy is compensated by the query efficiency of storing complete deltas rather than compressed deltas. They go on to define change detection as the computation of non-empty deltas and they argue that past version retrieval can be achieved by storing all complete deltas as well as a number of complete intermediate versions, finding the bounding versions of the desired ones and applying their corresponding deltas. Finally, they use a query language based on XQuery in order to enable longitudinal querying and they provide tag indices for each edit operation for faster delta application. While this approach deals with longitudinal querying by extending an existing standard, similar to our approach, they do not provide support for more complex semantic changes, or placeholders for capturing the evolution of other entity types, such as metadata and provenance annotations.

In [Bun+04], the authors propose a method for archiving scientific data from XML documents. The approach targets individual elements in the DOM tree of an XML document, rather than the whole versions themselves. They use time stamping in order to differentiate between the states of a particular element in different time intervals and they store each element only once in the archive. The timestamps are pushed down to the children of an element in order to reflect the changes at the corresponding level of the tree, an approach also followed in [PSG13]. Our approach is inspired by the hierarchical attribution of time and we adopt this model and partially adapt it to the case of RDF. Moreover, we extend this hierarchical attribution to generic metadata annotations instead of strictly temporal.

In [UVTH10] the authors study the change frequency of LOD sources and the implications on dataset dynamics. They differentiate between the document-centric and the

entity-centric perspectives of change dynamics, the latter further divided into the entity-per-document and global entity notions. We partially adopt this distinction in our work, as will be described further on. Specifically, we introduce a conceptual model that differentiates between entity types that represent both the structural aspects of a dataset, and the semantic ones.

SemVersion [VG06] is a system that computes the semantic differences as well as the structural differences between versions of the same graph but is limited to RDFS expressiveness. DSNotify [PH10] is an approach to deal with dataset dynamics in distributed LD. The authors identify several levels for the requirements of change dynamics, namely, vocabularies for describing dynamics, vocabularies for representing changes, protocols for change propagation and algorithms and applications for change detection. It implements a change detection framework which incorporates these points in a unified functionality scheme, having as main motivation the problem of link maintenance. Both these approaches only support full materialization of datasets, contrary to our approach that supports a hybrid model of storing datasets and semantic deltas. Furthermore, contrary to our approach, they do not deal with querying over time, changes and metadata.

Our approach differentiates itself by considering versioning, annotating, change management, and dataset heterogeneity as necessary components of an evolving dataset, and are thus tackled together. Furthermore, most of the work presented in this section addresses the temporal aspect of evolution in datasets, instead we chose to consider temporality as an inherent characteristic of versioning. It is trivial to explicitly create temporal operators for DIACHRON QL by evaluating datasets over their temporal metadata and translating temporal operators to version-based operators such as AT VERSION or BETWEEN VERSIONS.

2.3 An archive model for evolving datasets

Our modelling approach supports a format-independent archiving mechanism that maintains syntactic integrity by making sure that the original datasets are reproducible and at the same time takes advantage of information-rich content in these datasets. Format-independence enables different source models (e.g. relational, multidimensional, ontological) to be transformed to a common RDF representation, uniformly annotated with temporal and provenance information.

The DIACHRON model provides the basis for defining semantically richer entities that evolve with respect to their source datasets' history. At the core of the model lies the notion of the *evolving entity*, which captures both structural and semantic constructs of

a dataset and acts as a common placeholder for provenance, temporal, and other types of metadata.

Evolving entities are identifiable and citable objects. These entities all share a common ancestor, the *Diachronic Entity*, which allows the aforementioned requirements to be addressed on different levels. The different types of entities in the DIACHRON model and their interactions can be seen in Figure 2.2 and Figure 2.3. Specifically, Figure 2.2 shows a class diagram that describes the relationships between concepts in the DIACHRON model, while Figure 2.3 provides an aggregated space where concepts are partitioned in time-aware vs time-agnostic, and data (non-curated) vs curated information space. There, example instantiations between the different concepts in the data model are presented. An example drawn from the EFO ontology can be seen in Figure 2.4. The entities of the model are described in the following.

2.3.1 Diachronic datasets and dataset instantiations

Diachronic datasets are conceptual entities that represent a particular dataset from a time-agnostic point of view, which in turn is linked to its temporal instantiations or versions. Furthermore, diachronic dataset metadata comprise information that is not subject to change, such as diachronic dataset identifiers. These identifiers serve as ways to refer to the datasets in a time and/or version unaware fashion (i.e. diachronic citations). On the other hand, dataset instantiations define temporal versions of diachronic datasets, holding information on how and when a particular dataset was relevant and actively used.

Definition 1 ***Diachronic Dataset***

A diachronic dataset $\mathbf{D}$ is defined as a set $\mathbf{d}, \mathbf{m}$ where $\mathbf{d}$ is a set of dataset versions $d_1, \ldots, d_n$ and $\mathbf{m}$ is a collection of metadata annotations associated with $\mathbf{D}$. Diachronic datasets usually carry housekeeping information about creation, modification etc. in the archiving context, which is included in $\mathbf{m}$. In Figure 2.4, *ex:EFO* represents a diachronic dataset that describes the EFO ontology through time. The same example entity can be seen in Listing 7.1 in an example RDF serialization.

Definition 2 ***Dataset Version***

A dataset version, or instantiation, d is defined as a set $\{\mathbf{R}, \mathbf{S}, \mathbf{t}, \mathbf{m}\}$ where $\mathbf{R}$ is a record set and $\mathbf{S}$ is a schema set, while $\mathbf{t}$ is a collection of temporal information associated with d, and $\mathbf{m}$ is a collection of non-temporal annotations associated with d. In Figure 2.4, instantiations of *ex:EFO* can be seen as versions 2.35 and 2.36. These can also be seen in Listing 7.1 in their serialized form.

2.3.2 Record sets and Schema Sets

Record sets are collections of data entries (e.g. tuples, triples) over a given subject/primary key within a particular dataset instantiation. Given a record set and the dataset's metadata information, the dataset instantiation can be queried and reproduced in its original form. Similarly, a *schema set* contains all schema-related entities (e.g. table definitions in the relational case, ontology entities in the ontological case etc.). Keeping data objects separate from schema objects makes versions interpretable by different schemata (e.g. new schema on old data or vice versa).

Definition 3 ***Record Set***

A record set $\mathbf{R}$ is defined as a set $\{\mathbf{r}, \mathbf{m}\}$, where $\mathbf{r}$ is a set of records $\{r_1, \ldots, r_n\}$ and $\mathbf{m}$ is a collection of associated metadata for $\mathbf{R}$. A record set $\mathbf{R}$ is always enclosed in the scope of a dataset instantiation d, as discussed in Definition 2. The record set for version 2.35 of the EFO ontology can be seen in Figure 2.4 and Listing 7.1 as $ex : recordSet_2.35$.

Definition 4 ***Schema Set***

A schema set $\mathbf{S}$ is defined as a set $\{\mathbf{e}, \mathbf{m}\}$, where $\mathbf{e}$ is a set of schema objects $\{e_1, \ldots, e_n\}$ and $\mathbf{m}$ is a collection of associated metadata for $\mathbf{S}$. A schema set $\mathbf{S}$ is always enclosed in the scope of a dataset instantiation d, as discussed in Definition 2. The schema set for version 2.35 of the EFO ontology can be seen in Figure 2.4 as $ex : schemaSet_2.35$.

2.3.3 Data and Schema Objects

Data objects consist of *records* and *record attributes*. A record represents a most granular data entry about a particular evolving entity. Records are uniquely identified in order to make record-level annotation feasible in order to attribute provenance, temporality and changes on them. A record serves as a container of one or more record attributes. Every data record is broken down to assertions (facts) that can be expressed as RDF triples. In this sense, a record reifies the predicate-object pairs for a fixed subject. These predicate-object pairs are called record attributes. For instance, a tuple from a relational table is considered to be a record describing the tuple's primary key, with each relational attribute being a record attribute. In [Mei+14; Mei+13] we describe in details how data records from relational, multidimensional and RDF models can be mapped to data objects in our model. Schema objects represent the schema-related entities of the archived datasets given the dataset's source model. For instance, the classes along with their class restrictions of an ontology, the properties and their definitions (domains, ranges, meta properties depending on the expressivity) are modelled as schema objects. Similarly to

data objects, the goal is to provide a reusable modelling mechanism for identifying and referring to schema elements and their evolution across datasets. In this way, schema evolution is captured by annotating schema elements with schema changes.

Definition 5 ***Records***

A record r is defined as a set $\{s, \mathbf{a}, \mathbf{m}\}$ where s is the identifier, or subject, of r, $\mathbf{a}$ is a collection of record attributes $\{a_1, \ldots, a_n\}$ and $\mathbf{m}$ is a collection of associated metadata for r. In Figure 2.4, an example record can be seen as a part of $ex : recordSet_2.35$. A record describing the experimental factor *EFO_0000887* can be seen in Listing 7.1.

Definition 6 ***Record Attributes***

A record attribute $\mathbf{a}$ is defined as a set $\{p, o, \mathbf{m}\}$ where p and o are predicate-object pairs and $mathbfm$ is a collection of metadata associated with $\mathbf{a}$. In Figure 2.4, the record attributes for version 2.35 are the direct children of the aforementioned record. In Listing 7.1, two record attributes that describe the label of *EFO_0000887* are shown, with the use of the $rdfs : label$ property.

2.3.4 Diachronic Resources and Resource Instantiations

Similarly to diachronic datasets, a *diachronic resource* represents a time-agnostic information entity. The *resource instantiation* captures the resource evolution across time and its realization over a versioned dataset's records. The definition of a resource consists of two parts; the resource identification definition comprises of the way an instantiated resource is identified within the archive. The resource description definition provides the way a resource is evaluated over the records of a particular dataset instantiation. Resources can be versatile in nature across datasets and data formats. For example, given an ontology and its instantiation, each class instance can describe a resource identified by the respective URI. Given a table of employees in a relational database, a resource in this sense can be a particular employee identified by his primary key. Finally, in a multidimensional dataset, a resource can be a specific observation identified by the values of the constituent dimensions. More complex definitions of resources are allowed and, in fact, encouraged for capturing more high-level, curator specific semantics of evolution and dataset dynamics.

Definition 7 ***Diachronic Resources***

A diachronic resource $\mathbf{E}$ is defined as a set $\{E, q, \mathbf{m}\}$ where E is a set of resource instantiations $\{E_1, \ldots, E_n\}$, q is a description definition and $\mathbf{m}$ is a collection of metadata associated with E. The description definition q is a DIACHRON query.

Definition 8 ***Diachronic Resource Instantiations***

A resource instantiation E is defined as a set $\{g, t, \mathbf{m}\}$ where g is a set of data records $\{r_1, \ldots, r_n\}$, t is the temporal information associated with E and $\mathbf{m}$ is a collection of metadata associated with resource E.

2.3.5 Change Sets

Changes are compiled in *Change Sets* between two dataset instantiations of a diachronic dataset. These are comprised of changes between record sets, changes between schemata and changes between resource instantiations of the two datasets under comparison.

Definition 9 ***Change Sets***

A change set $\mathbf{C}$ is defined as a set $\{c, \mathbf{m}\}$ where c is a set of changes $\{c_1, \ldots, c_n\}$ and $\mathbf{m}$ is a collection of metadata associated with $\mathbf{C}$. The change set between versions 2.35 and 2.36 of the EFO ontology can be seen in Figure 2.4 as $ex : changeSet_2.35 - 2.36$. Furthermore, the same change set can be seen in Listing 7.1 in a serialized form.

The proposed data model provides a conceptual way of uniformly representing low-level and high-level evolving entities. Within the context of our model, an evolving entity is a dataset instantiation (affected by changes in its schema and contents), a schema object, a data object or finally a resource instantiation object. This gives us a uniform way to model evolution and annotate entities at different levels of granularities with information related to the changes affecting them. Furthermore, it enables us to enrich evolving entities with metadata related to the way these entities are published on remote sites and collected in the archive, such as provenance information, quality and trust.

2.4 The DIACHRON Query Language

2.4.1 Requirements and Overview

The DIACHRON model provides metadata placeholders in different granularities, from the dataset to the record level. In this section, we motivate the need for an appropriate query language that exploits the specificities of the data model and provides ways to achieve the following:

- *Dataset and version listing*: Retrieve lists of datasets stored in the archive, as well as lists of the available versions of a given dataset. These can either be exhaustive or filtered based on temporal, provenance or other metadata criteria.

```
ex:EFO rdf:type diachron:DiachronicDataset ;
        dcterms:creator "European Bioinformatics Institute" ;
        diachron:hasInstantiation ex:EFO_v2.35 ;
        diachron:hasInstantiation ex:EFO_v2.36 ;
        diachron:hasChangeSet ex:ChangeSet_2.35_2.36 .

ex:EFO_v2.35 rdf:type diachron:Dataset ;
        dcterms:date "2015-01-02"\^\^xsd:date ;
        diachron:hasRecordSet ex:RecordSet_2.35.

ex:EFO_v2.36 rdf:type diachron:Dataset ;
        dcterms:date "2015-02-02"\^\^xsd:date ;
        diachron:hasRecordSet ex:RecordSet_2.36.

ex:RecordSet_2.35 rdf:type diachron:RecordSet ;
        diachron:hasRecord ex:Record_1 .

ex:Record_1 diachron:subject efo:EF\_0000887 ;
        diachron:recordAttribue ex:RecordAttribute_1 .

ex:RecordAttribute_1 diachron:predicate rdfs:label ;
        diachron:object "liver" .

ex:RecordSet_2.36 rdf:type diachron:RecordSet ;
        diachron:hasRecord ex:Record_2 .

ex:Record_2 diachron:subject efo:EFO_0000887 ;
        diachron:recordAttribue ex:RecordAttribute_2 .

ex:RecordAttribute_2 diachron:predicate rdfs:label ;
        diachron:object "LIVER" .

ex:ChangeSet_2.35-2.36 rdf:type diachron:ChangeSet ;
        diachron:oldVersion ex:EFO_v2.35 ;
        diachron:newVersion ex:EFO_v2.36 ;
        diachron:hasChange ex:Change1 .

ex:Change1 rdf:type diachron:LabelModificationChange ;
        diachron:parameter1 ex:RecordAttibute_1 ;
        diachron:parameter2 ex:RecordAttibute_2 .
```

Listing 2.1: Example RDF serialization of a diachronic dataset ex:EFO, two dataset instantiations (versions) *ex:EFO_v.235* and *ex:EFO_v.236*, in their respective record sets *ex:RecordSet_2.35* and *ex:RecordSet_2.36*. The two record sets contain one record about *efo:EFO_0000887*, an original instance of the EFO ontology, which shows how its label changes its capitalization between versions. Note that the prefix *ex* is an example prefix. A change set containing a sample LabelModificationChange can also be seen.

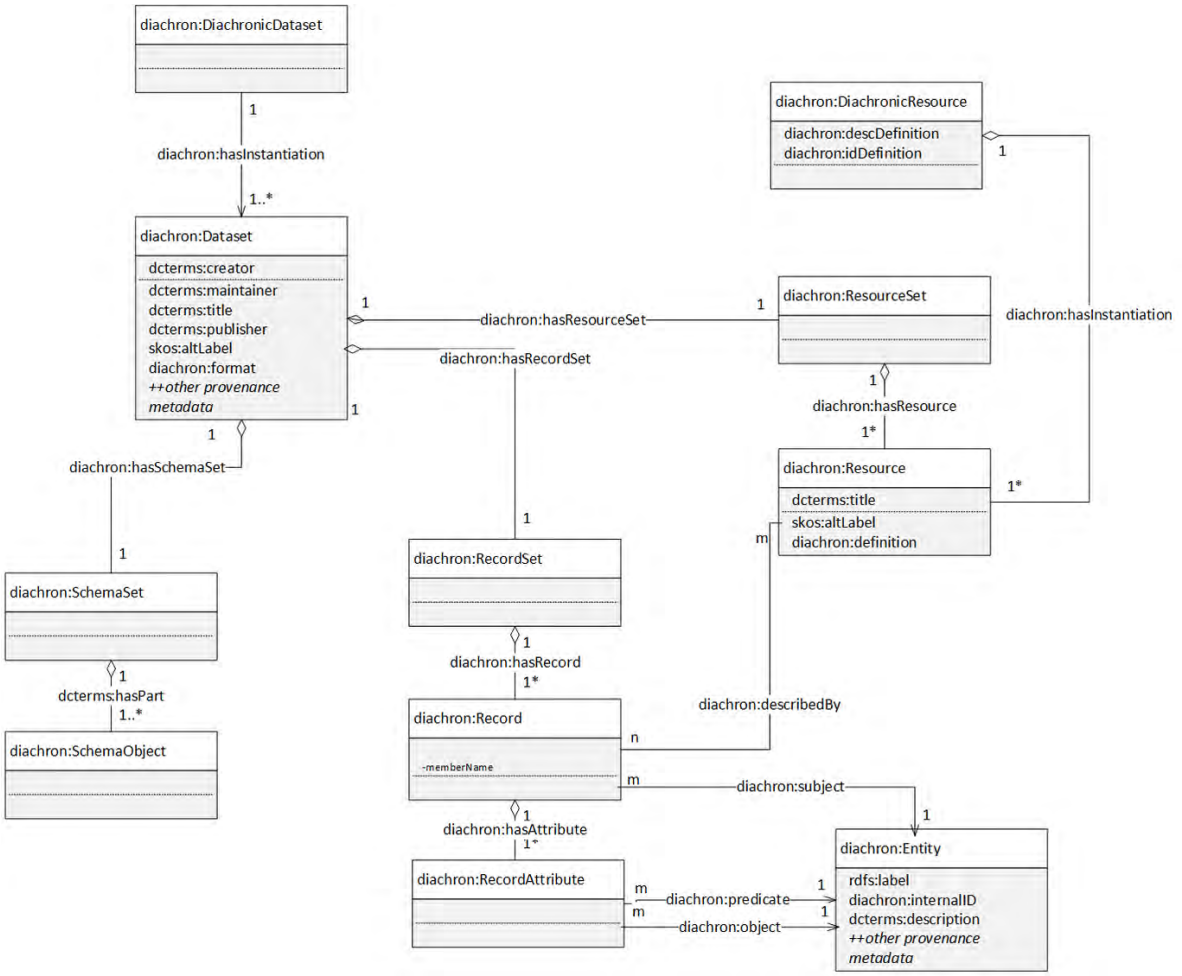


Figure 2.2: Class diagram for the DIACHRON model.

- *Data queries*: Retrieve part(s) of a dataset that match certain criteria.
- *Longitudinal queries*: As above but with the timeline of all types of diachronic entities. Temporal criteria can be applied to limit the timeline (specific versions or time periods), or successive versions.
- *Queries on Changes*: Retrieve changes between two concurrent versions of an entity (dataset, resource etc.). Limit results for specific type of changes, or for a specific part of the data.
- *Mixed Queries on Changes and Data*: Retrieve datasets or parts of datasets that are affected by specific types of changes.

In this section, we propose the *DIACHRON Query Language* (DIACHRON QL), to tackle these requirements, and we discuss its design and implementation as an extension of SPARQL. The basis of the query language is the DIACHRON graph pattern, which, in the context of extending SPARQL, is a specialization of a SPARQL graph pattern,

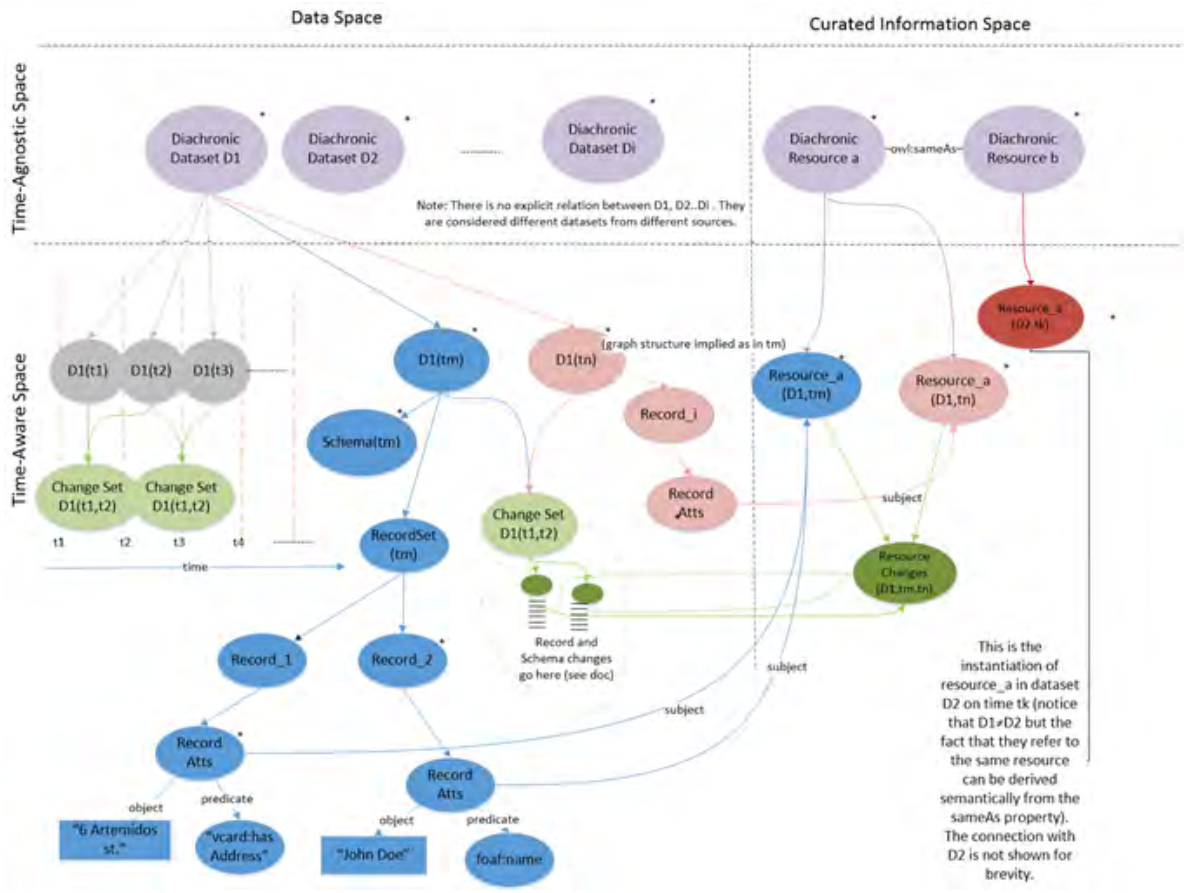


Figure 2.3: The DIACHRON model space.

thus making SPARQL queries valid DIACHRON QL queries. New keywords are defined in order to cover the model's characteristics and allow the user to query archived data intuitively, without the need to know the specificities of the implementation. In plain SPARQL engines, or any other query engine basis, the user would need to know how the DIACHRON model is implemented in the system, and how its entities are mapped to the system's underlying information retrieval engine. With the use of a dedicated query language, we abstract the implementation details to the DIACHRON QL syntax. DIACHRON QL introduces keywords that allow defining the scope of a query with respect to the matched diachronic datasets and their versions, their change sets, or both.

2.4.2 DIACHRON QL basics

Given the above, diachronic datasets, versions and change sets can be bound to variables with the use of DATASET or CHANGES. This is simply done by using variables instead of explicit URIs, inside the query body, i.e. not in a FROM clause. For example, consider the case where we want to retrieve all the information (predicate-object pairs) associated with the protein $efo : EFO_0004626$, and find out what the state of this information is for

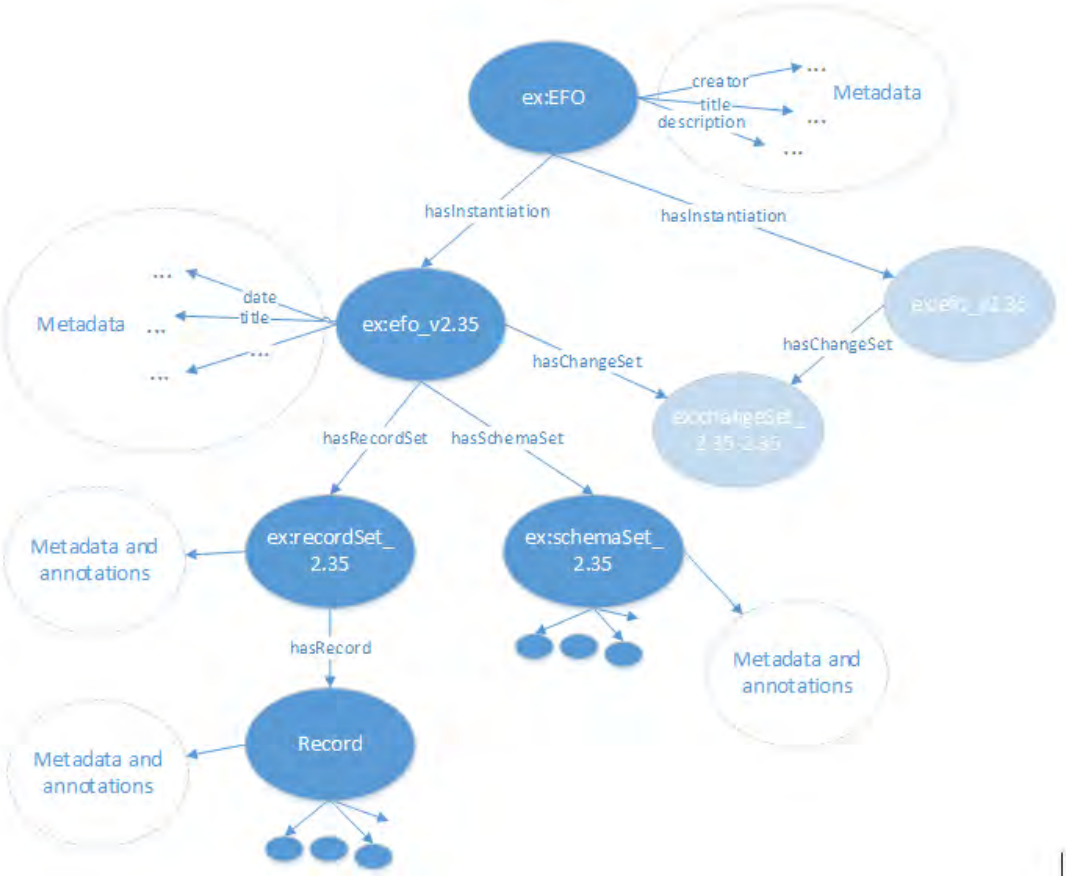


Figure 2.4: An example of a diachronic dataset (ex:EFO) that has two dataset instantiations (versions 2.35 and 2.36). The record and schema sets of version 2.35 can be seen in bold blue, while version 2.36 and the change set that is shared between 2.35 and 2.36 can be seen in pale blue.

all the dataset versions of the EFO ontology it appears in (and what are those versions). That is, the dataset versions as well as the actual information are to be retrieved. In DIACHRON QL this can be written as follows:

```
SELECT ?version ?p ?o WHERE {
        DATASET <EFO> AT VERSION ?version {
                efo:EFO_0004626 ?p ?o
        }
}
```

This will retrieve all versions of EFO joined with predicate-object pairs for the protein *efo:EFO_0004626.* If we want to retrieve the records these predicate-object pairs appear in, without querying for the particular dataset versions. We can retrieve the URIs of the DIACHRON records these triples appear in by modifying the query as follows:

```
SELECT ?rec ?p ?o FROM DATASET <EFO> WHERE {
```

```
        RECORD ?rec {efo:EFO_0004626 ?p ?o}
}
```

With the optional use of the RECATT keyword we can retrieve the URIs of the record attributes of a matched record. The previous query would become:

```
SELECT ?rec ?ra ?p ?o FROM DATASET <EFO> WHERE {
        RECORD ?rec { efo:EFO_0004626
                RECATT ?ra {?p ?o}
        }
}
```

When writing a DIACHRON graph pattern, the query can either contain simple triple patterns, or more verbose constructs that take into account the archive data model and structure. Specifically, the simple triples will match the dereified data, whereas the RECORD and RECATT (abbreviation of *record attribute*) blocks will also take into account a triple's record or record attribute.

This is further exemplified in Figure 2.5 where we show how term and variable use is reflected on the matched graph of a particular reified triple. This way, metadata (e.g. temporal, provenance) of the records and/or record attributes can be queried as well as combined with data queries. It should be noted that in the simplest case where only the data are of interest, the query does not need to include RECORD and RECATT blocks.

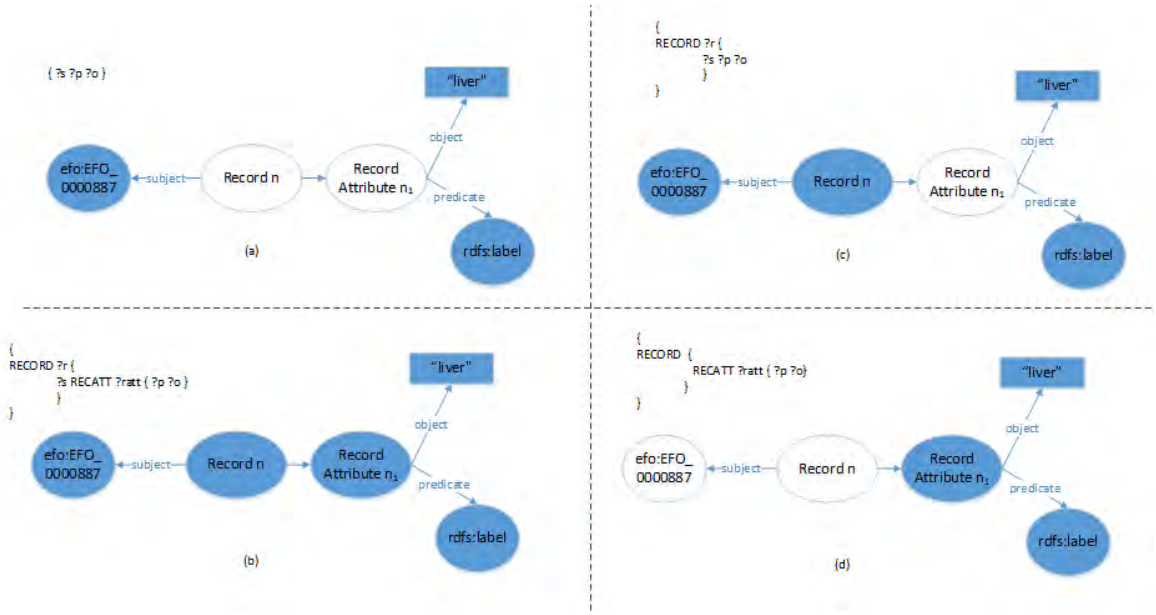


Figure 2.5: (a) matches in a simple triple query, (b) matches a blown-out version of the same query with the RECORD and RECATT terms, selecting both data and structural elements. (c) matches subject, predicate, object and record, (d) matches predicate, object and record attribute.

2.4.3 Query Syntax and Examples

DIACHRON QL clauses are formally described in the following section and an overview of them is presented in Table 2.1 in E-BNF form. In Table 2.2 usage examples are presented for all DIACHRON QL clauses.

```
FROM DATASET <diachronicURI> [[AT VERSION <instantiationURI>]]
```

The FROM DATASET keyword is followed by a URI of a diachronic dataset to declare the dataset scope of the query. If no FROM DATASET is given, then the whole corpus of datasets is queried. The optional AT VERSION keyword limits the selected diachronic dataset to a specific dataset instantiation. No variables can be given in any of the parameters of FROM DATASET AT VERSION.

Table 2.1: The DIACHRON query language syntax in E-BNF.

DiachronQuery :=	'DIACHRON' 'SELECT' ('DISTINCT')? (Var+|'*') *Source_Clause** 'WHERE' *Where_Clause**
Source_Clause :=	('FROM DATASET' <*URI*> ['AT VERSION' <*URI*>] | 'FROM CHANGES' <*URI*> ['BEFORE VERSION' <*URI*> | 'AFTER VERSION' <*URI*> | 'BETWEEN VERSIONS' <*URI*>+2])
Where_Clause :=	(*Diachron_Pattern* *['UNION' Diachron_Pattern]* *['OPTIONAL' Diachron_Pattern])*
Diachron_Pattern :=	(*Source_Pattern Basic_Archive_Graph_Pattern*)
Source_Pattern :=	(('DATASET' <VarOrURI> ['AT VERSION' <VarOrURI>]) | ('CHANGES' <VarOrURI> ['BEFORE VERSION' <VarOrURI>]) | ('CHANGES' <VarOrURI> ['AFTER VERSION' <VarOrURI>]) | ('CHANGES' <VarOrURI> ['BETWEEN VERSIONS' <VarOrURI>+2]))
Basic_Archive_Graph_Pattern :=	'{' *SPARQL_Triples_Block** *Record_Block** *Change_Block** '}'
Record_Block :=	'RECORD' <VarOrURI> '{' <VarOrURI> ((<VarOrURI>+2 '.')*) | ('RECATT' <VarOrURI> '{' <VarOrURI>+2 '}')* '}'
Change_Block :=	'CHANGE' <VarOrURI> '{' (<VarOrURI>+2 '.')* '}'

SPARQL_ Triples_Block :=	As defined in the SPARQL recommendation[2].

```
FROM CHANGES <diachronicURI> [[BETWEEN VERSIONS <version1URI> <
   version2URI> ] || [BEFORE VERSION <versionURI>] || [AFTER
   VERSION <versionURI>]]
```

FROM CHANGES is used to query change sets directly. It is immediately followed by a URI of a diachronic dataset that defines the diachronic dataset to be queried on its changes. If no URI is given, then all existing change sets will be used to match the query body. FROM CHANGES can optionally be used with BETWEEN VERSIONS, BEFORE or AFTER VERSION to limit the scope of the changes.

```
DATASET <URI | ?var> [[AT VERSION <URI | ?var>]] { (query) }
```

The DATASET keyword differs from FROM DATASET in that it is found inside a query body. It is followed by either a URI or variable of a diachronic dataset to declare or bind the scope of the graph. DATASET is inside a WHERE statement and is followed by a graph pattern, on which the dataset restriction is applied. It is optional, meaning that if no DATASET clause is defined, then the whole corpus of datasets will be queried, or the datasets defined in the FROM DATASET clause. The AT VERSION keyword, when applied to a DATASET statement inside a WHERE clause, is used to either define a specific dataset instantiation or bind dataset instantiations to a variable for the graph pattern that follows. However, AT VERSION is optional and if no specific dataset instantiation URI or variable is declared, AT VERSION is omitted. An example of matching both triples and versions can be seen in Figure 2.6.

```
RECORD <record_URI | ?record_var>
        {<subjectURI | ?subject_var > ATTRIBUTE_pattern}
```

RECORD is used inside the body of a graph pattern for querying either a specific DIACHRON record or to match DIACHRON records in the pattern. It is followed by a record URI/variable. If neither of those is declared, the RECORD keyword can be omitted. Following RECORD is a block containing a graph pattern that can either be of SPARQL form, or used in conjunction with the RECATT keyword.

Table 2.2: DIACHRON query language keywords and usage examples.

Keyword	Parameters	Usage example
SELECT	variable list	SELECT ?x, ?y, ?z

[2]http://www.w3.org/TR/sparql11-query/

Table 2.2: DIACHRON query language keywords and usage examples.

Keyword	Parameters	Usage example
FROM DATASET	URI of diachronic dataset	SELECT ?x, ?y, ?z FROM DATASET <efo>
FROM DATASET AT VERSION	URI of dataset instantiation	SELECT ?x, ?y, ?z FROM DATASET <efo> AT VERSION <v1>
FROM CHANGES	URI of diachronic dataset	SELECT ?x, ?y, ?z FROM CHANGES <efo>
FROM CHANGES BETWEEN VERSIONS (params)	URIs of dataset instantiations to define the change scope	SELECT ?x, ?y, ?z FROM CHANGES <efo> BETWEEN VERSIONS <v_m>, <v_n>
FROM CHANGES AFTER / BEFORE VERSION (params)	URI of dataset instantiation to define the start/end of the change scope	SELECT ?x, ?y, ?z FROM CHANGES <efo> AFTER / BEFORE VERSION <v_m>

WHERE { (params) }	DIACHRON patterns	SELECT ?x, ?y, ?z
		FROM DATASET <efo>
		WHERE {
		?x a efo:Protein ; ?y ?z .
		}

DATASET (params)	URI or variable of diachronic dataset	SELECT ?x, ?y
		WHERE {
		DATASET ?x {
		?s a efo:Protein.
		}
		DATASET ?y {
		?s dcterms:creator “EBI”
		}
		}

Table 2.2: DIACHRON query language keywords and usage examples.

Keyword	Parameters	Usage example

DATASET ... **AT VERSION** (params)	URI or variable of dataset instantiation	SELECT ?x, ?y
		WHERE {
		DATASET ?x AT VERSION ?var {
		?s a efo:Protein.
		}
		DATASET ?y AT VERSION <v1> {
		?s dcterms:creator “EBI”
		}
		}

RECORD (params)	URI or variable of DIACHRON record	SELECT ?x, ?r, ?y
		WHERE {
		DATASET ?x AT VERSION ?var {
		RECORD ?r {?s a efo:Protein}
		}
		DATASET ?y AT VERSION <v1> {
		?s dcterms:creator “EBI”
		}
		}

RECATT (params)	URI or variable of a DIACHRON record attribute	SELECT ?var, ?r, ?ra
		WHERE {
		DATASET <efo> AT VERSION ?var {
		RECORD ?r {
		?s RECATT ?ra
		{rdf:type efo:Protein}
		}
		}
		}

CHANGES (params)	URI of diachronic dataset or variable	SELECT ?c, ?param1, ?value1
		WHERE {
		CHANGE ?c {?param1 ?value1 }
		}

CHANGES BETWEEN VERSIONS (params)	URIs of dataset instantiations or variables to define the change scope	SELECT ?v1, ?v2, ?c
		WHERE {
		CHANGES <EFO> BETWEEN VERSIONS ?v1, ?v2 {
		?c rdf:type co:Add_Definition ;
		?p1 [co:param_value ?o3 . rdf:type co:ad_n1] ;
		?p2 [co:param_value ?o4 . rdf:type co:ad_n2]
		}
		}

Table 2.2: DIACHRON query language keywords and usage examples.

Keyword	Parameters	Usage example

CHANGES ... **AFTER / BEFORE VERSION** (params)	URI of dataset instantiation or variable to define the start/end of the change scope	SELECT ?s ?p ?o
		WHERE {
		CHANGES <efo>
		BEFORE/AFTER VERSION <v_m> {
		?s ?p ?o}
		}
		}

CHANGE (params)	URI of change or variable	SELECT ?v1, ?v2, ?c, ?p ?o WHERE {
		CHANGES <EFO>
		BETWEEN VERSIONS ?v1 ?v2 {
		CHANGE ?c {?p ?o}
		}

RECATT <recattURI | ?recatt_var>

{ <predicateURI | ?predicate_var> <objectURI | ?var> }

RECATT is used inside a RECORD block and separates the subject of a DIACHRON record with the record attributes that describe it. It is followed by a URI/variable. If no specific record attribute needs to be queried or matched in a variable, RECATT can be omitted.

CHANGES <diachronicURI | var> [[BETWEEN VERSIONS <version1URI | ?var1>] || [BEFORE VERSION <versionURI | var1>] || [> AFTER VERSION <versionURI | var1>]]

CHANGES is used to limit the scope of a block within a larger query into a particular change set, or match change sets to a variable. If no URI is given, then all existing change sets will be used to match the query body. CHANGES can optionally be used with BETWEEN VERSIONS, BEFORE VERSION or AFTER VERSION to limit the scope of the changes or bind the dataset versions that match the change set pattern to variables.

CHANGE <changeURI | ?change_var>

The CHANGE keyword is used to query a particular change in a fixed query block within a larger query pattern. It is followed by a specific change URI or a variable to be bound. The succeeding block is used to declare the change parameters in a predicate-object manner.

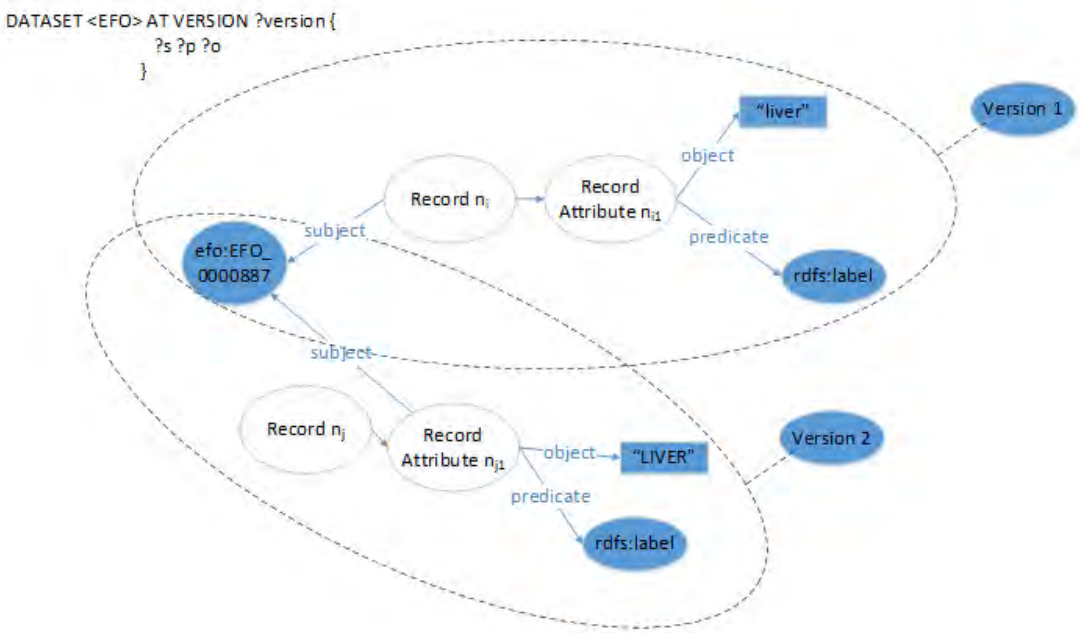


Figure 2.6: Matching a reified triple in a query with variable versions. Blue nodes are selected by the query.

2.4.4 DIACHRON QL formal definitions

In order to formally describe DIACHRON QL as a SPARQL extension, it is necessary to address the DIACHRON model as an extension of RDF, in a manner similar to [KK10; Lop+10; PAG06; PJS11]. Let I, B, L, V be infinite, pairwise disjoint sets of IRIs, blank nodes, literals and variables respectively. An RDF triple t is a triple $(s, p, o) \in (I \cup B) \times (I) \times (I \cup B \cup L)$, where s is the subject, p is the predicate and o is the object of the triple. An RDF graph is a collection of triples g = {t_1,t_2,...,t_n). The union $(I \cup B \cup L)$ is denoted as T and represents all possible bound values any node in an RDF graph can take. The set of all RDF graphs is denoted as G. Given the above, we define the DIACHRON model entities as follows:

Definition 10 ***Record Attribute***

A record attribute a is a tuple (t, g) where t is an RDF triple, and g is a metadata subgraph for a. In essence a record attribute associates an RDF triple t with its metadata, expressed as an RDF graph g. We denote as $G_a \subseteq G$ the set of all record attributes, and as $I_a \subseteq I$ the set of all record attribute IRI nodes.

Definition 11 ***Record***

A record r is defined as a tuple (G_a^s, g), where $G_a^s \subseteq G_a$ is a set of record attributes over subject s, and g is a metadata subgraph associated with r. The set G_a^s is only relevant to the particular context and is not meant to be an exhaustive list of triples with s as common subject. We denote as $G_r \subseteq G$ the set of all records, and as $I_r \subseteq I$ the set of all record IRI nodes.

Definition 12 ***Record Set***

A record set R is defined as a tuple $(G_r^{'}, g)$, where $G_r^{'} \subseteq G_r$ is a set of records, and g is a metadata subgraph associated with R. We denote as $G_R \subseteq G$ the set of all record sets, and as $I_R \subseteq I$ the set of all record set IRI nodes.

Definition 13 ***Schema Set***

A schema set S is defined as a tuple $(G_s^{'}, g)$, where $G_s^{'} \subseteq G_r$ is a set of schema elements, and g is a metadata subgraph associated with S . We denote as $G_s \subseteq G$ the set of all schema sets, and as $I_S \subseteq I$ the set of all schema set IRI nodes.

Definition 14 ***Dataset Instantiation***

A dataset instantiation d is a tuple $\left(G_R^{'}, G_S^{'}, g\right)$ where $G_R^{'} \subseteq G_R$ and $G_S^{'} \subseteq G_S$ are the record set and schema set of the instantiation. We denote as $G_d \subseteq G$ the set of all dataset instantiations, and as $I_d \subseteq I$ the set of all dataset instantiation IRI nodes.

Definition 15 ***Diachronic Dataset***

A diachronic dataset D is a tuple $\left(G_d^{'}, g\right)$ where $G_d^{'} \subseteq G_d$ is an arbitrary set of dataset instantiations as per Definition 14. We denote as $G_D \subseteq G$ the set of all diachronic datasets, and as $I_D \subseteq I$ the set of all diachronic dataset IRI nodes. Similarly to SPARQL we allow for blank nodes and literals to be identifier values, as well as triple subjects, even though in practice this is not supported by most frameworks. Note further that the metadata subgraph can be an empty graph. This allows for definitions of datasets and other DIACHRON entities without necessarily associating metadata with them.

The above definitions serve to regard the entities of the DIACHRON model as extensions of RDF. Examples of these are shown in Figure 2.5 and Table 2.2, as discussed in Section 3.

In order to define the syntax of DIACHRON QL, we briefly recall the notion of a SPARQL graph pattern presented in [PAG06]. A SPARQL graph pattern expression is defined recursively as follows:

1. A tuple from $(T \cup V) \times (I \cup V) \times (T \cup V)$ is a graph pattern.

2. If P_1 and P_2 are graph patterns, then (P_1 AND P_2), (P_1 OPT P_2), and (P_1 UNION P_2) are graph patterns.

3. If P is a graph pattern and R is a SPARQL built-in condition, then the expression (P FILTER R) is a graph pattern.

Given this, a DIACHRON QL graph pattern expression (DGP) is defined hierarchically and recursively as follows:

1. A SPARQL graph pattern P is a DGP.

2. If $X \in (I_a \cup V)$ then (X RECATT P) is a DGP (a record attribute pattern).

3. If $X \in (I_r \cup V)$ then (X RECORD P) is a DGP (a record pattern).

4. If P is a DGP, $X \in (I_D \cup V)$ and $Y,\ Z \in (I_d \cup V)$ then :

   a. (((DATASET X) AT VERSION Y) P),

   b. (((DATASET X) AFTER VERSION Y) P),

   c. (((DATASET X) BEFORE VERSION Y) P),

   d. (((DATASET X) AFTER VERSIONS Y, Z) P)

   e. ((DATASET X) P) are DGPs (dataset instantiation patterns).

5. If P_1 and P_2 are DGPs, then the following are DGPs:

   a. P_1 AND P_2

   b. P_1 OPT P_2

   c. P_1 UNION P_2

DIACHRON QL built-in conditions for filtering are similar to [PJS11] and are not further addressed in this chapter. Examples on all keywords and constructs of DIACHRON QL can be seen in Table 2.2.

2.4.5 Semantics of DIACHRON QL graph pattern expressions

We are now ready to define the semantics of DPG expressions. Borrowing the notation of [PJS11], a SPARQL mapping, or substitution, , is defined as a partial function μ : $V \rightarrow T$ for a subset V$^{'} \subseteq V$, such that the variables in $V^{'}$ are replaced with values from T as is defined in . The domain of : $dom(\mu)$ is the subset of V where is defined. A pair of mappings $\dot{\varphi}_1$ and $\dot{\varphi}_2$ exhibits compatibility when for all $v \in dom(\mu_1) \cap dom(\mu_2)$, it

holds that $\mu_1(v) = \mu_2(v)$. Let Ω_1 and Ω_2 be sets of mappings, then the join, the union, and the difference between Ω_1 and Ω_2 are defined as follows:

- $\Omega_1 \bowtie \Omega_2 = \{\mu_1 \cup \mu_2 | \mu_1 \in \Omega_1, \mu_2 \in \Omega_2$ are compatible$\}$,
- $\Omega_1 \cup \Omega_2 = \{\mu | \mu \in \Omega_1$ or $\mu \in \Omega_2\}$,
- $\Omega_1 \setminus \Omega_2 = \{\mu \in \Omega_1 | \forall \mu^{'} \in \Omega_2, \mu$ and $\mu^{'}$ are not compatible$\}$.

  Finally, the left outer-join (OPTIONAL) is defined as:

- $\Omega_1 \mathbin{⟕} \Omega_2 = (\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 \setminus \Omega_2)$.

Given the above, the notion of mapping remains the same in DIACHRON QL.

In DIACHRON QL, the hierarchical relationship between entities enables graph patterns to be limited in *scopes* , with respect to the DIACHRON model. Evaluating a triple pattern within the scope of two different record patterns can result in different output, and also enables pattern expressions involving the binding of DIACHRON model entities as well.

Formally, we need to define what the scope of a graph pattern is. A scope is a function $\sigma : P \rightarrow T^{'} \subseteq T$ that maps a graph pattern P to a closed set $T^{'}$, so that any mapping μ_p of P is only valid with respect to $\mathrm{T}^{'}$, i.e. $\mu_P \subseteq \sigma(P)$. Given this, we go on to define the *lowest wrapping scope* λ as a partial function $\lambda : P \rightarrow \sigma(I \cup V)$ that maps P with a scope, such that the variables in P are mapped to elements in that scope, and there exists no other scope that is a subset of the one derived from . This implies that any graph pattern P is equipped with a function $\lambda(P) \in \sigma(I \cup V)$ such that $\mu(P) \in \lambda(P)$ and $\nexists \lambda^{'}(P) \neq \lambda(P) \mid \lambda^{'}(P) \subseteq \lambda(P)$. Furthermore, we denote with $\lambda_{D'}(P)$ when $\lambda(P)$is limited to a specific subset of diachronic datasets and dataset instantiations $\mathrm{D}^{'}$. Intuitively, a lowest wrapping scope for a particular query is the lowest entity type in the DIACHRON model hierarchy where P is expressed. For example, a record attribute pattern $\mathrm{P_a}$ in a query is nested within a record pattern $\mathrm{P_r}$ and a dataset instantiation pattern $\mathrm{P_d}$. Then $(P_a) = \sigma(P_r)$ and $(P_r) = \sigma(P_d)$. An example of scoping in a DIACHRON query can be seen in Figure 2.7.

We are now ready to define the evaluation of a DIACHRON QL graph pattern. Given a diachronic dataset D with a set of dataset instantiations d over T , such that $D^{'} \subseteq D$ is the subset of D in which d exists, and DGPs P, P_1 and P_2 defined in $D^{'}$, $D^{'}_1$ and $D^{'}_2$ respectively, then the evaluation of a DGP denoted as $[[\bullet]]_{D'}$ is as follows:

- $[[P]]_{D'} = \{ \mid \mathrm{dom}(\mu) = var(P)$ and $\mu(P) \in \lambda_{D'}(P) \}$
- $[[P_1 \text{ AND } P_2]]_{D'_1, D'_2} = \{\mu = [[P_1]]_{D'_1} \bowtie [[P_2]]_{D'_2} \mid \mu \in \lambda_{D'_1}(P_1) \cap \lambda_{D'_2}(P_2)\}$

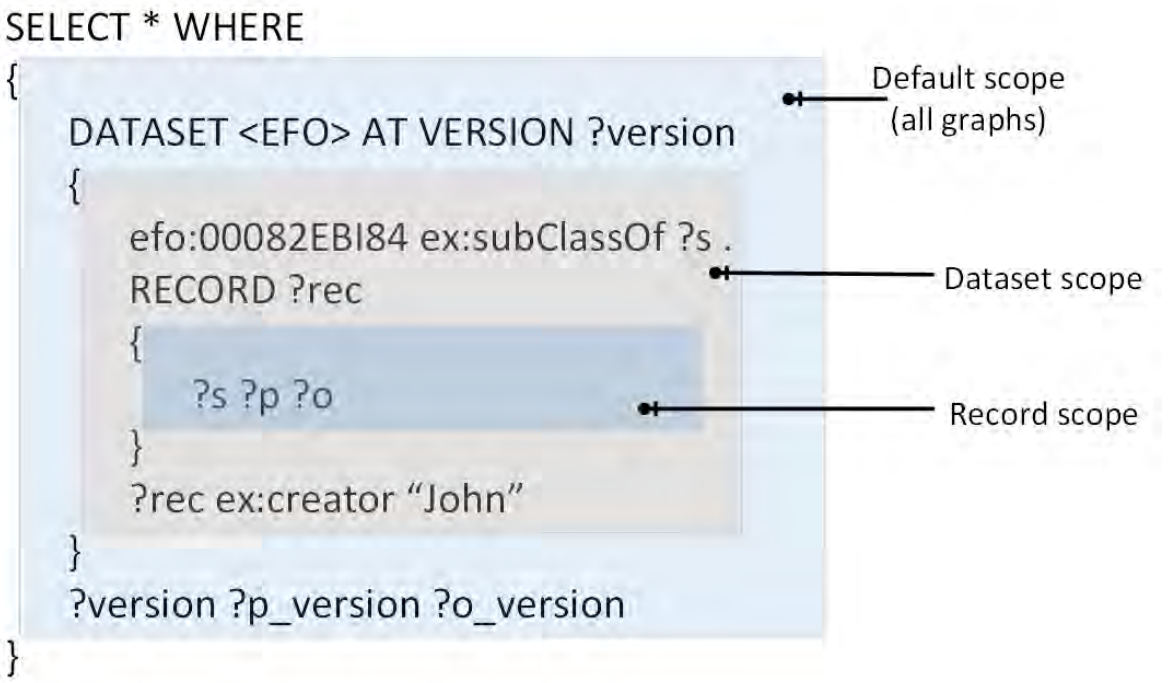


Figure 2.7: An example of scopes in a DIACHRON QL query.

- $[[P_1 \text{ UNION } P_2]]_{D'_1,D'_2} = \{\mu = \ [[P_1]]_{D'_1} \cup [[P_2]]_{D'_2} \mid \mu \in \ \lambda_{D'_1}(P_1) \cup \lambda_{D'_2}(P_2)\}$
- $[[P_1 \text{ OPT } P_2]]_{D'_1,D'_2} = \{\mu = \ [[P_1]]_{D'_1} \bowtie [[P_2]]_{D'_2} \mid \mu \in \ \lambda_{D'_1}(P_1) \cup \lambda_{D'_2}(P_2)\}$

Evaluation of filters remains the same as with the original SPARQL specification [PJS11] and is not reported herein. Finally, note that we do not consider the case of named graphs within DIACHRON graph patterns, because the general notion of a SPARQL named graph is specialized in the more refined DIACHRON entity types.

Given a DIACHRON graph pattern expression (((DATASET X)AT VERSION Y)P), its evaluation will be equal to the evaluation of P over diachronic dataset X at version Y, i.e. the set of all mappings such that $\mu(P) \in \lambda_{X'}(P)$, with X' being the subset of X that contains version Y.

2.5 Implementation

In this section we present the implementation of the proposed query language. We first provide an overview of the overall architecture of the DIACHRON archive. The archive employs the proposed DIACHRON model for storing evolving LOD datasets. The query engine is a core component of the archive, responsible for processing queries expressed in the DIACHRON QL and retrieving data out of the archive.

2.5.1 System architecture

The architecture of the archive and various components of the archive can be seen in Figure 2.8. The archive's web service interface is exposed via the HTTP protocol as the primary access mechanism of the archive through a RESTful web service API. The Data Access Manager provides low level data management functionality for the archive. It is bound to the specific technology of the underlying store, in our case Openlink Virtuoso 7.1[3], as well as external libraries that provide data access functionality for third-party vendors. For this we used the Jena semantic web framework[4]. It serves as an abstraction layer between the store and the query processor.

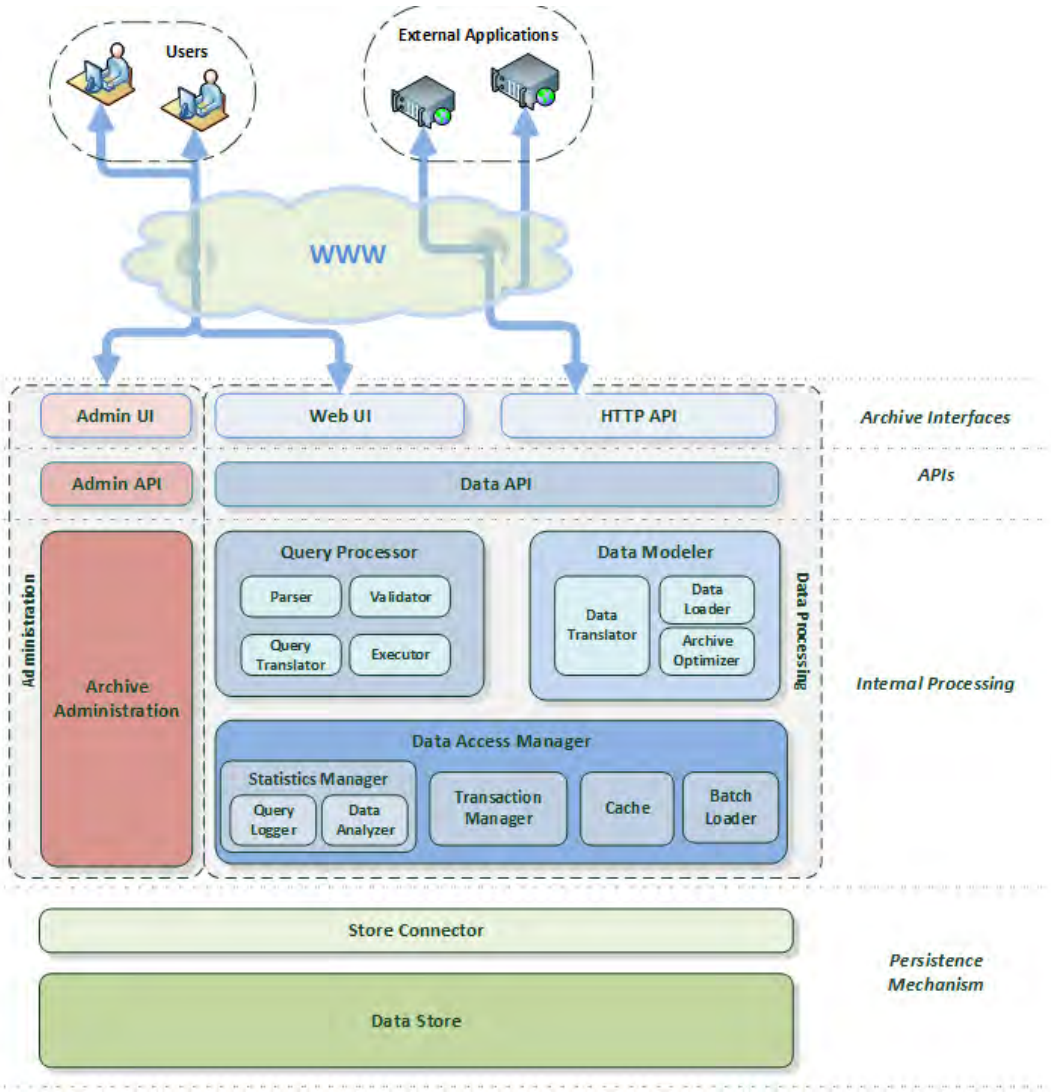


Figure 2.8: Architecture of the archive.

[3]http://virtuoso.openlinksw.com/
[4]https://jena.apache.org/

The archive employs a *Data Access Manager*, a *Store Connector*, a *Data Modeler*, an *Archive Optimizer* and a *Query Processor*. The Store Connector is the software package that provides an API to other components of the archiving module for communication and data exchange with the underlying store and is implemented with the Virtuoso JDBC Driver package[5]. The Data Store employs a Virtuoso 7.1 instance. The Data Modeler component handles the dataset input functionality and data transformations from the DIACHRON dataset model to the native data model of the store and vice versa, and consists of the Data Translator and the Data Loader. The Archive optimizer component supports the optimization of the datasets' storage method based on various archive strategies as shown in [SCF14]. It performs analysis of the dataset characteristics and chooses the most efficient storage strategy based on metrics.

The Query Processor component is the base mechanism for query processing and thus data access. It consists of the following subcomponents:

- **Validator:** validates the DIACHRON queries for syntactic validity against the DIACHRON QL syntax.
- **Query parser:** parses the queries in DIACHRON QL so as to create a structure of elements that correspond to DIACHRON Dataset Entities and DIACHRON query operators.
- **Query Translator:** creates the execution plan of DIACHRON queries by translating the queries in SPARQL. The translator also makes use of the various archive structures implemented in the persistence store and the appropriate indexes and dictionaries. The query translator is the subcomponent that ties the DIACHRON archive module to the specific storage technology of RDF and SPARQL. Translation is further described in the next subsection.
- **Executor:** executes the created execution plan step by step and retrieves the raw data from the store so as to build the result set of the query. It uses also the Data Modeler component in order to perform, if necessary, data transformations from the native data model of the underlying store to the DIACHRON dataset model.

2.5.2 Translation of DIACHRON QL to SPARQL

Our implementation is based on mature standards and state of the art triple stores that implement RDF storage and SPARQL querying. This imposes that DIACHRON entities are converted to RDF and queries are mapped to SPARQL expressions. In this

[5]http://docs.openlinksw.com/virtuoso/VirtuosoDriverJDBC.html

context, DIACHRON graph patterns can generally be translated to SPARQL as shown in Table 2.3. However, a direct mapping is not generally possible, as the two models differ conceptually. The actual translation to SPARQL is ultimately dependent on factors that are affected by the implementation at hand, such as the storage policies, the structure of the archive and its dictionary, and the pre-processing requirements of the query engine. For this reason, we have implemented a middle layer between the DIACHRON QL parser and the SPARQL query executor, where the following steps take place:

1. Identification of the query's relevant scope(s), and in-memory mapping to DIACHRON structural elements
2. Extraction and mapping of graph patterns to their respective scopes
3. Conversion of lowest level graph patterns to SPARQL
4. Detection of non-materialized dataset versions that contain possible scope candidates
5. Temporary materialization of non-materialized dataset versions
6. Mapping to final SPARQL query

In the above flow of actions, step 1 is responsible for extracting the scopes $\sigma(P_i)$ for all P_i that are sub-expressions of a DIACHRON query expression P. References to their respective URI nodes or variables point to their subsumed DGP and are stored in memory for future reference. In step 2, we map each scope to its respective DGP found in the query string, and populate the query object in-memory. In step 3, we identify the data-relevant part of the query (i.e. the part that references actual records and attributes), and rewrite it to SPARQL independently of its scope. In step 4, we detect whether a scope is actually materialized in the archive. This step deals with cases where the chosen storage policy differs from full materialization, however it is not in the scope of this work to address the implementation issues of storage policies, the storage-querying trade-off, or storage optimization for contexts with versioning. Furthermore, simple lookups in the dictionary for a given query's scopes is not sufficient to determine which σ are eventually referenced, because a scope can be unbound (i.e. a variable). These points are all taken into account in steps 4 and 5. Finally, step 6 relies on the output of the previous steps in order to build one or more SPARQL queries that will be executed by the query engine. Hence, in order to implement DIACHRON QL in a SPARQL setting, the added expressivity of DIACHRON QL over SPARQL is translated to a series of steps, rather than a direct 1:1 mapping of entities and graph pattern expressions.

Table 2.3: DIACHRON graph patterns and their translation to SPARQL.

DIACHRON Pattern (Parsed Syntax)	**SPARQL**
{?s ?p ?o}	{[a evo:Record ; evo:subject ?s ; evo:hasRecordAttribute [evo:predicate ?p ; evo:object ?o]]}
RECORD ?r {?s ?p ?o}	{?r a evo:Record ; evo:subject ?s ; evo:hasRecordAttribute [evo:predicate ?p ; evo:object ?o]}

RECORD ?r {
?s RECATT ?ra {?p ?o}
}
	{?r a evo:Record ;
	evo:subject ?s ;
	evo:hasRecordAttribute ?ra .
	?ra evo:predicate ?p ;
	evo:object ?o}

DATASET <EFO> AT VERSION ?v {
RECORD ?r {
?s RECATT ?ra
{?p ?o}
}
}
	{GRAPH <dataset_dictionary> {
	<EFO> evo:hasInstantiation ?v .
	?v evo:hasRecordSet ?rs
	}
	GRAPH ?rs{
	?r a evo:Record ;
	evo:subject ?s ;
	evo:hasRecordAttribute ?ra .
	?ra evo:predicate ?p ;
	evo:object ?o }}

FROM DATASET <EFO>
AT VERSION <EFO/v1>
{
RECORD ?r {
?s RECATT ?ra {?p ?o}
}
}
	{GRAPH <dataset_dictionary> {
	<EFO> evo:hasInstantiation <EFO/v1>
	.
	<EFO/v1> evo:hasRecordSet ?rs
	} GRAPH ?rs{
	?r a evo:Record ; evo:subject ?s ;
	evo:hasRecordAttribute ?ra .
	?ra evo:predicate ?p ;
	evo:object ?o }}

FROM CHANGES <EFO>
BETWEEN VERSIONS <EFO/v1>
<EFO/v2>
{
CHANGE ?c {?p ?o}
}
	{GRAPH <dataset_dictionary> {
	?cs a evo:ChangeSet ;
	evo:oldVersion <EFO/v1> ;
	evo:newVersion <EFO/v2>
	} GRAPH ?cs{
	?c a _:Change ; ?p ?o }}

2.6 Evaluation

In this section we present the evaluation of our approach over a real world evolving biological use case of the EFO ontology as well as use case concerning evolving multidimensional data of the statistical domain published on the web in LOD format following the Data Cube Vocabulary[6] approach. As a first step, in Table 2.4, we provide a qualitative evaluation of supported storage policies, querying scopes, supported change representation, and metadata granularity of a framework implementing the DIACHRON model and Query language, compared with related works previously discussed. Specifically, we compare our approach with traditional version control, as well as SemVersion [VG06], Auer et al [AH06], Im et al [ILK12], Hauptmann et al [HBW15] and Memento LD [Som+09; Som+10], and we find that these approaches cover parts of the functionality offered by a framework that implements DIACHRON. Furthermore, we conducted a performance evaluation and a usability evaluation. The performance evaluation aims at showing that there is no significant overhead imposed in query processing that introduces above-linear performance for queries of increasing difficulty. The usability evaluation aims at measuring with objective metrics the syntax overhead that the proposed DIACHRON Query Language introduces.

In the first case, we consider 15 consecutive versions of the ontology, that exhibit various types of changes, both simple and complex, as well as four multidimensional datasets each comprised of three consecutive versions. We load all datasets into the same archive instance, and in order to do so, the data are first converted to fit the RDF mapping of the DIACHRON model. For this, we implemented a conversion mechanism as part of the Data Modeller component presented in the previous section. The modeller reifies data to records and record attributes. Data are mapped to the DIACHRON data model in the following manner. First, classes and their definitions (domains, ranges) are modelled as schema objects. The triples are grouped by their subjects. For each subject URI, its corresponding predicate-object pairs are modelled as record attributes and grouped in records. The subject records are in turn connected with the record attributes created for each triple associated with a subject URI.

2.6.1 Experimental Evaluation

The goal of the experimental evaluation was to assess the performance of our implementation w.r.t three main aspects: the time overhead related to the initial loading of the archive, the time overhead related to the retrieval of the datasets in their original form

[6]http://www.w3.org/TR/vocab-data-cube/

Table 2.4: Qualitative comparison of each frameworks support for (a) storage policies, (b) querying scopes, (c) change representation, and (d) provenance and metadata granularity. (CB = change-based storage, FM = full materialization)

	Storage	Querying	Changes	Provenance Granularity
Version Control	CB (Sequential)	N/A	Low Level	None
SemVersion	FM	Graph Patterns	Low Level	None
Auer et al	CB (Sequential)	Changes	High Level	Changes
Im et al	CB (Aggregated)	Graph Patterns	Low Level	Datasets
Hauptmann et al	CB (Sequential)	Graph Patterns	Low Level	Datasets
Memento LD	FM	Resources	N/A	Resources
DIACHRON	**Hybrid**	**Datasets, Versions, Graph Patterns, Resources, Changes, Longitudinal**	**High Level**	**Datasets, Versions, Resources, Changes, Triples**

(de-reification and serialization) and the time overhead of executing queries of different difficulty. Specifically, we want to assess (i) the runtime performance of the pre-processing step for DIACHRON QL, and (ii) whether there is extra processing overhead that makes query processing non-linear with respect to query difficulty. Our approach was implemented in Java 1.7, and all experiments were performed on a server with Intel i7 3820 3.6GHz, running Debian with kernel version 3.2.0 and allocated memory of 8GB.

First, bulk operations on whole datasets have been tested, namely loading and retrieving full dataset versions. Loading and retrieval times can be seen in Figure 2.10 (a) and (b). A series of 10 tests were run for each version of the datasets and the averages have been used in computing execution time, using least squared sums. Loading a dataset in the archive implies splitting it into the corresponding structures, i.e. dataset, record set, schema set and change set, and storing it in different named graphs. The splits were done directly in the store using the SPARQL update language and basic pattern matching, thus no need to put a whole dataset in memory arose, which would be costly in terms of loading in and building the respective Java objects in Jena[7]. The increasing sizes of the input datasets are the effect of their evolution, as new triples are being added. In Figure 2.10 (b), retrieval times can be seen for the same datasets. Retrieval of a dataset is the process of de-reifying it to recreate the dataset version at its original form and structure. As can be seen, both loading times and retrieval fit into a linear regression w.r.t to the datasets' sizes as measured in record attributes and imply that no additional

[7]https://jena.apache.org/

time overhead is imposed that would destroy linearity as new versions of a dataset are stored in the archive.

Figures 2.10 (c)-(h) show running times of 14 queries we devised for this experiment. An analysis of the queries' characteristics can be seen in Table 2.5. In Figures 2.10 (c) and (d) we perform a series of queries on different dataset versions. Specifically, two sets of 5 queries have been devised to run on a fixed dataset. Each query is run on one particular version, and the total running time of all 5 queries in each set (c) and (d) is calculated after retrieving the results and storing them in memory, which implies a simple iteration on all results. The query sets are made up from SELECT queries that combine structural entities (records, record attributes etc.) with actual data entries (subject URIs etc.) in different levels of complexity. In Figure 2.10 (a) no aggregate functions, OPTIONALs or other complex querying capabilities have been used, while in Figure 2.10 (b) the queries consist of selecting, aggregating and filtering graph patterns. As in the case of loading and retrieval, the archive behaves in a linear way as the size of a dataset increases.

Finally, four queries, Q11-Q14, with variable datasets that search in the entire archive have been devised and run on an incrementally larger archive, that is, the queries have been tested on deployments of the archive where versions of datasets are being incrementally added to their corresponding diachronic datasets. The queries use dataset versions as variables. The results can be seen in Figures 2.10 (e)-(h) where linearity is still being preserved when new datasets are stored.

Running times for the pre-processing step can be seen in Figure 2.9. Specifically, we have measured the total running time required to create and populate a DIACHRON query object, prior to execution, as opposed to a SPARQL query object, for queries Q1-14 on an archive instance that contains the maximum number of tested versions. The pre-processing overhead for DIACHRON QL is proportional to the intermediate steps, but does not impose a large difference when compared with plain SPARQL queries in the majority of cases. The SPARQL queries appear to impose a constant overhead, while the time needed to pre-process DIACHRON queries increases along with the query expressivity and complexity of mapped scopes and DIACHRON elements. Even so, the pre-processing overhead is negligible (in most cases <100ms). For queries Q13 and Q14 the pre-processing step is very costly, because of the sequenced nature of pre-processing steps required to combine materialized and non-materialized datasets in queries with variable diachronic datasets and dataset instantiations.

Table 2.5: Characteristics of the experiment queries.

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14
DISTINCT	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Unbound predicates								✓	✓	✓	✓	✓	✓	✓
Filters											✓	✓	✓	✓
Aggregate Functions							✓	✓	✓	✓	✓	✓	✓	✓
ORDER BY							✓	✓	✓	✓	✓	✓	✓	✓
OPTIONAL												✓		
SELECT	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓
CONSTRUCT											✓			
Reified data	✓	✓	✓	✓	✓								✓	✓
De-reified pattern						✓	✓	✓	✓	✓	✓	✓	✓	✓
Diachronic metadata					✓	✓			✓	✓	✓	✓	✓	✓
Unbound named graphs											✓	✓	✓	✓
Non-materialized datasets													✓	✓

Table 2.6: Comparison of (i) number of keywords, (ii) number of triple (or record) patterns, and (iii) number of generated variables not existing in the original query.

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14
# keywords (SPARQL)	**4**	**4**	**4**	**4**	**4**	7	9	8	8	8	10	12	27	46
# keywords (DIACHRON)	6	6	5	5	5	**5**	**5**	**6**	**6**	**5**	**7**	**7**	**6**	**7**
# TPs (SPARQL)	5	5	5	5	10	6	7	9	9	9	19	21	33	44
# TPs (DIACHRON)	**1**	**1**	**1**	**1**	**1**	**2**	**2**	**2**	**2**	**2**	**4**	**4**	**2**	**2**
# non-TP vars (SPARQL)	2	2	1	2	3	2	2	3	2	2	4	5	6	9
# non-TP vars (DIACHRON)	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**

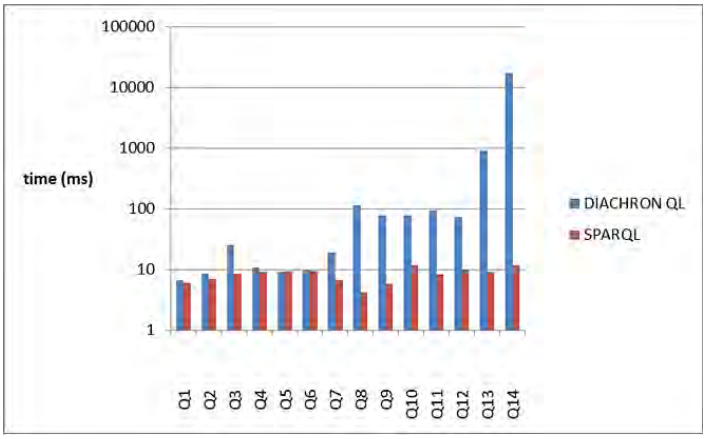


Figure 2.9: Logarithmic plot of pre-processing time (in milliseconds) for queries Q1-Q14.

2.6.2 Usability Evaluation

In order to evaluate usability, we make use of three objective metrics in order to compare the *compactness* , *expressiveness* and *usability* of DIACHRON QL with respect to SPARQL. Specifically, we compare (i) the number of language-specific keywords used in each of the 14 queries, (ii) the total number of triple/record patterns, and (iii) the number of intermediate variables that were neither part of the original query, nor requested by the user. The results can be seen in Table 2.6.

As the number of SPARQL TPs increases, the number of DIACHRON QL record patterns remains at low levels, thus abstracting the complexity of writing large queries. This is especially evident in queries Q13 and Q14, where we have used hybrid storage policies, thus forcing the query engine to decide on parse-time which dataset versions are materialized and which have to be materialized as nested graph patterns. For instance, query 13 that features a bound diachronic dataset with an unbound version (using AT_VERSION ?v) can be expressed with just two DIACHRON patterns, whereas the SPARQL query uses 33 triple patterns to cater for the versions that follow a mixed storage policy. Note, however, that independently of the underlying storage policies, even if the user was inclined to express a query in a language like SPARQL and rely on an existing query engine for execution, the set of intermediate steps executed by our system would be omitted in the process, thus limiting the expressivity of the possible queries.

The number of keywords used in each of the two languages for the 14 queries is smaller for small queries (queries Q1-Q5), but SPARQL tends to overcome DIACHRON in total number of language-related keywords as the query gets larger and more complicated. This is also dependent on the various scopes and filters used by a query. Finally, SPARQL eventually depends on a number of dynamically generated intermediate variables that are used in the translated query, which is not needed by DIACHRON. These variables bind dictionary elements, scopes, versions, record sets and so on to variables that are further used in GRAPH clauses and FILTERs in the SPARQL translation.

2.7 Conclusions

In this chapter, we have discussed the challenges and requirements for the preservation and evolution management of datasets published on the Data Web and we have presented an archiving approach that utilizes a novel conceptual model and query language for storing and querying evolving heterogeneous datasets and their metadata. The DIACHRON data model and QL have been applied to real world datasets from the life-sciences and open government statistical data domains. An archive that employs these ideas has been

implemented and its performance has been tested using real versions of datasets from the aforementioned domains over a series of loading, retrieval and querying operations.

The growing availability of open linked datasets has brought forth significant new problems related to the distributed nature and decentralized evolution of LOD and has posed the need for novel efficient solutions for dealing with these problems. In this respect, we have highlighted some possible directions and presented our work that tackles evolution and captures several dimensions regarding the management of evolving information resources on the Data Web.

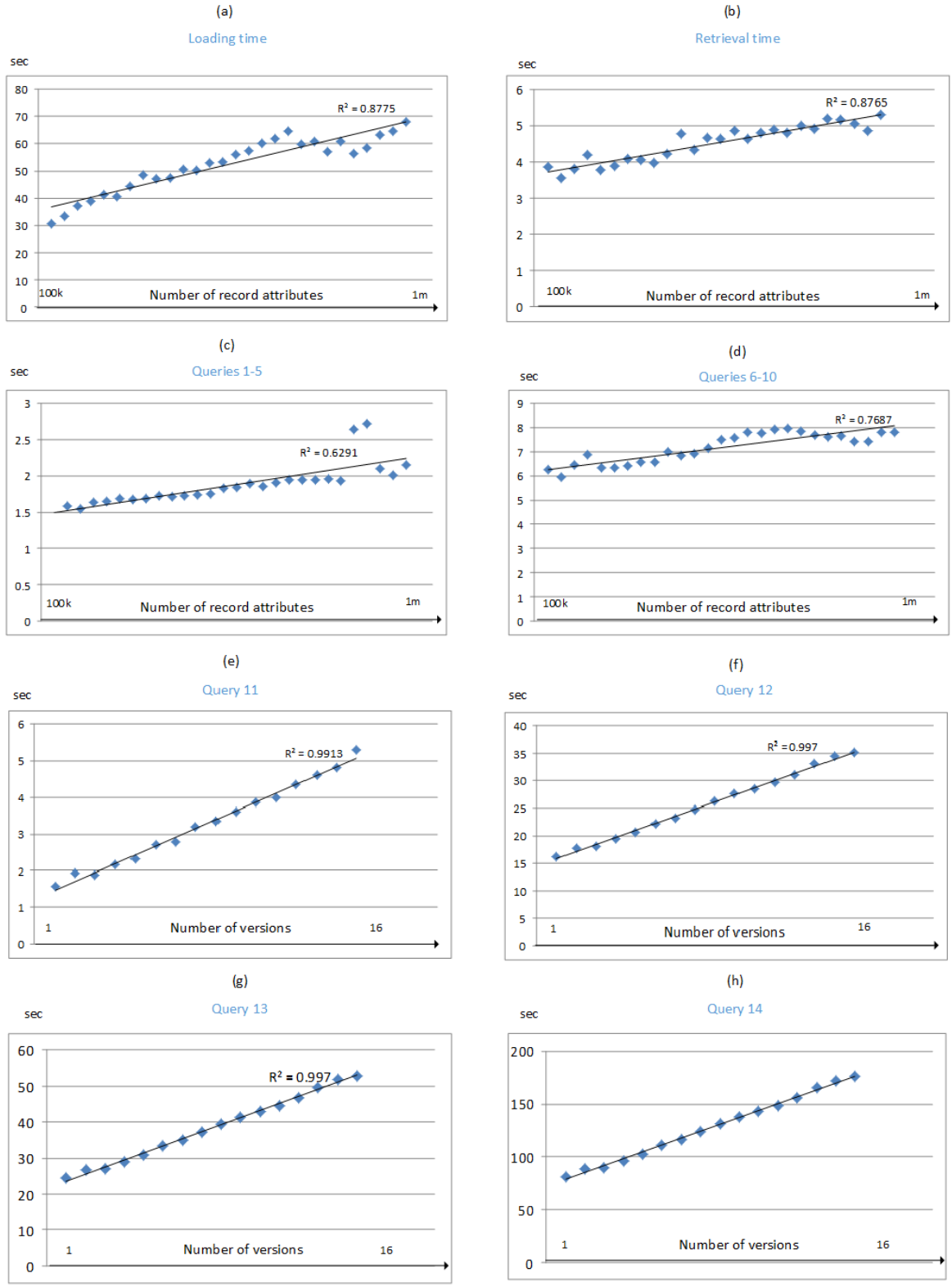


Figure 2.10: Loading times (a), retrieval times (b), select queries without filters and aggregates (c), select que-ries with filters and aggregates (d), select queries with variable datasets (e)-(h).

Chapter 3

Benchmarking Evolution Management Systems for Linked Data

3.1 Introduction

In the previous chapter, we have discussed the implications of storing, querying and managing evolving data. In this chapter, we will discuss the challenge of evaluating systems that handle RDF evolution, and we will present a novel benchmarking framework that is designed to evaluate and assess evolution management systems for RDF data.

Evolution in RDF data stems from low-level changes in the datasets, i.e., additions and deletions of triples through different time points. However, other parameters come in to play when benchmarking evolution management systems, such as schema vs instance evolution, change complexity, change distribution, and so on. With SPARQL being the standard for querying RDF, a remarkable number of efforts have emerged to address matters of efficiency in storing and querying RDF. However, the same issues have not been thoroughly addressed in versioning and evolving contexts. Hence, any experimentation on such systems is unable to rely on arbitrarily large or complex data, because of the aforementioned lack of synthetic versioned data generators. Therefore, benchmarking archiving and versioning systems is not a trivial task, as the benchmarking process is not easily configurable or tailored to custom needs based on existing approaches, an advantage that stems from synthetic data. Two main aspects must be considered towards this goal. First, systems must be able to generate synthetic datasets of varying sizes and schema complexity, in order to approximate different cases of evolution. Second, the performance evaluation of these tools requires the existence of representative query workloads and evolving operators.

In this chapter, we present *EvoGen*, a synthetic data generator for evolving RDF that is based on the widely adopted Lehigh University Benchmark generator [GPH05]. EvoGen addresses generation of consecutive versions with a configurable shift (i.e. change in size between versions) parameter, configurable schema evolution, as well as query workload generation functionality. To this end, we build on LUBM's existing benchmark queries, and we provide new ones that address the querying dimensions commonly found in dataset evolution, discussed in the previous chapter.

Contributions. The contributions of this chapter are summarized as follows:

- we present requirements and characteristics for generating synthetic versioned RDF,
- we extend the LUBM ontology with 10 new classes and 19 new properties,
- we propose *EvoGen* as a benchmarking system for evolving RDF,
- we implement a change logging mechanism, that produces RDF logs of the changes between consecutive versions following the representational schema of the change ontology described in [Pap+13b],
- we provide an implementation for adaptive query workload generation, based on the evolutional aspects of the data generation process.

This chapoter is outlined as follows. Section 2 provides an overview of related work. Section 3 discusses requirements for the benchmark, and Section 4 discusses the parameters of the benchmark in the context of the EvoGen system. Section 5 describes the system's implementation, and section 6 concludes the chapter.

3.2 Related Work

In this work, we extende the Lehigh University Benchmark (LUBM) [GPH05], a widely adopted benchmark for RDF and OWL datasets. LUBM includes an implementation for generating synthetic data in OWL and DAML formats. Its broader scope includes benchmarking reasoning systems, as well as RDF storage and SPARQL querying engines [Sto+08], [KKB15], [KKK10], [BSK07], [Bor+13], [Hus+10], [Pap+12]. It provides an ontology expressed in OWL, where there exist restrictions between classes so that reasoners can perform inferencing. Furthermore, LUBM comes with 14 SPARQL queries with varying sizes of query patterns, ranging from 1 to 6 triple patterns. Because of the fact that these can be limiting when stress testing SPARQL engines, they have been extended in the literature in order to provide more complex patterns (e.g. in [KKB15]). SP^{2}Bench

[Sch+09] is a generator for RDF data, with the purpose of evaluating SPARQL querying engines. Its scope is mostly query efficiency instead of inferencing, and has been widely adopted in the literature [SML10; Hus+10; Let+13]. Other approaches in the context of RDF and SPARQL benchmarking, such as FedBench [Sch+11] and the Berlin SPARQL Benchmark (BSBM) [BS09], provide fixed data rather than custom data generation, hence they are not readily capable of providing a benchmark for evolving and otherwise versioned datasets. The reader is referred to [Dua+11] for an extensive study and comparison of RDF benchmarks. Finally, Fernandez et al. [FPU15] discuss a series of metrics for benchmarking archiving systems in Linked Data contexts. Our approach aims at providing a highly customizable benchmarking suite for creating synthetic and evolving data, with instance-level and schema-level evolution and adaptive query workload generation.

3.3 Requirements

Benchmarking processes usually adhere to several functional and non-functional requirements concerning the generation of synthetic data and the relevant query workload. Especially in the case of evolving data, there is a multitude of dimensions to address when tailoring the benchmark to custom needs.

3.3.1 Configurability

The benchmark should be able to provide a viable degree of configurability through tunable parameters, regarding the data generation process, the context of the application that will be tested, and the adaptability of the query workload on the specificities of the generated evolving data. The differentiation between benchmarks for evolving settings, and benchmarks for static settings, is that the temporal dimension can randomize the data generation process, which implies that a dynamic and adaptive query workload generation process is required as well.

3.3.2 Extensibility

As evolving data are by definition dynamic in nature, new requirements are bound to arise as application contexts and systematic approaches expand. For this reason, the benchmark is not considered to be exhaustive. Instead, we consider extensibility to be a crucial requirement when designing the parameters of the data generation process and the query workload.

3.3.3 Mixed Workload

For the aforementioned reasons, the workload of the benchmark must be generated adaptively with respect to the required parameters and the generated data. Many traditional benchmarking techniques that include some sort of data provision, either fixed or dynamically generated, usually rely on standardized or otherwise fixed query workloads. For instance, the LUBM benchmark that provides the foundations to EvoGen, offers a set of 14 predefined queries that try to address a variety of interesting query patterns with varying complexities. We argue that in evolving and versioning contexts, the fixed queries can only represent static contexts, and it is thus crucial to be able to extend the workload and provide adaptive workloads that reflect the generation process, which is in turn tailored after the user's custom needs. For example, the number of versions and the variations, as well as the complexity of changes between the versions, leads to significantly different outcomes that can impact the same set of benchmarking tests in varying and possibly unpredictable ways.

3.4 EvoGen Characteristics

3.4.1 Generated data

EvoGen is based on the widely used LUBM generator, which uses an ontology of concepts drawn from the world of academia. Specifically, LUBM creates a configurable number of university entites, which are split in departments. Furthermore, LUBM generates entities that describe university staff and students, research groups, and publications. Most of these classes are provided in different types of specializations, as defined in the LUBM schema ontology. For example, the generator creates varying numbers of lecturers, full professors, associate professors and assistant professors, as well as undergraduate and postgraduate students. The created entities are interrelated via direct (e.g., a professors can be an advisor of a student) or indirect properties (e.g., professors and students can be co-authors in publications), and their cardinalities adhere to relative ranges that are hard-coded in the generator. LUBM heavily relies on randomization over these types of associations, however, it is guaranteed that the schema will be populated relatively evenly across different runs.

We extend the LUBM ontology by providing 10 new classes and 19 new properties, in order to implement schema evolution functionality, also maintaining backward compatibility with the original synthetic generator. The new classes are both specializations (subclasses) of existing ones (e.g. visiting professor, conference publication), and novel

```
ex:change1 rdf:type co:Add_Type_Class ;
co:atc_p1 lubm:VisitingProfessor .
ex:change2 rdf:type co:Add_Super_Class ;
co:asc_p1 lubm:VisitingProfessor ;
co:asc_p2 lubm:Professor .
ex:change3 rdf:type co:Add_Property_Instance ;
co:api_p1 AssociateProfessor13 ;
co:api_p2 lubm:doctoralDegreeFrom ;
co:api_p3 University609 .
```

Listing 3.1: Example RDF in the change log

concepts in the ontology (e.g., research project, scientific event). Through this extension we are able to implement schema evolution which was not supported in the original version, and at the same time keep the original LUBM schema intact to allow backwards compatibility with existing approaches.

3.4.2 Change Production

We design and implement a component for semantic change generation, which relies on the change representation scheme presented in [Pap+13b]. Changes are represented as entities of the Change Ontology, which is able to capture both high level changes, such as adding a superclass, and low level changes, such as triple insertions and deletions. The Change Ontology has been adopted by the community and used in change detection and change representation [Pap+13b] and in [Rou+15] for designing and representing multi-level changes. Also, it is tightly integrated with the temporal query language DIACHRON QL [Mei+16a]. EvoGen optionally creates a change set between two consecutive versions, that includes all changes between the versions, both on the instance and on the schema level.

3.4.3 EvoGen Parameters

We drive the generation process through a set of abstract parameters that reflect the user's needs with respect to the type and amount of changes. Specifically, we use the notions of *shift*, *monotonicity* and *strictness* as high level characteristics of the generation process, and we define a parameter for class-centric schema evolution. In what follows, we will describe these notions.

3.4.3.1 Parameters regarding instance evolution.

Following the definitions provided in the previous chapter, we treat evolution on the dataset level by default. In this context, a dataset D is diachronic, when it provides a time-independent, static representation of all its versions. Given this, let D be a diachronic dataset, and $D_i \dots D_{i+n}$ a set of dataset instantiations at time points $t_i \dots t_{i+n}$. Then, the *shift* of dataset D between t_i and t_{i+n}, denoted as $h(D)|_{t_i}^{t_{i+n}}$, is defined as the ratio of change in the size of the instantiations $D_i \dots D_{i+n}$ of D.

$$h(D)|_{t_i}^{t_{i+n}} = \frac{|D_{i+n}| - |D_i|}{|D_i|} \tag{3.1}$$

The *shift* parameter shows how a dataset evolves with respect to its size, through different points in time. Its directionality is captured by signed values, i.e., a positive shift points to *incremental* datasets, whereas a negative shift points to *decremental* datasets. It captures the relative different of additions and deletions between two fixed time points, and as a parameter it allows for generating increasingly larger or decreasingly smaller versions through the generation process. In EvoGen, $h(D)|_{t_i}^{t_{i+n}}$, is evenly distributed between all versions $D_i \dots D_{i+n}$.

The *monotonicity* of a dataset D determines whether an incremental or decremental shift changes D monotonically in a given time period $[t_i, t_j]$. A monotonic shift requires additions and deletions to not coexist within the same time period. Therefore, the set of triples that occur in a series of consecutive versions of D between t_i and t_j will be strictly increasing for a monotonic incremental shift, and strictly decreasing in a monotonic decremental shift. In order to make the ratio of low-level incremental (i.e., triple insertions) to decremental (i.e., triple deletions) changes quantifiable, we use the notion of *monotonicity rate*, denoted as $m(D)|_{t_i}^{t_{i+n}}$, as a parameter between 0 and 1:

$$m(D)|_{t_i}^{t_{i+n}} = \frac{|t_a|_i^{i+n}}{|t_a|_i^{i+n} + |t_d|_i^{i+n}} \tag{3.2}$$

where $|t_a|_k^l$ and $|t_d|_k^l$ the number of added and deleted triples between time points t_k and t_l. Formally, we define a dataset D to be monotonically incremental when:

$$h(D)|_{t_k}^{t_l} > 0 \quad \text{and} \quad m(D)|_{t_k}^{t_l} = 1$$

, or more intuitively, when the shift is incremental and there are no triple deletions between t_k and t_l. In a similar way, we define a dataset to be monotonically decremental when

$$h(D)|_{t_k}^{t_l} < 0 \quad \text{and} \quad m(D)|_{t_k}^{t_l} = 0$$

, or more intuitively, when the shift is decremental and there are no triple additions between t_k and t_l.

3.4.3.2 Parameters regarding schema evolution.

The *ontology evolution* parameter of a dataset represents the change on the ontology (i.e., schema) level, based on the change in the number of total classes in the schema. It can be used in conjunction with the *schema variation* parameter that will be defined in what follows. The *ontology evolution* parameter, denoted as $e(D)|_{t_k}^{t_l}$, is the ratio of new classes to the total number of classes in t_l:

$$e(D)|_{t_i}^{t_{i+n}} = \frac{|c_{i+n}| - |c_i|}{|c_i|} \tag{3.3}$$

where $|c_i|$ is the total number of ontology classes at time t_i. We then go on to define the *schema variation* parameter. Schema variation, denoted as $v(D)|_{t_k}^{t_l}$, is a property that represents that variations in the structure of the schema's properties that a dataset D exhibits through time. Because of the schema looseness typically associated with RDF, we recall the notion of Characteristic Sets [NM11] as the basis for $v(D)$. A characteristic set of a subject node s is essentially the collection of properties p that appear in triples with s as subject. Given an RDF dataset D, and a subject s, the Characteristic Set $S_c(s)$ of s is:

$$S_c(s) = \{p \mid \exists o : (s, p, o) \in D\}$$

and the set of all S_c for a dataset D at time t_i is:

$$S_c(D) = \{S_c(s) \mid \exists p, o : (s, p, o) \in D\}$$

The total number of permutations of the properties associated with a given class gives a maximum number of characteristic sets associated with that class, minus the empty set, which gives $2^n - 1$ total permutations of properties for a class c_i. Given this, we consider $v(D)|_{t_k}^{t_l}$ to be a constant parameter between 0 and 1 that quantifies the number of permutations that the generator will actually generate in the evolving process. Therefore, for all classes $|c|$ of a dataset D, the total number of permutations for a given time period is given by the following:

$$E(D)|_{t_k}^{t_l} = v(D)|_{t_k}^{t_l} \times \sum_{i=1}^{|c|} 2^i - 1 \tag{3.4}$$

We call E the *schema evolution* parameter. In essence, (4) quantifies the number and quality of schema changes in the dataset as time passes.

3.4.3.3 Parameters regarding query workload generation.

EvoGen generates a query workload that is based on six query types associated with evolving data defined in the previous chapter. We briefly provide an overview of the query types and the generated workload in the following:

1. Retrieval of a diachronic dataset. This type of query is used to retrieve all information associated with a particular diachronic dataset, for all of its instantiations. It is a workload-heavy CONSTRUCT query that either retrieves already fully materialized versions, or has to reconstruct past versions based on the associated changes.

2. Retrieval of a specific version. This is a specialization of the previous type, focusing on a specific (past) version of a dataset. The generator has to be aware of the context of the process, and create a query that refers to an existing past version.

3. Snapshot queries on the data. For this type of query, we use the original 14 LUBM queries and wrap them with a named graph associated with a generated version.

4. Longitudinal (temporal) queries. These queries retrieve the timeline of particular subgraphs, through a subset of past versions. For this reason, we use the 14 LUBM queries and wrap them with variables that take values from particular version ranges, and we order by ascending version in order to provide a valid timeline.

5. Queries on changes. This type of querying is associated with the high level changes that are logged in the change set between two successive versions. We provide a set of simple change queries that provide the ability to benchmark implementations that extract and store changes between RDF dataset versions, represented in the change ontology model.

6. Mixed queries. These queries use sub-queries from a mixture of the rest of the query types, and provide a way to test implementations that store changes alongside with the data and its past instantiations.

3.4.3.4 Parameters regarding type of archive.

Finally, we provide some degree of configurability with respect to EvoGen's serialized output. More specifically, we allow for the user to request fully materialized versions, or the full materialization of the first version followed by a series of deltas. This allows using the generated data in scenarios where the archiving process uses different archiving strategies, such as full materialization of datasets, delta-based storage, and hybrid storage, which is a combination of the two. For a discussion of different archiving strategies, the reader is referred to [FPU15; SCF14].

3.5 Implementation

The system extends the Lehigh University Benchmark (LUBM) generator, a Java based synthetic data generator. The LUBM ontology is extended to include 10 new classes and 19 new properties, which serve as a basis for implementing schema evolution functionality. Specifically, we have implemented schema evolution on top of the original ontology, without affecting the original ontology's structure for backwards compatibility.

The high-level architecture of EvoGen can be seen in Figure 3.1. The implemented functionality includes instance-level monotonic shifts, as well as schema-level evolution by class-centric generation of characteristic sets. The parameters that can be provided as input by the user in EvoGen are as follows:

1. number of versions: integer denoting the total number of consecutive versions. The number of versions needs to be larger than 1 for evolving data generation, else the original LUBM generator is triggered.
2. shift: the value of shift as defined in equation (3.1) of section 4, i.e., $h(D)|_{t_i}^{t_j}$, for a time range $[t_i, t_j]$, represents the percentage of change in size (measured as triples) between versions D_i and D_j. Currently, EvoGen generates monotonically incremental and decremental shifts between consecutive versions, and distributes the changes between all pairs of consecutive versions.
3. monotonicity: A boolean denoting the existence of monotonicity in the shift, or lack thereof.
4. ontology evolution: this parameter denotes the change in ontology classes with respect to the original ontology of LUBM, as given by equation (3.3).
5. schema variation: this parameter is used to quantify the total number of permuted characteristic sets that will be created for each new class introduced in the schema, as defined by equation (3.4).

The *Version Management* component and the *Change Creation* component are the main components that deal with translating the input parameters to actual instance/schema cardinalities and weights. They compute how many new instances have to be created or how many existing instances have to be deleted for each class of the LUBM ontology, without affecting the structure of the data and the distribution of instances per class.

The functionality is exposed through a Java API that can be invoked by importing EvoGen's libraries into third party projects[1].

[1]Source code is available at: https://github.com/mmeimaris/EvoGen

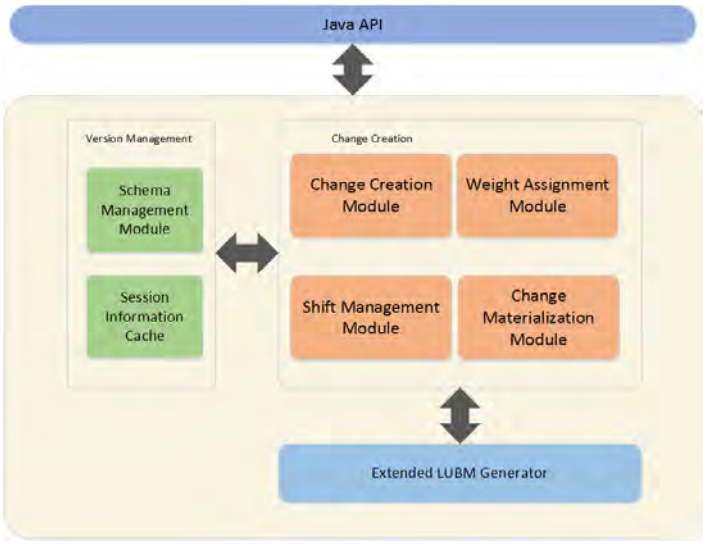


Figure 3.1: High-level architecture of *EvoGen*.

The actual distribution of triple insertions and deletions is performed dynamically in a process that takes into account session information on the evolution context of the generation. The process also involves several degrees of randomization with respect to URI and literal values, cardinalities of inter-class properties, selection of characteristic set permutations and so on. This component is responsible for all interactions with the *Extended LUBM Generator component*, which performs the actual serialization of dataset versions in the file system. In order to distribute the computed changes, we perform weighting to each class and derive concrete numbers for the instance cardinalities. This weighting is done in the *Weight Assignment Module*, which uses normalized weights in the range of 0..1 for each class, based on studying LUBM's original data structure and total instances per class for various input dataset sizes. By multiplying these weights with the desired shift value $h(D)|_{t_i}^{t_j}$, we end up with an approximation for the total number of instances per class.

The Change Materialization module is responsible for creating the change log file. It interacts with the Change Creation module sequentially, and creates an instance of the Change Ontology for each insertion and deletion of class instances.

The Version Management component keeps session information on each version during runtime, the schema of the dataset, the newly introduced classes and characteristic sets per version, the mapping of dataset versions to their respective files and folders in the file system and so on. Also, it is responsible for generating different types of archives, based on the user input; it can generate successive full materialized datasets without any change set produced, or change-based archives that includes an initial dataset with all successive deltas, or finally combinations of these approaches (hybrid storage).

3.6 Evaluation

In order to evaluate and validate the output of *EvoGen*, we perform a series of generation tasks for different combinations of numbers of universities and changes, and a fixed number of 10 versions, and we measure the achieved *shift* with respect to the required one. Specifically, we perform 10 runs of generations for three different values of h, namely $h(D) = 0.2$, $h(D) = 0.4$, and $h(D) = 0.6$ and we report the percentage difference between the mean of the achieved h and the required one. The results can be seen in Figure 3.2. With a small number of universities, the achieved shift differs significantly with respect to the required one, but as the number of universities, i.e. the dataset size, increases, the error decreases. Therefore, for a reasonably large number of dataset size, (e.g. > 5 universities), *EvoGen* performs as expected. Note, however, that this evaluation does not take into account scalability and efficiency issues, which is left as future work.

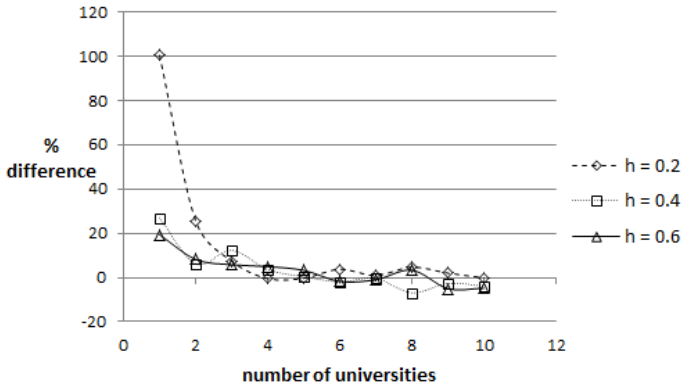


Figure 3.2: Average of achieved shift over 10 runs for 10 versions, for increasing number of universities.

3.7 Conclusions and Future Work

In this chapter, we described *EvoGen*, a system for synthetic and evolving data generation, with instance and schema level capabilities. Furthermore, EvoGen provides adaptive workload generation, that creates queries based on the user's choice of query types and the context of the generated data.

As future work, we intend to address issues of scalability and efficiency, as well as provide thorough experimental evaluation of the system by using it to benchmark existing RDF versioning solutions.

Chapter 4

Indexing and Query Optimization in RDF Graph Datasets

4.1 Introduction

In recent years, the Web of Data has been established as a vast source of data from diverse domains, such as biology, statistics, finance, and health. As these data become larger and wider in range, complex queries start to emerge, calling for improvements in the performance of RDF storage and querying engines.

In the case of indexing and query processing, traditional approaches often rely on permuting a single table with three columns, representing the subject, predicate and object (SPO) of a triple, in order to store the triples with different relative orderings. For example, the high-performance store RDF-3x [NW10] uses all six permutations of the SPO table, namely SPO, SOP, PSO, POS, OSP, and OPS, and maintains interesting orders on the index attributes in order to allow for as many merge joins as possible in a given query plan. In a similar way, the open source version of Virtuoso 7.2 relies on full and partial permutations, also catering for named graphs. Query planning and execution on these systems rely on the *data independence assumption*, which ignores any inherent structure in the data. Thus, optimizers mostly rely on first-level statistics, such as the number of distinct triples with a particular property, and heuristic estimations on join cardinalities. While these techniques are efficient for evaluating queries with small numbers of unbound variables and short paths, their performance degrades when adding complex, multi-join query patterns with potentially low selectivity and large intermediate results between joins. This shortcoming is more evident in Table 4.1, where we present the running times from the execution of two queries requiring multiple joins[1] on the Reactome and

[1]Query Q9 from the LUBM experiments and Q8 from the Reactome experiments

Table 4.1: Runtimes in seconds

	axonDB	RDF-3x	Virtuoso 7.2	TripleBit
Reactome	0.016	4.7	8.1	2.6
LUBM	0.23	8.2	timeout	timeout

the LUBM100 datasets using three state of the art RDF query engines, namely RDF-3x, Virtuoso Opensource 7.2[EM10], and TripleBit[Yua+13]. Even though the datasets are relatively small (∼16m triples in Reactome, ∼17m triples in LUBM100 with transitive closure), the engines fail to produce results fast. Other approaches, like property tables [Wil06] and vertical partitioning [Aba+07], have been proposed for dealing with diversity in the structure. These group or partition the data based on the sets of properties emitting from each node class. However, these approaches suffer from added space overhead for null values in sparse properties [Aba+07] and performance reduction as the dataset size increases [JK05].

This is in part due to the generic nature of the indexes that do not take into account the inherent structure of the data, and in that it involves relying on first-level statistics, such as the number of distinct triples with a particular property, or estimations on join cardinalities. This bias gives rise to optimizers that perform very well on particular query patterns, but poorly on others. For this reason, large complex queries that involve long paths in the data can lead to erroneous plans with large intermediate results, or plans that work on very large subsets of the original data in the case of queries with low selectivity.

Specifically, these approaches tend to be problematic when answering queries that contain long paths (*chains*) in the data and descriptive star patterns around the chain nodes, i.e., queries with an abundance of subject-object, and subject-subject joins, which we call *multi-chain-star* queries herein. Such an example is shown at the top of Fig. 4.1; its evaluation on the RDF graph is marked with bold edges at the left of the figure. In fact, these types of joins are very frequent in real world data, making up for 35% of all joined patterns[2] in empirical studies [Ari+11]. Recent approaches [Pap+14], [Sch+16b], [Wu+14], [Zen+13] attempt to speed up the query performance over very large datasets by distributing data and scaling out joins into multiple nodes; still, their query processing, although distributed, relies on the data independence assumption, thus moving the aforementioned limitations to a distributed setting.

In this chapter, we focus on the limitations of the core modules (i.e., indexing schemes) of RDF systems to answer complex queries even in relatively small datasets. We present

[2]In the same study, 60% of the join types are subject-subject joins, thus subject-subject and object-subject joins make up for 95% of all join types.

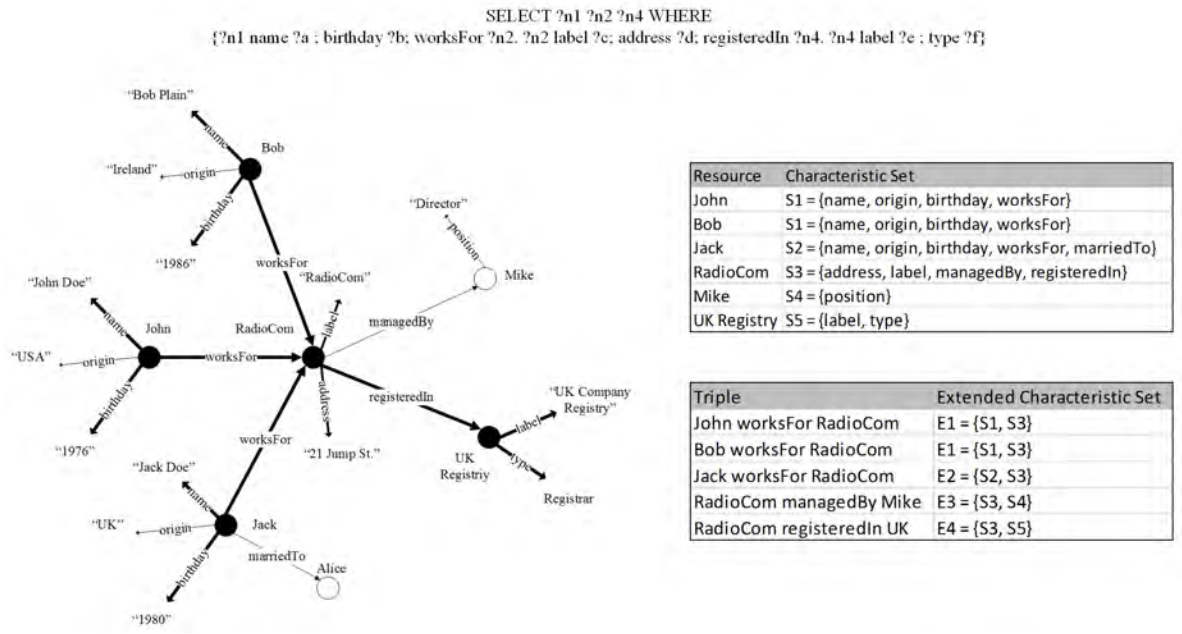


Figure 4.1: An RDF graph (left), its Characteristic Sets (top right), and Extended Characteristic Sets (bottom right). The evaluation of the query shown on the top of the figure is marked with bold nodes and edges on the graph.

a novel indexing scheme, the *ECS index*, that aims at accelerating query processing for conjunctive queries with multi-chain-star patterns. The ECS index is based on the notion of *Extended Characteristic Set* (ECS), which captures the inherent structure of subject-object relationships in an RDF graph. An ECS corresponds to a different type of subject-object relationship by comprising the different types of triples (i.e., properties) of the adjacent nodes. We construct an index that maps a triple to an ECS, and we present an efficient approach that evaluates conjunctive SPARQL queries with multi-chain-star patterns based on this index. An example RDF graph with four ECSs is shown in Fig. 4.1. ECS E_1 corresponds to the type of relationship between the nodes *John* and *RadioCom*, as well as *Bob* and *RadioCom*; it comprises the properties (*name,origin,birthday,worksFor*) and (*address,label,managedBy, registeredIn*), respectively. In the same way, all ECSs of Fig. 4.1 are constructed and all triples in the RDF graph are partitioned based on the ECS they belong to. ECS are defined as an extension of *Characteristic Sets* [NM11] to represent *the structure of triples*, as opposed to nodes, in the data.

This partitioning requires less storage overhead, compared to the permutation approaches, by not relying on excessive replications, and decreases the effects of bad estimates by quickly accessing triples that collectively participate in multiple joins. In short, the contributions of this work are the following:

- We define *Extended Characteristic Sets* (ECS) as a schema abstraction for collections of triples, based on the work in [MP16a] and [Mei+17],
- we present an algorithm for efficient extraction of ECSs in RDF datasets, as well as extraction of ECS graphs that represent paths in the data,
- we present an algorithm for query processing on top of an ECS index,
- we implement the approach in *axonDB*, a reference engine for ECS indexing and query processing that handles conjunctive multi-chain-star queries, and
- we evaluate its performance on one synthetic and two real datasets with respect to storage and querying, and we compare it with three widely used systems; our tool outperforms the competition by 1-3 orders of magnitude, both in the case of selective and unselective queries.

The rest of this chapter is organized as follows. Section 2 provides preliminary definitions for RDF and SPARQL, and defines Extended Characteristic Sets (ECSs) and ECS graphs. Section 3 presents algorithms for extracting characteristic sets and extended characteristic sets, and for constructing the index. In Section 4, we discuss query processing based on this index, and in Section 5 we present an experimental evaluation on synthetic and real-world data. Finally, Section 6 presents related work, and Section 7 concludes the chapter.

4.2 Related Work

RDF data management systems follow three storage schemes, namely *triples tables*, *property tables*, and *vertical partitioning*. A triples table has three columns, representing the subject, predicate and object (SPO) of a triple. This technique usually replicates data in different orderings of SPO in order to facilitate sort-merge joins. For example, RDF-3X [NW10] and Hexastore [WKB08] build tables on all six permutations of SPO, while RDF-3x also employs indexes for binary and unary projections of the original SPO data. Similarly, Virtuoso [EM10] uses a large 4-column table for quads, and a combination of full and partial indexes, while Jena TDB relies on three permutations. Other centralized RDF systems are built on top of relational backbones, such as Jena SDB, Virtuoso, and DB2RDF [Bor+13]. These methods have been established in centralized systems and in fact work well for selective queries with small numbers of joins, however, they tend to degrade with increasing dataset sizes, large numbers of unbound variables and decreasing selectivity, as the required index scans become larger. Furthermore, the storage overhead can become a limiting factor when scaling for very large datasets.

In distributed settings, a growing body of literature exists, with systems such as H2RDF+ [Pap+14], S2RDF [Sch+16b], SemStore [Wu+14], and TrinityRDF [Zen+13]. H2RDF+ employs all six permutations of the triples table, implemented over Hadoop and HBase. S2RDF uses a vertical partitioning schema named ExtVP, that takes into account the joins between vertical partitioning tables, while SemStore focuses on the partitioning aspects of data in different nodes, and uses TripleBit[Yua+13] in its reference implementation. TrinityRDF is designed to work in memory, and thus has no disk-based storage component. However, our focus is on the limitations of the core aspects of centralized RDF systems to answer complex queries even in relatively small datasets, such as Reactome in our experiments. Thus, it is out of scope to perform a quantitative comparison with distributed RDF engines, and we leave it as future work to assess how ECS indexing can work on distributed settings.

Property Tables [Wil06; Aba+07] is a technique that places data in one or multiple tables, the columns of which correspond to the properties of the dataset. Each row identifies a subject node and holds the value of each property in the corresponding cells. However, this causes extra space overhead for null values in cases of sparse properties for a given class[Aba+07]. Also, it raises performance issues when handling complex queries with many self-joins, as the amounts of intermediate results tend to be significant, especially for increasing sizes of datasets [JK05].

Vertical partitioning is a technique that partitions data in tables with two columns. Each table corresponds to a property in the data, and each row to a subject node [Aba+07]. This approach provides great performance when evaluating queries with bound objects, but tends to suffer when the sizes of the tables have large variations in size [Sid+08]. TripleBit [Yua+13] is an RDF store that broadly falls under the vertical partitioning type, but uses bitmaps to store the occurrence or absence of predicate-object pairs in a table where rows represent subject nodes. In TripleBit, the data is vertically partitioned in chunks per predicate. While this approach is efficient for reducing the amount of replication in the data, it suffers from the same problems as property tables. It does not consider the inherent schema of the triples in order to speed up the evaluation of complex query patterns, as is the case for axonDB.

Emergent schema extraction has been studied in [PB16], the authors group together CSs based on semantics and structure, in order to form a much smaller set of tables compared to the entire set of CSs. Our work, although in the same direction, is technically different, because we use the notion of ECS, and focus on ECS-based methods. CSs have been introduced as an abstraction of node types, and used for provision of better estimates of join cardinalities [NM11]. In this regard, Brodt et al [BSM11] present their approach on

how the SPO index can be used to identify CSs and assist query processing by decreasing the number of SS joins that are common in star patterns. We follow this approach in axonDB with the use of the CS index, which is an SPO permutation partitioned among all CSs of a dataset. Our notion of the ECS is in fact inspired by the Characteristic Set, but focuses on triples, rather than nodes. In [MP16a] we have presented a layout of similar indexing, without providing an implementation or algorithmic contributions. To the best of our knowledge, this is the first work to use such a structure for RDF indexing and query processing.

4.3 Preliminaries

RDF and SPARQL. RDF models facts about entities in a triple format consisting of a subject s, a predicate p and an object o. A collection of triples is usually represented as a directed labelled graph with subjects and objects being the nodes, and predicates being the edges of the graph. Formally, let I, B, L be infinite, pairwise disjoint sets of IRIs, blank nodes and literals, respectively. Then, an RDF triple t is represented by a triple $(s, p, o) \in (I \cup B) \times (I) \times (I \cup B \cup L)$ and a collection of triples $\{t_1, t_2, \ldots, t_n\}$ is represented by an RDF graph, in which every node $n \in T = (I \cup B \cup L)$ and every edge $e \in I$.

Following this notation, a SPARQL query defines a set of triple patterns of the form $(T \cup V) \times (I \cup V) \times (T \cup V)$, where V is the set of variables that can be bound to T. Triple patterns can be recursively combined via AND, $OPTIONAL$ and $UNION$ operators.

4.3.1 Extended Characteristic Sets (ECS)

One of the benefits of RDF is that it is loosely structured; one can extend and modify the schema at will, by adding or deleting new triples for properties and classes. Neumann and Moerkotte [NM11] introduced the notion of *characteristic sets* as a means to capture the underlying structure of an RDF dataset. A characteristic set CS identifies node types based on the set of properties they emit. Formally, given a collection of triples D, and a node s, the characteristic set $S_c(s)$ of s (or simply S_s) is:

$$S_c(s) = \{p \mid \exists o : (s, p, o) \in D\} \tag{4.1}$$

and the set of all S_c for a dataset D is:

$$S_c(D) = \{S_c(s) \mid \exists p, o : (s, p, o) \in D\} \tag{4.2}$$

Characteristic sets provide a node-centric partitioning of an RDF dataset, based on the structure of a node, and they have been used effectively in the characterisation of joins and cardinality estimation [NM11]. However, they cannot capture the different relationships of nodes in a dataset, i.e., how triples, instead of nodes, can be partitioned based on their characteristics. For this reason, we introduce the *Extended Characteristic Set* (ECS), the *triple-level analogue* of the node-based characteristic set. An ECS captures the inherent schema of triples, based on the properties of their adjacent nodes, i.e., the characteristic sets of the subject and the object. Formally, given a triple (s, p, o), the ECS $E_c(s, o)$ is an ordered set containing the characteristic sets of s and o:

$$E_c(s, o) = \{S_c(s), S_c(o) \mid \exists p : (s, p, o) \in D\} \tag{4.3}$$

which is shortly denoted as $E_{s,o}$. The set of all ECS in D is:

$$E_c(D) = \{E_c(s, o) \mid \exists p : (s, p, o) \in D\} \tag{4.4}$$

An ECS helps to quickly identify the largest superset of graph patterns that contain a star pattern around s, a star pattern around o, and an edge from s towards o. In the example of Fig. 4.1, where nodes $John$ and $RadioCom$ are present in the same triple $\langle John, worksFor, RadioCom\rangle$ as subject and object respectively, and descriptive star patterns are present for each of the two nodes. In a similar manner, an ECS is formed between Bob and $RadioCom$, $Jack$ and $RadioCom$, $RadioCom$ and $Mike$, as well as $RadioCom$ and $UK_{Registry}$. Note that, by definition, if two nodes n_1 and n_2 are linked with multiple properties, these are part of the same ECS, which is defined by all the properties from n_1 to n_2, along with the rest of the properties emitting from n_1 and n_2.

Each triple (s, p, o) corresponds to one and only one ECS, i.e., $E(s, o)$. The upper bound for $|S_c(D)|$ is the distinct number of subject nodes, i.e., nodes that emit property edges, however, the existence of an inherent structure in RDF data makes the distinct set of Characteristic Sets that appear in real-world data small [NM11]. Similarly, the maximum number of ECSs in a given dataset is $|S_c(D)|^2$, that is, one ECS for each pair of characteristic sets. However, in practice, we observe that triples are partitioned in tractable numbers of ECSs, as it can be seen in Table 4.2 for several real-world and synthetic datasets.

4.3.2 ECS Graphs and ECS Query Graphs

ECSs can be combined to form a directed graph that captures transitive relationships between characteristic sets in an RDF dataset. This is useful for representing paths between types of s, o pairs. An ECS graph is a directed graph $G_E = (V_E, E_E)$ where $V_E \in$

Table 4.2: Observed cardinalities of properties, CS and ECS in synthetic and real data.

	LUBM	BSBM	WordNet	Reactome	EFO	GeoNames	DBLP
#properties	18	40	64	65	80	36	26
#CS	14	44	779	112	520	851	95
#ECS	68	374	7250	346	2515	12136	733

$E_c(D)$, and $E_E \in (V_E \times V_E)$ are the nodes and edges of the graph, respectively. A node in G_E corresponds to an ECS of the RDF dataset. A directed edge $e = (E_{n_1,n_2}, E_{n_2,n_3})$ exists when there is at least one triple t_a with ECS E_{n_1,n_2}, whose object is the subject of a triple t_b with ECS E_{n_2,n_3}. In other words, an edge between two ECSs represents two sets of triples whose schemas form subject-object joins in the dataset. An ECS chain c_E is a path formed by consecutive edges between ECSs in an ECS graph. An example of an ECS graph for a given RDF graph is depicted on the right of Fig. 4.1, where we show the object-subject joins between ECSs of the RDF graph. Consider the query q listed at the top of the figure. The query defines a chain from n_1 to n_4 through n_2, along with star patterns around n_1, n_2 and n_4. Its evaluation can be seen with bold edges in both the RDF and the ECS graph.

An ECS graph provides a suitable abstraction for traversing long paths in the RDF graph efficiently, without spending computational resources in the execution of subject-subject self-joins that usually have low selectivity and generate large intermediate results [Tsi+12]. Instead, it treats subject-object joins as first-class citizens. With the help of ECS-based preprocessing and indexing, queries can be evaluated on top of the dataset's ECS graph, by (i) quickly assessing the existence of one or more ECS sub-graphs that are super-sets of the query graph, and (ii) finding a minimal set of triples that contribute to the evaluation of the query. The first point is important for determining whether large, complicated queries have non-empty results, while the second point allows us to access and process a small subset of the data that is sure to contribute to the query processing stage. The latter point is of particular interest when handling complicated queries of long paths with many unbound variables, and helps avoid large intermediate results.

Given the above, we propose to extract the ECSs out of a query graph and map them to the dataset's ECS graph space. A query pattern q is mapped to the ECS query graph Q_E based on the identification and extraction of the ECSs of the triple patterns in q. Formally, a small modification to the ranges in the original definition of characteristic sets [NM11] is needed in order to allow variable nodes to instantiate characteristic sets as well. Specifically, a characteristic set $S_c(s_q)$ of a node s_q in a query pattern is allowed to be defined over unbound, as well as bound instances of s_q, and unbound or bound

instances of predicates and objects in the triple patterns with s_q as subject, i.e., $S_c(s_q) = \{p_q \mid \exists o_q : (s_q, p_q, o_q) \in q\}$ $, (s_q, p_q, o_q) \in (I \cup B \cup V) \times (I \cup V) \times (I \cup B \cup L \cup V)$.

We have implemented all aforementioned concepts in a native RDF engine, called *axonDB*. Its overall architecture is shown in Fig. 4.2. There are three core modules, responsible for a) loading a new RDF dataset and extracting the CS and ECS, b) constructing and storing the CS and ECS indexes and c) processing a SPARQL query and fetching the results. Next sections present the technical details for each module.

4.4 Loading and Indexing

In this section, we provide methods and algorithms for efficient extraction of CS and ECS from datasets, and show how the ECS structure is used for triple storage and indexing.

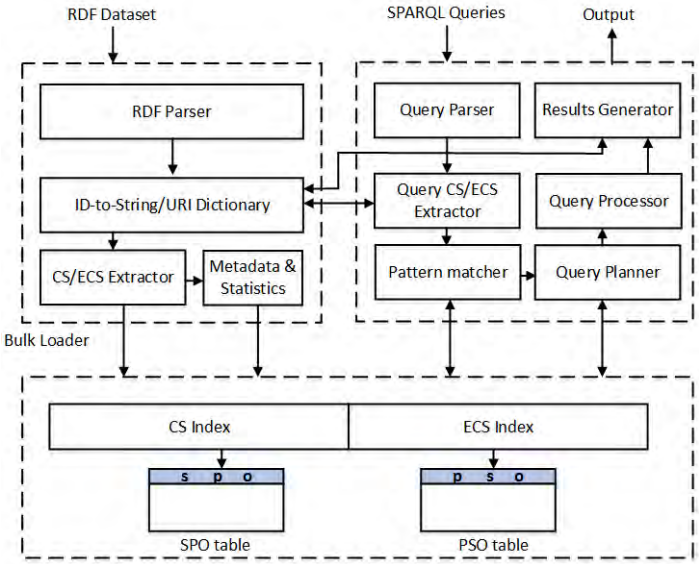


Figure 4.2: Overview of system architecture.

4.4.1 Data Loading.

In axonDB, triples are stored on disk as three consecutive integers of 4 bytes, one for each triple component, namely subject, predicate and object, as is typically done in RDF stores [Atr+10; NW10; Yua+13]. The id assignment is performed during initial parsing of the input, and the references are stored in memory during the loading phase until they are flushed to disk in bulk. During the loading phase, each triple is modelled as a vector of size 4. The first three positions hold the subject, predicate, and object ids, and the last position points to the CS of its subject.

Notice that we reserve 4 bytes (32-bit integers) for each component during the loading phase, instead of maintaining an encoding of varying size. The usefulness of this verbosity will become clear later as we use this structure in order to sort the triples both by subject and CS. Furthermore, it is relatively affordable, as even for 1 billion distinct ids, the system needs 4 GB of RAM while loading the data. In any case, this structure is held off-heap and is backed by a memory mapped file in order to avoid overflows during data loading.

A dictionary is built during parsing, that holds values for the node and predicate ids (IRIs), as well as for the literals. IRIs are compressed based on their prefixes in order to avoid tedious duplications of strings that occur frequently in RDF datasets. The dictionary is then used during query parsing, in order to map bound values from the query to the actual RDF data in the system, as well as to generate the human readable results.

4.4.2 Characteristic Set Extraction and CS Index.

A characteristic set $S_c(s)$ is a a set of common properties $p_1, \ldots, p_n$ that are emitted from a set of subject nodes. The set of all CS $S_c(D)$ can be easily retrieved with a linear scan on the triples of a dataset [BSM11],[NM11]. The algorithm is presented in Algorithm 1. We sort the triples by subject and construct a new CS each time a new combination of properties is found in a subject, i.e., while we iterate through triples with the same subject, we aggregate the properties of these triples, and when the iteration moves on to the next subject, we hash the bitmap of the aggregated properties and check if it already exists. If not, we create a new CS with these properties. Each CS is assigned a unique integer identifier, and holds a bitmap of the properties that define it, where each bit corresponds to the presence of a property[3] in D (e.g., for k properties in D, we construct a bitmap of length k; in typical datasets - see *properties* row of Table 4.2 - k is small enough for the bitmap to fit in a few bytes). This is useful for fast subset checking during the query preprocessing phase, as will be discussed. During this iteration, we associate a triple to a CS, based on the CS of the *subject node*, by setting the fourth element of the triple's vector to the integer identifier assigned to the CS.

We then sort the triples by their CS, maintaining the subject as the secondary sort key, and iterate to construct a big triples table in the SPO ordering for persistent storage. Algorithm 1 does not show this step, as it returns a mapping from CSs to sets of triples. It is trivial to iterate through the keys of this mapping (*csMap*) and flush each CS's

[3]The properties are ordered as they appear in the first iteration of the input triples. We use this predicate ordering as a reference for all other stuctures and indexes that use it.

triples to the persistent storage sequentially. The *CS index* is constructed on top of this table as a B^+-tree, where the keys are defined by the id of the CSs. We can use this index to get the triples associated with a specific CS, by maintaining the start and end indexes of each CS in the SPO table. This way, the *CS Index* partitions all triples based on their subject's CS and allows us to easily evaluate properties in star patterns around a given node or variable, with simple range scans. For our running example, the SPO table and its CS index can be seen in Fig. 4.3. The CS id's refer to the characteristic sets that were shown in Fig. 4.1.

Algorithm 1: *extractCharacteristicSets*

Data: *triples*: A $N \times 4$ table of ids, where N is the number of triples in the input. The first three columns are used for subject, predicate and object ids, and the fourth column is used for CS ids.

Result: *csMap*: An inverted index, with CS ids as keys, and sets of triples as values.

```
1  sort(triples) by subject ;
2  properties ← new Set() ;
3  previousSubject ← triples[0][0] ;
4  lastIndex ← 0;
5  for each i = 1; i ∈ triples do
6  |  subject ← triples[i][0];
7  |  if previousSubject ≠ subject then
8  |  |  cs ← newCharacteristicSet(csId, properties);
9  |  |  for each j = lastIndex; j < i; j++ do
10 |  |  |  triples[j][3] ← csId ;
11 |  |  end
12 |  |  csId++;
13 |  |  lastIndex ← i + 1;
14 |  |  properties.clear();
15 |  properties.add(triples[i][1]);
16 |  previousSubject ← subject;
17 end
18 sort(triples) by CS ;
19 triplesToAdd ← newSet();
20 lastCS ← triples[0][3];
21 for each i ∈ triples do
22 |  if lastCS ≠ triples[i][3] then
23 |  |  csMap.put(lastCS, triplesToAdd);
24 |  triplesToAdd.add(triple[i]);
25 |  lastCS ← triples[i][3];
26 end
27 return csMap;
```

Analysis. Identification and extractions of CSs comprises sorting the triples once by

subject, and scanning. This costs $O(nlogn + n)$ for n triples. Furthermore, we sort the triples a second time, and iterate over them once more in order to store them on disk at the order of the CS's appearance, i.e., $O(nlogn + n)$. Not counting the cost of disk I/O, the time complexity of this step is $O(2nlogn + 2n)$.

4.4.3 Extended Characteristic Set Extraction and ECS Index.

The next step is to extract the ECSs, and build the *ECS index*. A naive way of extracting the ECSs is to perform an object-subject join on the whole dataset, scan the resulting rows and create a new ECS for each different combination of the subjects' and objects' CSs. A more efficient way is to take advantage of the previously computed CS Index.

Specifically, we utilize the CS Index and iterate through all *pairs* of CSs looking for subject-object joins in their chunks of triples that are held in *csMap* (see Algorithm 1). When the result of the join between the triples of S_1 and S_2 is non-empty, we construct a new ECS $E(S_1, S_2)$, based on the CSs of the triples' subjects. This join process enables us to identify an ECS and retrieve all triples associated with that ECS at the same step. In other words, given two characteristic sets S_1 and S_2, and two sets of triples T_1 and T_2, whose subjects belong to S_1 and S_2 respectively, the object-subject join between T_1 and T_2 will be non-empty when there exist triples that belong in S_2, whose subjects are objects in triples of S_1. We can then store the ECS $E(S_1, S_2)$ along with references to the identifiers of its subject and object CSs, as well as the triples contained in it. As with CSs, each ECS is assigned a unique integer identifier. In contrast to the *CS Index* that partitions all of the triples in a dataset, the *ECS Index* partitions only the triples that pertain to a valid ECS, i.e., whose subject and object have non-empty CSs. These are triples that describe paths between resources. We store these triples as a PSO table, and build the ECS Index as a B^+-tree on top of this table, where each ECS defines a range of consecutive triples that belong to it. This way, fetching the triples of a specific ECS requires a simple range scan over the PSO table. In our running example, the PSO table and ECS Index can be seen at the bottom of Fig. 4.3. Note that the size of the PSO table is smaller than the SPO table, which contains all triples of the input data. This is due to the fact that many triples do not belong to a valid ECS. These are either triples with literal objects, or triples with objects that do not have any emitting edges, and are thus described by an empty CS.

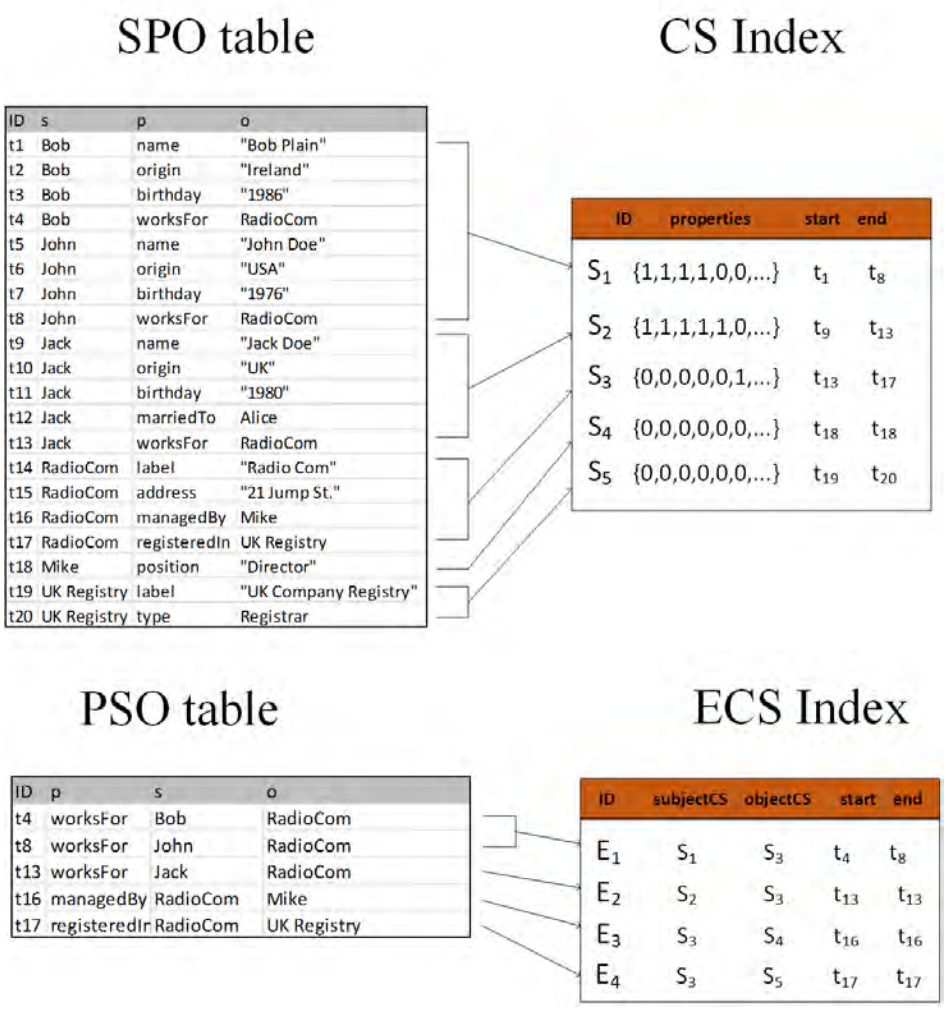


Figure 4.3: Example instantiation of the CS (top) and the ECS (bottom) indexes. The CS contains the bitmap of a set of properties $p_i..p_k$, while the ECS is a composition of a subject CS and an object CS.

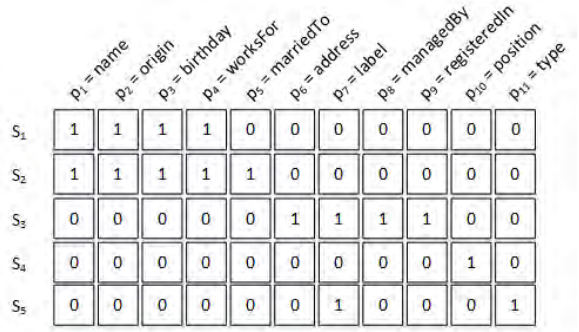


Figure 4.4: Property bitmaps of CSs $S_1 \ldots S_5$.

Algorithm 2: *extractExtendedCharacteristicSets*

Data: *csMap*: An inverted index that maps characteristic sets to sets of triples based on the characteristic set that is defined by a triple's subject node.

Result: *ecsMap*: An inverted index that maps extended characteristic sets to sets of triples, *ecsLinks*: A set of adjacency lists with links between the retrieved ECSs.

```
1  ecsMap, subjectCSMap, objectCSMap ← new Map();
2  for each S_i ∈ csMap do
3      for each S_j ∈ csMap do
           /* Perform an object-subject join of triples in S_i, S_j        */
4          triples ← (csMap.get(S_i) ⋈|_{o-s} csMap.get(S_j));
5          if triples.size()! = 0 then
6              ecs ← newECS(S_i, S_j);
7              ecsMap.put(ecs, sort(triples));
8              subjectCSMap.get(S_i).add(ecs);
9              objectCSMap.get(S_j).add(ecs);
10     end
11 end
   /* Find links between ECSs                                              */
12 ecsLinks ← newMap();
13 for each S_i ∈ objectCSMap.keys() do
14     if S_i ∉ subjectCSMap.keys() then
15         continue;;
16     end
17     for each ecs_left ∈ objectCSMap.get(S_i) do
18         for each ecs_right ∈ subjectCSMap.get(S_i) do
19             ecsLinks.get(ecs_left).add(ecs_right);
20         end
21     end
22 end
23 return ecsMap, ecsLinks;
```

Algorithm 2 shows how ECSs are extracted and mapped to triples. The algorithm takes as input the *csMap* and results in two outputs: (i) a mapping of ECSs to sets of triples, and (ii) and a graph in the form of adjacency lists that represents the links between joinable ECSs. It iterates through all pairs S_i, S_j of CSs (lines 2-4), performs an object-subject hash-join of their triples T_i, T_j, (line 5) and if the result is not empty, it creates a new ECS and maps the triples to it, sorted in the PSO order (lines 6-8). This ordering is useful for early filtering of triples with properties not in the query pattern. After retrieving the ECSs, the algorithm finds directed links between ECSs (lines 11-15) to construct the *ECS graph*. It first populates the *subjectCSMap* and *objectCSMap* that link CSs to ECSs based on their position in the ECSs (lines 9-10). Then, it looks for CSs that are both objects and subjects in different sets of ECSs, and links these together. The resulting

adjacency list represents the *ECS graph*, and is stored as part of the indexing scheme in axonDB. This is essential in the query preprocessing stage, in which an incoming query is matched to existing ECS paths in the data.

Analysis. For p CSs, the step of extracting ECSs contains iterating p^2 pairs of CSs. For each pair S_i, S_j, the asymptotic cost of a hash join over the triples T_i, T_j of S_i, S_j respectively, is $O(|T_i|+|T_j|)$. Assuming an even distribution of $|D|/p$ triples per CS, where D is the input dataset, the total cost of ECS extraction is $O(p^2 2\,|D|/p)$, or $O(p2\,|D|)$. The computation of *ecsLinks* entails building the $subjectCSMap$ and $objectCSMap$, which in the worst case when all p^2 pairs are valid, costs $O(p^2)$ insertions. Then, we have to find object CSs that are both in $subjectCSMap$ and $objectCSMap$, in order to ensure that an object CS of an ECS is also a subject CS of another ECS. This can be performed by iterating over $objectCSMap$, and for each CS iterating over pairs of ECSs that have the current CS as key in both $objectCSMap$ and $subjectCSMap$. Thus, the total cost of this step is $O(p^2 + p2\,|D|)$.

4.4.4 ECS Hierarchy

ECSs pertain to a hierarchical structure that defines ancestral, parent-child relationships between them. We consider that an ECS E_1 is a specialization of another ECS E_2 if it contains all properties of E_2, i.e., $E_1 \succ E_2$. As will be discussed in the next section, a query is broken down to query ECSs, derived from the query's graph pattern. In order to match a query ECS with an ECS from our index, we perform subset checking between the query ECS and the ECSs in the index. As a derivative, a query ECS is evaluated by all ECSs that contain all properties of the query ECS, as they appear in the respective subject and object CSs. As many of these matched ECSs in the index are hierarchically related, because they contain *at least* the same subset of properties, we can use this ECS hierarchy to improve disk I/O and naturally group together triples that belong to the same ECS families. In our running example, Fig. 4.4 shows the bitmaps of the CSs, where each cell denotes the presence (1) or absence (0) of a property p_i. The two ECSs E_1 and E_2 of Fig.4.3 are hierarchically related, namely $E_1 \succ E_2$, because $S_1 \subset S_2$, and S_3 is the same for both. To take advantage of this observation, we implement an optimization that sorts ECSs based on the pre-order traversal of this hierarchy, and stores triples in the order defined by this traversal. This means that the ECSs of consecutive chunks of triples on the disk will often be hierarchically related, in an attempt to minimize reading redundant pages from persistent storage. For computing the hierarchy from *ecsLinks*, we sort the ECSs based on the number of the properties they contain, because the less properties it contains, the more generic and higher in the hierarchy the ECS is. Then, we

iterate through the sorted ECSs, traverse their links from *ecsLinks* and build paths for each one, taking care to add only one level of children to each previous level of ancestors. This process results in a graph lattice, where the root nodes are the most generic ECSs (containing fewer properties), and the leaves are the most specialized ECSs (containing more properties).

4.4.5 Metadata and statistics

During the loading phase, axonDB computes and stores along with the ECS index, auxiliary metadata and statistics in order to assist the pre-processing stage of query evaluation. First, each ECS maintains pointers to the first occurrences of each property in the indexed PSO table. This helps us avoiding logarithmic searches in large triple collections during query evaluation. Then, it extracts all edges between ECSs in order to be able to traverse the ECS graph using standard graph traversal algorithms. The algorithm for extracting edges is based on finding ECSs that exhibit object-subject joins on the CS level. This is shown in lines 10-17 of Algorithm 2. Finally, it computes the cardinality of distinct properties in the triples of each ECS, as well as the cardinalities of distinct subjects and objects per ECS. These statistics are used by the query planner.

4.5 Query Processing

In this section, we discuss how query processing is performed on top of the ECS Index. Our goal is to employ the derived index structures in order to reduce the number of scans over the triples, number of joins, and the amount of intermediate results when evaluating SPARQL queries with multi-chain-star graph patterns. Still, our approach is efficient for simple query patterns as well. An overview of the query processing steps is shown in Fig. 4.5, from top to bottom. Given a query over our example dataset, we first parse the query statement and identify the characteristic sets around the chain variables, i.e., S_x, S_y, S_z, and S_w for $?x$, $?y$, $?z$, and $?w$ respectively. Then, we extract the query ECSs $Q_{x,y}, Q_{y,z}, Q_{y,w}$, and identify the chains between them as well as the type of joins to be performed; OS correspond to object-subject joins where the triples of an object's CS are joined with the triples of the subject's CS and SS denotes a subject-subject join. Finally, we match each query ECS to ECSs in the data and we generate the plan that retrieves and joins triples to output the result. Note that we consider the union of triples from E_1 and E_2 as $Q_{x,y}$ is matched to both ECSs. For simplicity reasons, we have omitted the step of processing the restriction of the bound *"Director"* node. This is performed when

retrieving E_3, by doing a semi-join with the triples of its object CS and filtering out by the bound object.

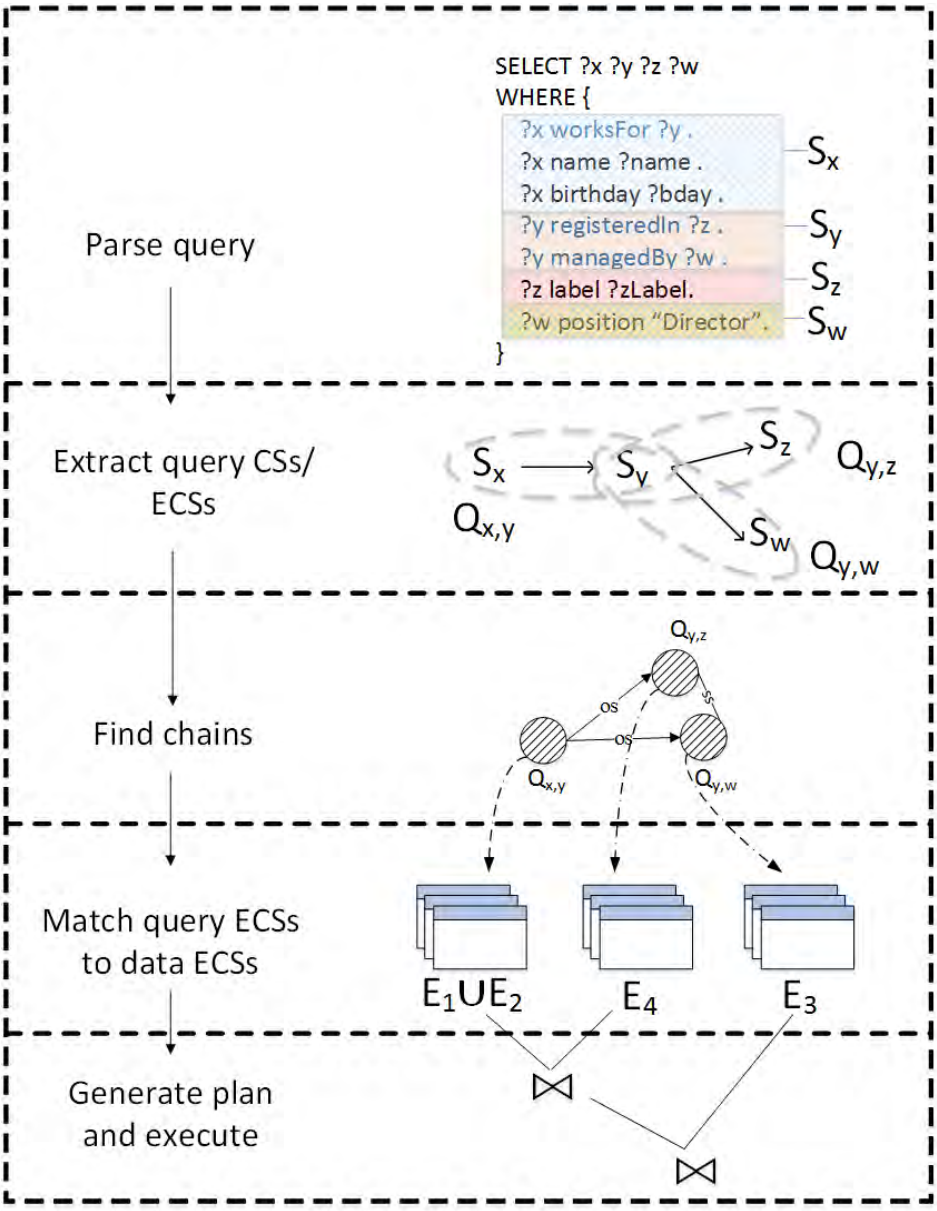


Figure 4.5: Query processing for two chain patterns of three query ECSs. Notice that $Q_{x,y}$ matches both E_1 and E_1.

4.5.1 Query parsing and ECS query graph extraction

Incoming queries are first converted to ECS query graphs by the query parser. This is achieved by first exracting the characteristic sets of the query's nodes, then applying Algorithm 2 to find the ECSs on the query pattern, to create adjacency lists between the query ECSs. This procedure is identical to the ECS extraction when loading the

data, but this time it is performed on the triple patterns of the query. During this step, the dictionary is used for id resolution of predicates and any other bound nodes in the patterns. Having identified the query ECSs and the links between them, we traverse these links in the order of their occurrence, in order to identify chains (i.e., series of object-subject joins) in the ECS query graph. This results in a set of *chain patterns* $c_1 \dots c_n$. Finally, we remove chains that are fully contained in other chains. Given that their number is rather small, this step is efficiently performed with a single nested loop over the set of chains.

4.5.2 Matching of query ECSs to the ECS index

Each query ECS Q_i is matched to zero or more ECSs in the ECS index. We say that there exists a match between a query ECS $Q_i = \{S_{q,left}, S_{q,right}\}$ and an indexed ECS $E_j = \{S_{j,left}, S_{j,right}\}$, when the following are true:

$$S_{q,left} \subseteq S_{j,left} \tag{4.5}$$

$$S_{q,right} \subseteq S_{j,right} \tag{4.6}$$

$$p(Q_i) \in p(E_j) \tag{4.7}$$

where $p(Q_i)$, $p(E_j)$) are properties of the triple patterns whose subject and object CSs form ECS Q_i and E_j, respectively. In our running example, this is the set of properties that appear in the P column of the PSO table for the same ECS. If we denote the set of all ECSs that match Q_i as $matches(Q_i)$, then when (5)-(7) are true, it holds that $E_j \in matches(Q_i)$. Aggregating the triples of all ECSs in $matches(Q_i)$, gives the *evaluation eval* of Q_i, which is given by the following:

$$eval(Q_i) = \bigcup_{n=1}^{k} T(E_n), E_n \in matches(Q_i) \tag{4.8}$$

$T(E_n)$ corresponds to the triples associated with E_n.

Matching of the query ECSs to the ECS index is performed through a depth-first traversal on the ECS graph. As shown in Algorithm 3, we iterate over the edges of *ecsLinks* (line 2), an adjacency list with the ECS graph, and for each ECS as a starting point, we search for matching ECSs. The DFS traversal is performed by the recursive method *matchDataPatterns* (lines 2-4) as detailed in Algorithm 4. The *matchDataPatterns* method takes as input a query ECS, an indexed ECS (the starting/current node for the dfs traversal) and the two ECS graphs (i.e., the query and the indexed ECS graph) and returns a linked list of ECSs in the data that match the query ECSs. It first evaluates whether the input query ECS and the ECS in the data satisfy the conditions (5)-(7)

(line 1-4), otherwise it returns an empty list. Subset checking is performed with bitwise operations on the property bitmaps. Specifically, a bitmap b_1 is a subset of a bitmap b_2 when it holds that $b_1 AND b_2 = b_1$. If all conditions are satisfied, it checks whether the examined ECS has already been visited or the query chain is empty (line 5) otherwise it marks the matching ECS as visited and adds it to the matching list of the input query ECS (Line 7-8). It then proceeds with the dfs traversal on the ECS graph (Line 9), and evaluates the matching ECSs for the rest of the chain starting from q_1, i.e., consecutive node in the query ECS chain. By performing depth-first traversal on the ECS graph, it is guaranteed that consecutively matched ECSs over the query are actually linked in the data, because each reached ECS will be a child of the preceding one. The output of this process is a set of ECS chains in the ECS graph that match the query's ECS chains. If the property of the ECS is unbound, then we match it to all properties found in the region of the PSO table that matches the rest of the ECS restrictions.

Algorithm 3: *matchQueryToECSIndex*

Data: *ecsLinks*: The ECS adjacency list, $c(q_0 \dots q_{n-1})$: A chain of query ECSs
Result: *ecsMatches*: A linked list of ECS sets that match the ECSs in c

```
1  ecsMatches ← newMap();
2  for each e ∈ ecsLinks.keySet() do
3  |  matchDataPatterns(e, ecsLinks, c(q_0 ... q_{n-1}), ecsMatches);
4  end
5  return ecsMatches;
```

Algorithm 4: *matchDataPatterns*

Data: *ecsLinks*: The ECS adjacency list, e: The ECS of the current iteration, $c(q_0 \dots q_{n-1})$: A chain of query ECSs, *ecsMatches*: A linked list of ECS sets that match the ECSs in c
Result: *ecsMatches*: A linked list of ECS sets that match the ECSs in c

```
1  if q_0.subjectCS.bitmap ⊈ e.subjectCS.bitmap
2  OR q_0.objectCS.bitmap ⊈ e.objectCS.bitmap
3  OR q_0.property ∉ e.properties then
4  |  return null;
5  if visited(e) OR c.size == 1 then
6  |  return ecsMatches;
7  visited.add(e);
8  ecsMatches.get(q_0).add(e);
9  for each e_child ∈ ecsLinks.get(e) do
10 |  matchDataPatterns(e_child, ecsLinks, c_{q_1...q_{n-1}}, ecsMatches);
11 end
```

In our running example of Fig. 4.1, the query of Fig. 4.5 defines three query ECSs, namely $Q_{x,y}$, $Q_{y,z}$ and $Q_{y,w}$, as can be seen in Fig. 4.5. The algorithm will match E_1, E_2 to $Q_{x,y}$ because the bitmap of S_x is a subset of both S_1 and S_2 that constitute the subject

CSs of E_1 and E_2 respectively, and the bitmap of S_y is a subset of S_3 which is the common object CS for both E_1 and E_2. In a similar manner, E_4 will be matched to to $Q_{y,z}$ and E_3 to $Q_{y,w}$.

4.5.3 Query planning.

The query planner decides the join execution order for the various sets of triples corresponding to the matched ECS chains of the previous step. The planner distinguishes between the *outer ordering* (evaluation of different chains) and the *inner ordering* (evaluation of a specific chain). The outer ordering is useful for filtering out triples as early as possible based on the common attributes of the different chain patterns. The inner ordering helps reduce intermediate results in object-subject joins between ECSs, that do not contribute to the final result.

To get the outer order of chains, each chain's cost is computed. The general rule is to order chains based on ascending cost, i.e., $cost(c_i) \leq cost(c_{i+1})$. The cost of evaluating a query ECS Q_i with unbound nodes (e.g. $?x$, $?y$ for $Q_{x,y}$ in Fig. 4.5) is the cost of reading all triples of $eval(Q_i)$, or its cardinality, that is, $cost(Q_i) = \sum_{n=1}^{k} |T(E_n)|$, with $E_n \in matches(Q_i)$. If either or both of Q_i are bound, we estimate the cost of its evaluation as a constant 1. For consecutive ECSs in a chain $c_k = Q_1 \bowtie Q_2 \bowtie \ldots \bowtie Q_k$, we estimate the cost of the resulting series of joins with the following recursive formula:

$$cost(c_{Q_1 \ldots Q_k}) = cost(c_{Q_1 \ldots Q_{k-1}}) \times m_{f,os}(Q_k) \tag{4.9}$$

where $m_{f,os}(Q_k)$ is the multiplication factor of Q_k for an object-subject join. The cost of a chain consisting of one ECS is given as the cardinality of the ECS, which is the base case of the recursion. The multiplication factor m_f of an ECS $E_i = \{S_{1,i}, S_{2,i}\}$, where $S_{1,i}$ and $S_{2,i}$ are the subject CS and object CS of E_i respectively, is an estimation of how many rows will be generated by performing an object-subject join with E_i at the right side. We define it as the ratio of (distinct) objects per subject in E_i, i.e., $m_{f,os}(E_i) = |o_{E_i}| \,/\, |s_{E_i}|$, where $|o_{E_i}|$ and $|s_{E_i}|$ are the distinct subject nodes of $S_{1,i}$ and $S_{2,i}$ respectively. We can use m_f instead of assuming independence between consecutive ECSs, because it is guaranteed from Algorithm 3 that the consecutive ECSs will be joined on the same sets of CSs, and not on the whole body of triples. We can afford adopting this type of estimation, because the cardinalities of the ECSs are generally bound to values much lower than the total size of the dataset.

For queries with bound nodes in the CSs of Q_i, we retrieve the triples of $eval(Q_i)$ by first retrieving the respective CSs, and scanning the regions of the SPO table that refer to the matched CSs. This, however, may affect the cardinalities of the ECSs; thus, the cost

model adjusts the counts of distinct object and subject nodes to the numbers derived by the retrieved triples from the SPO table.

To get the inner ordering, we take into account the fact that all ECSs in a chain are linked with an object-subject join. This allows us to expand an existing node or sub-chain either left or right, one ECS at a time. Based on this, we employ a simple heuristic that starts from the ECS with the lowest cardinality, and expands the chain selecting the ECS with the minimum cardinality from the left or right.

While other approaches use Dynamic Programming algorithms for finding the optimal join order based on the employed statistics in order to reduce intermediate results, in axonDB a large amount of the filtering of triples is already performed at the pattern matching stage. Thus, the order does not heavily affect the performance of the query processor, an observation that is reflected in our experiments, where we tested the system with the planner both disabled and enabled, and we found that while there is indeed a speed-up factor of 2-3 when the planner is enabled, the improvement is less than an order of magnitude for all experiments.

4.5.4 Query execution

Each query chain pattern is executed individually, by looking up the ECS index and joining the triples of each ECS of the matched chains. Multiple chain patterns are joined in the final step of the execution using hash joins on their common attributes, the join tables of which are created dynamically during the evaluation of individual chains. Note that, execution of a chain pattern does not take into account the star pattern variables when joining consecutive ECSs. Retrieval of the attributes in the star pattern of the subject and/or object of an ECS is instead achieved when retrieving the ECS from disk, by performing a merge-join between the ECS's triples and the triples of the subject/object CS from the CS Index. In fact, a merge-join is possible because the CS Index maintains the interesting order of the subject node. However, this will not happen in the case where none of these variables are part of the query projection. In this case, as is the case for the queries of Figures 4.1 and 4.5, the restriction for the chain nodes to emit the bound properties is already enforced by the ECS definition.

For a query ECS Q_j, assuming that E_i is the most generic ECS that is matched to Q_j, this entails that *all supersets* of E_i, i.e., $E_i \succ E_{i'} \succ \cdots \succ E_{i''}$ will also belong to $matches(Q_j)$, thus the evaluation $eval(Q_j)$ must be the union of the triples in the hierarchically related ECSs that match the pattern, i.e., $T(E_i) \cup T(E_{i'}) \cup \cdots \cup T(E_{i''})$. Therefore, it is often expected to read the evaluations of ancestors or children of an ECS

Table 4.3: Size on disk (GB) and loading times (minutes)

	# triples	input	axonDB		RDF-3x		TripleBit		Virtuoso	
			size	time	size	time	size	time	size	time
LUBM2000	370m	54.2	8.12	68	16.54	58	10.88	45	14.6	45
Reactome	16m	2.8	0.71	3	1.07	2	0.74	2	0.91	2
Geonames	172m	18.8	8.24	81	12.48	34	8.6	20	8.56	27

in the same evaluation process. This is the main reasoning behind our approach to store the triples of hierarchically related ECSs in close locality (see Section III), and essentially extend the range scan of a match to all of its matching neighbours as well.

4.6 Evaluation

4.6.1 Experimental Setup

We have conducted an extensive experimental evaluation on *axonDB* with both synthetic and real-world data, and a comparative study with three high-performance RDF engines, namely *RDF-3x*, *Virtuoso opensource 7.2* and *TripleBit*. We have selected three native, high-performance, and centralized competitors that use different indexing approaches, in order to perform a system-wide comparison. For axonDB, we experiment with all four available optimization alternatives, i.e., a base configuration with the ECS hierarchy and query planner off (denoted with *axonDB*), two alternatives with one of them on (*axonDB-h* and *axonDB-qp*, respectively), and an optimized configuration with both features on (denoted with *axonDB+*), and assess the effect of these components to the performance of the system. All experiments were performed on a server with Intel i7 3820 3.6GHz, running Debian with kernel version 3.2.0 and allocated memory of 16GB. For Virtuoso, we used the recommended tuning parameters given by Openlink, for RDF-3x and TripleBit we used the default deployment, which is non-tunable.

The aim of the experiments is to assess the performance of axonDB in *data loading*, *query execution* and *scalability* with synthetic data of increasing sizes. For the query runtime experiments, we execute the queries 20 times and report the best time. Furthermore, the experiments have been performed with cold caches, each time dropping the cache with the use of the sync; echo 1 > /proc/sys/vm/drop_caches command in linux. Our metrics are: *query execution time*, *loading time* and *disk storage size*. For query execution time, we set an upper timeout limit at 30 minutes.

Implementation. We have implemented axonDB as an open-source project[4], using

[4]All code and queries are available in `http://github.com/mmeimaris/axonDB`

Java 1.8 and the mapDB[5] library, a high-performance key-value engine with drop-in replacements for sets, such as hash tables. axonDB uses mapDB for object serialization/deserialization and disk I/O on native Java objects. This is also the default way of serializing and deserializing ECS and CS objects, as well as all auxiliary indexes. All data structures except for the SPO and PSO tables, are stored as serialized Java objects using mapdBD. For triple serialization and persistence, axonDB uses byte arrays and random access files and writes all data in a single binary file, similar to RDF-3x and Virtuoso. The triples in the SPO/PSO tables are serialized as contiguous arrays of integers, and can be retrieved using range scans defined by the ECS/CS indexes. This format carries the benefit of being easily partitioned, while reducing disk reads to the number of matched ECSs per query. The ID-to-String/URI dictionary, which holds values for the compressed node and predicate ids, as well as the literals, is stored in the form of a clustered B^+-tree, with keys being sorted in ascending order. For this release, axonDB only supports conjunctive SPARQL queries with equi-joins.

Datasets and Queries. We employ one synthetic and two real-world datasets, widely used in the literature [NM11], [Wu+14], [Yua+13]. For synthetic data, we have used the *Lehigh University Benchmark* (LUBM) data generator to create RDF datasets of increasing sizes, from 15 (LUBM10) to 370 (LUBM2000) million triples. LUBM uses an academic ontology of universities, with entities for departments, courses, members of faculty and so on. Since axonDB does not support inferencing, we extended the LUBM generator to add all superclasses of an instance's class, in order to generate the transitive closure of the subclass relationships, as well as the *memberOf* and *hasAlumnus* properties. For our real-world experiments, we have chosen the *Reactome*[6] dataset, which contains information about biological pathways, and is rich in long paths with branching components, and *Geonames*[7], an ontology of geographical features that contains a diverse schema of varying properties (i.e., large number of CS/ECS) among the same types of entities, as shown in Table 2.

Regarding the queries, we create two sets of queries for LUBM, one set for Reactome, and one set for Geonames. LUBM defines 14 queries; most of them are simple, pertaining to a range of 1 to 6 triple patterns. From these, we select 6 representative queries, namely 2, 4, 7, 8, 9, and 12, which are the most challenging and have the largest numbers of triple patterns, in order to assess the performance of the system on traditional settings. To assess the performance on more complex queries, we create a second set of queries, by modifying 7 of the original queries (2, 3, 4, 8, 10, 11 and 12), converting all bound

[5]www.mapdb.org
[6]http://www.ebi.ac.uk/rdf/services/reactome
[7]http://www.geonames.org/ontology/documentation.html

Table 4.4: Comparison of different optimization settings for representative queries.

	LUBM					Reactome					Geonames				
	Q1	Q5	Q8	Q12	GM	Q2	Q3	Q7	Q8	GM	Q1	Q2	Q4	Q6	GM
axonDB	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
axonDB-h	0.97	**0.79**	0.83	0.82	0.79	0.76	0.56	0.73	0.99	0.82	0.73	0.81	0.75	**0.70**	0.74
axonDB-qp	0.77	1.01	0.86	0.98	0.83	0.85	0.57	0.53	**0.61**	0.73	0.71	0.56	0.65	1.05	0.72
axonDB+	**0.69**	0.87	**0.66**	**0.82**	**0.73**	**0.68**	**0.38**	**0.49**	**0.61**	**0.62**	**0.69**	**0.49**	**0.51**	**0.70**	**0.64**

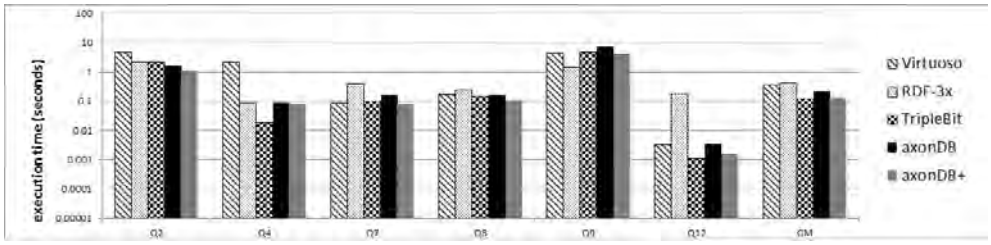
nodes to variables, and extending their characteristic sets, and we also define 5 additional ones. The queries are ordered by complexity[8], and Q1-8 are highly selective, while Q9-12 are low in selectivity. For the Reactome and Geonames datasets, we construct 8 and 6 queries, respectively, with increasing selectivity and numbers of chain patterns, i.e., 1-3 chains and 3-6 query ECSs. These take advantage of the long paths in the two datasets, and have progressively larger result sizes. The queries can be seen in the Appendix of this thesis. In what follows, we present the results.

4.6.2 Experimental Results

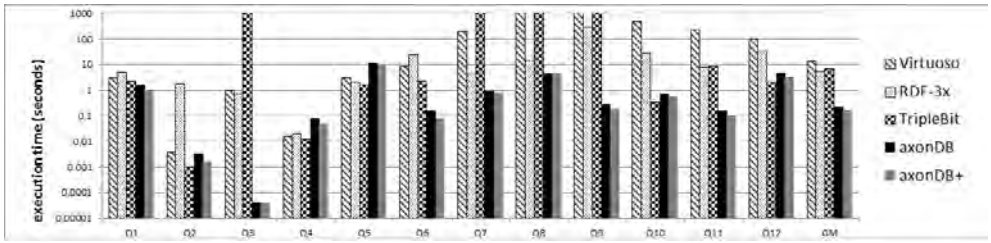
Loading. The size in GB and loading time in minutes for the two real datasets and LUBM2000 can be seen in Table 4.3. Overall, axonDB exhibits the lowest space overhead for the input data, along with TripleBit which comes second. This is a derivative of the low degree of data replication imposed by ECS indexing, and the fact that it only uses two triples tables (SPO and PSO). However, axonDB suffers from longer loading times compared to all three competitors, because of the added complexity of retrieving the inherent schema of nodes (i.e., CS index), and triples (i.e., ECS index). Especially for Geonames, the loading time is significantly longer, because of the large number of ECSs.

Comparison of different optimizations. Table 4.4 compares the four versions of axonDB (based on the employed optimizations). We experiment with all queries from the modified LUBM and the two real-world datasets and we report the GM of all queries as well as the performance for four representative queries that exhibit the highest complexity in each dataset. The numbers denote the ratio of runtime of each configuration w.r.t. the runtime of the base implementation, which is shown at the first line. Overall, the relative performance improvement with all optimizations on is most cases better than its respective counterparts. The effects of the planner are minimized when the queries have only one chain, as the outer ordering and thus the cost model are redundant in such cases (e.g., LUBM Q5, Q12, Geonames Q6). The hierarchy optimization affects most queries, as multiple related ECSs often differ by a small number of properties in the data.

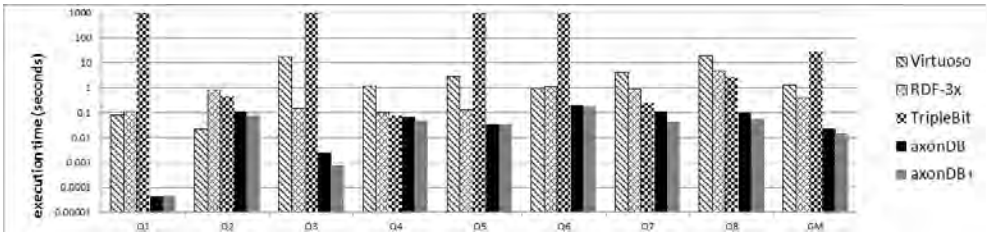
[8]Calculated as the product of (#triple patterns)×(#chains)



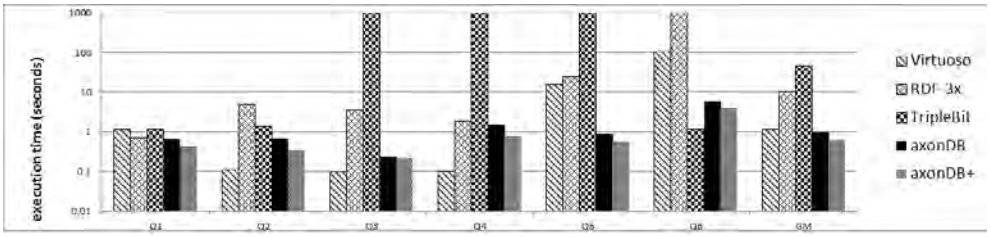
(a) Query runtimes for the LUBM dataset (original)



(b) Query runtimes for the LUBM dataset (modified)



(c) Query runtimes for the Reactome dataset



(d) Query runtimes for the Geonames dataset

Figure 4.6: Query runtimes in seconds

Next, we only consider the worst and optimal configurations to compare them with the competitors.

Query performance - LUBM. All systems can address quite efficiently the original LUBM queries . This can be seen from the geometric means of the queries in Fig. 4.6(a), where we experiment with queries 2, 4, 7, 8, 9 and 12, which are the most challenging, and have more than one triple patterns. Overall, the performance of all systems lies in the same order of magnitude. Even though axonDB is designed to address more complex queries, this experiment shows that it can handle simple patterns efficiently as well.

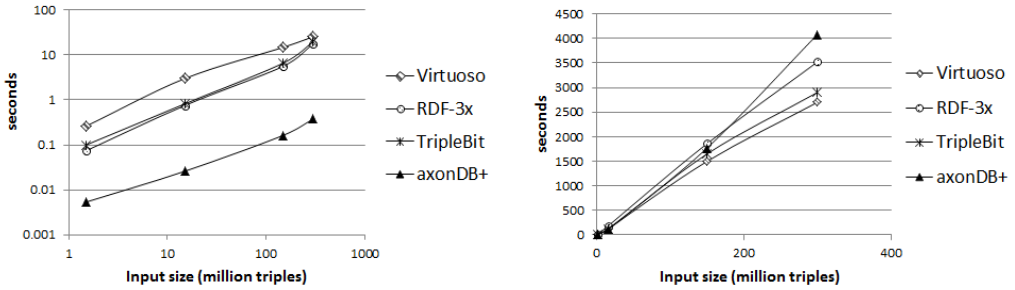
The runtimes for *axonDB* and *axonDB+* against the three competitors for the modified

LUBM queries can be seen in Fig. 4.6(b) along with their geometric mean (GM). The actual GM for Virtuoso and TripleBit is equal to, or greater than the maximum depicted in the figure, as we do not show running times above 30 minutes, or timed-out queries. As shown, both axonDB configurations outperform the rest, with their geometric mean improving the competition by at least[9] 1 order of magnitude. Especially in the case of queries with complex patterns (Q7-12), axonDB is better by several orders of magnitude, while Virtuoso and TripleBit suffer several 30-minute timeouts. For Q3, which does not yield any results, the preprocessor cannot match the query graph to any ECS chains in the data, and thus does not perform any joins, giving axonDB an advantage of up to 4 orders of magnitude compared to RDF-3x and Virtuoso (TripleBit timed out). In the more selective queries Q4 and Q5, axonDB is outmatched by the rest of the systems, because it does not have permuted indexes that quickly filter out triples that do not contribute to the solution, and has to suffer a full scan of the matched ECSs instead. However, in Q7, Q8 and Q9, both TripleBit and Virtuoso exhibited times over 30 minutes. These queries have long chains with up to 14 triple patterns with all nodes unbound except the predicates. Thus, the optimizers of these systems spend a lot of time dealing with large intermediate results created by the abundance of variables.

Reactome. The results are shown in Fig. 4.6(c). Again, both axonDB and axonDB+ outperform the rest for all queries. Even though the dataset is relatively small, the complexity of the data can lead to queries with non-trivial patterns. This is evident by the relatively large number of ECSs (346). For the queries with the lowest selectivity (Q6, Q7 and Q8), axonDB improves the competition by at least one order of magnitude, while TripleBit fails to answer four queries within 30 minutes. As in LUBM, these queries (Q1, Q3, Q5, Q6) exhibit a large amount of unbound variables, in long chain patterns. This provides an intuitive insight that the nature of ECS indexing facilitates the evaluation of complex query patterns by isolating smaller subsets of the data that contribute to the result, and thus decreasing the intermediate results that would be present in traditional indexing paradigms. Instead, an ECS graph matches a query to smaller and more relevant subsets of the data, and reduces the number of self-joins and the cardinality of intermediate results.

Geonames. The results for Geonames are shown in Fig. 4.6(d). While axonDB and axonDB+ configurations outperform in all queries but Q4 and Q6, the improvements against RDF-3x and Virtuoso are not at the same scale with the previous datasets. Geonames has over 10,000 different ECSs, thus invoking costly disk reads even for ECSs with small cardinalities. In fact, this reflects a drawback in the ECS indexing approach,

[9]In reality, it is more for the timeouts.



(a) (b)

Figure 4.7: Query execution (a) and dataset loading (b) for increasing sizes of LUBM

where partitioning of the triples by their associated ECS can become a bottleneck when the partitioning is volatile with respect to the triple cardinality of each ECS. In any case, axonDB improves the competition by one order of magnitude overall, based on the observed GM. While TripleBit performs very fast on Q1, Q2 and Q6, it fails to answer three queries under the 30-minute timeframe, because its vertical partitioning storage scheme suffers from large intermediate joins in queries with long chains. This is an indication that such approaches that use inherent schema retrieval, are not suited for highly versatile RDF datasets.

Scalability. We have experimented with increasing input sizes of LUBM, starting from 15M triples, up to 370M triples. In Fig. 4.7, we report the GM of Q1-Q12 (a), and the loading time (b) for all four systems, in log-log scales. The query performance of axonDB+ scales *linearly* and retains its relative difference by 1-3 orders of magnitude with the rest of the systems for all input sizes. Loading also appears to scale linearly with respect to input size, however, due to the ECS extraction of the loading phase, axonDB+ is outperformed by Virtuoso and RDF-3x as the input size increases. In any case, our experiments indicate that the methods presented herein are indeed scalable for larger input sizes.

4.7 Conclusions and Future Work

In this chapter, we have presented *axonDB*, a native RDF engine that employs ECS indexing, and discussed its implications on SPARQL processing. To this end *Extended Characteristic Sets*, and *ECS graphs* were introduced, along with methods and algorithms for ECS retrieval and querying. These have been implemented with two optimizations, that take into account query planning and hierarchical relationships between ECSs. Finally, we performed an extensive experimental evaluation against three high-performance

RDF stores. The experimental evaluation has shown that axonDB outperforms the state of the art approaches, especially for answering complex query patterns with low selectivity. As future work, we will address data updates in existing ECS indexes, and study the application of the approach in a distributed setting.

Chapter 5

Optimizing SPARQL Query Planning

5.1 Introduction

5.1.1 Motivation

SPARQL is a W3C recommendation for querying graph data expressed in the Resource Description Framework (RDF). SPARQL query engines are implemented on either native RDF stores, or physical implementations of other paradigms, such as relational databases. In either case, query optimization involves a layer of logical optimization, where the query plan is constructed. Logical optimization of SPARQL queries heavily depends on the ability of a query optimizer to provide good plans. This involves reordering of a query's triple patterns, in order to minimize intermediate results.

Optimizers use statistics and heuristics in order to search the vast space of potential query plans and reach a solution without trading off much pre-processing time for actual query execution time. They rely on first-level statistics, such as the number of distinct triples with a particular property, and assumptions on the values of deeper level statistics (e.g. join cardinalities). In practice, this is not always efficient, because the bias imposed by these techniques gives rise to optimizers that perform very well for particular query patterns, but poorly on others. As an example, consider the following query, where the triple patterns are labelled on the left as $t_1 \dots t_5$:

Listing 5.1: Example query for the LUBM[GPH05] dataset.

```
  SELECT ?X ?Y
  WHERE {
t1: ?X rdf:type ub:Student .
t2: ?Y rdf:type ub:Department .
t3: ?X ub:memberOf ?Y .
```

```
t4: ?Y ub:subOrganizationOf <http://www.University0.edu> .
t5: ?X ub:emailAddress ?Z}
```

Assume that the query processor evaluates the query in the order t_1, t_5, t_3, t_2, t_4, relying on statistics that the total number of *Students* is 99k. It will first attempt to match ?X to all 99k *Students* and their *email addresses*, and then retrieve all *ub:memberOf* associations for these. Most of the resulting triples will be discarded in the evaluation of t_2 and t_4, because we are only interested in *Students* that are members of a *Department* that is a *subOrganization* of *University0*. On the other hand, if the order is t_2, t_4, t_3, t_1, t_5, then the triple pattern t_3 will not be evaluated on all *Students*, but only on the members of the matched results for *?Y*. Hence, the evaluation of t_1 and t_5 will be performed on a much smaller set of intermediate results, thus optimizing query answering. This kind of optimization can yield results that are orders of magnitude faster than in the case where no such optimization takes place [NM11; GN14].

In this chapter of the thesis, we introduce and discuss a novel statistical approach for reordering triple patterns in SPARQL queries, which searches a quadratic plan space and builds a plan on a bottom-up manner by taking into account relative differences between triple patterns. This method generates binary left-deep trees, and our evaluation is centered around this sub-problem. Preliminary experiments show that it outperforms traditional statistics- and heuristics- based approaches in query planning.

5.1.2 Related Work

There exists a large body of literature on SPARQL query optimization, where many approaches operate under the data independence assumption, which states that data are uncorrelated, and cardinalities of joined patterns are computed using heuristic, cross-product approaches. Stocker et al [Sto+08] propose several statistical and heuristic-based planning algorithms that involve cardinality estimation both with and without joins. Neumann and Weikum [NW10] in the RDF-3X store use a Dynamic Programming algorithm and cardinality histograms. However, the optimizer sometimes spends more time than the actual query execution, some times orders of magnitude slower [GN14]. TripleBit [Yua+13] uses a Dynamic Query Plan Generation Algorithm (DQPGA) for queries with multiple consecutive joins, which can be costly for pattern-rich queries. Gubichev and Neumann provide better estimates by extending Characteristic Sets [GN14]. Kalayci et al [KKB15] use ant-colonization algorithms for dynamic optimization. The latest version of Virtuoso uses a greedy query optimization approach that is further assisted by dynamic sampling and a combination of full and partial indexes [BEP14]. Finally, Tsialiamanis et

al[Tsi+12] rely solely on triple pattern heuristics to reach good plans without the need for statistics and indexing.

5.2 Distance-Based Reordering

5.2.1 Query Representation

Let T be the set of all triple patterns in an incoming query q. Each triple pattern $t_i \in T$ consists of three nodes, either bound (URIs, blank node IDs or literals) or unbound (variable), i.e., the subject, predicate and object of t_i. Furthermore, let M represent the set of all unique nodes in the subject and object positions in T, both bound and unbound. We map each triple pattern t_i from T to a $|M|$-dimensional vector space, where each vector attribute is an element from M, i.e., a subject or object node from the query. In this preliminary approach, we assume a solution space of left-deep trees, where the original problem of finding the best order of triple evaluation is factorial. This is because for a query q consisting of $|T|$ triple patterns, there are $|T|!$ different permutations of triples. The factorial problem, commonly solved by dynamic programming algorithms, guarantees that the solution will be optimal with respect to the chosen cost model. Unfortunately, recent studies have suggested that even industrial-level query optimizers do not yield satisfactory results when they are heavily dependent on the cost model [Lei+15], especially when cost model errors propagate through multiple joins.

In our approach, we map triple patterns from a query into a multidimensional space, and decide the join order based on the spatial correlations of the patterns, i.e., by comparing the pair-wise distances of the vectors that represent the triple patterns in the multiple dimensions, and ranking the patterns based on this comparison. The multidimensional space is built based on cardinality statistics that are held in a separate index. These pre-computed statistics are easily calculated in RDF stores, and include the cardinality of triples with a bound subject, property, and object, as well as the size of the whole dataset. Specifically, we assume that each triple pattern becomes a m-dimensional vector, and the value of an attribute m_i for a given triple pattern $t = (s, p, o)$, is given by the following function:

$$f(t, m_i) = \begin{cases} card(t), & \text{if } m_i \in (s, o) \\ 0, & \text{otherwise} \end{cases}$$

Where $card(t)$ is the cardinality of the triple pattern. This is calculated using the following rules:

- $card(t) = card(p)$, if p is bound and s,o are unbound,

- $card(t) = max(1, \frac{card(p)}{|S|})$, if p and o are bound, and s is unbound,
- $card(t) = 1$, if p and s are bound, and o is either bound or unbound,
- $card(t) = |D|$, for all other cases

The cardinality for a bound property $card(p)$ is simply given as the number of triples with p as the predicate. $|D|$ and $|S|$ are the number of triples in the dataset D, and the number of distinct subject nodes S respectively. Notice that we do not represent predicate attributes, which means that the query space will be built entirely based on the subject and object nodes of all triple patterns in the query. Instead, the information that comes from the predicates in the triple patterns becomes quantized in the values of the attributes, as is given by the aforementioned cardinality estimation rules. Especially for the *rdf:type* property, we assume the existence of an aggregate index that holds the exact cardinalities of particular class types. Therefore, in this special case, the second rule is altered to reflect the exact enumeration of subjects with a specific type. For instance, in our running example, the cardinality of t_1 is 99k, as there exist 99k subjects with *rdf:type ub:Student*.

Function $f(t, m)$ can be used to calculate the values of a $|T| \times |M|$ matrix, denoted as Q_m, where each row represents a triple pattern in T, and each column represents an attribute in M, or simply a node in the query pattern. By applying a distance function and processing the pair-wise distances between rows in Q_m, the search space becomes quadratic with respect to $|T|$, and the comparisons will be of $O(|T|^2)$ complexity, rather than the original factorial one. Moreover, as we do not evaluate self-distances and permuations of the same pair (e.g., $[t_1, t_2]$ and $[t_2, t_1]$, the actual number of comparisons will be $\binom{|T|}{2}$, or $|T| \times (|T| - 1)/2$. Even though this reduction in cost comes with the loss of guarantee of optimality, our experiments show that this approach tends to work well for all types of queries, even with large numbers of triple patterns.

The matrix Q_m for the query of Listing 5.1, is shown in Table 1 for the synthetic dataset LUBM10 with 1.5m triples. To construct this for dataset D, we apply the estimation rules based on the pre-computed statistics. For instance, there are 99k triples with *rdf:type* as a property and *ub:Student* as an object, which can be seen in columns 1 and 2 for t_1. Similarly, there are 106k triples with *ub:emailAddress* as a predicate, which is encoded in t_5. Application of a distance function will generate a $|5| \times |5|$ distance matrix of pair-wise distances between the triple patterns $t_1 - t_5$.

5.2.2 Plan Generation

When Q_m is constructed and the distance matrix is calculated, we build sub-plans for the query, based on the ascending ranking of the pair-wise distances of the triple patterns in T. The algorithm to build sub-plans can be seen in Algorithm 5. Specifically, we sort pair-wise distances in ascending order and create an empty queue for sub-plans (Lines 1-3). Then, we iterate through each pair of sorted triple patterns t_a, t_b (Line 4). If there exists a sub-plan p_i that contains only t_a or only t_b, we append the non-contained triple pattern to p_i (Lines 5-10). If none of t_a, t_b are contained in a sub-plan, we create a new sub-plan and add t_a, t_b, then push the new sub-plan to the existing queue. The order in which we add these two depends on the cost of each triple pattern (Lines 11-15). In case both patterns t_a and t_b exist in sub-plans, we continue to the next pair.

After we have created a series of (ordered) sub-plans, we iterate through consecutive sub-plans and attempt to reorder their patterns in order to better capture join relationships that occur between these. More accurately, if there are more than one sub-plans found, then we try to rearrange the triples that share common variables between adjacent sub-plans, so that the joined triple patterns between the two sub-plans are closely located. This can be seen in Algorithm 6.

Specifically, if there is only one sub-plan, then this is returned as the final plan (Lines 1-3). Else, the algorithm iterates through the *subPlans* queue with two pointers (Line 4), and prioritizes each pair of sub-plans (Line 5). The *prioritizePair* function checks for joins between two consecutive sub-plans (Line 12), i.e. it checks whether the last triple (tail) in p_i is joined with the first triple (head) in p_{i+1}. If they are joined, it returns the pair as is (Lines 13-15). If not, it finds the first triple in p_{i+1} that can be joined with the tail of p_i (if one exists), pushes it at the tail of p_i and recursively checks the same pair (Lines 17-21). Finally, the *prioritizePair* function returns an array of two plans, the first of which is inserted in the final plan, while the second is used as the first sub-plan of the next iteration, as it holds the updated ordering. This means that the triple patterns that are ranked lower in p_{i+1} will be bubbled-up and re-ranked in order to reflect the join

relationship between p_i and p_{i+1}.

Algorithm 5: *generateSubPlans*

Data: *map*: A matrix of triple patterns from the query as rows, and query nodes as columns

Result: *subPlans*: An ordered list of sub-sets of query triple patterns, that constitute subplans.

```
1  subPlans ← newQueue();
2  distances ← distanceMatrix(map);
3  sort(distances);
4  for each pair t_i, t_j ∈ distances | t_i ≠ t_j do
5      if ∃p_i ∈ subPlans | t_a ∈ p_i AND t_b ∉ p_i then
6          append t_b to p_i;
7          continue;
8      else if ∃p_i ∈ subPlans | t_a ∉ p_i AND t_b ∈ p_i then
9          append t_a to p_i;
10         continue;
11     else if ∄p_i ∈ subPlans | t_a ∈ p_i OR t_b ∈ p_i then
12         p ← newList();
13         append minCost(t_a, t_b) to p;
14         append maxCost(t_a, t_b) to p;
15         subPlans.push(p);
16     else
17         continue;
18 end
19 return subPlans;
```

5.2.3 Experiments

We implemented a proof-of-concept version of the query planner in Jena ARQ, and conducted experiments on Jena TDB[1], comparing against several statistical reordering approaches, namely Stocker et al's PFJ and ONS [Sto+08], Kalayci et al's Ant System [KKB15] as well as Jena TDB's Fixed (JF) and Weighted (JW) optimizers, and the reordering performed by the open source edition of Openlink Virtuoso 7.2. As triple pattern ordering is a problem that is orthogonal to the low-level implementation specifics and design choices (e.g., join implementations, indexing), we use Jena TDB as a common testbed for all approaches to strictly assess and compare the effect of triple pattern orderings on query processing. We used LUBM[GPH05] to generate a synthetic dataset of 15m triples, and measured execution times for 29 queries on the LUBM dataset. The queries consist of 14 original queries provided by LUBM, and an additional 15 queries of

[1]https://jena.apache.org/documentation/tdb/

Algorithm 6: *reorderSubPlans*

Data: $subPlans(p_0, \ldots, p_{n-1})$: A queue of sub-plans
Result: $finalPlan$: An ordered list of triple patterns, to be executed by the query engine.

```
1  if subPlans.size == 1 then
2  |   finalPlan.push(p_0);
3  |   return finalPlan;
4  for each p_i, p_{i+1} ∈ subPlans do
5  |   nextPair ← prioritizePair(p_i, p_{i+1});
6  |   finalPlan.push(nextPair[0]);
7  |   p_{i+1} ← nextPair[1];
8  |   if p_{i+1}.isEmpty then
9  |   |   subPlans.remove(p_{i+1});
10 end
11 return finalPlan;
12 function prioritizePair(p_i, p_{i+1})
13 |   newPair ← array[2];
14 |   if p_i.tail is joined with p_{i+1}.head then
15 |   |   newPair[0]← p_i;
16 |   |   newPair[1]← p_{i+1};
17 |   |   return newPair ;
18 |   else
19 |   |   for each t_k, ∈ p_{i+1} do
20 |   |   |   if p_i.tail is joined with t_k then
21 |   |   |   |   p_i.push(t_k);
22 |   |   |   |   p_{i+1}.remove(t_k);
23 |   |   |   |   newPair ←prioritizePair(p_i, p_{i+1});
24 |   |   end
25 |   |   newPair[0]← p_i;
26 |   |   newPair[1]← p_{i+1};
27 |   |   return newPair;
```

increasingly complex shapes and sizes, used in [KKB15]. All code and queries are available online[2]. For each approach, we extract an ordering, and feed that directly to the Jena query processor. Table 2 summarizes the percentages of best plans for all queries. These are computed by measuring, for each approach, how many plans were the fastest[3] in relation to the other approaches, for all queries. Because of the fact that the fastest plan can be reached by more than one method, we differentiate between plans based on the generated orderings, and then we compare their execution times. As can be seen, our method outperforms all other methods for queries of different patterns, namely the original 14 queries, and the 15 additional queries of star, chain, chain-star, and cyclic patterns, achieving the best plan 90% of the time, with Virtuoso coming second with 66% of its orderings being the best.

In order to assess how our planner performs with respect to different query types, we use 15 extended queries originally constructed in [KKB15]. These represent different query types, and are thus classified into star, chain, cyclic, and chain-star query types, of varying triple pattern sizes. Their original intention was to provide more complex query structures of up to 14 triple patterns for LUBM dataset, because the original queries are limited to 1-6 triple patterns. We measure execution times for each query type separately, and report these in Tables 3, 4, 5, and 6, for star, chain, cyclic, and chain-star queries respectively. For each query, the best ordering is marked with a bold execution time. While we do not show the actual triple orders that were derived by each method, multiple bold values in the same row indicate that the same order has been achieved by more than one approach. Note that small differences in execution times for the same orderings can be attributed to lags imposed by I/O, available CPU resources and other external factors.

For computing the distance matrix from Q_m, we used a simple Euclidean distance function that measuring distances between triple patterns in the M-dimensional space. Overall, the results are encouraging for further pursuing this direction. In fact, in 90% of the queries, our method generated the fastest plan in relation to the other approaches, of whom the best (VIRT) achieved a rate of 66%, and the rest achieved less than 60%.

5.3 Conclusions and Future Work

Our preliminary results show potential value in a full implementation of a logical query optimizer. As future work, we intend to design a more mature algorithm for the plan gen-

[2]https://github.com/mmeimaris/sparqlDistancePlanner

[3]We measure execution times by executing each query 10 times and reporting the mean.

Table 5.1: Q_m matrix for reference query.

	?X	Student	?Y	Dpt	Univ0	?Z
t_1	99k	99k	0	0	0	0
t_2	0	0	189	189	0	0
t_3	106k	0	106k	0	0	0
t_4	0	0	239	0	239	0
t_5	106k	0	0	0	0	106k

Table 5.2: Percentage of plans that are best compared to other methods for all queries.

JW	JF	ONS	PFJ	ANT	VIRT	OUR
59%	48%	28%	59%	31%	66%	90%

eration, which takes into account a mix of dataset statistics, historical queries and graph summarization in order to provide better estimates that lead to more efficient plans. Taking into account join cardinality estimates will also be implemented. Moreover, we plan on experimenting with different distance functions in order to capture distances between triple patterns more accurately. Finally, we intent to perform exhaustive comparisons with large datasets and queries, as well as other state of the art methods, in order to assess the scalability of the approach, as well as its implementation in parallel settings. Towards this end, we will explore how our method can generate flat bushy n-ary join trees (instead of binary left-deep trees) that tend to be more easily parallelized.

Table 5.3: Query execution times (seconds) for the star queries.

	JW	JF	ONS	PFJ	ANT	VIRT	OUR
Q1	1.01	1.14	1.07	**0.61**	1.05	0.63	**0.57**
Q2	0.91	0.92	4.71	0.76	0.79	0.74	**0.67**
Q3	0.23	0.27	0.23	0.15	3.62	0.21	**0.13**
Q4	0.17	0.08	0.07	0.07	0.42	**0.05**	**0.03**

Table 5.4: Query execution times (seconds) for the chain queries.

	JW	JF	ONS	PFJ	ANT	VIRT	OUR
Q1	0.30	0.33	0.37	**0.14**	1.09	0.39	**0.14**
Q2	**0.49**	**0.56**	**0.52**	0.77	**0.50**	**0.48**	**0.48**
Q3	322	323	131	292	139	**55**	**53**
Q4	**54**	**53**	**53**	163	209	182	**49**

Table 5.5: Query execution times (seconds) for the cyclic queries.

	JW	JF	ONS	PFJ	ANT	VIRT	OUR
Q1	0.05	0.07	0.05	**0.04**	**0.04**	**0.04**	**0.04**
Q2	0.25	0.24	0.19	**0.02**	0.20	0.18	**0.02**
Q3	**0.06**	**0.06**	0.31	2.37	0.74	**0.04**	4.50
Q4	1.06	0.94	1.30	0.64	0.81	**0.04**	0.52

Table 5.6: Query execution times (seconds) for the chain-star queries.

	JW	JF	ONS	PFJ	ANT	VIRT	OUR
Q1	**4.16**	**4.16**	**4.15**	7.39	**4.15**	5.15	**4.15**
Q2	5.95	5.92	5.93	5.83	6.67	**0.49**	**0.42**
Q3	6.11	4.98	4.75	214.92	6.37	**2.35**	**2.31**

Chapter 6

Hierarchical Characteristic Set Merging

6.1 Introduction

Recent works in the state of the art in RDF data management have shown that extraction and exploitation of the implicit schema of the data can be beneficial in both storage and SPARQL query performance [Pha+15][PB16][Mei+17][MSMH17]. This was also apparent in the previous two Chapters, where ECS indexing came short of addressing complicated schema structures, as in the case of Geonames. In order to organize on disk, index and query triples efficiently, these trends heavily rely on two structural components of an RDF dataset, namely (i) the notion of *characteristic sets* (CS), i.e., different property sets that characterize subject nodes, and (ii) the join links between CSs. For the latter, in the previous chapter, we introduced *Extended Characteristic Sets* (ECS)[Mei+17], which are typed links between CSs that exist only when there are object-subject joins between their triples, and we showed how RDF data management can rely extensively on CSs and ECSs for both storage and indexing, yielding significant performance benefits in heavy SPARQL workloads. However, this approach failed to address schema heterogeneity in loosely-structured datasets, as this implied a large number of CSs and ECSs (e.g., Geonames contains 851 CSs and 12136 CS links), and thus, skewed data distributions that impose large overheads in the extraction, storage and disk-based retrieval[Pha+15][Mei+17].

In this chapter, we exploit the hierarchical relationships between CSs, as captured by subsumption of their respective property sets, in order to merge related CSs. We follow a relational implementation approach by storing all triples corresponding to a set of merged CSs into a separate relational table and by executing queries through a SPARQL to SQL

transformation. Although, alternative storage technologies can be considered (key-value, graph stores,etc), we have selected well-established technologies and database systems for the implementation of our approach, in order to take advantage of existing data indexing and query processing techniques that have been proven to scale efficiently in large and complex datasets. To this end, we present a novel system, named *raxonDB*, that exploits these hierarchies in order to merge together hierarchically related CSs and decrease the number of CSs and the links between them, resulting in a more compact schema with better data distribution. The resulting system, built on top of PostgreSQL, provides significant performance improvements in both storage and query performance of RDF data.

In short, our contributions are as follows:

- We introduce a novel CS merging algorithm that takes advantage of CS hierarchies,
- we implement *raxonDB*, an RDF engine built on top of a relational backbone that takes advantage of this merging for both storing and query processing,
- we perform an experimental evaluation that indicates significant performance improvements for various parameter configurations.

6.2 Related Work

RDF data management systems generally follow three storage schemes, namely *triples tables*, *property tables*, and *vertical partitioning*. A triples table has three columns, representing the subject, predicate and object (SPO) of an RDF triple. This technique replicates data in different orderings in order to facilitate sort-merge joins. RDF-3X [NW10] and Hexastore [WKB08] build tables on all six permutations of SPO. Built on a relational backbone, Virtuoso [EM10] uses a 4-column table for quads, and a combination of full and partial indexes. These methods work well for queries with small numbers of joins, however, they degrade with increasing sizes, unbound variables and joins.

Property Tables places data in tables with columns corresponding to properties of the dataset, where each table identifies a specific resource type. Each row identifies a subject node and holds the value of each property. This technique has been implemented experimentally in Jena [Wil06] and DB2RDF [Bor+13], and shows promising results when resource types and their properties are well-defined. However, this causes extra space overhead for null values in cases of sparse properties [Aba+07]. Also, it raises performance issues when handling complex queries with many joins, as the amounts of intermediate results increase [JK05].

Vertical partitioning segments data in two-column tables. Each table corresponds to a property, and each row to a subject node [Aba+07]. This provides great performance for queries with bound objects, but suffers when the table sizes have large variations in size [Sid+08]. TripleBit [Yua+13] broadly falls under vertical partitioning. In TripleBit, the data is vertically partitioned in chunks per predicate. While this reduces replication, it suffers from the same problems as property tables. It does not consider the inherent schema of the triples in order to speed up the evaluation of complex query patterns.

In distributed settings, a growing body of literature exists, with systems such as Sempala [Sch+14], H2RDF [Pap+14] and S2RDF [Sch+16a]. However, these are based on parallelization of centralized indexing and query evaluation schemes.

For these reasons, latest state of the art approaches rely on implicit schema detection in order to derive a hidden schema from RDF data and index/store triples based on this schema. Furthremore, due to the tabular structure that tends to implicitly underly RDF data, recent works have been implemented in relational backbones. In our previous work [Mei+17], we defined *Extended Characteristic Sets (ECSs)* as typed links betwen CSs, and we showed how ECSs can be used to index triples and greatly improve query performance. In [Pha+15], the authors identify and merge CSs, similar to our approach, into what they call an *emergent schema*. However, their main focus is to extract a human-readable schema with appropriate relation labelling. They do not use hierarchical information of CSs, rather they use semantics to drive the merging process. In [PB16] it is shown how this *emergent schema* approach can assist query performance, however, the approach is limited by the constraints of human-readable schema discovery. In our work, query performance, indexing and storage optimization are the main aims of the merging process, and thus we are not concerned about providing human-readable schema information or any form of schema exploration. In [MSMH17], the authors use CSs and ECSs in order to assist cost estimation for federated queries, while in [GN14], the authors use CSs in order to provide better triple reordering plans. To the best of our knowledge, this is the first work to exploit hierarchical CS relations in order to merge CSs and improve query performance.

6.3 Hierarchical CS Merging

6.3.1 Preliminaries

The RDF model does not generally enforce structural rules in the representation of triples; within the same dataset there can be largely diverse sets of predicates emitting from nodes

of the same semantic type [Mei+17; Pha+15; NM11]. *Characteristic Sets* (CS)[NM11] capture this diversity by representing implied node types based on the set of properties they emit. Formally, given a collection of triples D, and a node s, the characteristic set $cs(s)$ of s is $cs(s) = \{p \mid \exists o : (s, p, o) \in D\}$.

The set of properties of a CS cs_i is denoted with P_i. Furthermore, in a given dataset, each CS represents a set of records identified by a subject node, and all of the values of the subject node (i.e., objects) for the predicates in P_i. We denote the set of all records of cs_i as r_i, while cs_i is represented by a relational table c_i that is defined by these two elements, i.e., $c_i = (P_i, r_i)$. The tuples in c_i are of the form $(s, p_{i,1}, \ldots, p_{i,k})$, where s is the identifier column (e.g. URI) of a subject node and $p_{i,1}, p_{i,2}, \ldots, p_{i,k}$ are the values, i.e. object nodes, of the properties in P_i for s. In the context of this chapter, with the term *Characteristic Set* we will refer collectively to the properties and records of a CS, i.e., its relational table, rather than just the set of properties proposed in the original definition, for the sake of simplicity.

Within a given dataset, CSs often exhibit hierarchical relationships, as a result of the overlaps in their comprising sets of properties. For example, consider two CSs, c_1, c_2, describing human beings, with $P_1 = \{type, name\}$ and $P_2 = \{type, name, marriedTo\}$. It can be seen that $P_1 \subset P_2$ and thus c_1 is a parent of c_2. This relationship entails an overlap of properties that define the CSs, and can be exploited in order to provide a means to merge common CSs based on the specialization or generalization of the node types they describe. In what follows, we formally define the notions of CS *subsumption*, *hierarchy* and *ancestral sub-graphs*.

Definition 1. ***(CS Subsumption)***. Given two CSs, c_i and c_j, and their property sets P_i and P_j, then c_i subsumes c_j, or $c_i \succ c_j$, when the property set of c_i is a proper subset of the property set of c_j, or $P_i \subset P_j$. This subsumption forms parent-child relationships between CSs. CS subsumption relationships can be seen in Figure 6.1(a) as directed edges between nodes. The set of all parent-child relationships defines a CS hierarchy as defined in the following.

Definition 2. ***(CS Hierarchy and Inferred Hierarchy)***. CS subsumption creates a partial ordering that essentially defines a *hierarchy* such that when $c_i \succ c_j$, then c_i is a parent of c_j. Formally, a CS hierarchy is a graph lattice $L = (V, E)$ where $V \in C$ and $E \in (V \times V)$. A directed edge between two CS nodes c_1, c_2 exists in L, when $c_1 \succ c_2$ and there exists no other c_i such that $c_1 \succ c_i \succ c_2$. An example CS hierarchy can be seen in Figure 6.1(a). Given a hierarchy L, we denote the *hierarchical closure* of L with L_c, so that L_c extends L to contain inferred edges between hierarchically related nodes that are

not consecutive, e.g. a node and its grandchildren. An example inferred hierarchy can be seen in Figure 6.1(c) for a sub-graph of the graph in Figure 6.1(a), with the inferred relationships in dashed lines. In the remainder of this chapter, we refer to L_c as the *inferred hierarchy* of L.

Definition 3. ***(CS Ancestral Sub-graphs)***. Given an inferred hierarchy $L_c = (V, E)$, a CS c_{base} and set of CSs $c_1, \ldots, c_k$, then $a = (V', E')$ is an ancestral sub-graph with c_{base} as the lowermost child when $\forall i \in [1..k]$, it holds that $c_i \succ c_{base}$, and $(c_i, c_{base}) \in E'$. This means that any sub-graph with c_{base} as a sink node will be an ancestral sub-graph of c_{base}. Thus, it holds that $a \subset L_c$. For instance, in Figure 6.1(c), nodes c_7, c_4, c_2 form an ancestral sub-graph with c_7 as the base CS. Similarly, nodes c_6, c_4, c_2 and c_6, c_5, c_2 form ancestral sub-graphs with c_6 as base CS.

Logically, we map each CS to a relational table, so that for a CS c_i we create a relational table $t_i = (s, p_{i,1}, p_{i,2}, \ldots, p_{i,k})$, where s is the id of the subject and $p_{i,1} \ldots, p_{i,k}$ are the properties that belong to P_i, and then we use the CS hierarchy in order to merge the nodes of an ancestral sub-graph with c_i as base into a single table. Specifically, we exploit the property set overlap in order to merge together smaller parent CSs with larger child CSs, in order to minimize the effect of NULL values that will appear for properties in smaller CSs that do no exist in the larger CSs. Thus, c_{base} will be the most specialized CS in its ancestral sub-graph. For this reason, we define a merge operator, *hier_merge*, as follows.

Definition 4. ***(Hierarchical CS Merge)***. Given an ancestral sub-graph $a = (V', E')$, where $V' = \{c_1 = (P_1, r_1), c_2 = (P_2, r_2), \ldots, c_k = (P_k, r_k)\}$ as defined above, then a hierarchical merge of a is given as follows: $hier\_merge(a) = c_a$, where $c_a = (P_1, r_a)$. Here, P_1 is the most specialized property set in a, as c_1 does not have any children in a, while $r_a = \bigcup_{i=1}^{k} r_i'$ is the UNION of the records of all CSs in V', where r_i' is the projection of r_i on P_1. This means that r_i' will contain NULL values for all the non-shared properties of P_1 and P_i, i.e., $P_1 \backslash P_i$. In essence, *hier_merge* is an *edge contraction* operator that merges all nodes of an ancestral sub-graph into one, while removing the edges that connect them. For instance, assume that $V' = \{c_0 = (P_0, r_0), c_1 = (P_1, r_1), c_2 = (P_2, r_2)\}$ is the set of vertices of an ancestral sub-graph with three CSs, with $P_0 = \{p_a, p_b\}$, $P_1 = \{p_a, p_b, p_c\}$ and $P_2 = \{p_a, p_b, p_c, p_d\}$. Thus, $c_0 \succ c_1 \succ c_2$. Hierarchical merging can be seen in Figure 6.2.

Definition 5. ***(Merge Graph)***. Given an inferred CS hierarchy $L_c = (V, E)$, a merge graph is a graph $L' = (V', E')$ that consists of a set of n ancestral sub-graphs, and has the following properties: (i) L' contains all nodes in L such that $V' \equiv V$, i.e., it covers

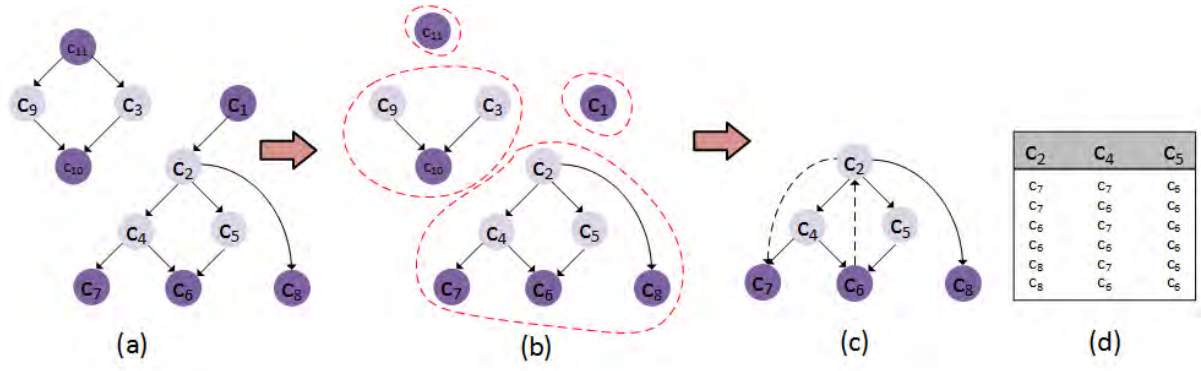


Figure 6.1: (a) A CS hierarchy graph with dense nodes colored in deep purple, (b) the connected components derived by cutting off descendants from dense nodes, (c) a connected component with dashed lines representing inferred hierarchical relationships, (d) all possible assignments of dense nodes to non-dense nodes.

all CSs in the input dataset, (ii) $L^{'}$ contains a subset of the edges in L such that $E^{'} \subset E$, (iii) each node is contained in exactly one ancestral sub-graph a_i, (iv) all ancestral sub-graphs are pair-wise disconnected, i.e., there exist no edges between the nodes of different ancestral sub-graphs. Thus, each ancestral sub-graph can be contracted into one node unambiguously, using the *hier_merge* operator. Also, the total number of relational tables will be equal to the number of ancestral sub-graphs in the merge graph.

Problem Formulation. Given an inferred CS hierarchy $L_c = (V, E)$, the problem is to find a merge graph $L^{'} = (V, E^{'})$ in the form of a set of disconnected ancestral sub-graphs, that provides an optimal way to merge CS nodes. In other words, the problem is to find the best set of ancestral sub-graphs from an inferred hierarchy L_c that minimize an objective cost function $cost(x)$, or more formally:

$$L^{'} = argmin_{x \subset L_c} cost(x) \tag{6.1}$$

This formulation entails several problems. First, the notion of cost depends on possibly subjective factors, such as the query workload, the storage technology, the input dataset and so on. There is no universal cost model that can be deployed in order to assess the effectiveness of a merge graph. Moreover, neither the number of ancestral sub-graphs, nor the set of sub-graph roots is known as part of the input. A CS hierarchy of n nodes can potentially create 2^n sub-graphs, while the number of possible sub-graph roots is also exponential with the respect to the hierarchy size. Thus, given an arbitrary cost function, this is a problem of *non-uniform graph partitioning* on the inferred hierarchy L_c, which is known to be NP-Hard. That is, even with a deployed cost model, it is still an exponential problem to enumerate all possible sub-graphs and find the one with the minimum cost. For these reasons, we approach the problem by deploying a set of rules

and heuristics that find a good merge graph efficiently and offer improved storage and query performance, as will be shown in the experiments.

6.3.2 CS Retrieval and Merging

The primary focus of this work is to improve the efficiency of the storage and query capabilities of relational RDF engines by exploiting the implicit schema of the data in the form of CSs. However, CS merging results in several problems that need to be addressed in this context. These are discussed in what follows.

First, the problem of selecting ancestral sub-graphs is a computationally hard one, as mentioned earlier. For this reason, we rely on a simple heuristic in order to seed the process and provide an initial set of ancestral sub-graph *sink nodes*, that will form the bases of the final merged tables, as defined in Definition 3. For this, we identify *dense* CS nodes in the hierarchy (i.e, with large cardinalities) and use these nodes as the bases of the ancestral sub-graphs. While node density can be defined in many different ways, in the context of this work we define a c_i to be dense, if its cardinality is larger than a linear function of the maximum cardinality of CSs in D, i.e., a function $d : N \rightarrow R$, with $d(c_i) = m \times |r_{max}|$. Here, $m \in [0, 1]$ is called the *density factor*, and r_{max} is the cardinality of the largest CS in D. This means that, by definition, if $m = 0$, no CSs will be merged, because all CSs will be considered dense and thus each CS will define its own ancestral sub-graph, while if $m = 1$, all no ancestral sub-graph will be defined, and all CSs will be merged to one large table, as no CS has a cardinality larger than that of the largest CS. With a given m, the problem is reduced to finding the optimal ancestral sub-graph for each given dense node.

Second, merging tables results in the introduction of NULL values for the non-shared columns, which can degrade performance. Specifically, merging CSs with different attribute sets can result in large numbers of NULL values in the resulting table. Given a parent CS $c_1 = (P_1, r_1)$ and a child CS $c_2 = (P_2, r_2)$ with $|P_1| < |P_2|$ and $|r_1| >> |r_2|$, the resulting $|P_2 \setminus P_1| \times |r_1|$ NULL cells will be significantly large compared to the total number of $r_1 + r_2$ records, thus potentially causing poor storage and querying performance[Pha+15]. For this reason, CS merging must be performed in a way that will minimize the presence of NULL values. The following function captures the NULL-value effect of the merge of two CSs $c_i = (P_i, r_i), c_j = (P_j, r_j)$ with $c_i \succ c_j$:

$$r_{null}(c_i, c_j) = \frac{|P_j \setminus P_i| \times |r_i|}{(|r_j|)} \tag{6.2}$$

Intuitively, r_{null} represents the ratio of null values to the cardinality of the base CS in the merge. The numerator of the fraction represents the total number of cell values that will

be null, as the product of the number of non-shared properties and the cardinality of the parent CS. The denominator represents the cardinality of the base CS. Hence, the base CS must be a descendant (i.e., with more properties) in order to minimize the presence of NULLs.

In order to assess an ancestral sub-graph, we use a generalized version of r_{null} that captures the NULL value effect on the whole sub-graph:

$$r^{g}_{null}(g)|_{c_d} = \frac{\sum_{i=1}^{|g|} |P_d \setminus P_i| \times |r_i|}{|r_d| + \sum_{i=1}^{|g|}(|r_i|)} \tag{6.3}$$

Here, $c_d = (P_d, r_d)$ is the dense root of sub-graph g. However, merging a parent to a dense child changes the structure of the input graph, as the cardinality of the dense node is increased. To accommodate this, we define a cost function that works on the graph level, as follows:

$$cost(g) = \sum_{i=1}^{n} r^{g}_{null}(g_i)|_{c_{di}} \tag{6.4}$$

where n is the number of dense nodes, c_{di} is a dense node and g_i is the ancestral sub-graph with c_{di} as the base node.

Given this cost model and a pre-defined set of dense nodes, our exact algorithm will find the optimal sub-graph for each dense node. An inferred hierarchy graph can be converted to a set of connected components that are derived by removing the outgoing edges from dense nodes, since we are not interested in merging children to parents, but only parents to children. An example of this can be seen in Figure 6.1(b). For each component, we can compute $cost(g)$ as the sum of the costs of these components. The main idea is to identify all connected components in the CS graph, iterate through these components, enumerate all sub-graphs within the components that start from the given set of dense nodes, and select the optimal partitioning for each component.

The algorithm can be seen in Algorithm 7. The algorithm works by first identifying all connected components of the inferred hierarchy (Line 2). Identifying connected components is trivially done using standard DFS traversal, and is not shown in the Algorithm. Then, we iterate each component (Line 3), and for each component, we generate all possible sub-graphs. Then, we calculate the cost of each sub-graph (Line 7) and if it is smaller than the current minimum, the minimum cost and best sub-graph are updated (Lines 8-9). Finally, we add the best sub-graph to the final list (Line 11) and move to the next component.

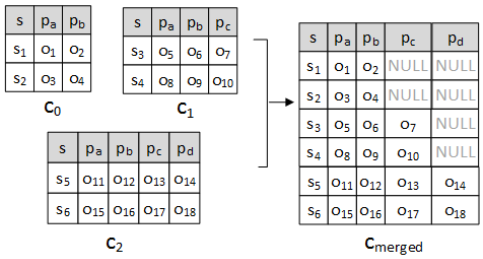


Figure 6.2: Merging the tables of c_0, c_1 and c_2.

To generate the sub-graphs, we do not need to do an exhaustive generation of 2^n combinations, but we can rely on the observation that each non-dense node must be merged to exactly one dense node. Therefore, sub-graph generation is reduced to finding all possible assignments of dense nodes to the non-dense nodes. An example of this can be seen in figure 6.1. In the figure, nodes c_2, c_4, c_5 are non-dense, while nodes c_6, c_7, c_8 are dense. All possible and meaningful sub-graphs are enumerated in the table at the right of the figure, where we assign a dense node to each of the non-dense nodes. An assignment is only possible if there exists a parent-child relationship between a non-dense node and a dense node, even if it is an inferred one (e.g. c_2 is an inferred parent of c_7). Hence, the problem of sub-graph generation becomes one of generating combinations from different lists, by selecting one element from each list. The number of lists is equal to the number of non-dense nodes, and the elements of each list are the dense nodes that are related to the non-dense node.

Complexity Analysis. Assuming that a connected component g has k non-dense nodes and d dense nodes, and each non-dense node k_i is related to $e(k_i)$ dense nodes, then the number of sub-graphs that need to be enumerated are $\prod_{i=1}^{k} e(k_i)$. In the example of figure 6.1, the total number of sub-graphs is $e(c_2) \times e(c_4) \times e(c_5) = 3 \times 2 \times 1 = 6$. In the worst case all k nodes are parents of all d nodes. Then, the number of total sub-graphs is k^d, which makes the asymptotic time complexity of the algorithm $O(k^d)$.

6.3.3 Greedy Heuristic

For very small d (e.g. $d < 4$), the asymptotic complexity of $O(k^d)$ is acceptable. However, in real-world cases, the number of connected components can be small, making d large. For this reason, we introduce a heuristic algorithm for approximating the problem, that does not need to enumerate all possible combinations, but instead relies on a greedy objective function that attempts to find the local minimum with respect to our defined cost model for each non-dense node. Note that it lies beyond the scope of this

Algorithm 7: *optimalMerge*

Data: An inferred hierarchy lattice L_c as a adjacency list , and a set of dense CSs D
Result: A set of optimal ancestral sub-graphs

```
1  init finalList;
2  connectedComponents ← findConnectedComponents(L_c);
3  for each connectedComponent do
4  |   init min ← MAX_VALUE;
5  |   init bestSubgraph ;
6  |   while next ← connectedComponent.generateNextSubgraph() do
7  |   |   if cost(next) < min then
8  |   |   |   min ← cost(next);
9  |   |   |   bestSubgraph ← next;
10 |   end
11 |   finalList.add(bestSubgraph);
12 end
13 return finalList;
```

work to compute the degree of approximation to the optimal solution, however, in our experiments, the heuristic solution is shown to provide significant performance gains.

The main idea behind the algorithm is to iterate the non-dense nodes, and for each non-dense node, calculate the r_{null} function and find the dense node that minimizes this function for the given non-dense node. Then, the cardinalities will be recomputed and the next non-dense node will be examined. The algorithm can be seen in Algorithm 8. In the beginning, the algorithm initiates a hash table, $mergeMap$, with an empty list for each dense node (Lines 1-4). Then, the algorithm iterates all non-dense nodes (Line 5), and for each dense node, it calculates the cost r_{null} of merging it to each of its connected dense nodes (Lines 5-13), keeping the current minimum cost and dense node. In the end, the current non-dense node is added to the list of the dense node that minimizes r_{null} (Line 14). Notice that we do not need to split the hierarchy into connected components in order for $greedyMerge$ to work.

Complexity Analysis. Given k non-dense nodes and d dense nodes, where each non-dense node k_i is related to $e(k_i)$ dense nodes, the $greedyMerge$ algorithm needs $\sum_{i=1}^{k} e(k_i)$ iterations, because we need to iterate all $e(k_i)$ nodes for each k_i. In the worst case, every k_i is related to all d dense nodes, requiring kd iterations. Assuming a constant cost for the computation of r_{null}, then the asymptotic complexity of the greedy algorithm is $O(kd)$, which is a significant performance when compared to the exponential complexity of $optimalMerge$.

Obviously, this process does not necessarily cover all CSs of the input dataset. The

Algorithm 8: *greedyMerge*

Data: A hash table p mapping non-dense CSs to their dense descendants, a set of dense CSs D, and a set of non-dense CSs K

Result: A hash table mapping dense CSs to sets of non-dense CSs to be merged

```
1  init mergeMap;
2  for each d ∈ D do
3  |  mergeMap.put(d, newList());
4  end
5  for each k ∈ K do
6  |  min ← MAX_VALUE;
7  |  init bestDense;
8  |  for each d_k ∈ p.get(k) do
9  |  |  cost ← r_null(k, d_k);
10 |  |  if cost < min then
11 |  |  |  min ← cost;
12 |  |  |  bestDense ← d_k;
13 |  end
14 |  mergeMap.get(bestDense).add(k);
15 end
16 return mergeMap;
```

percentage of the dataset that is covered by this process is called *dense CS coverage*. The remainder of the CSs that are not contained by any merge path are aggregated into one large table containing all of their predicates. If the total coverage of the merging process is large, then this large table does not impose a heavy overhead in query performance, as will be shown in the experiments. Finally, we load the data in the corresponding tables.

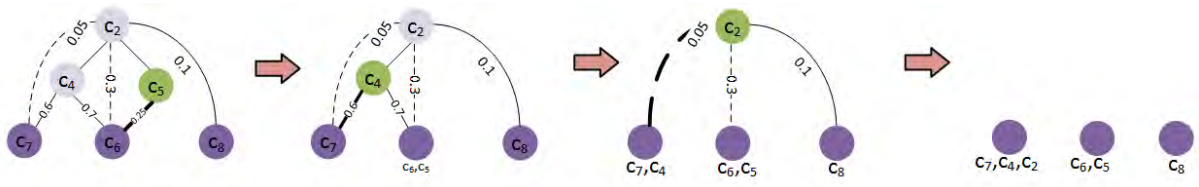


Figure 6.3: An example of greedy merging. Dense nodes are coloured in deep purple. At each step, the non-dense node under examination is coloured with green, while the edge that minimizes r_{null} can be seen in bold.

6.3.4 Implementation

We implemented *raxonDB* as a storage and querying engine that supports hierarchical CS merging, and can be deployed on top of standard RDBMS solutions. Specifically, we used PostgreSQL 9.6, but *raxonDB* can be adapted for other relational databases as well. The architecture of *raxonDB* can be seen in Figure 6.4.

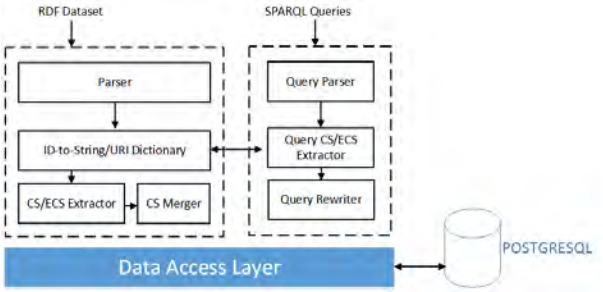


Figure 6.4: Architecture of *raxonDB*.

CS Retrieval and Merging. The processes of retrieving and merging CSs take place during the loading stage of an incoming RDF dataset. CS retrieval is a trivial procedure that requires scanning the whole dataset and storing the unique sets of properties that are emitted from the subject nodes in the incoming triples, and is adopted from our previous work in [Mei+17] where it is described in detail. After retrieving the CSs, the main idea is to compute the inferred CS hierarchy and apply one of the described merging algorithms. Finally, each set of merged CSs is stored in a relational table. In each table, the first column represents the subject identifier, while the rest of the columns represent the union of the property sets of the merged CSs. For multi-valued properties, we use PostgreSQL's array data type in order to avoid duplication of the rows.

Indexing. We deploy several indexes in *raxonDB*. First off, we index the subject id for each row. We also build foreign-key indexes on object-subject links between rows in different CSs, i.e., when a value of a property in one CS is the subject id of another CS. Next, we use standard B+tree for indexing single-valued property columns, while we use PostgreSQL's GIN indexes, which apply to array datatypes for indexing multi-valued properties. This enables fast access to CS chain queries, i.e., queries that apply successive joins for object-subject relationships. Furthermore, we store these links on the schema level as well, i.e., we keep an index of CS pairs that are linked with at least one object-subject pair of records. These links are called Extended Characteristic Sets (ECSs) and are based on our previous work in [Mei+17]. With the ECS index, we can quickly filter out CSs that are guaranteed not to be related, i.e., no joins exist between them, even if they are individually matched in a chain of query CSs. Other metadata and indexes include the property sets of CSs, and which properties can contain multiple values in the same CS.

Query Processing. Processing SPARQL queries on top of merged CSs entails (i) parsing the queries, (ii) retrieving the query CSs, (iii) identifying the joins between them, and (iv) mapping them to merged tables in the database. Steps (i)-(iii) are inherited from our previous work in [Mei+17]. For (iv), a query CS can match with more than one

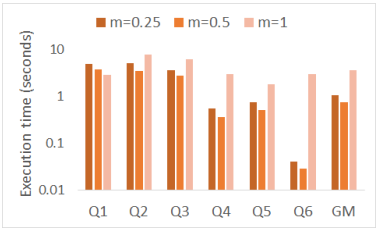
table in the database. For instance, consider a query containing a chain of three CSs, $q_1 \bowtie q_2 \bowtie q_3$, joined sequentially with object-subject joins. Each query CS q_i matches with all tables whose property sets are supersets of the property set of q_i. Thus, each join in the initial query creates a set of *permutations* of table joins that need to be evaluated. For instance, assume that q_1 matches with c_1, c_2, while q_2 matches with c_3 and q_3 matches with c_4, c_5. Furthermore, by looking up the ECS index, we derived that the links $[c_1, c_3]$, $[c_2, c_3]$, $[c_3, c_4]$ and $[c_3, c_5]$ are all valid, i.e., they correspond to candidate joins in the data. Then, $[c_1, c_3, c_4]$, $[c_1, c_3, c_5]$, $[c_2, c_3, c_4]$ and $[c_2, c_3, c_5]$ are all valid table permutations that must be processed. Two strategies can be employed here. The first is to join the UNIONs of the matching tables for each q_i, and the other is to process each permutation of tables separately and append the results. Given the filtering performed by the ECS indexing approach, where we can pre-filter CSs based on the relationships between them, the UNION would impose significant overhead and eliminate the advantage of ECS indexing. Therefore, we have implemented the second approach, that is, process a separate query for each permutation. Finally, due to the existence of NULL values in the merged tables, we must add explicit IS NOT NULL restrictions for all the properties that are contained in each matched CS and are not part of any other restriction or filter in the original query.

6.4 Experimental Evaluation

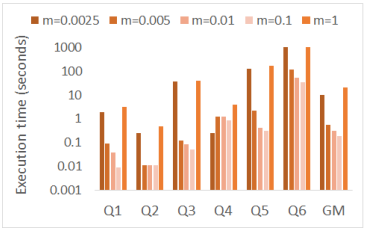
We implemented *raxonDB* on top of PostgreSQL[1]. We did not extend our previous native RDF implementation of *axonDB* [Mei+17], because given the underlying relational schema of the CS tables, we decided to rely on a well-established relational engine for both the planning and the execution of queries, instead of re-implementing it. As the focus of this chapter is to improve RDF storage and querying efficiency in relational settings, we rely on existing mechanisms within PostgreSQL for I/O operations, physical storage and query planning. In this set of experiments, we report results of implementing *hier_merge* with the greedy approximation algorithm, as experimenting with the optimal algorithm failed to finish the merging process even in datasets with small numbers of CSs.

Datasets. For this set of experiments, we used two synthetic datasets, namely *LUBM2000* (≈300m triples), and WatDiv (≈100m triples), as well as two real-world datasets, namely *Geonames* (≈170m triples) and *Reactome* (≈15m triples). LUBM [GPH05] is a customizable generator of synthetic data that describes academic information about universities, departments, faculty, and so on. Similarly, WatDiv[Alu+14] is a customizable generator

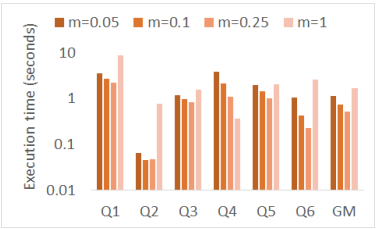
[1]The code and queries are available in https://github.com/mmeimaris/raxonDB



(a) Execution time (seconds) for LUBM



(b) Execution time (seconds) for Geonames



(c) Execution time (seconds) for Reactome

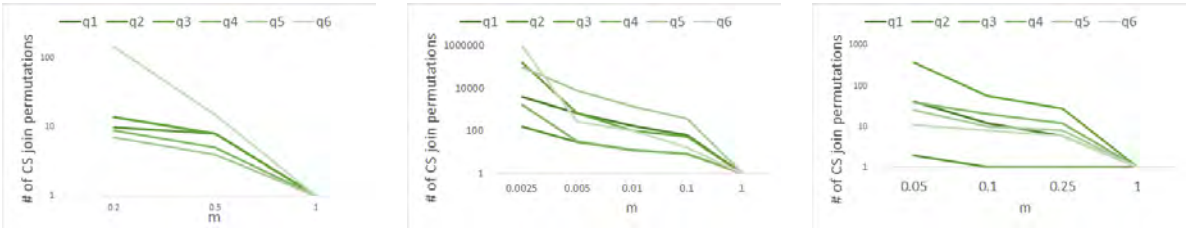
Figure 6.5: Query execution times in milliseconds

with more options for the production and distribution of triples to classes. *Reactome*[2] is a biological dataset that describes biological pathways, and *Geonames*[3] is a widely used ontology of geographical entities with varying properties. Geonames maintains a rich graph structure as there is a heavy usage of hierarchical area features on a multitude of levels.

Loading. In order to assess the effect of hierarchical merging in the loading phase, we performed a series of experiments using all four datasets. For this experiment, we measure the size on disk, the loading time, the final number of merged tables, as well as the number of ECSs (joins between merged tables) and the percentage of triple coverage by CSs included in the merging process, for varying values of the density factor $m \in [0, 1]$. The results are summarized in Table 6.1. As can be seen, the number of CS, and consequently tables, is greatly reduced with increasing values of m. As the number of CSs is reduced, the expected number of joins between CSs is also reduced, which can be seen in the column that measures ECSs. Consequently, the number of tables can be decreased significantly without trading off large amounts of coverage by dense CSs, i.e. large tables with many null values. Loading time tends to be slightly greater as the number of CSs decreases, and thus the number of merges increases, the only exception being WatDiv, where loading time is actually decreased. This is a side-effect of the excessive number of tables (= 5667) in the simple case which imposes large overheads for the persistence of the tables on disk and the generation of indexes and statistics for each one.

Query Performance. In order to assess the effect of the density factor parameter m during query processing, we perform a series of experiments on LUBM, Reactome and Geonames. For the workload, we used the sets of queries from [Mei+17]. We employ two metrics, namely *execution time* and *number of table permutations*. The results can be seen in Figures 6.5 and 6.6. As can be seen, hierarchical CS merging can help speed up

[2]http://www.ebi.ac.uk/rdf/services/reactome
[3]http://www.geonames.org/ontology/documentation.html



(a) # of CS permutations for LUBM

(b) # of CS permutations for Geonames

(c) # of CS permutations for Reactome

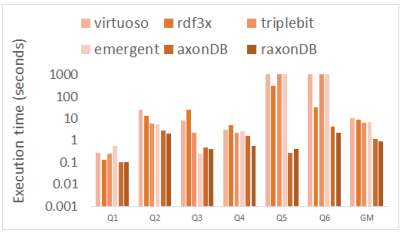
Figure 6.6: # of CS permutations for increasing m

Table 6.1: Loading experiments for all datasets

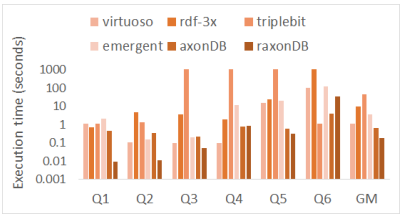
Dataset	**Size (MB)**	**Time**	**# Tables (CSs)**	**# of ECSs**	**Dense CS** Coverage
Reactome Simple	781	3min	112	346	100%
Reactome (m=0.05)	675	4min	35	252	97%
Reactome (m=0.25)	865	4min	14	73	77%
Geonames Simple	4991	69min	851	12136	100%
Geonames (m=0.0025)	4999	70min	82	2455	97%
Geonames (m=0.05)	5093	91min	19	76	87%
Geonames (m=0.1)	5104	92min	6	28	83%
LUBM Simple	591	3min	14	68	100%
LUBM (m=0.25)	610	3min	6	21	90%
LUBM (m=0.5)	620	3min	3	6	58%
WatDiv Simple	4910	97min	5667	802	100%
WatDiv (m=0.01)	5094	75min	67	99	77%
WatDiv (m=0.1)	5250	75min	25	23	63%
WatDiv (m=0.5)	5250	77min	16	19	55%

query performance significantly as long as the dense coverage remains high. For example, in all datasets, query performance degrades dramatically when $m = 1$, in which case the merging process cannot find any dense CSs. In this case, all rows are added to one large table, which makes the database only contain one table with many NULL cells. These findings are consistent across all three datasets and require further future work in order to identify the optimal value for m.

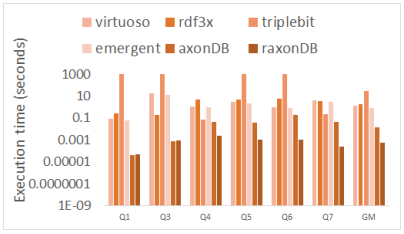
In order to assess the performance of *raxonDB* and establish that no overhead is imposed by the relational backbone, we performed a series of queries on LUBM2000, Geonames and Reactome, assuming the best merging of CSs is employed as captured by m with respect to our previous findings. We also compared the query performance with rdf-3x, Virtuoso 7.1, TripleBit and the emergent schema approach described in [PB16]. The results can be seen in Figure 6.7 and indicate that *raxonDB* provides equal or better performance from the original *axonDB* implementation, as well as the rest of the systems, including the emergent schema approach, which is the only direct competitor for merging CSs.



(a) Execution time (seconds) for LUBM2000



(b) Execution time (seconds) for Geonames



(c) Execution time (seconds) for Reactome

Figure 6.7: Query execution times in milliseconds for different RDF engines

Especially for queries with large intermediate results and low selectivity that correspond to a few CSs and ECSs (e.g. LUBM Q5 and Q6, Geonames Q5 and Q6) several of the other approaches fail to answer fast and in some cases time out.

6.5 Conclusions and Future Work

In this chapter, we tackled the problem of merging characteristic sets based on their hierarchical relationships. As future work, we will study computation of the optimal value for m, taking into consideration workload characteristics as well as a more refined cost model for the ancestral paths. Furthermore, we will study application of these findings in a distributed architecture, in order to further scale the capabilities of *raxonDB*.

Chapter 7

Computational Methods for Containment and Complementarity in RDF Cubes

7.1 Introduction

The increasing adoption of RDF as the de facto Semantic Web standard has led the industrial, government, and academic sectors to leverage Linked Data technologies [CRT13; VT+11] in order to publish, re-use and extend big amounts of proprietary data. A large subset of data on the web consists of multidimensional data about policies, demographics, socio-economics and health data among others [Tam16].

Statistical multidimensional data is often represented in the form of data cubes. Under this model, a single data record, named *observation* or *fact*, is broadly defined as the value of a specific measure over several different observed dimensions [CD97]. For example, Germany's population for the year 2001 can be represented as an observation with *population* as the measure, and *location* and *time* as the dimensions, with the values *Germany* and *2001* instantiating these dimensions. The use of hierarchical values enables the representation of information on multiple combinations of levels, such as the *female population of a country in the last decade*, or *the total population of a city in the last year*. An example of dimension hierarchies can be seen in Figure 7.1. The RDF Data Cube Vocabulary (QB) [CRT13] provides a schema for RDF multidimensional data, allowing for the representation of schemas, dimensions, measures, hierarchies, observations, among others. Considering the aforementioned example, its mapping to the RDF QB vocabulary can be seen in Listing 7.1. RDF QB enables different data publishers to fit their data in a common meta-schema, and reuse common entities across different sources.

Hence, remote datasets often exhibit overlaps in the values that instantiate their dimensions and measures, this way creating implicit relationships between observations among remote sources; for example, an observation can be a specialization or generalization of another observation from a different dataset, an observation can partially aggregate information contained in other observations, or finally different observations can capture complementary knowledge and can be combined together.

Inspired by the notion of fusion cubes [Abe+13] towards self-service analytics, we define instance-level relationships for multidimensional observations, and we address the challenge of efficient computation of these relationships over multiple data cubes.

In order to better illustrate the defined relationships, we discuss an example scenario that will be used throughout this chapter. In this scenario, the user has gathered data from several remote sources in order to explore unemployment and population demographics. The gathered data are in the form of observations and originate from Linked Data sources. As such, they exhibit heavy re-use of the same hierarchies and code lists[1]. The example hierarchies are shown in Figure 7.1, and a snapshot of the gathered data is shown in Figure 7.2, where the analyst has gathered data from three different datasets, namely D_1, D_2 and D_3.

Observations o_{11} and o_{31} have the same values for the *refArea* and *refPeriod* dimensions, while the *sex* dimension has the most general value possible, i.e., *Total*. Intuitively, this means that the two observations measure different things for the same setting, and are thus *complementary*. Furthermore, observations o_{21}, o_{22} measuring unemployment in Greece and Italy for the year 2001, are generalizations of o_{32} and o_{33}, because the latter measure unemployment in Athens and Rome, which are sub-parts of Greece and Italy respectively, for a sub-period of 2011. For the data of the example, these relationships can be seen in Figure 7.3.

Discovering relationships such as the above is useful in several tasks. Multidimensional data enable third parties to study, process and visualize information in order to perform more complex analytics such as combining different datasets, discovering new knowledge, assisting socio-centric processes such as data journalism, as well as enabling evidence-based policy making on the government and industry levels[NDS00; Böh+10; PS+96]. As potential users, we consider data scientists, such as data analysts or, data journalists and business users, who collect data from external and corporate sources in their personal data cube for analysis purposes. Then, the added value of detecting such relationships

[1]Some degree of schema alignment is often necessary in realistic scenarios. This type of alignment is used in the following two prominent cases: (a) traditional BI settings, where all dimensions provide a reconciled dimension bus, and (b) user-initiated data collections from the web.

can be summarized in the following. First and foremost, observations that originate from different datasets become linked, and thus comparable in future analytical tasks, as in the case of fusion cubes [Abe+13] towards self-service analytics. Furthermore, navigation and exploration of aggregations of datasets is facilitated with the existence of links on the instance level. Traditional OLAP tasks such as rolling up or drilling down can be applied for the exploration of remote cubes. These types of relationships can help quantify the degree of relatedness across remote datasets and this way provide recommendations for online browsing. Finally, materializing these relationships speeds up online exploration, as well as computation of k-dominance [Cha+06], skylines and k-dominant skylines.

Finding implicit knowledge across different sources is a non-trivial, computationally challenging task [Böh+10], that is inherently quadratic at its core, since all pairs of records must be examined. Traditional query processing methods such as SPARQL engines, and inference-based methods fail to address this issue efficiently as the volume of data increases. For instance, our experiments with recursive, property-path based SPARQL queries show that even for small numbers of records (≈20,000 observations from 7 datasets) require more than one hour in commodity hardware to detect and materialize pair-wise containment relationships. Similary, inference-based methods such as SWRL [Hor+05] and Jena Rules [Car+04] fail to scale due to the transitive nature and the universal restrictions of these relationships; the search space expands exponentially [Don03]. Hence, the need arises to establish more efficient methods that can scale to the size of the web of data.

Approach Overview. In this chapter, we address efficient computation of three specific types of relationships in multidimensional data from different sources, namely *full containment*, *partial containment* and *complementarity*. Full containment between observations occurs when all dimension values in two observations are hierarchically related in the same direction, i.e., the containing observation is a generalization of the contained observation, while partial containment occurs when at least one, but not all of the dimension values are hierarchically related. Complementarity occurs when two observations identify the same setting but measure different aspects, and thus hold complementary information. Specifically for complementarity, we extend the notion of *schema complement*[DS+12] to fit observations.

We present a quadratic baseline algorithm for computation of these relationships, and introduce three alternative methods that target efficiency and scalability, an approach based on pruning the required comparisons by clustering together related observations, and two approaches based on the notion of the multidimensional lattice, a data structure that groups observations based on their defined combination of dimension levels. The

```
ex:obs1 a qb:Observation ;
  qb:dataSet ex:dataset ;
  ex:refPeriod ex:Y2001 ;
  sdmx-attr:unitMeasure ex:unit ;
  ex:refArea ex:DE ;
  ex:population "82,350,000"^^xmls:integer .
```

Listing 7.1: Example RDF Data Cube observation

Figure 7.1: Hierarchical code list for the dimensions in Figure 7.2.

first approach exploits the dimension levels in order to reduce the required comparisons, whereas the optimized approach makes use of the inherent hierarchical structure of the lattice to further speed up the detection process. We perform an extensive experimental evaluation of the 4 methods over 7 real-world multidimensional datasets, and compare their efficiency with two traditional approaches, namely SPARQL querying and rule-based inferencing. Finally, we evaluate the scalability of our approach in an artificially generated dataset.

Contributions. The contributions of this chapter are summarized as follows:

- we formally define the notions of *full containment*, *partial containment* and *complementarity*,
- we present four algorithms, a baseline, data-driven technique for computing these properties in memory, and three alternative approaches with the scope of improving performance with respect to efficiency and scalability,
- we perform an extensive experimental evaluation of the achieved efficiency and scalability over both real-world and synthetic datasets, comparing between the proposed methods, a SPARQL-based and a rule-based approach.

The remainder of this chapter is organized as follows. Section 2 discusses related work, Section 3 presents the preliminary definitions and formulates the problem. Section 4 presents the proposed approaches. Section 5 describes the experimental evaluation. Finally, Section 6 concludes this chapter.

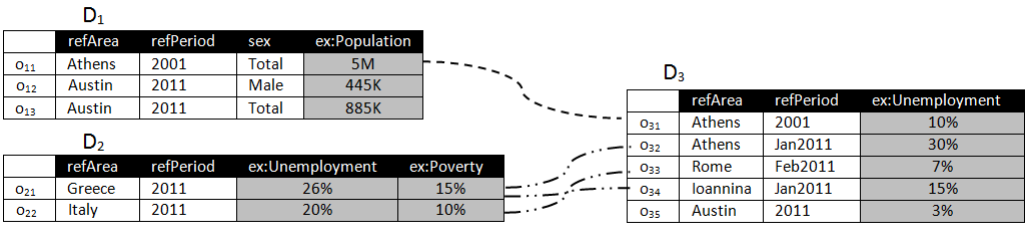


Figure 7.2: Candidate relationships between observations.

	refArea	refPeriod	sex	ex:Unemployment	ex:Poverty	ex:Population
o21 **contains:** --o32 --o34	Greece **contains:** --*Athens* --*Ioannina*	2011 **contains:** --*Jan2011* --*Jan2011*	Total *Total* *Total*	26% *30%* *15%*	15% - -	- - -
o22 **contains:** --o33	Italy **contains:** --*Rome*	2011 **contains:** --*Feb2011*	Total *Total*	20% *7%*	10% -	- -
o11 **complements** --o31	Athens *Athens*	2001 *2001*	Total *Total*	- *10%*	- -	5M -
o13 **complements** --o35	Austin *Austin*	2011 *2011*	Total *Total*	*3%*	-	885K

Figure 7.3: Derived containment and complementarity relationships from datasets D_1, D_2 and D_3 of Figure 7.2.

7.2 Related Work

In the context of RDF, there exists a growing body of research focused on the provision of tools, methods and techniques for representing, analysing and processing multidimensional data. In this chapter, we build on, and extend the work presented in [Mei+16b][MP14], where we introduced the notions of full containment, partial containment and complementarity, and discussed three approaches for efficient computation of these relationships. We formally define the problem components and relationships, we improve upon the presented approaches, and we introduce a novel optimization of the cube masking approach that targets performance, complementing it with an extended experimental evaluation.

The general problem of detecting similarities between resources is central in the fields of entity resolution, record linkage and interlinking [Men+11; NA11; Vol+09; Pap+11; Eft+17; Pap+16]. However, these approaches are focused on finding links between resources from different datasets without taking into account multidimensional features such as dimension values. To the best of our knowledge, this is the first work centred on the definition, representation and computation of relationships between instance-level multidimensional data.

7.2.1 Schema-Level Hierarchy Extraction for OLAP

Traditional OLAP and data warehousing systems and frameworks are often used for performing analytical queries with operators that can generalize (roll-up) or specialize (drill-down) specific records, based on their defined dimension hierarchies. These are most commonly built upon a relational backbone, or native data cube implementations [VZ14], and rely on the management of data from trusted sources with known schemas and interconnections. The latter assumption does not apply to data analysis in the Data Web, because the input potentially originates from remote, implicitly related sources. Furthermore, these types of systems are not built for detecting instance-level relationships such as containment and complementarity. For instance, early approaches on automatic concept hierarchy detection have been proposed, but deal with hierarchy construction on the schema or the attribute level, rather than the instance level[HF94][Han+96]. Similarly, extraction of concept hierarchies from web tables and transformation to data cubes has been studied in [AL12], and extraction of dimension hierarchies from ontological data has been addressed in [RA07]. Thus, the process of detecting instance-level relationships must be translated to queries over the employed format (e.g. SQL or SPARQL queries) in the form of query operators, which makes the detection costly, as will be shown in the experiments, or derived with the use of customized ETL processes.

7.2.2 Analytical Mining in the presence of hierarchies

The problem of finding related observations in multidimensional data spaces has been addressed in the field of Online Analytical Mining (OLAM)[MRB99], which refers to the integration of data mining techniques into traditional OLAP. These methods have been successfully used for tasks such as classification of observations and detection of outliers [MRB99][AY01], exploration recommendation[Ali+14][BRV11], intelligent exploratory query recommendation [SS01], discovery of implicit knowledge[Gia+09] and optimized OLAP querying[MRB99]. In [RGG15], the authors introduce the *shrink* operator that exploits hierarchies as a means to provide summarized and shortened cubes. It achieves this by clustering together and consequently merging similar facts, in order to assist human-readability. Their work is not driven by efficiency, as in our case, rather they focus on improving the presentation of cubes in graphical form. In [CCM15], the authors tackle the problem of performing roll-up and drill-down operations on continuous dimensions, rather than fixed dimension values as in our case, and to this end they employ hierarchical clustering on the numerical values of the dimensions.

7.2.3 Partial Materialization

In [DT16], the authors target efficiency in the performance of OLAP related tasks by studying partial materialization techniques for aggregation and summarization of multi-dimensional observations. Similarly, in [Hal01] the authors propose materialized views for efficient processing of aggregation queries. These two approaches resemble our notion of observation containment, and can indeed be complemented by the efficient computation of this type of relationship on the observation level. In [Xie+16], the authors propose a probabilistic approach for providing full and partial materialization over aggregate analytics at the cube level. Ibragimov et al. [Ibr+16] use materialized views formulated as SPARQL queries in order to address the lack of support for incomplete data with implicit information, and they evaluate their approach on multidimensional RDF data represented with the QB4OLAP model [EV12], which is an extension of the RDF QB vocabulary. This way, they provide scalable support for aggregate queries that include roll-up and drill-down exploration over incomplete data. This work is complementary to our methods for efficient computation of aggregate relationships between observations (i.e., observation containment) and the (partial) materialization of RDF views can be complemented by the optimizations presented in this chapter.

7.2.4 Skyline Computation

The computation of containment relationships has been addressed in different contexts, with *skyline computation* being the most prominent one. Specifically, skyline computation is based on the definition of observation *dominance*, and asserts the existence of points in the multidimensional dataset that are not dominated (i.e., fully contained) by other points [Yua+05; TEO+01; KRR02]. The set of these points comprises the skyline of a dataset, and has found important applications in summarization and recommendation tasks in data warehousing. In this regard, full containment is a generalization of the skyline problem, where we are interested in all intermittent skylines at all of the level combinations of the hierarchies. Similarly, partial containment in the same context is referred as the *k-dominance* problem in [Cha+06], where the authors propose a methodology for efficient computation of partial skylines in subsets of the original dimension set of the input. The problem of Subspace Skyline computation is presented in [Rah+17], and is defined as the computation of partial skylines in subsets of the dimensions of a given dataset. This is relevant to our definition of partial containment, however, partial containment can be defined in several different subsets of the dimensions between two observations at the same time, which makes our problem more computationally complicated.

7.2.5 Observation Relationships via Similarity Metrics

As a metric of relatedness, containment and complementarity relationships have the potential to highlight similarity between observations as well as datasets, even though this is not the main focus of this work. In this regard, Aligon et al.[Ali+14] use query features in OLAP sessions in order to define distance functions that capture instance-level similarities. In a related context, Baikousi et al.[BRV11] propose several similarity metrics in the form of distance functions that specifically address distances in hierarchical code lists. In [HL11] the authors propose a set of scalable multidimensional methods via hierarchical clustering in order to measure similarity between reports in the same cubes. In the broader context of web-based data sources, the work in [DS+12] defines the notions of *schema* and *entity complement*, the latter of which is the basis for our definition of observation complementarity.

Recent works in entity resolution (ER) have been shown to perform efficiently in cases when duplicate entities need to be identified based on pre-defined similarity metrics. As ER is mainly a quadratic problem, in the sense that all pair-wise comparisons are needed in order to identify duplicate or similar entries, these works usually focus on providing fast ways of partitioning the search space in smaller chunks, or blocks, and limiting the pair-wise comparisons of records within the same, or nearby blocks. Examples of these have been addressed in [Ben+09][Pap+11][Pap+13a], while the reader is referred to [Pap+15] for an extensive experimental evaluation of recent schema-less and schema-aware techniques. While these approaches aim at identifying similar entries, they do not address cases where the examined attributes (i.e., dimension values) exhibit hierarchical relationships, as in our case. Furthermore, they provide approximate solutions, rather than exact ones.

7.2.6 Multidimensional Linked Data Related Approaches

The versatility of the RDF model has enabled the creation of several schemas, vocabularies and ontologies that are used for the representation of multidimensional data, concept hierarchies, code lists and so on, with the most prominent example being the RDF Data Cube Vocabulary (QB). Furthermore, many high-level representation models such as RDFS[2] and SKOS[3] provide conventions for representing hierarchical dependencies, such as *rdfs:subClassOf*, and *skos:broader/skos:narrower*. In fact, in this work, we rely on *skos* concepts and hierarchical properties in order to detect and represent hierarchical dependencies between values in code lists that are shared among different datasets.

[2]https://www.w3.org/TR/rdf-schema/
[3]https://www.w3.org/2004/02/skos/

In the context of Linked Data, a thorough survey of how OLAP exploration tasks and processes are performed in the context of the Semantic Web, is given in [Abe+15]. The authors perform a classification of research works that leverage Semantic Web technologies for OLAP schema design and data provisioning according to five criteria, namely *materialization*, *transformations*, *freshness*, *structuredness*, and *extensibility*, and further analyzed these technologies with respect to *Reasoning*, *Computation* and *Expressivity*. In this regard, our work can be categorized as a *computational* approach with instance-level inferred materialization as the ultimate goal, in order to allow for constant-time access to more complex exploration tasks, such as querying implicit information. The work in [KSH14] addresses the problem of finding related cube entities amongst different and remote sources with the use of an extended *Drill-Across* operator. The authors tackle relatedness on the level of the cube schema, and to that end they define relatedness by quantifying the difficulty of tasks such as conversion between cubes and merging of different cubes. In [KH13] the authors advocate the development of native engines that translate traditional OLAP to SPARQL queries and materialized views in order to tackle the lack of support for analytical workflows in traditional RDF management systems. In [Kom+16] the authors propose a SPARQL-based ETL framework for extracting multidimensional star-pattern data and hierarchies from RDF and Linked Data using dynamically generated SPARQL queries, but the authors note the lack of functionality regarding information extraction in the form of aggregation functions in their approach. In [EV16], the authors propose *CQL*, a conceptual algebra for querying multidimensional RDF data, which they use to translate SPARQL queries and apply traditional SPARQL query optimization methods. In [BG17], the authors propose a method for discovering and merging OLAP cubes in the context of RDF. While this is an interesting approach, it is not centred on the detection of instance-level relationships, as is the main focus in our work. In [EV12][Var+16a], the authors present the QB4OLAP vocabulary, an extension of the RDF QB vocabulary with OLAP constructs such dimension levels, with the aim to go beyond the representational capabilities of QB and enable native support for traditional OLAP tasks in Linked Open multidimensional datasets. Extending on this work, in [Var+16b] the authors implement a tool for performing OLAP-related tasks on QB Linked Data without requiring SPARQL expertise. To this end, they provide functionality for semi-automatic transformation of existing QB datasets to QB4OLAP, and high-level query formulation using the generic QL language. Furthermore, they implement an enrichment module that is able to extract hierarchies and code lists from remote Linked Data sources.Even though the scope of these works is not to provide efficient computation of instance-level relationships between observations, as is our focus, they are complementary to our approach.

7.3 Problem Definition

In this section, we present preliminaries of our approach and formulate the problem addressed in this chapter. As was noted in the introduction, in the context of this work we are interested in processing linked open multidimensional datasets with OLAP cube characteristics. These datasets must exhibit several characteristics, the main of which is the conformance to a representational model that allows the description of cubes and cube facts, i.e., observations. Furthermore, linked open data technologies use commonly agreed ontologies for describing data across different sites. This enables us to process datasets which, although being published by different sources, are following the same semantics for the description of the schema, the values of the dimensions, the unit of measurements, etc. Under this scope, we consider a problem space consisting of n input datasets, each of which follows a multidimensional schema in the form of one or more cubes, containing observation instances. In the following, we present and define the components of the problem.

Definition 16 ***Dimension Schema.***

Following the definitions in [GGV12], a dimension schema P is a tuple $(L, \rightarrow)$ where L is a non-empty finite set of values $h_1, h_2, \ldots, h_n$ along with a top value concept named *All*, and $\rightarrow$ is a partial ordering of the values in L. This partial ordering essentially defines a *hierarchy* in the values. In the setting of this work, L is a fixed code list, that is represented by URIs. Furthermore, the $\rightarrow$ operator in the definition of P defines a hierarchy such that when $h_i \succ h_j$, where h_i, h_j are values in L, then h_i is a hierarhical parent of h_j. The concept $h_{root} = All$ is defined as the top level concept in each code list, i.e., an ancestor of every other value in L, such that $\forall h_i : h_{root} \succ h_i$. This hierarchical ancestry is reflexive, i.e. $\forall h_i : h_i \succ h_i$. Figure 7.1 shows several code list values in their respective hierarchies.

Definition 17 ***Cube Schema.***

A cube schema CS is a tuple (P, M), where P is a dimension schema, and M is a finite set of *measures*. Measures are essentially measurements of a specific metric that are instantiated over a point in the multidimensional space defined by P.

Definition 18 ***Observation.***

An observation is a cube instance that defines a single point in the multidimensional space. More specifically, an observation o is a tuple of the form $o_a = (h_a^1, h_a^2, \ldots, h_a^l, v_a^1, v_a^2, \ldots, v_a^m)$, where h_a^i is the value of dimension P_i, and v_a^i is the value of measure M_i for observation o_a. In other words, an observation is an entity that instantiates all of the dimensions

and measures that are defined in its respective dataset. In our running example, the values in the white cells represent dimension values (e.g. *"Athens"* is a value for dimension *refArea*), while grey cells represent the values of measures, such as *10% unemployment*.

Definition 19 ***Dataset Structure.***

Let D = $\{D_1, \ldots, D_n\}$ be the set of all input datasets. A dataset $D_i \in D$ consists of a set of data observations $O_i = o_1, \ldots, o_k$, as well as a set of dimension schemas $\mathbf{P_i} = P_1, \ldots, P_l$ and a set of measures $\mathbf{M_i} = M_1, \ldots, M_m$. Thus, D_i is a tuple of the form (O_i, CS_i), where $CS_i = (\mathbf{P_i}, \mathbf{M_i})$. This means that D is defined as the union of the respective components of the input datasets, that is, $D = (O_D, CS_D)$, with $O_D = \bigcup_{i=1}^{n} O_i$, $CS_D = (P_D, M_D)$, with $P_D = \bigcup_{i=1}^{n} \mathbf{P_i}$ and $M_D = \bigcup_{i=1}^{n} \mathbf{M_i}$. In the running example, all three datasets D_1, D_2, D_3 share the dimensions *refArea* and *refPeriod*. Furthermore, D_2 and D_3 share the measure *ex:unemployment*.

These datasets originate from possibly remote, linked open data sources, and can exhibit overlap in both their records, and the used/reused vocabularies. Hence, dimension values that instantiate observation instances are drawn from linked open codelists and vocabularies and can be shared across datasets. This creates the possibility of linkage between datasets on the instance level, i.e., observations can be related across remote datasets. For this reason, we will define three types of relationships that pairs of observations can exhibit, namely *full containment*, *partial containment*, and *complementarity*.

Definition 20 ***Observation Complementarity.***

Complementarity is a binary relationship between a pair of observations. Specifically, we define complementarity as a function $compl : O \times O \rightarrow B$, where B is the boolean set $B = \{0, 1\}$. Let o_a and o_b be two observations that originate from datasets $D_a = (O_a, CS_a)$ and $D_b = (O_b, CS_b)$ respectively, with $CS_a = (P_a, M_a)$ and $CS_b = (P_b, M_b)$. Then, o_a complements o_b when the following conditions hold:

$$\forall P_i \in \mathbf{P_a} \cap \mathbf{P_b} : h_a^i = h_b^i \tag{7.1}$$

$$\forall P_j \in \mathbf{P_a} \triangle \mathbf{P_b} : h_b^j = h_{root} \tag{7.2}$$

where $\mathbf{P_a} \triangle \mathbf{P_b}$ is the symmetric difference of sets $\mathbf{P_a}$ and $\mathbf{P_b}$. When both conditions hold true, there is a complementarity relationship between o_a and o_b, i.e., (1) $\wedge$ (2) $\Rightarrow compl(o_a, o_b) = 1$. We denote this with $o_a \overset{c}{=} o_b$. This definition essentially relates the two observations as occupying the same point in the multidimensional space defined by their shared dimensions. These shared dimensions $\mathbf{P_a} \cap \mathbf{P_b}$ must be instantiated with the same values from the respective code lists (condition (1)), and all non-shared

dimensions, i.e., $\mathbf{P_a} \triangle \mathbf{P_b}$, must be equal to the root of the dimension hierarchy, i.e. the value $h_{root} = All$, thus providing no further specialization, (condition (2)). This relationship indicates that the two observations basically *identify the same setting*. The complementarity relationship is symmetric, thus $o_a \overset{c}{=} o_b$ also implies $o_b \overset{c}{=} o_a$.

For instance, in a one-dimensional setup where the only dimension is *refArea*, an observation that measures *poverty* for the value *Greece* in this dimension, exhibits complementarity with an observation that measures *population* in Greece. If the second observation originates from a dataset that includes the dimension *sex*, then the two would complement each other only if the non-shared dimension (i.e., *sex*) provides no specialization in its respective observation. In our example, observations o_{11} and o_{31} are complementary, in that they measure different things for Athens in 2001. Condition (2) holds for o_{31} in the sex dimension, where absence of the dimension implies existence of the root value $h_{root} = All$. The fact that o_{11} refers to all values from the sex dimension does not provide any further specialization and is inherently found in o_{31} as well.

Definition 21 ***Observation Containment.***

A special type of directed relationship between a pair of observations exists when one of the two observations is a specialization of the other. We call this a *containment relationship*. For instance, the population of Greece implicitly contains all the populations of Greece's cities. However, there are cases where only a subset of the dimensions exhibits this type of relationship between two observations. This is an important relationship as it shows which dimensions need to be abstracted (i.e., rolled-up) in order for two observations to become comparable and/or relatable. For this reason, we define two notions of containment, namely *full containment* and *partial containment*. Full containment is exhibited when all dimension values of one observation are subsumed by the values of the respective dimensions of another observation, while partial containment is exhibited when at least one, but not all dimension values are subsumed from one observation to another.

More specifically, we define the existence of containment as a function $cont : O \times O \rightarrow [0, 1]$, where a value of 0 means that no containment relationship exists, while a value of 1 means that there exists absolute containment between a pair of observations. When $cont(o_i, o_j) = 1$, we call this *full containment*. On the other hand, when $0 < cont(o_i, o_j) < 1$, we call this *partial containment*. These are defined as follows.

Definition 21.1 ***Full Containment.***

Let o_a and o_b be two observations from datasets $D_a = (O_a, CS_a)$ and $D_b = (O_b, CS_b)$ respectively, with $CS_a = (\mathbf{P_a}, \mathbf{M_a})$ and $CS_b = (\mathbf{P_b}, \mathbf{M_b})$. Full containment between two

observations, $o_a \in O_a$ and $o_b \in O_b$, exists when the following conditions hold:

$$\mathbf{P_a} \cap \mathbf{P_b} \neq \emptyset \tag{7.3}$$

$$\forall P_i \in \mathbf{P_a} \cap \mathbf{P_b} : h_a^i \succ h_b^i \tag{7.4}$$

Furthermore, the non-shared dimensions must not provide any further specialization, as stated in condition (2). When all conditions are true, the pair of observations exhibits full containment. Therefore, (2) $\wedge$ (3) $\wedge$ (4) $\Rightarrow$ $cont(o_a, o_b) = 1$. We denote this with $o_a \overset{f}{\succ} o_b$. The intuition behind these conditions relies on several facts. An observation o_a fully contains o_b when values of all shared dimensions for o_a are hierarchical ancestors of the values for the same dimensions in o_b as stated in (4). Furthermore, the conjunction of the two dimension sets must be non-empty, as stated in (3). This condition is needed because the universal condition in (4) would be evaluated to true in the case that there are no shared dimensions. Observe that the containment property is not symmetric, i.e., given $o_a \overset{f}{\succ} o_b$, then $o_b \overset{f}{\succ} o_a$ is not implied. In the example, o_{21} fully contains o_{32} and o_{34}.

Definition 21.2 ***Partial Containment.***

Let o_a and o_b be two observations from datasets $D_a = (O_a, CS_a)$ and $D_b = (O_b, CS_b)$ respectively, with $CS_a = (\mathbf{P_a}, \mathbf{M_a})$ and $CS_b = (\mathbf{P_b}, \mathbf{M_b})$. Then, o_a partly contains o_b when there exists at least one dimension whose value for o_a is a hierarchical ancestor of the value of the same dimension in o_b, as stated in the following condition:

$$\exists P_i \in \mathbf{P_a} \cap \mathbf{P_b} : h_a^i \succ h_b^i \tag{7.5}$$

Thus, partial containment is a generalized case of full containment. We denote this as $o_a \overset{p}{\succ} o_b$. In the example, observation o_{21} partially contains o_{31}, because *Greece* contains *Athens* but *2001* does not contain *2011*. The notation is summarized in Table 7.1.

Problem Definition. Based on the above, our problem is formulated as follows. Given a set D of source datasets, and a set O of observations in D, for each pair of observations $o_i, o_j \in O, i \neq j$, assess whether a) $o_i \overset{f}{\succ} o_j$, b) $o_i \overset{p}{\succ} o_j$ and c) $o_i \overset{c}{=} o_j$. In the following section, we provide our techniques for computing these properties.

7.4 Algorithms for computing complementarity and containment

In this section, we present four methods for the computation of the proposed relationships, i.e, full/partial containment and complementarity. We first present a baseline method

Table 7.1: Notation

Notation	Description
o_i	The i-th observation in a set O
$\mathbf{P}$	A set of dimension schemas
$\mathbf{M}$	A set of measure schemas
P_i	The i-th dimension in a set $\mathbf{P}$
M_i	The i-th measure in a set $\mathbf{M}$
h_a^i	Value of dimension P_i for observation o_a
$h_a^i \succ h_b^i$	h_a^i is a parent of h_b^i
h_{root}	The root value *All*
$compl(o_a, o_b)$	Complementarity function
$cont(o_a, o_b)$	Containment function
$o_a \overset{c}{=} o_b$	o_a complements o_b
$o_a \overset{f}{\succ} o_b$	o_a fully contains o_b
$o_a \overset{p}{\succ} o_b$	o_a partially contains o_b

that requires quadratic computations, i.e., comparisons for all pairs of observations, and then we propose three efficient and scalable alternatives. The first uses clustering to group related observations together and limit comparisons within clusters, the second uses the notion of a cube mask lattice [HRU96] in order to take advantage of the hierarchical relationships between the levels of the dimensions of each observation and limit comparisons between hierarchically related containers, and the third proposes an optimization over the cube masking method.

7.4.1 Baseline

First, we present a *baseline* method, which performs comparisons between all pairs of observations in the input dataset. Performing all pair-wise comparisons makes the *baseline* algorithm quadratic, and thus not efficient for large datasets. However, this method requires minimal preprocessing and is thus suited for smaller input sizes.

Representation. The *baseline* algorithm works under the premise that observations are represented as bit vectors in a large bitmap, which essentially defines a multidimensional feature space. Let $D_1 = (O_1, CS_1), \ldots, D_n = (O_n, CS_n)$ be n input datasets, then this bitmap is represented by an *occurrence matrix* **OM**, where each row is defined by an observation key in $\bigcup_{i=1}^{n} O_i$, and each column represents a specific value in the codelist hierarchies of the union of all dimension schemas $P = \bigcup_{i=1}^{n} P_i$, with $P_i \in CS_i$. That is, for each observation, we set the bits that correspond to the dimension values of the observation. This representation also captures the ancestral relationships between hierarchical values of the dimensions, by encoding the occurrence of a dimension value together with

all of its parents. For this, we set the value of 1 to all columns that are ancestors of this value.

Prior to creating the representation space, it is often an implicit requirement of the input to perform dimension alignment, and have a reconciled dimension bus in the multidimensional space. This can be achieved by applying established entity resolution techniques for interlinking dimension values across different datasets. Even though we use the inherent linkage of Linked Open Data, it is often necessary to further resolve disambiguations and similarities in the data. Note however that Entity resolution tasks are beyond the scope of this work, which focuses on data analytics rather than on data integration problems. Thus, we consider that schema alignment and mapping of values is feasible and amortized over time (especially when data is collected from already processed sources at a regular basis) .

The occurrence matrix **OM** is a matrix that is defined over the union of all input datasets, and encodes each observation with respect to the values, all the way to the root, of its dimensions. Thus, presence of a dimension value is denoted with the corresponding bit of the column of the dimension set to 1. Furthermore, hierarchical occurrence is also represented in **OM**, by setting all parents of the dimension value to 1, up to the root.

Each observation is defined over this matrix of dimensions $|O| \times |L|$ as a bit vector representing the occurrences of codelist values in their respective dimensions, that is, each value $h_i \in L$ becomes a feature, i.e., a column in **OM**. For example, given an observation o_a, and its value $h_a^{P_j}$ for dimension P_j, then the value as well as its hierarchical subsumption is encoded in **OM** by assigning a set bit in the column that represents $h_i = h_a^j$, as well as all of the parents of h_i. Finally, we set the columns representing h_{root} for all observations that do not contain P_j in their schema. This means that dimensions not appearing in a cube schema are assigned the *top* concept, this way marking the distinct lack of specialization in the absence of a dimension.

Conceptually, **OM** can be vertically partitioned into a series of sub-matrices, each one representing one dimension in the unified schema of the input, i.e., $\mathbf{OM} = [\mathbf{OM_1}, \ldots, \mathbf{OM_{|P|}}]$, where $\mathbf{OM}_i$ is a sub-matrix that represents occurrences for all values of dimension p_i. For the example of Figure 7.2, and given the hierarchical code lists shown in Figure 7.1, the **OM** matrix is depicted n Table 7.2. The baseline algorithm uses **OM** for computing containment scores, and encoding these scores in a pair-wise *containment matrix*. The latter is used for the computation of both the complementarity and the containment relationships.

Table 7.2: Matrix OM for the example of Figure 7.2

	refArea										**refPeriod**					**sex**		
	WLD	EUR	AM	GR	IT	Ath	Rom	US	TX	Aus	ALL	2001	2011	Jan11	Feb11	T	F	M
obs_{11}	1	1	0	1	0	1	0	0	0	0	1	1	0	0	0	1	0	0
obs_{12}	1	0	1	0	0	0	0	1	1	1	1	0	1	0	0	1	0	1
obs_{21}	1	1	0	1	0	0	0	0	0	0	1	0	1	0	0	1	0	0
obs_{22}	1	1	0	0	1	0	0	0	0	0	1	0	1	1	0	1	0	0
obs_{31}	1	1	0	1	0	1	0	0	0	0	1	1	0	0	0	1	0	0
obs_{32}	1	1	0	1	0	1	0	0	0	0	1	0	1	1	0	1	0	0
obs_{33}	1	1	0	0	1	0	1	0	0	0	1	0	1	0	1	1	0	0

Table 7.3: (a) Matrix CM_1 for dimension refArea of the example of Figure 1, (b) Matrix OCM for the example of Figure 1

(a)

	obs_{11}	obs_{12}	obs_{21}	obs_{22}	obs_{31}	obs_{32}	obs_{33}
obs_{11}	1	0	0	0	1	1	0
obs_{12}	0	1	0	0	0	0	0
obs_{21}	1	0	1	0	1	1	0
obs_{22}	0	0	0	1	0	0	1
obs_{31}	1	0	0	0	1	1	0
obs_{32}	1	0	0	0	1	1	0
obs_{33}	0	0	0	0	0	0	1

(b)

	obs_{11}	obs_{12}	obs_{21}	obs_{22}	obs_{31}	obs_{32}	obs_{33}
obs_{11}	1	0	0.33	0.33	1	0.66	0.33
obs_{12}	0.33	1	0.66	0.66	0.33	0.66	0.66
obs_{21}	0.66	0.33	1	0.66	0.66	1	0.66
obs_{22}	0.33	0	0.33	1	0.33	0.66	0.66
obs_{31}	1	0	0.33	0.33	1	0.66	0.33
obs_{32}	0.66	0	0.33	0.66	0.66	1	0.33
obs_{33}	0.33	0.33	0.66	0.33	0.33	0.33	1

Specifically, for each pair of observations, or rows in **OM**, we calculate a score that denotes containment. This score is basically a normalized indicator of how many dimensions exhibit subsumption between the (ordered) pair of observations. Given a specific dimension p_i, the $|O| \times |O|$ matrix that is computed for all pairs of observations is called the *containment matrix* for dimension p_i. To compute a containment score for a pair of observations with respect to a specific occurrence matrix $\mathbf{OM_i}$ of dimension $p_i = (L_i, \rightarrow)$, as per the definition of Section 2, we define a conditional boolean function $s_f : B^k \times B^k \rightarrow B$, where B is the boolean realm, i.e., $B = \{0, 1\}$, and B^k represents the set of bit vectors of size $k = L_i$. Assuming there is a transformation $b_v : O \rightarrow B^k$ that transforms the values of an observation to its respective bit vector for a particular dimension, then, s_f is defined as follows:

$$s_f(o_a, o_b) \mid_{\mathbf{OM_i}} = \begin{cases} 1, & \text{if } b_v(b) \subseteq b_v(a) \\ 0, & \text{otherwise} \end{cases}$$

In other words, containment exists if the bit vector of the right-hand observation is a subset of the bit vector of the left-hand observation. This can easily be computed as a logical AND operation between the bit vectors of the rows, i.e., if $b_v(b) \wedge b_v(a) = 1$, then $b_v(b) \subseteq b_v(a)$. We apply sf for o_a and o_b for dimension p_i in $\mathbf{OM_i}$. Application of this function for each dimension returns a set of $|P|$ containment matrices, $\mathbf{CM_1}, \ldots, \mathbf{CM_k}$. Adding these matrices yields the *Overall Containment Matrix* **OCM**:

$$\mathbf{OCM} = \sum_{i=1}^{k} \mathbf{CM_i}$$

The values in **OCM** are normalized between $[0, 1]$, with 0 denoting absence and 1 denoting presence of containment in all involved dimensions. This means that *full containment* between a pair of observations is derived when a cell has a value of 1, and *partial containment* when a cell has a value between 0 and 1 (non-inclusive). To assert which particular dimensions exhibit containment in a partial relationship, we examine the cells in $\mathbf{CM_i}$ being equal to 1. The occurrence of a 0 value indicates that full containment and complementarity can not hold. Note also that measure overlaps can be easily detected with a simple lookup. The construction of the **OCM** matrix is explained in Algorithm 9 *computeOCM*. We then calculate containment and complementarity using the **OCM**-based Algorithm 10 *baseline*.

Algorithm 9: *buildContainmentMatrix*

Data: An occurrence matrix **OM**, a set P of dimensions and their start indices in **OM**

Result: An *overall containment matrix* **OCM**

```
1  initialize OCM;
2  for each p_i ∈ P do
3  |   initialize CM_pi;
4  |   for each pair o_j, o_k ∈ OM_pi do
5  |   |   if o_j AND o_k == o_j then
6  |   |   |   CM_pi[j][k] ← 1;
7  |   |   else
8  |   |   end
9  |   |   OCM[j][k] ← OCM[j][k] + (CM(p_i)/|P|);
10 |   end
11 end
```

Computation of complementarity. Recalling the definition of complementarity, and specifically condition (1), we can take advantage of the reflexivity of complementarity and assert that two observations are complementary when (1) and (2) hold bi-directionally. Specifically, the existence of two equal values c_i, c_j implies that there exists a bi-directional hierarchical ancestry relationship, i.e., $c_i \succ c_j$ and $c_j \succ c_i$. In this context, if two observations are related with bi-directional full containment, then they are asserted to be complementary. Therefore, during the same process of computing containment, we can also compute the complementarity relationships. For this reason, we use **OCM** to assess whether a pair of observations exhibits full containment in both directions, i.e. $o_a \overset{f}{\succ} o_b$ and at the same time $o_b \overset{f}{\succ} o_a$. For example, in Table 3, the *obs11* and *obs31* are complementary, whereas the *obs21* and *obs32* are not complementary although exhibit full containment.

Complexity Analysis. Building the containment matrix requires n^2 iterations over the full observation vectors, where n is the number of observations in the input. Even though Algorithm 9 iterates over observations $|\mathbf{P}|$ times, one for each dimension, the input for each iteration is a subset of the observation, as we take into account only the dimension values for the particular dimension. Hence, assuming that the size of a bit vector is b for all $|\mathbf{P}|$ dimensions, and $b = \sum_{i=1}^{|\mathbf{P}|} b_i$, where b_i is the size of the bit vector for dimension P_i, then the number of checks with respect to the bit vector size for dimension P_i is $b_i n^2$. Adding these for all dimensions, $\sum_{i=1}^{|\mathbf{P}|} b_i n^2 = b_1 n^2 + \cdots + b_{|\mathbf{P}|} n^2 = n^2 \sum_{i=1}^{|\mathbf{P}|} b_i = n^2 b$. Therefore, for fixed vectors of size b, the total complexity of this step is $O(n^2)$ for n observations. Then, we iterate once again all of the pairs of observations in Algorithm 10. Therefore, the total iterations required by the baseline approach are $2n^2$, with an asymptotic time complexity of $O(n^2)$.

The *baseline* algorithm operates by applying all possible pair-wise comparisons between observations in the input datasets. Thus, given n observations, the complexity is $O(n^2)$. However, during the iteration of the set of **CM** matrices, if a 0 is found at any point, we can skip further computation of full containment and complementarity, because the pair under comparison is no longer candidate for these relationships, per their definitions.

Storage-wise, **OM** needs $n \times |P|$ space for n observations and $|P|$ dimension properties in the input, following a multi-dimensional array approach. However, in our implementation, a sparse matrix implementation is adopted in order to reduce the space complexity.

7.4.2 Computation with Clustering

The baseline approach requires n^2 comparisons and thus quickly becomes inefficient for large datasets as it fails to scale as a result of this complexity. The first proposed alternative method aims at improving performance by reducing the search space and executing fewer comparisons between observations. It is based on pre-clustering the input observations based on their distances in the multidimensional space, and limiting the comparisons between observations that belong to the same cluster. This approach is shown in Algorithm 11. The occurrence matrix **OM** is the input of the algorithm, and all rows are clustered into smaller occurrence matrices (Line 1). Then, the algorithm iterates through each of these clusters and applies the *buildContainmentMatrix* and *baseline* algorithms to each separate cluster (Lines 3-5). At each step of the iteration, the return arrays are updated to include the newly retrieved relationships (Line 6).

Notes on the Clustering Step. In our experiments, we employed three clustering algorithms, namely k/x-means [PM+00], agglomerative clustering and fast canopy clus-

Algorithm 10: *baseline*

Data: An overall containment matrix **OCM**.

Result: S_F, S_p, S_c sets of full, partial containment and complementarity relationships, and a map of partial containment relationships map_P with the dimensions they exhibit containment in.

```
initialize S_F, S_p, S_c;
for each pair o_j, o_k ∈ OCM do
    if OCM[i][j] == 1 then
        S_F ← S_F ∪ (o_i, o_j);
        if OCM[j][i] == 1 then
            S_C = S_C ∪ (o_i, o_j);
    else if OCM[i][j] > 0 then
        S_P = S_P ∪ (o_i, o_j);
        for each p_i ∈ P do
            if CM_pi[i][j] == 1 then
                map_P(o_i, o_j, p_i) = true
        end
    else
        continue;
    end
end
```

tering [MNU00]. The input for the distance function of the clustering step is a vector with the dimension values of the row. While more features such as other semantic and RDF metadata can be taken into account, previous related work [BRV11] has shown that simple hierarchical distances of the values of the hierarchy are adequate to characterize the distance between dimension values. It is out of scope to find the optimal clustering approach for the computation of the relationships, as finding the optimal clustering parameterization or a close approximation is a non-trivial task. More sophisticated clustering approaches can be employed, however we base our selection on evaluating our approach on three representative clustering algorithms, a centroid-based (k/x-means, a hierarchical (agglomerative) and a fast pre-clustering approach (fast canopy). In order to optimize the pre-processing step of creating the clusters and assigning points to them, we first cluster a small sample of the data (in our experiments 10% of the input size), then we assign the rest of the input to the created clusters.

Complexity Analysis. Time and space complexity of the clustering step depends on the complexity of the chosen clustering algorithm, the number of clusters and the distribution of observations in the clusters. The baseline algorithm will run times equal to the number k of clusters. However, the distribution of observations in clusters is not known for a given collection of datasets. In the centroid-based case (canopy, k/x-means), assuming an equal

Algorithm 11: *baselineWithClustering*

Data: An occurrence matrix **OM**

Result: S_F, S_p, S_c sets for fully, partial containment and complementarity relationships.

```
clusters ← cluster(OM);
initialize OCM;
for i = 1 to clusters.size do
    OCM_i ← buildContainmentMatrix(clusters[i], P);
    S_Fi, S_Pi, S_Ci ← baseline(OCM_i);
    S_F, S_P, S_C ← (S_F, S_P S_C) ∪ (S_Fi, S_Pi, S_Ci);
end
return S_F, S_P, S_C;
```

distribution of $\frac{n}{k}$ observations per cluster, then the time complexity for each cluster is $\Theta(\frac{n}{k})^2$ thus making the total time complexity $\Theta(\frac{n^2}{k})$. Following a rule of thumb where $k = \sqrt{\frac{n}{2}}$, this becomes $\Theta(n^{1.5})$, at the cost of information loss, as will be shown in the experiments. This does not, however, account for the complexity of the actual clustering step, which in general is a hard problem of at least quadratic nature (e.g., hierarchical clustering requires $n^2 logn$ steps, while k-means can be solved in n^{dk} steps when the number of dimensions d and the number of centroids k are fixed).

7.4.3 Computation with Cube Masking

In this section, we present an alternative pre-processing method that enables flexible processing and identification of the containment and complementarity relationships in the data. The method is based on the notion of *cube masks*, which are structures that represent a fixed *level* instantiation of all the dimensions, derived from the observations in $|D|$, and the cube lattice, which represents the cube masks and their interrelationships into a graph lattice. Following, we provide the definition of these notions.

Definition 22 ***Cube Masks.***

Given a globally fixed dimension ordering, a cube mask c_i is a tuple $c_i = (l_{p_1}, l_{p_2}, \ldots, l_{p_n})$, where $p_1 \ldots p_n$ are dimensions in P, and l_{p_k} is an integer denoting the level of dimension p_k as defined in c_i. Note that P is an ordered set, and the set of all cube masks in a dataset D is denoted with C_D. In order to derive cube masks from a given dataset D, we use a function $l : C \rightarrow \mathbb{N}$ that maps codelist values to the hierarchy level they belong, and a function $mask : O \rightarrow C_D$ that maps an observation to a specific cube mask. Given the above, the mask for an observation o_i is given as:

$$mask(o_i) = (l(h_1^i), l(h_2^i), \ldots, l(h_k^i)) \tag{7.6}$$

A cube mask can be used as a container structure that holds references to all the observations that exhibit this level signature. In this sense, a cube mask container $||c_i||$ can be defined as the set of all observations for which the evaluation of the *mask* function is equal to c_i, i.e.:

$$||c_i|| = \bigcup_{i=1}^{n} o_i, mask(o_i) = c_i \tag{7.7}$$

Each observation is assigned to exactly one cube mask. This means that for a given dataset D with k cube masks, the set of all observations O in D is given as the union of all cube mask containers $||c_i||$, i.e., $O = \bigcup_{i=1}^{k} ||c_i||$.

Definition 23 ***Cube Lattice.***

Given a dataset D and set of cube masks C_D as defined, we can build a graph lattice [HRU96][SDN+98], where each cube mask is a node, and each edge between two nodes denotes a direct subsumption relationship between the two nodes. In this sense, an edge in the lattice represents a difference of exactly one level in exactly one dimension between the two cube mask nodes. Formally, a cube lattice is a graph $L = (V, E)$ where $V \in C_D$ and $E \in (V \times V)$. Specifically, a directed edge between two nodes exists in L, when the two nodes exhibit pair-wise subsumption in exactly one dimension, with all other dimension levels being equal, i.e. the following is true:

$$\begin{aligned} E(c_i, c_j) \in L \iff \exists p_k \in P : l^i_{p_k} = l^j_{p_k} + 1, \\ \forall p_m \neq p_k : l^i_{p_m} = l^j_{p_m} \end{aligned} \tag{7.8}$$

Furthermore, given two cube masks c_i and c_j, the relationship $c_i \succ_{cube} c_j$ is used to denote that for all dimensions in P, c_i is defined in a level that is the same or higher than c_j, and c_i is thus a hierarchical ancestor of c_j. Formally, this means that there exists a directed path in L between c_i and c_j, or that there exists a sequence of vertices $path_{ij} = (c_i = v_1, v_2, \ldots, v_k = c_j)$ such that $(v_m, v_{m+1}) \in E$ for $1 \leqslant m < k$. This further entails that each directed pair of nodes from $path_{ij}$ exhibits pair-wise subsumption, i.e. given a path $path_{ij} = (c_i = v_1, v_2, \ldots, v_k = c_j)$, then $\forall m, n \in [1 \ldots k], m < n \rightarrow v_m \succ_{cube} v_n$.

An example lattice for the three hierarchies of Figure 7.1 can be seen in Figure 7.4. Using the lattice, we can immediately prune out comparisons between observations that belong in cube masks that are not hierarchically related. For instance, in the lattice of the figure, it is unnecessary to compare $||c_{020}||$ with $||c_{310}||$ for full containment, because the relationship $c_{310} \succ_{cube} c_{020}$ is not satisfied, i.e., while the cube with signature 310 is defined on a higher level for dimensions *refArea* and *sex* ($3 \geqslant 0$ and $0 \geqslant 0$ respectively), the same is not true for dimension *refPeriod* ($1 \not\geqslant 2$). Thus, it is guaranteed that no full containment relationships can be found between $||c_{020}||$ and $||c_{310}||$.

Lattice Creation. The multidimensional cube mask lattice L can be created with one scan on the observations of D, by applying the *mask* function on each observation and storing the hash signatures of the unique cube masks into an appropriate data structure, such as a hash map. Furthermore, during the same iteration, we can derive the set of all $||c_i||$ in D. This process can be seen in Algorithm 12. First, we initiate a single scan through all observations (Line 2), then for each observation we apply the *mask* function in order to derive the cube mask of the observation (Line 4), and finally we add the observation to *cubeMaskMap* with the found mask as key (Line 5). Then, we iterate through all the detected cubes in a nested loop and check for subsumption between the cubes (Lines 7-13).

Algorithm 12: *latticeCreation*

Result: A mapping of observations to unique cube masks, and an adjacency list with the edges between hierarchically related cubes.

```
1  initialize cubeMaskMap, lattice;
2  for each o_i ∈ O do
3  |  initialize cube;
4  |  cube ← mask(o_i);
5  |  cubeMaskMap.put(cube, cubeMaskMap.get(cube).add(o_i)) ;
6  end
7  for each c_i ∈ cubeMaskMap.keys() do
8  |  for each c_j ∈ cubeMaskMap.keys() do
9  |  |  if c_i ≻_cube c_j then
10 |  |  |  lattice.get(c_i).add(c_j)
11 |  |  end
12 |  end
13 end
14 return cubeMaskMap, lattice;
```

Baseline Computation using the Lattice. In the cases of full containment and complementarity, we do not need to compare observations that belong to lattice nodes that are not hierarchically related, such as node "121" with node "311". In the case of partial containment we look for at least one dimension inclusion (i.e. path) in the lattice before comparing the contents.

Based on these observations, we propose the *cubeMasking* algorithm (Algorithm 13). The algorithm first identifies cubes in the input datasets and populates the lattice, mapping observations to cubes (Line 2). Then, it iterates through cubes (Line 3) and does a pair-wise check for the cube containment criterion (Line 5). Finally it compares observations between pairs of cubes that fulfils this criterion (Lines 9-11). In order to perform these steps, we use a hash table to ensure that a value's level can be checked in constant

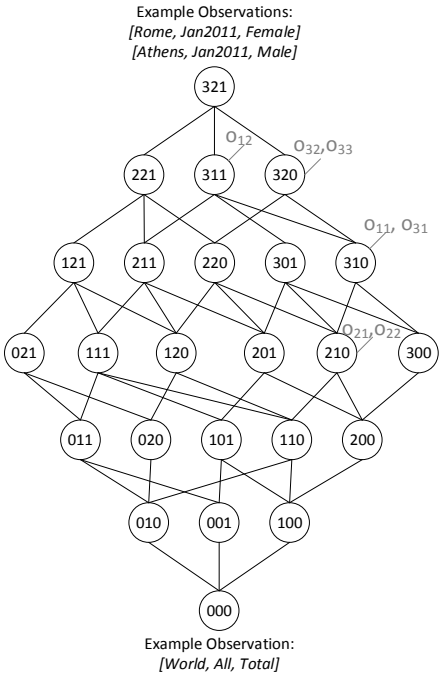


Figure 7.4: The lattice for the three hierarchies of Figure 2. Observations in Figure 1 are mapped to the appropriate node. The number in each node corresponds to the level of each dimension.

time. We then go on to identify the cubes and build the lattice by iterating through all observations and extracting their unique combinations of dimensions and levels. To do so, we apply a hash function on each observation that both identifies and populates its cube at the same step. Finally, we iterate through the identified cubes and by doing a pair-wise check for the containment and complementarity criteria, all meaningful observation comparisons are identified. This can be seen in Algorithm 13.

alysis. In this approach, only the observations between comparable cubes are compared for the candidate relationships; the multidimensional lattice ensures that the contents of cube masks that are not hierarchically related will not be iterated quadratically. Thus, in the worst case, the maximum number of cube masks is defined as the number of permutations of dimensions and levels, i.e. $k^{(|P|)}$, where k is the maximum level of all hierarchies and $|P|$ is the number of dimensions. In order to check for comparable pairs of cube masks, we need to identify if two cube masks belong in the same ancestral path. Thus, a full (directed) traversal of the lattice is required, starting from the root. The complexity is equal to the number of vertices, i.e., $k^{(|P|)}$. In the worst case, there will exist only one mask containing all observations, and all pairs of observations will have to be compared, thus still making this approach a quadratic one. However, assuming that there exist $k^{(|P|)} > 1$ vertices, with c comparable mask pairs found in the traversal,

Algorithm 13: *cubeMasking*

Data: A list C with all code list terms as they appear in the datasets, a hash table *levels* with a mapping of hierarchical values to their levels, and a list O observations

Result: S_F, S_p, S_c sets for full, partial containment and complementarity

```
initialize cubeMaskMap;
cubes, lattice ← latticeCreation();
for each pair c_i, c_j ∈ cubes do
    for each p_i ∈ P do
        if not(cube_j.p_i ≺ cube_k.p_i) then
            break
        for each o_i ∈ cube_j do
            for each o_j ∈ cube_k do
                SF[o_i, o_j] ← checkFullContainment(o_i, o_j);
                SP[o_i, o_j] ← checkPartialContainment(o_i, o_j);
                SC[o_i, o_j] ← checkComplementarity(o_i, o_j);
            end
        end
    end
end
return cubeMaskMap;
function checkFullContainment
    for each p_i ∈ P do
        if not isParent(o_i.p_i, o_j.p_i) then
            return false;
        else return true;
    end
function checkPartialContainment
    for each p_i ∈ P do
        if isParent(o_i.p_i, o_j.p_i) then
            return true;
        else return false;
    end
function checkComplementarity
    if checkFullContainment(o_i.p_i, o_j.p_i) && checkFullContainment(o_j.p_i, o_i.p_i)
     then
        return true;
    else return false;
```

and an average of $p << n$ observations per mask, then the algorithm will require cp^2 comparisons instead of n^2, with a total cost of $O(k^{(|P|)} + cp^2)$. The reason we expect $p << n$ is that we assume a distribution of all input observations in a tractable and small number of cube masks. In real world cases, the cube schema is usually defined

beforehand and populated by observations. Furthermore, the number of cube masks is restricted by the number and combinations of hierarchical levels in the input dimensions. As there are usually significantly more dimension values than hierarchy levels, we expect that all observations will be distributed to a small number of cube masks. This is also discussed in the experiments, where it can be seen in Figure 7.9 that as the number of input observations increases, the rate of new cube masks with respect to the input size decreases.

7.4.4 Optimized Cube Masking

In order to further optimize the computation of the containment and complementarity relationships, we can take advantage of the relative differences in dimensions between cube masks compared in the lattice structure that was proposed in Algorithm 13. Specifically, given two cube masks c_i and c_j, where $c_i \succ_{cube} c_j$, we can take advantage of the relative difference in dimension levels between c_i and c_j in order to define a mapping function that, given an observation $o_a \in ||c_i||$, will output the signature of the potential parent observation o_b, for which it holds that $mask(o_b) = c_j$. The requirement for this hash function is that the relative difference in dimensions and their levels between c_i and c_j is explicitly known. Formally, assume that there exists a function $L_{diff} : C_D \rightarrow C_D$ that is defined as follows:

$$L_{diff}(c_i, c_j) = (l^i_{p1} - l^j_{p1}, l^i_{p2} - l^j_{p2} \dots l^i_{pk} - l^j_{pk}) \tag{7.9}$$

where $p_1 \dots p_k$ are the dimensions, and it holds that $c_i \succ_{cube} c_j$. L_{diff} creates a *level difference mask* between the parent and child cubes, essentially capturing the level distance for each dimension p_i between c_i and c_j. The result of L_{diff} will be a level mask that represents the distances in levels between the dimensions of the two cube masks. For instance, consider cube masks c_{321} and c_{111} in Figure 7.4. Then, $L_{diff}(c_{321}, c_{111}) = (2, 1, 0)$, which means that the two cube masks have a difference of two levels in the $refArea$ dimension, a difference of one level in the $refPeriod$ dimension, and a difference of zero levels in the *sex* dimension. We denote the level difference mask between c_i and c_j as L^j_i.

With the use of the L_{diff} function, we can further define a function $H_{diff} : O \rightarrow O$, that, given an observation o_a and a level difference mask L^j_i, i.e., the output of an instance of L_{diff}, H_{diff} will output the potential observations that are parents of o_a and also conform to the parent cube mask of the L_{diff} function, i.e., c_i. This function is defined as follows:

$$H_{diff}(o_a)|_{L_{diff}(c_i,c_j)} = \mathbf{O_a} \tag{7.10}$$

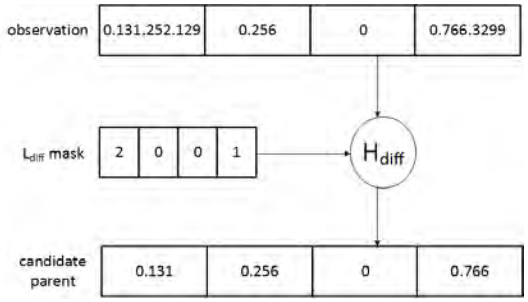


Figure 7.5: The lattice for the three hierarchies of Figure 2. Observations in Figure 1 are mapped to the appropriate node. The number in each node corresponds to the level of each dimension.

where $\mathbf{O_a} = \bigcup_{i=1}^{m} o_i$ such that for all i, $o_i \overset{f}{\succ} o_a$, and $mask(o_i) = c_j$, or $o_i \in ||c_j||$. In other words, H_{diff} outputs a set $\mathbf{O_a}$ that contains all potential observations that are parents of the input observation o_a with respect to the given L_{diff} mask. An example of the application of H_{diff} can be seen in Figure 7.5. In the top of the figure the input observation is represented as an array of dimension value signatures, pertaining to the fixed dimension ordering. The representation of the values follows the Dewey Decimal System, with parent and child values separated by dots. In the middle of the figure, the L_{diff} mask is $(2, 0, 0, 1)$. The mask and the observation instantiate the H_{diff} function which in turn outputs a candidate parent for the input observation, where each value differs in the number of levels defined by L_{diff}.

7.4.5 Computation of Full Containment and Complementarity

Thus, the steps that we then follow in order to derive full containment and complementarity are as follows:

1. Find next comparable pair of cube masks c_i, c_j from the lattice
2. Compute L_i^j by applying $L_{diff}(c_i, c_j)$
3. For each observation o_a in $||c_j||$, apply $H_{diff}(o_a)|_{L_i^j}$
4. Check for existence of the output of $H_{diff}(o_a)|_{L_i^j}$ in $||c_i||$

If step 4 is successful, there is a full containment relationship between c_i, c_j. These steps are described in detail in Algorithm 14, which also includes computation of complementarity. Lines 2-4 define an iteration of all nodes in the lattice. Starting from each node, we traverse the node's children by using simple recursive pre-order traversal. In Lines 6-19 the recursive function is defined. The condition for termination is that a node does not have any other children (Lines 7-8). The algorithm first iterates through the node

where the traversal initiated and each of the node's children (Line 9). For each pair of observations, the L_{diff} masking function is applied (Line 10). Then, the algorithm iterates through the contents of the parent node (Line 11) and checks for complementary observations in the child node (Line 11-13). Then, the algorithm applies the H_{diff} function on the observations of the child cube mask (Line 14) and checks if the candidate parent (i.e., the result of H_{diff} is contained in the parent cube mask (Lines 14-17). Finally, the recursion continues in the child cube mask (Line 19).

Algorithm 14: *optimizedCubeMasking*

Data: A map *cubes* containing cube masks and their links in the lattice
Result: S_F, S_p, S_c sets for full containment and complementarity

```
1  for each  c_i ∈ cubes do
2  |  traverse(c_i);
3  end
4  function traverse(c_i)
5  |  if cubes.getChildren(c_i) == ∅ then
6  |  |  return;
7  |  for each c_child ∈ cubes.getChildren(c_i) do
8  |  |  L_i^child ← L_diff(c_i, c_child);
9  |  |  for each o_i ∈ ||c_child|| do
10 |  |  |  if o_i ∈ ||c_i|| then
11 |  |  |  |  SC[o_i, o_j] ← 1;
12 |  |  |  candidate_parent ← H_diff(o_a)|_{L_i^child};
13 |  |  |  if candidate_parent ∈ ||c_i|| then
14 |  |  |  |  o_j ← candidate_parent;
15 |  |  |  |  SF[o_i, o_j] ← 1;
16 |  |  end
17 |  |  traverse(c_child);
18 |  end
```

In order for the computation of H_{diff} to be both feasible and efficient, we need fast access to the parents of every codelist value. For this reason, we adopt a representation for observations that can capture, with small overhead, all parents of a given hierarchical value up to the root. Under this scheme, the parents of each value are encoded within the signature of the value. For example, the value *Greece* can be represented as *All.Europe.Greece*, essentially resembling the Dewey Decimal System. Then, with simple operations we can get the parent of the value that conforms to the defined level difference.

For instance, consider the case where we are comparing cube masks c_{121} and c_{100}, and we want to apply H_{diff} on an observation $o_a \in c_{121}$, with dimension values *(All.2011, All.Europe.Greece, All.Male)*, using $L_{c100}^{c121} = (0, 2, 1)$. The value of the first dimension ($refPeriod$) stays the same, i.e., 2011, as the level difference is 0. However, the values

for dimensions $refArea$ and sex will both become All, because the level differences are 2 and 1 respectively, which are both the distances of $Greece$ and $Male$ from All respectively. Eventually, the result of H_{diff} in this case is an observation $o_a^{'} = (2011, All, All)$. By definition, the $mask$ of $o_a^{'}$ is c_{100}, however, it is not guaranteed that $o_a^{'}$ exists in the dataset. If $o_a^{'} \in ||c_{100}||$, we can derive that $o_a^{'}$ fully contains o_a, and we can mark the relationship as computed.

Complexity Analysis. With this optimization, the extra space overhead required for encoding parent values into the signature of every value in the hierarchies is traded off for a significant decrease in the required comparisons at the observation level. Following the analysis of the time complexity of the simple *cubeMasking* algorithm presented in the previous section, we assume a total of $k^{(|P|)} > 1$ cube masks. The optimized masking algorithm will require $k^{(2|P|)}$ comparisons of cube masks if all cube masks are candidates for containment, and assuming that the average amount of observations in a cube mask is $p << n$, then an iteration of all the observations in one of the two compared masks is required, thus making the total cost $k^{(2|P|)} + p$. Thus, assuming that the average number of observations is asymptotically larger than the average number of cube masks, the asymptotic cost of the optimized method is $O(p)$ for the computation of full containment and complementarity, based on these assumptions. In other words, the optimized algorithm takes advantage of the knowledge of dimension level differences in a pair of comparable cube masks in order to directly retrieve the potential parents of the existing observations, thus eliminating the need for quadratic comparisons between observations of a pair of cube masks. It should be noted, however, that in the worst case, $O(n^2)$ comparisons will still be needed if only one cube masks exists, containing all n observations.

7.4.6 Computation of Partial Containment

In the case of partial containment, we cannot apply directly the same steps as in full containment, because partial containment does not require the existence of hierarchical subsumption between the values of all dimensions of two observations, but instead, a non-empty subset of these (condition 3). There are two main problems that we have to overcome in the computation of partial containment using the optimized cube masking approach. First, given a set of dimensions $\mathbf{P} = \{P_1, P_2, \ldots, P_n\}$ in a cube schema, only a subset $\mathbf{P}^{'} \subset \mathbf{P}$ with $\mathbf{P}^{'} \neq \emptyset$ will exhibit this subsumption between two cube masks. The set $\mathbf{P}^{'}$ cannot be determined a priori given the lattice structure, but only when the cube masks containing the observations are compared, i.e., all combinations of cube masks must be checked, which makes this a costly procedure. Second, the values of the dimensions which lie in $\mathbf{P} \setminus \mathbf{P}^{'}$ are not expected to exhibit hierarchical relationships between

the compared observations. Thus, H_{diff} cannot be applied in the form of Definition 9 for detecting partial containment.On the contrary, we must first make two cube masks comparable by identifying the dimensions that are candidates for containment, and generalize to the root value the values of the remaining dimensions.

To address these issues, we propose an alternative optimization based on the optimized cube masking approach. Given two cube masks c_i, c_j, after computing $L_{diff}(c_i, c_j)$, we isolate the *positive values* in the resulting mask, as these denote the dimensions that exhibit hierarchical subsumption between the pair of cube masks, and generalize the rest of the dimension values to h_{root}. For example, given cube masks c_{321}, c_{222}, then $L_{diff}(c_{321}, c_{222}) = (1, 0, -1)$; the only positive value is the value of the first dimension. The last two dimensions must be generalized with the root value in order for the observations in the two masks to become comparable. For this, we refine the definition of H_{diff} by introducing a variant, named $H^{'}_{diff}$, defined as follows:

$$H^{'}_{diff}(o_a, \mathbf{P}^{'})|_{L_{diff}(c_i,c_j)} = \mathbf{O}^{'}_{\mathbf{a}} \tag{7.11}$$

where $\mathbf{O}^{'}_{\mathbf{a}} = \bigcup_{i=1}^{m} o_i$ such that for all i, it holds that (i) $o_i \overset{p}{\succ} o_a$ and (ii) $h^j_i = h_{root}$, where $p_j \in \mathbf{P} \setminus \mathbf{P}^{'}$. Note that, because of the generalization of the values of the dimensions in $\mathbf{P} \setminus \mathbf{P}^{'}$, the contents of $\mathbf{O}^{'}_{\mathbf{a}}$ are observations that do not exist in the c_i cube mask, but basically represent candidate parents of o_a with respect to the dimensions in $P^{'}$.

For computing partial containment, we scan both $||c_i||$ and $||c_j||$ once. For each observation in $||c_i||$, we create a set of candidate parents by generalizing the dimensions in $\mathbf{P} \setminus \mathbf{P}^{'}$, and for each observation in $||c_j||$, we calculate $H^{'}_{diff}$ on $\mathbf{P}^{'}$, generalizing as well the values of $\mathbf{P} \setminus \mathbf{P}^{'}$. This procedure outputs a set of candidates in each one of the cube mask containers. Finally, we check for candidates that exist in both of these sets. In brief, the steps for computing partial containment are as follows:

1. Iterate through *all* pairs of cube masks c_i, c_j from the lattice
2. Compute L^j_i by applying $L_{diff}(c_i, c_j)$
3. Derive $\mathbf{P}^{'}$ by isolating the positive values in L^j_i
4. For each observation o_a in $||c_i||$, generalize all values of the dimensions that are not in $\mathbf{P}^{'}$ to the h_{root} value, and store the result in set $\mathbf{O}^{'}_{\mathbf{a}}$
5. For each observation o_b in $||c_j||$, apply $H^{'}_{diff}(o_b, \mathbf{P}^{'})|_{L^j_i}$ and store the result in set $\mathbf{O}^{'}_{\mathbf{b}}$
6. For each entry $o^{'}_b$ in $\mathbf{O}^{'}_{\mathbf{b}}$, check for existence in $\mathbf{O}^{'}_{\mathbf{a}}$

These steps are shown in Algorithm 15. The algorithm iterates through all possible pairs of cube masks (Lines 1-2), as opposed to the case of full containment, where only child cube masks are traversed. Then, L_{diff} is computed (Line 3) and consequently the set of containment dimensions $P^{'}$ is derived (Lines 4-7). The algorithm iterates through the container of the outer cube mask (Line 9), and for each observation, it generalizes the dimension values to the h_{root} value (Line 10), mapping the newly created observation to the original one (Line 11). Next, the algorithm iterates through the contents of the inner cube mask (Line 13) and applies $H^{'}_{diff}$ on the contained observations (Line 14). Each created observation is then checked for existence in the candidates of the outer set, marking the relationship as partial containment in case of success (Line 15-17).

Algorithm 15: *optimizedCubeMaskingPartial*

Data: A map *cubes* containing cube masks and their links in the lattice
Result: S_p set for partial containment

```
1  for each c_i ∈ cubes do
2      for each c_j ∈ cubes do
3          L_i^j ← L_diff(c_i, c_j);
4          for each l_p ∈ L_i^j do
5              if l_p > 0 then
6                  P' ← p;
7          end
8          initialize candidate_set_i;
9          for each o_a ∈ ||c_i|| do
10             o_a' ← generalize(o_a, P \ P');
11             candidate_set_i.put(o_a', o_a);
12         end
13         for each o_b ∈ ||c_j|| do
14             o_b' ← H'_diff(o_b, P');
15             if candidate_set_i.contains(o_b') then
16                 o_a ← candidate_set_i.get(o_b');
17                 SP[o_a, o_b] ← 1;
18         end
19     end
20 end
```

Complexity Analysis. As in the case for full containment, the extra space overhead required for encoding the candidate sets for a given pair of cube masks is traded off for a decrease in the required comparisons at the observation level. Assuming a total of $k^{(|P|)}$ cube masks, then we need to compare all pairs of cube masks for partial containment, or $k^{(2|P|)}$. For each compared pair of cube masks, we scan both cube masks and apply the $H^{'}_{diff}$ function in their contents, which makes the total cost $k^{(2|P|)} + 2p$. Asymptotically,

this amounts to a linear time complexity of $O(p)$, which is still linear with respect to the average observation cardinality in the cube masks. As in the other cube masking approaches, however, in the worst case, there will only exist one cube mask containing the whole set of observations, which makes the complexity $O(n^2)$ for n observations. In real cases, the performance improvements are still significant and fall under the assumptions made in these analyses.

7.5 Experimental Evaluation

The goal of the experimental evaluation is to assess the performance of all the above methods in terms of time efficiency, accuracy and scalability of the proposed algorithms and evaluate our methods in comparison with inference and SPARQL query processing techniques widely used for detecting relationships in RDF data. We demonstrate that our approach achieves small execution time in commodity hardware and outperforms traditional techniques, which fail to scale up as the number of observations increase. In the following sections, we first provide the experimental setting, i.e., the datasets used for the experiments, the metrics and the setup of the experimental environment. Then we proceed with the evaluation of the proposed metrics, and we present and discuss the results.

7.5.1 Setting

We have selected seven real-world datasets on government and demographic statistics. The datasets were taken from Eurostat[4], the Eurostat Linked Data Wrapper[5], World Bank[6] and the linked-statistics.gr project [7]. Eurostat offers a wide variety of statistical data on countries of the EU region, and while it did not provide data in RDF format at the time of writing, some of the datasets contained therein are provided through the Eurostat Linked Data Wrapper in the RDF Data Cube (QB) representation format. World Bank is a rich source of statistical data for all countries of the world, and linked-statistics.gr is a project that offers data from the official Greek statistics authority, converted to RDF with the QB representation. In the case of datasets in the CSV format, we converted them to QB by adopting the approach of [SS01]. Notably, several other tools can also

[4] `http://epp.eurostat.ec.europa.eu/portal/page/portal/statistics/search_database`
[5] `http://estatwrap.ontologycentral.com/`
[6] `http://data.worldbank.org/`
[7] `http://linked-statistics.gr/`

be used for the conversion, such as CSV2RDF[8], OpenCube[9] and Open Refine[10]. In this context, the header labels of the CSV files are converted to dimensions (each assigned with a unique URI), and each row becomes an observation. The cell values are matched automatically to existing terms in the shared code lists.

The datasets consist of a total of 11 dimensions, 6 measures and more than 2,500 unique hierarchical terms, for a total of $\tilde{2}$60,000 observation records. The seven datasets cover demographic statistical data on population, unemployment, births/deaths, national economy (GDP) and internet adoption by household number, while the dimensions cover features such as geographical region, reference periods, country of citizenship, human genre (sex), level of education, and household size. The dataset details can be seen in Table 4.

We preprocessed the code lists in order to align dimension and hierarchy values across the input data space, by employing LIMES [NA11], a Linked Data interlinking tool that is commonly used for term alignment in the LOD cloud. LIMES can be configured to use restriction rules on the input (for example, candidate matches must exhibit the same *rdf:type* values), and has a customizable distance metric parameter, that can be programmed to use combined distance functions such as aggregates (maximum, average etc.) on multiple distance functions, such as cosine similarity, jaccard distance and levenshtein distance. For our experiments, we configured LIMES to match the code list terms by comparing the string URIs, and used their cosine distance in order to discover matches based on the URI suffixes.

Metrics. The main aim of the experimental evaluation was to compare the performance of all the proposed methods with respect to *execution time*. Specifically, we measure *execution time* by including the pre-processing and computation steps for the defined relationships. Furthermore, we report the total number of observations that are accessed/compared for each method. Especially for the case of the clustering approach, we are also interested in the achieved *recall* of the computed relationships, as this method is the only one that does not guarantee 100% recall. In this sense, *recall* is defined as the ratio of correctly found relationships to the number of all relationships in the datasets, as discovered by the other methods. Note that we do not consider any decrease in recall induced by the tool used in the dimension alignment step. The alignment process is performed as a pre-processing step for all algorithms, providing the same input in each case, and is thus independent from the achieved recall.

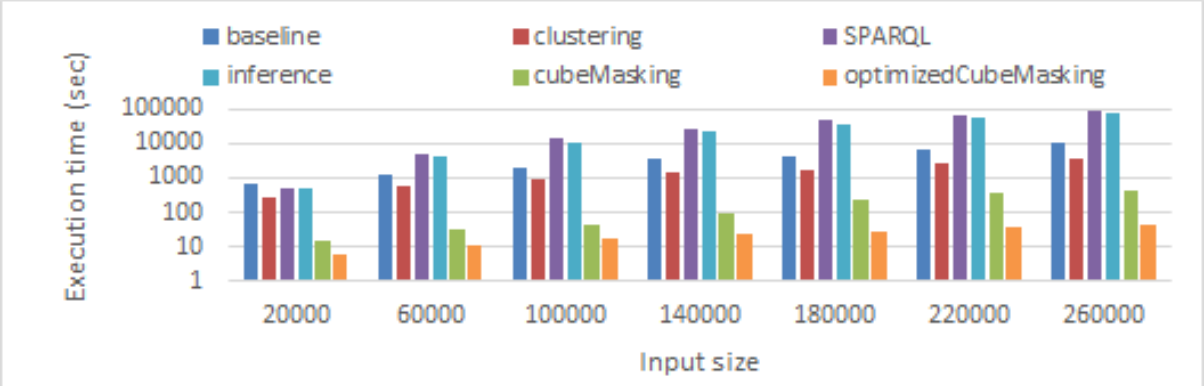
[8] `http://www.w3.org/TR/csv2rdf/`
[9] `http://opencube-toolkit.eu/`
[10] `http://refine.deri.ie/`

Table 7.4: Dataset dimensions, amount of observations and respective measures

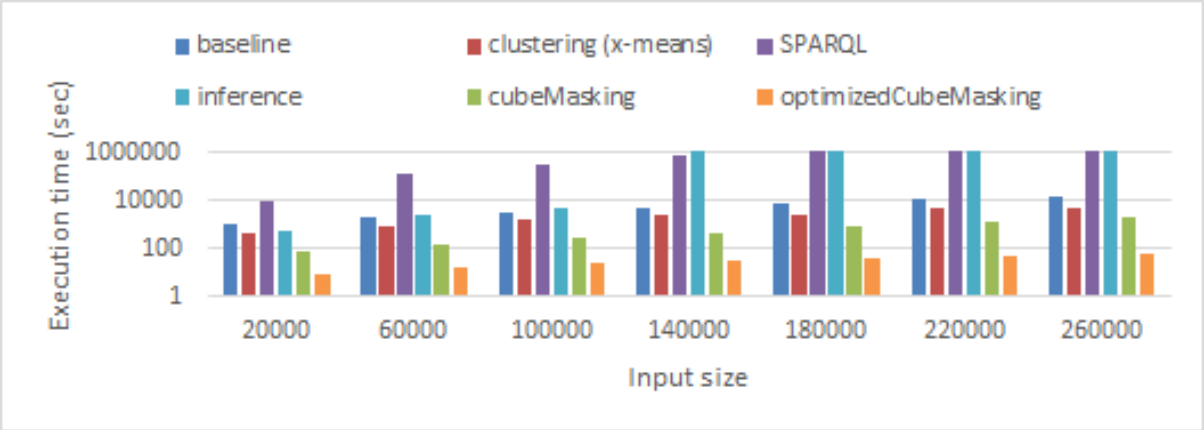
Dataset (# of obs)	refArea	refPeriod	sex	unit	age	economic activities	citizenship	education	household size	measure
D_1 (58k)	Y	Y	Y	Y	Y	N	Y	N	N	Population
D_2 (4.2k)	Y	Y	N	Y	N	N	N	N	Y	Members
D_3 (6.7k)	Y	Y	Y	Y	Y	N	N	Y	N	Population
D_4 (15k)	Y	Y	N	Y	N	N	N	N	N	Births
D_5 (68k)	Y	Y	Y	Y	Y	N	Y	N	N	Deaths
D_6 (73k)	Y	Y	N	Y	N	N	N	N	N	GDP
D_7 (21.6k)	Y	Y	N	N	N	Y	N	N	N	Compensation

Experimental Setup. We implement our approach in Java 1.8 on a single machine with allocated memory of 16GB DDR3. For the experiments, we gradually increased the input size, starting from $\tilde{2}$,000 observations and increasing it with a fixed step of 20,000 observations. For the *clustering* method, we have experimented with three clustering algorithms, namely fast canopy, x-means, and hierarchical clustering, using Jaccard similarity as a distance metric, and applying them on random 10% samples of the full input. In the series of experiments, we performed comparisons between the proposed algorithms, as well as a SPARQL-based and an inference-based alternative.

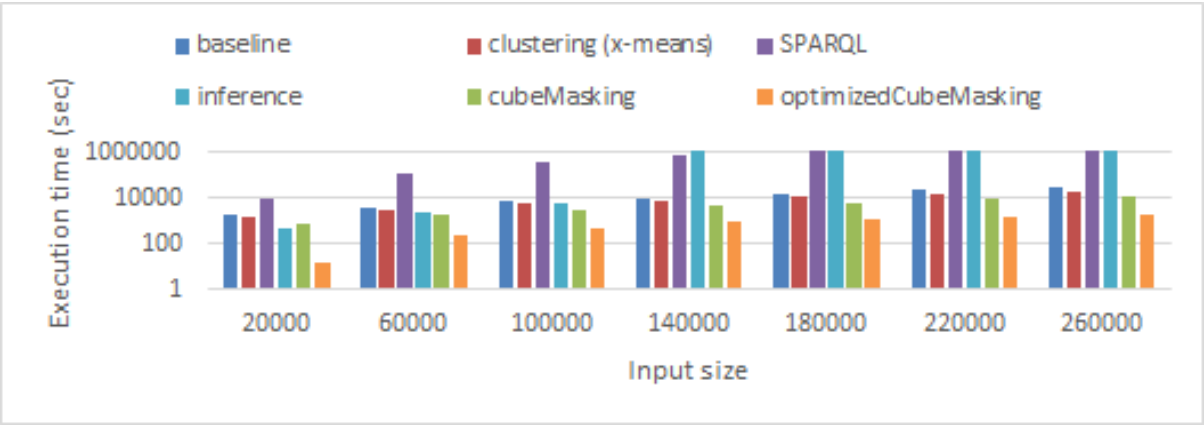
SPARQL-based approach. We consider a SPARQL-based alternative approach, for which we devised three queries for the detection of the underlying relationships of containment and complementarity. For the containment relationships, the subsumption between dimension values can occur in any level, and thus must be modelled using wildcard-enabled property paths, which are directly supported by the SPARQL 1.1 recommendation and most of its implementations. However, a requirement for the case of full containment is the universal restriction over the subsumption of the dimension values. As SPARQL does not explicitly allow universal quantification in its syntax, it has to be mimicked with the use of negation and nested recursion, which makes the query complicated to write and costly to execute. Partial containment can be easily detected, but it is complicated to derive the exact dimensions that do not exhibit containment. Given the above, and for the sake of simplicity, we design the three queries with the scope of detecting the *existence* of the relationships, and we do not quantify it like in the computation of the OCM matrix. The queries can be seen in Appendix B.



(a) Execution time (seconds) for complementarity



(b) Execution time (seconds) for full containment

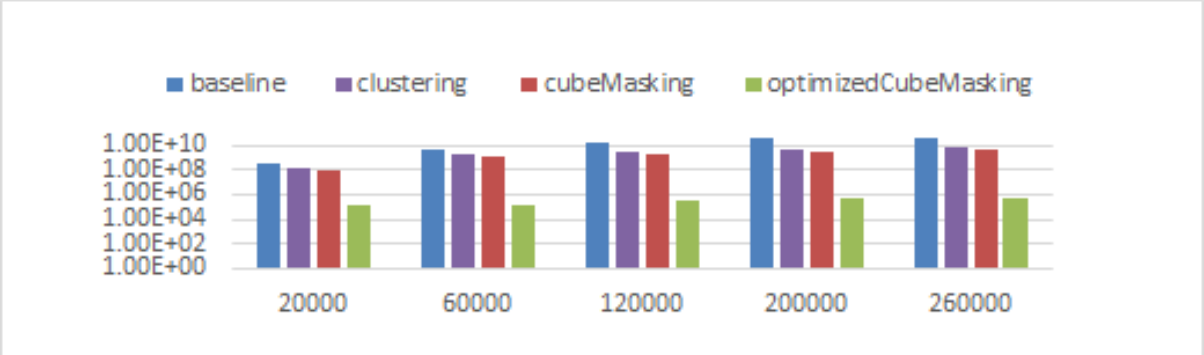


(c) Execution time (seconds) for partial containment

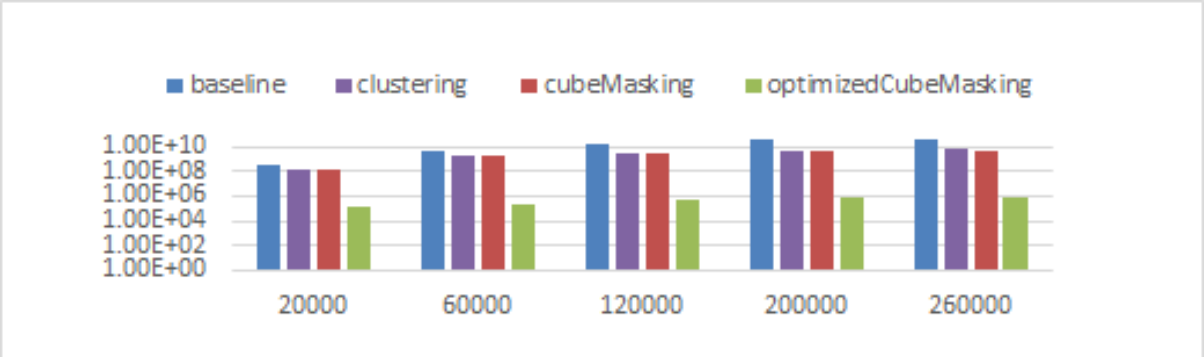
Figure 7.6: Execution performance experiments

Rule-based. The rule-based approach consists of three forward-chaining rules implemented in Jena, as the Jena generic rule reasoner is simple to use and offers the required expressiveness. The rules can be seen in Appendix C.

The datasets and code are available online at `http://github.com/mmeimaris`.



(a) Number of observation accesses for full containment



(b) Number of observation accesses for partial containment

Figure 7.7: Quantified accesses to individual observations for containment relationships

7.5.2 Experimental Results

The set of conducted experiments suggests that the baseline algorithm can be improved significantly by all three of the optimized methods. In what follows, we describe the achieved results for each of the algorithms.

7.5.2.1 Baseline

The *baseline* algorithm behaves quadratically with respect to input size, as it performs n^2 comparisons for n input observation rows. The results are shown in Figure 7.6(a-c). When computing full containment and complementarity, the required checks are decreased, because we can quickly skip pairs of rows that fail the subsumption criterion at least once. Furthermore, recall from Algorithm 2, that complementarity and full containment relationships are computed at the same pass, i.e., a bilateral full containment relationship between two observations implies their complementarity. The total number of observation pairs compared in the baseline method can be seen in Figure 7.7, and includes all possible pairs of observations in the datasets.

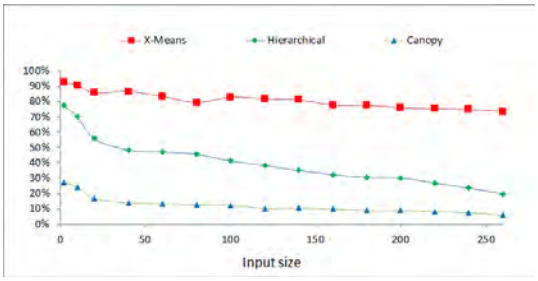


Figure 7.8: Achieved recall for the clustering approaches

7.5.2.2 Clustering

The *clustering* approach has been implemented on top of the baseline, by configuring the code to cluster the input observation rows and then perform the baseline on each cluster. In this set of experiments, we run three different settings, changing the clustering algorithm in each one. The three algorithms we have used, two centroid-based and one agglomerative, are (i) x-means, which is a variant of k-means that automatically configures the number of centroids based on the input, (ii) fast canopy clustering, and (iii) hierarchical clustering. For all approaches, we configured the system to use a sample 10% of the input size, and then assigned the remainder of the input to the detected clusters. We achieved varying degrees of recall, which can be seen in Figure 7.8. According to our results, the k-means variant achieved the higher degree of recall. In Figure 7.6(a-c) we report the execution times with x-means, compared with the other approaches. Furthermore, the total number of compared observation pairs can be seen in Figure 7.7 for the cases of full and partial containment. As a general note, the clustering approach outperformed the naive baseline algorithm, despite the eventual trade-off between runtime performance and relationship recall.

7.5.2.3 Cube Masking

The performance of the *cubeMasking* algorithm yields a substantial improvement with respect to both the baseline and the baseline with clustering. This is attributed to several factors. First, the cost of identifying the cube masks and building the lattice is linear with respect to the input size, as it requires one scan over the data. Second, the number of comparisons is significantly decreased, as comparisons are limited only between hierarchically related cube masks, while maintaining full recall. These results can be seen in Figures 7.6 and 7.7. A potential drawback of this approach is the loss of the performance advantage (i.e., the decrease of comparisons) when the number of cube masks is very large with respect to the input rows, or the distribution of the data is uneven (e.g., a small number of cube masks that cover a large percentage of the input).

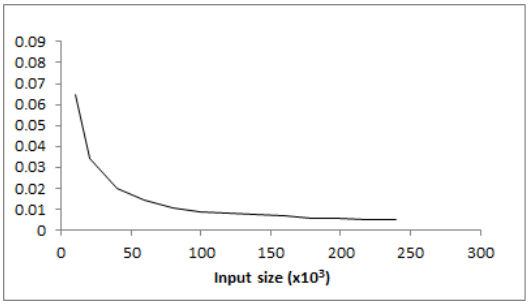


Figure 7.9: Rate of cube masks per row

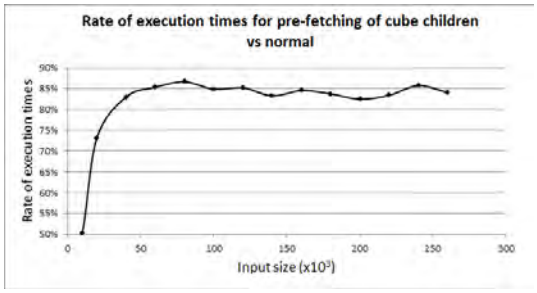


Figure 7.10: Execution rate (pre-fetching vs non-pre-fetching)

However, the rate of cube masks per input rows tends to converge logarithmically as the input size increases. This can be seen in Figure 7.9.

While *cubeMasking* operates in a similar way to the clustering method, it is more efficient because of the hierarchical relationships between the cube masks, which allow for less and more relevant comparisons between the observations. We implement the lattice as a graph data structure, so that we can have fast access to the children of each cube mask. However, we also experimented with pre-processing the lattice in order to derive the parent-child relationships and store them in memory for constant time access for each cube mask. This pre-fetching yielded a 15% improvement of the execution time, as can be seen in Figure 7.10, at the cost of an extra scan of the cube masks, and a minor imposed overhead in the storage of the system, for explicitly storing all paths between cube masks in the lattice (in our experiments, this was less than 1MB for the full dataset).

7.5.2.4 Optimized Cube Masking

The optimized cube masking algorithm yields a substantial improvement in the runtime performance for the computation of full containment, partial containment and complementarity, as can be seen in the execution times of Figure 7.6, as well as the total number of accessed observations in Figure 7.7. This is attributed to its decreased computational complexity with respect to the original cube masking approach. However, for large numbers of dimensions, the H_{diff} function becomes costlier as the observation signatures

require larger numbers of hierarchy abstractions. For the case of partial containment, the optimization exhibits a smaller relative advantage when compared to the original cube masking approach as the input size increases, as can be seen in Figure 7.6. However, it is still the fastest of all the approaches we experimented with.

The optimized cube masking approach is the fastest of all the tested approaches, as it takes advantage of exact hash signatures which can be checked in constant time. This is reflected in the experiments shown in Figure 7.6.

7.5.2.5 SPARQL and Rule-based

The runtime performance for these two alternatives is satisfactory for very small input sizes, as shown in Figure 7.6(a), (b) and (c) (less than 40k observations). However, they become intractable fast, as they either hit the time-out limits or they have vast memory requirements. This renders them non-scalable and thus not deployable in realistic settings over large real world datasets. The quality of performance is dependent on the transitive nature of the relationships, which quickly makes the search space large. The SPARQL queries timed out quickly as the number of rows increases when executed in Virtuoso (see Figure 7.6(c)). In the experiments the SPARQL method was still inadequate after the query relaxation described in Appendix A. For the rule based approach, the space overhead became large quickly, triggering several out of memory errors.

7.5.3 Scalability

We conducted a set of experiments to test the scalability of the proposed approaches, by creating a complementary synthetic dataset (x10 of the full size of the real world datasets), by following a similar approach as in [Din+11] and extending the existing data by creating observation rows that follow a projected distribution of the data with respect to the real-world datasets. More specifically, we used a number of cube masks derived from Figure 7.9 for 2.5 million observations, and populated the newly created cube masks accordingly.

As expected, the experiments show that the SPARQL-based and rule-based approaches do not scale. The results for the rest of the methods are shown in Figure 7.11, with the dotted line representing the projection of the baseline approach, as it took more than 7 days to complete.

These results show that the *clustering*, *cubeMasking* and *optimizedCubeMasking* methods are scalable for larger input sizes, with the latter two having a more clear advantage because of the reduced number of needed comparisons. It should be noted, however,

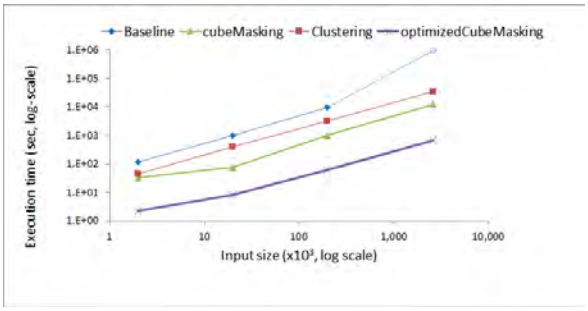


Figure 7.11: Execution time (log-log) with synthetic dataset

that in some edge cases where the input contains very large numbers of cube masks with sparse and even distributions of observations in these cube masks, the *cubeMasking* and *optimizedCubeMasking* approaches will lose their relative advantage to the quadratic baseline. Such cases justify the need for probabilistic approaches such as *clustering*, especially when runtime performance is more important than the achieved recall.

7.6 Conclusions

In this chapter, we have presented and compared four approaches for discovering three types of instance-level relationships between observations of multidimensional RDF data cubes. To this end, we have formally defined *full containment* , *partial containment*, and *complementarity* between multidimensional observations. We performed an extensive experimental evaluation between the proposed approaches and with two traditional approaches, namely a SPARQL-based and a rule-based method, and we found that our algorithms outperform the traditional approaches in both execution time and scalability. As future work, we intend to tackle incremental updates of the relationships in dynamically growing datasets. Finally, we plan to study the performance of the proposed methods in distributed and parallel contexts.

Chapter 8

Conclusions

In this thesis, we have presented novel models, methods and experimental results that focus on three directions of RDF data management and SPARQL query processing.

Specifically, the first direction dealt with management of temporal evolution in RDF databases, and to this end, a novel query language has been proposed as a solution for enabling RDF archival engines to assign temporal characteristics to SPARQL queries. Furthermore, a synthetic data generation method has been proposed in order to provide a highly customizable way of creating synthetic versioned data that evolves over time. This allows the evaluation of the capabilities of versioning and evolution management systems and archives.

The second direction dealt with the challenges that arise from the semi-structured and schema-generic nature of RDF, which gives rise to loosely-structured datasets the heterogeneity of which creates problem in the storage, indexing and query answering of modern RDF engines. In this context, novel methods for indexing, disk-based storage and query optimization were proposed and evaluated. Furthermore, methods and algorithms for efficient extraction and compaction of the underlying graph schema of RDF data have been proposed, improving the state of the art by orders of magnitude.

Finally, the third direction focused on providing algorithms for efficient and scalable mining of interrelatedness in remote, multidimensional RDF data. Several algorithms have been proposed, focusing on three particular types of relationships. The conducted experimental evaluation provides evidence that the proposed methods improve the state of the art and are scalable to large amounts of data.

Bibliography

[Aba+07] Daniel J Abadi et al. "Scalable semantic web data management using vertical partitioning". In: *VLDB*. VLDB Endowment. 2007, pp. 411–422.

[Abe+13] Alberto Abelló et al. "Fusion cubes: towards self-service business intelligence". In: (2013).

[Abe+15] Alberto Abelló et al. "Using semantic web technologies for exploratory OLAP: a survey". In: *IEEE transactions on knowledge and data engineering* 27.2 (2015), pp. 571–588.

[AH06] Sören Auer and Heinrich Herre. "A versioning and evolution framework for RDF knowledge bases". In: *International Andrei Ershov Memorial Conference on Perspectives of System Informatics*. Springer. 2006, pp. 55–69.

[AL12] Norah Alrayes and Wo-Shun Luk. "Automatic transformation of multidimensional web tables into data cubes". In: *Data Warehousing and Knowledge Discovery* (2012), pp. 81–92.

[Ali+14] Julien Aligon et al. "Similarity measures for OLAP sessions". In: *Knowledge and information systems* 39.2 (2014), pp. 463–489.

[Alu+14] Güneş Aluç et al. "Diversified stress testing of RDF data management systems". In: *ISWC*. 2014.

[Ari+11] Mario Arias et al. "An empirical study of real-world SPARQL queries". In: *arXiv preprint arXiv:1103.5043* (2011).

[Atr+10] Medha Atre et al. "Matrix Bit loaded: a scalable lightweight join query processor for RDF data". In: *WWW*. ACM. 2010, pp. 41–50.

[AY01] Charu C Aggarwal and Philip S Yu. "Outlier detection for high dimensional data". In: *ACM Sigmod Record*. Vol. 30(2). ACM. 2001, pp. 37–46.

[Ben+09] Omar Benjelloun et al. "Swoosh: a generic approach to entity resolution". In: *The VLDB JournalThe International Journal on Very Large Data Bases* 18.1 (2009), pp. 255–276.

[BEP14] Peter Boncz, Orri Erling, and Minh-Duc Pham. "Advances in Large-Scale RDF Data Management". In: *Linked Open Data–Creating Knowledge Out of Interlinked Data*. Springer, 2014, pp. 21–44.

[BG17] Sebastian Bayerl and Michael Granitzer. "Discovering, Ranking and Merging RDF Data Cubes". In: *Semantic Computing (ICSC), 2017 IEEE 11th International Conference on*. IEEE. 2017, pp. 133–140.

[Bor+13] Mihaela A Bornea et al. "Building an efficient RDF store over a relational database". In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM. 2013, pp. 121–132.

[Bro+11] Brad Brown et al. “Big data: the next frontier for innovation, competition, and productivity”. In: *McKinsey Global Institute* (2011).

[BRV11] Eftychia Baikousi, Georgios Rogkakos, and Panos Vassiliadis. “Similarity measures for multidimensional data”. In: *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*. IEEE. 2011, pp. 171–182.

[BS09] Christian Bizer and Andreas Schultz. *The berlin sparql benchmark*. 2009.

[BSK07] Abraham Bernstein, Markus Stocker, and Christoph Kiefer. “SPARQL query optimization using selectivity estimation”. In: *Poster Proceedings of the 6th International Semantic Web Conference (ISWC)*. 2007.

[BSM11] Andreas Brodt, Oliver Schiller, and Bernhard Mitschang. “Efficient resource attribute retrieval in RDF triple stores”. In: *CIKM*. ACM. 2011, pp. 1445–1454.

[BSP11] Sotiris Batsakis, Kostas Stravoskoufos, and Euripides GM Petrakis. “Temporal reasoning for supporting temporal queries in OWL 2.0”. In: *International Conference on Knowledge-Based and Intelligent Information and Engineering Systems*. Springer. 2011, pp. 558–567.

[Bun+04] Peter Buneman et al. “Archiving scientific data”. In: *ACM Transactions on Database Systems (TODS)* 29.1 (2004), pp. 2–42.

[Böh+10] Christoph Böhm et al. “Linking open government data: what journalists wish they had known”. In: *Proceedings of the 6th International Conference on Semantic Systems*. ACM. 2010, p. 34.

[Car+04] Jeremy J Carroll et al. “Jena: implementing the semantic web recommendations”. In: *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*. ACM. 2004, pp. 74–83.

[CCM15] Michelangelo Ceci, Alfredo Cuzzocrea, and Donato Malerba. “Effectively and efficiently supporting roll-up and drill-down OLAP operations over continuous dimensions via hierarchical clustering”. In: *Journal of Intelligent Information Systems* 44.3 (2015), pp. 309–333.

[CD97] Surajit Chaudhuri and Umeshwar Dayal. “An overview of data warehousing and OLAP technology”. In: *ACM Sigmod record* 26.1 (1997), pp. 65–74.

[Cha+06] Chee-Yong Chan et al. “Finding k-dominant skylines in high dimensional space”. In: *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. ACM. 2006, pp. 503–514.

[Cic+13] Paolo Ciccarese et al. “PAV ontology: provenance, authoring and versioning”. In: *Journal of biomedical semantics* 4.1 (2013), p. 37.

[Con14] World Wide Web Consortium. *Resource Description Framework (RDF)*. 2014. URL: `https://www.w3.org/RDF/` (visited on 08/02/2017).

[CRT13] Richard Cyganiak, Dave Reynolds, and Jeni Tennison. “The RDF data cube vocabulary”. In: *W3C Recommendation (January 2014)* (2013).

[Din+11] Bolin Ding et al. “Differentially private data cubes: optimizing noise sources and consistency”. In: *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. ACM. 2011, pp. 217–228.

[Don03] Francesco M Donini. “Complexity of reasoning”. In: *The description logic handbook*. Cambridge University Press. 2003, pp. 96–136.

[DS+12] Anish Das Sarma et al. “Finding related tables”. In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM. 2012, pp. 817–828.

[DT16] Grzegorz Drzadzewski and Frank Wm Tompa. “Partial materialization for online analytical processing over multi-tagged document collections”. In: *Knowledge and Information Systems* 47.3 (2016), pp. 697–732.

[Dua+11] Songyun Duan et al. “Apples and oranges: a comparison of RDF benchmarks and real RDF datasets”. In: *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. ACM. 2011, pp. 145–156.

[Eft+17] Vasilis Efthymiou et al. “Parallel meta-blocking for scaling entity resolution over big heterogeneous data”. In: *Information Systems* 65 (2017), pp. 137–157.

[EM10] Orri Erling and Ivan Mikhailov. *Virtuoso: RDF support in a native RDBMS*. Springer, 2010.

[EV12] Lorena Etcheverry and Alejandro A Vaisman. “QB4OLAP: a new vocabulary for OLAP cubes on the semantic web”. In: *Proceedings of the Third International Conference on Consuming Linked Data-Volume 905*. CEUR-WS. org. 2012, pp. 27–38.

[EV16] Lorena Etcheverry and Alejandro A Vaisman. “Querying Semantic Web Data Cubes.” In: *AMW*. 2016.

[FPU15] Javier D Fernández, Axel Polleres, and Jürgen Umbrich. “Towards Efficient Archiving of Dynamic Linked Open Data”. In: *Proceedings of the 1st DIACHRON workshop*. 2015.

[GGV12] Leticia I Gómez, Silvia A Gómez, and Alejandro A Vaisman. “A generic data model and query language for spatiotemporal OLAP cube analysis”. In: *Proceedings of the 15th International Conference on Extending Database Technology*. ACM. 2012, pp. 300–311.

[Gia+09] Arnaud Giacometti et al. “Query recommendations for OLAP discovery driven analysis”. In: *Proceedings of the ACM twelfth international workshop on Data warehousing and OLAP*. ACM. 2009, pp. 81–88.

[GN14] Andrey Gubichev and Thomas Neumann. “Exploiting the query structure for efficient join ordering in SPARQL queries.” In: *EDBT*. 2014, pp. 439–450.

[GPH05] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. “LUBM: A benchmark for OWL knowledge base systems”. In: *Web Semantics: Science, Services and Agents on the World Wide Web* 3.2 (2005), pp. 158–182.

[Gra11] Fabio Grandi. “Light-weight ontology versioning with multi-temporal RDF schema”. In: *SEMAPRO 2011, The Fifth International Conference on Advances in Semantic Processing*. 2011, pp. 42–48.

[Hal01] Alon Y Halevy. “Answering queries using views: A survey”. In: *The VLDB Journal* 10.4 (2001), pp. 270–294.

[Han+96] Jiawei Han et al. “DBMiner: A System for Mining Knowledge in Large Relational Databases.” In: *KDD*. Vol. 96. 1996, pp. 250–255.

[HBW15] Claudius Hauptmann, Michele Brocco, and Wolfgang Wörndl. “Scalable Semantic Version Control for Linked Data Management.” In: *LDQ@ ESWC*. 2015.

[HF94] Jiawei Han and Yongjian Fu. “Dynamic Generation and Refinement of Concept Hierarchies for Knowledge Discovery in Databases.” In: *KDD Workshop*. 1994, pp. 157–168.

[HL11] Kevin Chihcheng Hsu and Ming-Zhong Li. “Techniques for finding similarity knowledge in OLAP reports”. In: *Expert Systems with Applications* 38.4 (2011), pp. 3743–3756.

[Hor+05] Ian Horrocks et al. “OWL rules: A proposal and prototype implementation”. In: *Web Semantics: Science, Services and Agents on the World Wide Web* 3.1 (2005), pp. 23–40.

[HRU96] Venky Harinarayan, Anand Rajaraman, and Jeffrey D Ullman. “Implementing data cubes efficiently”. In: *ACM SIGMOD Record*. Vol. 25. 2. ACM. 1996, pp. 205–216.

[Hus+10] Mohammad Farhan Husain et al. “Data intensive query processing for large RDF graphs using cloud computing tools”. In: *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*. IEEE. 2010, pp. 1–10.

[Ibr+16] Dilshod Ibragimov et al. “Optimizing Aggregate SPARQL Queries Using Materialized RDF Views.” In: *International Semantic Web Conference (1)*. 2016, pp. 341–359.

[ILK12] Dong-Hyuk Im, Sang-Won Lee, and Hyoung-Joo Kim. “A version management framework for RDF triple stores”. In: *International Journal of Software Engineering and Knowledge Engineering* 22.01 (2012), pp. 85–106.

[JK05] Maciej Janik and Krys Kochut. “BRAHMS: a workbench RDF store and high performance memory system for semantic association discovery”. In: *ISWC*. Springer, 2005, pp. 431–445.

[Kei+11] Iman Keivanloo et al. “Towards sharing source code facts using linked data”. In: *Proceedings of the 3rd International Workshop on Search-Driven Development: Users, Infrastructure, Tools, and Evaluation*. ACM. 2011, pp. 25–28.

[KH13] Benedikt Kämpgen and Andreas Harth. “No size fits all–running the star schema benchmark with SPARQL and RDF aggregate views”. In: *The Semantic Web: Semantics and Big Data*. Springer, 2013, pp. 290–304.

[KK10] Manolis Koubarakis and Kostis Kyzirakos. “Modeling and querying metadata in the semantic sensor web: The model stRDF and the query language stSPARQL”. In: *The semantic web: research and applications* (2010), pp. 425–439.

[KKB15] Elem Guzel Kalayci, Tahir Emre Kalayci, and Derya Birant. “An ant colony optimisation approach for optimising SPARQL queries by reordering triple patterns”. In: *Information Systems* 50 (2015), pp. 51–68.

[KKK10] Zoi Kaoudi, Kostis Kyzirakos, and Manolis Koubarakis. “SPARQL query optimization on top of DHTs”. In: *The Semantic Web–ISWC 2010*. Springer, 2010, pp. 418–435.

[KKK12] Kostis Kyzirakos, Manos Karpathiotakis, and Manolis Koubarakis. “Strabon: a semantic geospatial DBMS”. In: *The Semantic Web–ISWC 2012* (2012), pp. 295–311.

[Kom+16] Takahiro Komamizu et al. “H-SPOOL: A SPARQL-based ETL framework for OLAP over linked data with dimension hierarchy extraction”. In: *International Journal of Web Information Systems* 12.3 (2016), pp. 359–378.

[KRR02] Donald Kossmann, Frank Ramsak, and Steffen Rost. “Shooting stars in the sky: An online algorithm for skyline queries”. In: *Proceedings of the 28th international conference on Very Large Data Bases*. VLDB Endowment. 2002, pp. 275–286.

[KSH14] Benedikt Kämpgen, Steffen Stadtmüller, and Andreas Harth. “Querying the global cube: Integration of multidimensional datasets from the web”. In: *Knowledge Engineering and Knowledge Management*. Springer, 2014, pp. 250–265.

[Leb+13] Timothy Lebo et al. “Prov-o: The prov ontology”. In: *W3C recommendation* 30 (2013).

[Lei+15] Viktor Leis et al. “How good are query optimizers, really?” In: *Proceedings of the VLDB Endowment* 9.3 (2015), pp. 204–215.

[Let+13] Andrés Letelier et al. “Static analysis and optimization of semantic web queries”. In: *ACM Transactions on Database Systems (TODS)* 38.4 (2013), p. 25.

[Lop+10] Nuno Lopes et al. “AnQL: SPARQLing up annotated RDFS”. In: *The Semantic Web–ISWC 2010* (2010), pp. 518–533.

[Mal+10] James Malone et al. “Modeling sample variables with an Experimental Factor Ontology”. In: *Bioinformatics* 26.8 (2010), pp. 1112–1118.

[Mei+13] Marios Meimaris et al. “DIACHRON archiving structures and associated metadata”. In: *Deliverable 4.1 of project FP7* 601043 (2013).

[Mei+14] Marios Meimaris et al. “Towards a Framework for Managing Evolving Information Resources on the Data Web.” In: *PROFILES@ ESWC*. 2014.

[Mei+16a] Marios Meimaris et al. “A query language for multi-version data web archives”. In: *Expert Systems* 33.4 (2016), pp. 383–404.

[Mei+16b] Marios Meimaris et al. “Efficient Computation of Containment and Complementarity in RDF Data Cubes.” In: *EDBT*. 2016, pp. 281–292.

[Mei+17] Marios Meimaris et al. “Extended Characteristic Sets: Graph Indexing for SPARQL Query Optimization”. In: *Data Engineering (ICDE), 2017 IEEE 33rd International Conference on*. IEEE. 2017, pp. 497–508.

[Mei+18] Marios Meimaris et al. “Computational methods and optimizations for containment and complementarity in web data cubes”. In: *Information Systems* 75 (2018), pp. 56–74.

[Mei16] Marios Meimaris. “EvoGen: a Generator for Synthetic Versioned RDF”. In: *Proceedings of the Workshops of the EDBT/ICDT 2016 Joint Conference (EDBT/ICDT 2016) (EDBT/ICDT)*. (Bordeaux, France, Mar. 15, 2016). Ed. by Themis Palpanas et al. CEUR Workshop Proceedings 1558. Aachen, 2016. URL: `http://ceur-ws.org/Vol-1558/paper9.pdf`.

[Men+11] Pablo N Mendes et al. “DBpedia spotlight: shedding light on the web of documents”. In: *Proceedings of the 7th International Conference on Semantic Systems*. ACM. 2011, pp. 1–8.

[MNU00] Andrew McCallum, Kamal Nigam, and Lyle H Ungar. “Efficient clustering of high-dimensional data sets with application to reference matching”. In:

Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining. ACM. 2000, pp. 169–178.

[MP14] Marios Meimaris and George Papastefanatos. "Containment and complementarity relationships in multidimensional linked open data". In: *Second International Workshop for Semantic Statistics SemStats*. 2014.

[MP16a] Marios Meimaris and George Papastefanatos. "Double Chain-Star: an RDF indexing scheme for fast processing of SPARQL joins." In: *EDBT*. 2016, pp. 668–669.

[MP16b] Marios Meimaris and George Papastefanatos. "The EvoGen Benchmark Suite for Evolving RDF Data." In: *MEPDaW/LDQ@ ESWC*. 2016, pp. 20–35.

[MP17] Marios Meimaris and George Papastefanatos. "Distance-Based Triple Reordering for SPARQL Query Optimization". In: *Data Engineering (ICDE), 2017 IEEE 33rd International Conference on*. IEEE. 2017, pp. 1559–1562.

[MPP15] Marios Meimaris, George Papastefanatos, and Christos Pateritsas. "An Archiving System for Managing Evolution in the Data Web". In: *Proceedings of the 1st DIACHRON workshop*. 2015.

[MRB99] Volker Markl, Frank Ramsak, and Rudolf Bayer. "Improving OLAP performance by multidimensional hierarchical clustering". In: *Database Engineering and Applications, 1999. IDEAS'99. International Symposium Proceedings*. IEEE. 1999, pp. 165–177.

[MSMH17] G. Montoya, H. Skaf-Molli, and K. Hose. "The Odyssey Approach for Optimizing Federated SPARQL Queries". In: *ISWC*. 2017.

[NA11] Axel-Cyrille Ngonga Ngomo and Sören Auer. "Limes-a time-efficient approach for large-scale link discovery on the web of data". In: *integration* 15 (2011), p. 3.

[NDS00] Sandra M Nutley, Huw TO Davies, and Peter C Smith. *What works?: Evidence-based policy and practice in public services*. MIT Press, 2000.

[NM11] Thomas Neumann and Guido Moerkotte. "Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins". In: *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*. IEEE. 2011, pp. 984–994.

[NW10] Thomas Neumann and Gerhard Weikum. "The RDF-3X engine for scalable management of RDF data". In: *The VLDB Journal* 19.1 (2010), pp. 91–113.

[PAG06] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. "Semantics and Complexity of SPARQL". In: *International semantic web conference*. Springer. 2006, pp. 30–43.

[Pap+11] George Papadakis et al. "Efficient entity resolution for large heterogeneous information spaces". In: *Proceedings of the fourth ACM international conference on Web search and data mining*. ACM. 2011, pp. 535–544.

[Pap+12] Nikolaos Papailiou et al. "H2RDF: adaptive query processing on RDF data in the cloud." In: *Proceedings of the 21st international conference companion on World Wide Web*. ACM. 2012, pp. 397–400.

[Pap+13a] George Papadakis et al. "A blocking framework for entity resolution in highly heterogeneous information spaces". In: *IEEE Transactions on Knowledge and Data Engineering* 25.12 (2013), pp. 2665–2682.

[Pap+13b] Vicky Papavasileiou et al. “High-level change detection in RDF (S) KBs”. In: *ACM Transactions on Database Systems (TODS)* 38.1 (2013), p. 1.

[Pap+14] Nikolaos Papailiou et al. “H 2 RDF+: an efficient data management system for big RDF graphs”. In: *SIGMOD*. ACM. 2014, pp. 909–912.

[Pap+15] George Papadakis et al. “Schema-agnostic vs schema-based configurations for blocking methods on homogeneous data”. In: *Proceedings of the VLDB Endowment* 9.4 (2015), pp. 312–323.

[Pap+16] George Papadakis et al. “Scaling Entity Resolution to Large, Heterogeneous Data with Enhanced Meta-blocking.” In: *EDBT*. 2016, pp. 221–232.

[Pap13] George Papastefanatos. “Challenges and Opportunities in the Evolving Data web”. In: *International Conference on Conceptual Modeling*. Springer. 2013, pp. 23–28.

[PB16] Minh-Duc Pham and Peter Boncz. “Exploiting Emergent Schemas to make RDF systems more efficient”. In: *ISWC*. Springer. 2016, pp. 463–479.

[PH10] Niko P Popitsch and Bernhard Haslhofer. “DSNotify: handling broken links in the web of data”. In: *Proceedings of the 19th international conference on World wide web*. ACM. 2010, pp. 761–770.

[Pha+15] MD. Pham et al. “Deriving an emergent relational schema from rdf data”. In: *WWW*. 2015.

[PJS11] Matthew Perry, Prateek Jain, and Amit P Sheth. “Sparql-st: Extending sparql to support spatiotemporal queries”. In: *Geospatial semantics and the semantic web*. Springer, 2011, pp. 61–86.

[PM+00] Dan Pelleg, Andrew W Moore, et al. “X-means: Extending K-means with Efficient Estimation of the Number of Clusters.” In: *ICML*. 2000, pp. 727–734.

[PS+06] Eric Prud, Andy Seaborne, et al. “SPARQL query language for RDF”. In: (2006).

[PS+96] Gregory Piatetsky-Shapiro et al. “An Overview of Issues in Developing Industrial Data Mining and Knowledge Discovery Applications.” In: *KDD*. Vol. 96. 1996, pp. 89–95.

[PSG13] George Papastefanatos, Yannis Stavrakas, and Theodora Galani. “Capturing the history and change structure of evolving data”. In: *DBKDA 2013, The Fifth International Conference on Advances in Databases, Knowledge, and Data Applications*. 2013, pp. 235–241.

[RA07] Oscar Romero and Alberto Abelló. “Automating multidimensional design from ontologies”. In: *Proceedings of the ACM tenth international workshop on Data warehousing and OLAP*. ACM. 2007, pp. 1–8.

[Rah+17] Md Farhadur Rahmany et al. “Efficient Computation of Subspace Skyline over Categorical Domains”. In: *arXiv preprint arXiv:1703.00080* (2017).

[RGG15] Stefano Rizzi, Matteo Golfarelli, and Simone Graziani. “An OLAM Operator for Multi-Dimensional Shrink”. In: *International Journal of Data Warehousing and Mining (IJDWM)* 11.3 (2015), pp. 68–97.

[Rou+15] Yannis Roussakis et al. “A flexible framework for understanding the dynamics of evolving RDF datasets”. In: *The Semantic Web-ISWC 2015*. Springer, 2015, pp. 495–512.

[SCF14] Kostas Stefanidis, Ioannis Chrysakis, and Giorgos Flouris. "On designing archiving policies for evolving RDF datasets on the Web". In: *International Conference on Conceptual Modeling*. Springer. 2014, pp. 43–56.

[Sch+09] M Schmidt et al. "SP 2 Bench: A SPARQL performance benchmark, ICDE". In: *Shanghai, China* (2009).

[Sch+11] Michael Schmidt et al. "Fedbench: A benchmark suite for federated semantic data query processing". In: *The Semantic Web–ISWC 2011*. Springer, 2011, pp. 585–600.

[Sch+14] Alexander Schätzle et al. "Sempala: Interactive SPARQL query processing on hadoop". In: *International Semantic Web Conference*. Springer. 2014, pp. 164–179.

[Sch+16a] Alexander Schätzle et al. "S2RDF: RDF querying with SPARQL on spark". In: *VLDB*. 2016.

[Sch+16b] Alexander Schätzle et al. "S2RDF: RDF Querying with SPARQL on Spark". In: *Proc. VLDB Endow.* 9.10 (June 2016), pp. 804–815. ISSN: 2150-8097. DOI: 10.14778/2977797.2977806.

[SDN+98] Amit Shukla, Prasad Deshpande, Jeffrey F Naughton, et al. "Materialized view selection for multidimensional datasets". In: *VLDB*. Vol. 98. 1998, pp. 488–499.

[Sid+08] Lefteris Sidirourgos et al. "Column-store support for RDF data management: not all swans are white". In: *VLDB* 1.2 (2008), pp. 1553–1563.

[SML10] Michael Schmidt, Michael Meier, and Georg Lausen. "Foundations of SPARQL query optimization". In: *Proceedings of the 13th International Conference on Database Theory*. ACM. 2010, pp. 4–33.

[Som+09] Herbert Van de Sompel et al. "Memento: Time travel for the web". In: *arXiv preprint arXiv:0911.1112* (2009).

[Som+10] Herbert Van de Sompel et al. "An HTTP-based versioning mechanism for linked data". In: *arXiv preprint arXiv:1003.3661* (2010).

[SS01] Gayatri Sathe and Sunita Sarawagi. "Intelligent rollups in multidimensional OLAP data". In: *VLDB*. Vol. 1. 2001, pp. 531–540.

[Sto+08] Markus Stocker et al. "SPARQL basic graph pattern optimization using selectivity estimation". In: *Proceedings of the 17th international conference on World Wide Web*. ACM. 2008, pp. 595–604.

[Tam16] Efthimios Tambouris. "Multidimensional Open Government Data". In: *JeDEM-eJournal of eDemocracy and Open Government* 8.3 (2016), pp. 1–11.

[TEO+01] Kian-Lee Tan, Pin-Kwang Eng, Beng Chin Ooi, et al. "Efficient progressive skyline computation". In: *VLDB*. Vol. 1. 2001, pp. 301–310.

[Tsi+12] Petros Tsialiamanis et al. "Heuristics-based query optimisation for SPARQL". In: *EDBT*. ACM. 2012, pp. 324–335.

[UVTH10] Jürgen Umbrich, Boris Villazón-Terrazas, and Michael Hausenblas. "Dataset dynamics compendium: A comparative study". In: *Proceedings of the First International Conference on Consuming Linked Data-Volume 665*. CEUR-WS. org. 2010, pp. 49–60.

[Var+16a] Jovan Varga et al. "Dimensional enrichment of statistical linked open data". In: *Web Semantics: Science, Services and Agents on the World Wide Web* 40 (2016), pp. 22–51.

[Var+16b] Jovan Varga et al. “QB2OLAP: Enabling OLAP on statistical linked open data”. In: *Data Engineering (ICDE), 2016 IEEE 32nd International Conference on*. IEEE. 2016, pp. 1346–1349.

[VG06] Max Völkel and Tudor Groza. “SemVersion: An RDF-based ontology versioning system”. In: *Proceedings of the IADIS international conference WWW/Internet*. Vol. 2006. 2006, p. 44.

[Vol+09] Julius Volz et al. “Silk-A Link Discovery Framework for the Web of Data.” In: *LDOW* 538 (2009).

[VT+11] Boris Villazón-Terrazas et al. “Methodological guidelines for publishing government linked data”. In: *Linking government data*. Springer, 2011, pp. 27–49.

[VZ14] Alejandro Vaisman and Esteban Zimányi. *Data Warehouse Systems: Design and Implementation*. Springer, 2014.

[Wil06] Kevin Wilkinson. *Jena property table implementation*. 2006.

[WKB08] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. “Hexastore: sextuple indexing for semantic web data management”. In: *VLDB* 1.1 (2008), pp. 1008–1019.

[WL02] Raymond K Wong and Nicole Lam. “Managing and querying multi-version XML data with update logging”. In: *Proceedings of the 2002 ACM symposium on Document engineering*. ACM. 2002, pp. 74–81.

[Wu+14] Buwen Wu et al. “Semstore: A semantic-preserving distributed RDF triple store”. In: *CIKM*. ACM. 2014, pp. 509–518.

[Xie+16] Xike Xie et al. “OLAP over probabilistic data cubes I: Aggregating, materializing, and querying”. In: *Data Engineering (ICDE), 2016 IEEE 32nd International Conference on*. IEEE. 2016, pp. 799–810.

[Yua+05] Yidong Yuan et al. “Efficient computation of the skyline cube”. In: *Proceedings of the 31st international conference on Very large data bases*. VLDB Endowment. 2005, pp. 241–252.

[Yua+13] Pingpeng Yuan et al. “TripleBit: a fast and compact system for large scale RDF data”. In: *VLDB* 6.7 (2013), pp. 517–528.

[Zen+13] Kai Zeng et al. “A distributed graph engine for web scale RDF data”. In: *Proceedings of the VLDB Endowment*. Vol. 6. 4. VLDB Endowment. 2013, pp. 265–276.

Appendix A

SPARQL Queries

A.1 Lehigh University Benchmark Original Queries

In this section, the original 14 queries of the Lehigh University Benchmark (LUBM) are presented.

Query1

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl
   #>
SELECT ?X
WHERE
{ ?X rdf:type ub:GraduateStudent .
  ?X ub:takesCourse http://www.Department0.University0.edu/
     GraduateCourse0}
```

Query2

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl
   #>
SELECT ?X, ?Y, ?Z
WHERE
{?X rdf:type ub:GraduateStudent .
  ?Y rdf:type ub:University .
  ?Z rdf:type ub:Department .
  ?X ub:memberOf ?Z .
  ?Z ub:subOrganizationOf ?Y .
  ?X ub:undergraduateDegreeFrom ?Y}
```

Query3

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
```

```
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl
   #>
SELECT ?X
WHERE
{?X rdf:type ub:Publication .
  ?X ub:publicationAuthor
        http://www.Department0.University0.edu/
           AssistantProfessor0}
```

Query4

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl
   #>
SELECT ?X, ?Y1, ?Y2, ?Y3
WHERE
{?X rdf:type ub:Professor .
  ?X ub:worksFor <http://www.Department0.University0.edu> .
  ?X ub:name ?Y1 .
  ?X ub:emailAddress ?Y2 .
  ?X ub:telephone ?Y3}
```

Query5

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl
   #>
SELECT ?X
WHERE
{?X rdf:type ub:Person .
  ?X ub:memberOf <http://www.Department0.University0.edu>}
```

Query6

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl
   #>
SELECT ?X WHERE {?X rdf:type ub:Student}
```

Query7

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl
   #>
SELECT ?X, ?Y
WHERE
{?X rdf:type ub:Student .
  ?Y rdf:type ub:Course .
  ?X ub:takesCourse ?Y .
  <http://www.Department0.University0.edu/AssociateProfessor0>,
        ub:teacherOf, ?Y}
```

Query8

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl
   #>
SELECT ?X, ?Y, ?Z
WHERE
{?X rdf:type ub:Student .
  ?Y rdf:type ub:Department .
  ?X ub:memberOf ?Y .
  ?Y ub:subOrganizationOf <http://www.University0.edu> .
  ?X ub:emailAddress ?Z}
```

Query9

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl
   #>
SELECT ?X, ?Y, ?Z
WHERE
{?X rdf:type ub:Student .
  ?Y rdf:type ub:Faculty .
  ?Z rdf:type ub:Course .
  ?X ub:advisor ?Y .
  ?Y ub:teacherOf ?Z .
  ?X ub:takesCourse ?Z}
```

Query10

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl
   #>
SELECT ?X
WHERE
{?X rdf:type ub:Student .
  ?X ub:takesCourse
<http://www.Department0.University0.edu/GraduateCourse0>}
```

Query11

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl
   #>
SELECT ?X
WHERE
{?X rdf:type ub:ResearchGroup .
  ?X ub:subOrganizationOf <http://www.University0.edu>}
```

Query12

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl
   #>
SELECT ?X, ?Y
WHERE
{?X rdf:type ub:Chair .
  ?Y rdf:type ub:Department .
  ?X ub:worksFor ?Y .
  ?Y ub:subOrganizationOf <http://www.University0.edu>}
```

Query13

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl
   #>
SELECT ?X
WHERE
{?X rdf:type ub:Person .
  <http://www.University0.edu> ub:hasAlumnus ?X}
```

Query14

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl
   #>
SELECT ?X
WHERE {?X rdf:type ub:UndergraduateStudent}
```

A.2 New LUBM Queries

In this section, the newly defined LUBM queries for the experiments of Chapter 5 are presented.

Query 2.1

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://swat.cse.lehigh.edu/onto/univ-bench.owl#>
SELECT DISTINCT ?s ?y ?z ?w WHERE {
 ?s ub:researchInterest ?o2 ;
 ub:mastersDegreeFrom ?o3 ;
 ub:doctoralDegreeFrom ?o4 ;
 ub:memberOf  ?y ;
 rdf:type ?o .
 ?y rdf:type ?o5 ;
 ub:subOrganizationOf ?z .
 ?z rdf:type ?o6 ;
 ub:hasAlumnus ?w .
 ?w ub:name ?o8
}
```

Query 2.2

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://swat.cse.lehigh.edu/onto/univ-bench.owl#>
SELECT DISTINCT ?s ?y ?z ?w WHERE {
?s ub:researchInterest ?o2 ;
 ub:mastersDegreeFrom ?o3 ;
 ub:emailAddress ?o44 ;
 ub:worksFor  ?y ;
 rdf:type ub:UndergraduateStudent .
 ?y rdf:type ?o5 ;
 ub:subOrganizationOf ?z .
 ?z rdf:type ?o6 ;
 ub:hasAlumnus ?o7 .
 ?s ub:advisor ?w .
 ?w rdf:type ?o88
}
```

Query 2.3

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://swat.cse.lehigh.edu/onto/univ-bench.owl#>
SELECT DISTINCT ?s ?y ?z ?course WHERE {
?s ub:researchInterest ?o2 ;
 ub:mastersDegreeFrom ?o3 ;
 ub:emailAddress ?o44 ;
 ub:worksFor  ?y ;
 ub:teacherOf ?course ;
 rdf:type ?profType .
 ?course rdf:type ?courseType .
 ?course ub:name ?courseName .
 ?student ub:takesCourse ?course .
 ?student rdf:type ub:UndergraduateStudent .
 ?student ub:memberOf ?sm .
 ?sm rdf:type ?smType .
}
```

Query 2.4

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://swat.cse.lehigh.edu/onto/univ-bench.owl#>
SELECT DISTINCT ?s1 ?pub ?dept WHERE {
        ?s1 rdf:type ?studentType .
        ?s1 ub:undergraduateDegreeFrom ?uguni .
        ?s1 ub:worksFor ?dept .
        ?dept rdf:type ?deptType .
        ?dept ub:subOrganizationOf ?sub .
        ?pub rdf:type ?pubtype .
        ?pub ub:publicationAuthor ?s1 .
}
```

Query 2.5

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://swat.cse.lehigh.edu/onto/univ-bench.owl#>
SELECT DISTINCT ?s1 ?pub ?dept WHERE {
        ?s1 rdf:type ?studentType .
        ?s1 ub:undergraduateDegreeFrom ?uguni .
        ?s1 ub:worksFor ?dept .
        ?dept rdf:type ub:Department .
        ?dept rdf:type ?deptType .
        ?dept ub:subOrganizationOf ?sub .
        ?sub rdf:type ub:University .
        ?pub rdf:type ub:Publication .
        ?pub rdf:type ?pubtype .
        ?pub ub:publicationAuthor ?s1 .
        ?s1 ub:advisor ?teacher .
        ?teacher rdf:type ub:Professor .
        ?teacher ub:name ?tname .
        ?teacher ub:doctoralDegreeFrom ?tdf .
}
```

A.3 Reactome Queries

In this section, the queries for the Reactome dataset are presented.

Query 3.1

```
SELECT DISTINCT ?pathway ?reaction ?complex ?protein ?ref
WHERE
{       ?pathway rdf:type biopax3:Pathway .
        ?pathway biopax3:displayName ?pathwayname .
        ?pathway biopax3:pathwayComponent ?reaction .
        ?reaction rdf:type biopax3:BiochemicalReaction .
        ?reaction biopax3:right ?complex .
        ?complex rdf:type biopax3:Complex .
        ?complex biopax3:component ?protein .
        ?protein rdf:type biopax3:Protein .
        ?protein biopax3:entityReference ?ref .
        ?ref biopax3:id ?id ; rdf:type ?refType
}
```

Query 3.2

```
SELECT ?pathway ?reaction ?complex ?protein
WHERE
{       ?pathway rdf:type biopax3:Pathway .
        ?pathway biopax3:displayName ?pathwayname .
        ?pathway biopax3:pathwayComponent ?reaction .
        ?reaction rdf:type biopax3:BiochemicalReaction .
```

```
        ?reaction  ?rel  ?complex .
        ?reaction biopax3:left ?left .
        ?complex rdf:type biopax3:Complex .
        ?complex biopax3:component ?protein .
        ?protein rdf:type biopax3:Protein .
        ?protein biopax3:entityReference <http://purl.uniprot.org
           /uniprot/P01308>
}
```

Query 3.3

```
SELECT DISTINCT ?x
WHERE
{
        ?x biopax3:dataSource ?x1 .
        ?x biopax3:organism ?x2 .
        ?x biopax3:pathwayComponent ?x3 .
        ?x biopax3:comment ?x4 .
        ?x biopax3:evidence ?x5 .
        ?x5 ?p ?o .
}
```

Query 3.4

```
SELECT DISTINCT ?pathway ?reaction ?entity
WHERE
{       ?pathway rdf:type biopax3:Pathway .
        ?pathway biopax3:displayName ?pathwayname .
        ?pathway biopax3:pathwayComponent ?reaction .
        ?reaction rdf:type biopax3:BiochemicalReaction .
        ?reaction biopax3:left ?entity .
        ?entity biopax3:cellularLocation <http://purl.obolibrary.
           org/obo/GO_0005886> .
        ?pathway biopax3:dataSource ?source .
        ?source biopax3:name ?sourceName .
}
```

Query 3.5

```
SELECT DISTINCT ?pathway ?organism ?ref
WHERE
{
        ?pathway rdf:type biopax3:Pathway .
        ?pathway biopax3:displayName ?pathwayname .
        ?pathway biopax3:organism ?organism .
        ?organism biopax3:name ?organismName .
        ?organism rdf:type ?orgType.
        ?organism biopax3:xref ?ref .
        ?ref biopax3:id ?id ;
                 rdf:type ?refType
```

```
}
```

Query 3.6

```
SELECT DISTINCT ?pathway ?organism
WHERE
{
        ?pathway rdf:type ?pathType .
        ?pathway biopax3:organism ?organism .
        ?organism biopax3:name ?organismName .
        ?organism rdf:type ?orgType.
        ?organism biopax3:xref ?ref .
        ?ref biopax3:id ?id ;
                 rdf:type ?refType
}
```

Query 3.7

```
SELECT DISTINCT ?organism ?ref
WHERE
{
        ?organism biopax3:name ?organismName .
        ?organism rdf:type ?orgType.
        ?organism biopax3:xref ?ref .
        ?ref biopax3:id ?id ;
                rdf:type ?refType
}
```

Query 3.8

```
SELECT DISTINCT *
WHERE
{
        ?x biopax3:pathwayComponent ?pw1 .
        ?pw1 rdf:type ?type1 .
        ?pw1 biopax3:pathwayComponent ?pw2 .
        ?pw2 rdf:type ?type2 .
        ?pw2 biopax3:pathwayComponent ?pw3 .
        ?pw3 rdf:type ?type3 .
        ?pw3 biopax3:pathwayComponent ?pw4 .
        ?pw4 rdf:type ?type4 .
        ?pw2 biopax3:pathwayComponent ?pw5 .
        ?pw5 rdf:type ?type5 .
}
```

A.4 Geonames Queries

In this section, the queries for the Geonames dataset are presented.

Query 4.1

```
SELECT ?f1 ?f2 ?f3 WHERE
{
        ?f1 rdf:type ?ft1 ;
                gn:parentFeature ?f2 ;
                gn:postalCode ?fp1 .
        ?f2 rdf:type ?ft2 ;
                gn:parentFeature ?f3 .
        ?f3 rdf:type ?ft3 .
}
```

Query 4.2

```
SELECT ?f1 ?f2 ?f3 WHERE
{
        ?f1 rdf:type ?ft1 ;
                gn:parentFeature ?f2 ;
                gn:postalCode ?fp1 .
        ?f2 rdf:type ?ft2 ;
                gn:parentFeature ?f3 .
        ?f3 rdf:type ?ft3 .
        ?f1 gn:parentCountry ?f3 .
}
```

Query 4.3

```
SELECT ?f1 ?f2 ?f3 ?4 WHERE
{
        ?f1 rdf:type ?ft1 ;
                gn:parentFeature ?f2 ;
                gn:postalCode ?fp1 ;
                gn:parentADM4 ?fadm1 .
        ?f2 rdf:type ?ft2 ;
                gn:parentFeature ?f3 .
        ?f2 gn:parentADM3 ?fadm2 .
        ?f3 rdf:type ?ft3 ;
                gn:wikipediaArticle ?fwiki3 .
        ?f3 gn:parentFeature ?f4 .
        ?f4 rdf:type ?ft4 .
}
```

Query 4.4

```
SELECT ?f1 ?f2 ?f3 ?4 WHERE
{
        ?f1 rdf:type ?ft1 ;
                gn:parentFeature ?f2 ;
                gn:postalCode ?fp1 .
        ?f2 rdf:type ?ft2 ;
                gn:parentFeature ?f3 .
```

```
        ?f3 rdf:type ?ft3 .
        ?f1 gn:parentFeature ?f3 .
        ?f3 gn:parentFeature ?f4 .
        ?f4 rdf:type ?ft4 .
}
```

Query 4.5

```
SELECT ?f1 ?f2 ?f3 ?adm1 ?fadm1 WHERE {
        ?f1 rdf:type gn:Feature ;
                gn:parentFeature ?f2 ;
                gn:postalCode ?fp1 .
        ?f2 rdf:type ?ft2 ;
                gn:parentFeature ?f3 .
        ?f3 rdf:type ?ft3 .
        ?f1 gn:parentADM1 ?adm1 .
        ?adm1 ?p ?fadm1 .
        ?fadm1 a gn:Feature
}
```

Query 4.6

```
SELECT ?f1 ?adm1 WHERE {
        ?f1 rdf:type gn:Feature ;
                gn:parentADM1 ?adm1 .
        ?adm1 gn:parentCountry ?fadm1 .
}
```

Appendix B

Notes on the SPARQL Approach

Notes on the SPARQL-based approach. As it has been argued in this thesis, property paths are directly supported by SPARQL 1.1 and are necessary for computing whether two values are related hierarchically. A different alternative is to compute the tran-sitive closure of the data and materialize these relationships, however we do not ad-dress efficient materialization of transitivity in RDF datasets. Universal quantification must be mimicked by using a negation construct that in-cludes a nested recursion. This negation actually ensures that there is no dimension between two candidate observations that does not exhibit hierarchically related val-ues. This is useful for computing full containment. On the other hand, partial con-tainment can be detected by SPARQL ASK queries merely by checking whether at least one occurrence of hierarchically related values is present in any of the shared dimensions between two observations.

In the case of *partial containment*, the queries for materializing and detecting pairs of observations are as follows:

Partial Containment (materialization):

```
CONSTRUCT {
        [
  rdf:type imis:PartialContainment ;
          imis:containedObservation ?o1 ;
          imis:containingObservation ?o2 ;
#       other metadata can be added about the containment here
        ]
}
WHERE {
?o1 a qb:Observation .
?o2 a qb:Observation .
?o1 ?d1 ?v1.
?o2 ?d1 ?v2.
?v1 skos:broaderTransitive/skos:broaderTransitive* ?v2
```

```
}
```

Partial Containment (detection):

```
SELECT DISTINCT ?o1, ?o2
WHERE {
        ?o1 a qb:Observation .
        ?o2 a qb:Observation .
        ?o1 ?d1 ?v1.
        ?o2 ?d1 ?v2.
        ?v1 skos:broaderTransitive/skos:broaderTransitive* ?v2
        FILTER(?o1 != ?o2)
}
```

The above query will select pairs of *?o1* and *?o2* that have at least one dimension with ancestral values; *?v1* must be a parent of *?v2*. The above query does not provide the number of dimensions that participate in the *partial containment*; this would make the query more complicated.

In the case of *full containment*, the queries for materializing and detecting pairs of observations are as follows:

Full Containment (Materialization):

```
CONSTRUCT {
        [
  rdf:type imis:FullContainment ;
          imis:containedObservation ?o1 ;
          imis:containingObservation ?o2 ;
#       other metadata can be added about the containment here
        ]
}
WHERE {
?o1 a qb:Observation .
?o2 a qb:Observation .
?o1 ?d1 ?v1.
?o2 ?d1 ?v2.
?v1 skos:broaderTransitive/skos:broaderTransitive* ?v2
        ?o1 ?d2 ?v12 .
        ?o2 ?d2 ?v22 .
        FILTER NOT EXISTS {
OPTIONAL {
                ?v12 skos:broaderTransitive/skos:
                   broaderTransitive* ?v22
}
FILTER (!BOUND(?v22))
}
}
```

Full Containment (Detection):

```
SELECT DISTINCT ?o1, ?o2 WHERE {
?o1 a qb:Observation .
?o2 a qb:Observation .
?o1 ?d1 ?v1.
?o2 ?d1 ?v2.
?v1 skos:broaderTransitive/skos:broaderTransitive* ?v2
        ?o1 ?d2 ?v12 .
        ?o2 ?d2 ?v22 .
        FILTER NOT EXISTS {
OPTIONAL {
                ?v12 skos:broaderTransitive/skos:
                   broaderTransitive* ?v22
}
FILTER (!BOUND(?v22))
}
}
```

The above queries return pairs observations, *?o1* and *?o2*, that have all dimension values exhibiting ancestral relationships; all *?v1* must be a parent of *?v2* using property paths of undefined length.

Complementarity (Materialization):

In the case of *complementarity*, we tested the data against the following SPARQL queries:

```
CONSTRUCT {
        [
  rdf:type imis:Complement;
          imis:observation ?o1 ;
          imis:observation ?o2 ;
#       other metadata can be added about the containment here
        ]
}
WHERE {
?o1 a qb:Observation .
?o2 a qb:Observation .
?o1 ?d1 ?v1.
?o2 ?d1 ?v1.
        FILTER NOT EXISTS {
                ?o1 ?d2 ?v12 .
                ?o2 ?d2 ?v22 .
                FILTER(?v12!=?v22)
        }
}
```

Complementarity (Detection):

```
SELECT DISTINCT ?o1, ?o2
```

```
WHERE {
?o1 a qb:Observation .
?o2 a qb:Observation .
?o1 ?d1 ?v1.
?o2 ?d1 ?v1.
        FILTER NOT EXISTS {
                ?o1 ?d2 ?v12 .
                ?o2 ?d2 ?v22 .
                FILTER(?v12!=?v22)
        }
}
```

The queries will return pairs of observations with no different values in their shared dimensions.

Note that in all of the queries, we have relaxed the conditions presented in section 2, as the runtimes would be even slower, and their syntax complicated.

Appendix C

Notes on the Rule-Based Approach

Notes on the rule-based approach. The rule based approach is performed with forward-chaining rules for the cases of containment and complementarity. For *full containment*, we check for pairs of observations that exhibit both existential and universal quantification in the subsumption of their respective dimension values. The existential quantification is needed to ensure that there exists at least one relationship, while the universal is needed to ensure that all relationships exist. The rule for *full containment* is as follows:

```
observation(o1) ∧ observation(o2)
∧ (o1 ≠ o2)
∧ ∃p.(has_dimension_value(o1,p,v1)
        ∧ has_dimension_value(o2,p,v2)
        ∧ is_ancestor(v1,v2))
∧ ∀p.( has_dimension_value(o1,p,v1)
        ∧ has_dimension_value(o2,p,v2)
        ∧ is_ancestor(v1,v2))
⇒ full_containment(o1,o2)
```

Similarly, the rule for *partial containment* checks the existential restriction; that is, we need at least one pair of dimension values to exhibit a containment relationship between o_1 and o_2. Therefore, the rule is as follows:

```
observation(o1) ∧ observation(o2)
∧ (o1 ≠ o2)
∧ ∃p.(has_dimension_value(o1,p,v)
        ∧ has_dimension_value(o2,p,v))
⇒ partial_containment(o1,o2)
```

The rule for *complementarity* is activated when two different observations have the same values for all of their shared dimensions and is summarized in the following:

```
observation(o1) ∧ observation(o2)
∧ (o1 ≠ o2)
∧ ∃p.(has_dimension_value(o1,p,v)
        ∧ has_dimension_value(o2,p,v))
∧ ∀p.(has_dimension_value(o1,p,v)
        ∧ has_dimension_value(o2,p,v))
⇒ complement(o1,o2)
```