DEPARTMENT OF INFORMATICS
UNIVERSITY OF FRIBOURG (SWITZERLAND)

Efficient, Scalable, and Provenance-Aware Management of Linked Data

THESIS

presented to the Faculty of Science of the University of Fribourg (Switzerland) in consideration for the award of the academic grade of *Doctor scientiarum informaticarum*

by

MARCIN WYLOT

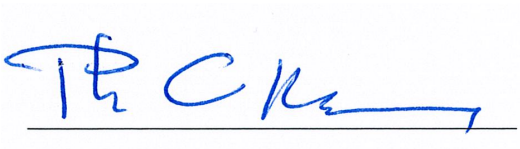
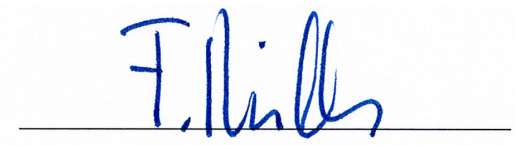
from

POLAND

Thesis No: 1903
UniPrint
2015

Accepted by the Faculty of Science of the University of Fribourg (Switzerland) upon the recommendation of Prof. Dr. Paul Groth, Prof. Dr. Manfred Hauswirth, Prof. Dr. Béat Hirsbrunner, and Prof. Dr. Marino Widmer.

Fribourg, June 16, 2015

Thesis supervisor	Dean
	
Prof. Dr. Philippe Cudré-Mauroux	Prof. Dr. Fritz Müller

The proliferation of heterogeneous Linked Data on the Web requires data management systems to constantly improve their scalability and efficiency. Despite recent advances in distributed Linked Data management, efficiently processing large-amounts of Linked Data in a scalable way is still very challenging. In spite of their seemingly simple data models, Linked Data actually encode rich and complex graphs mixing both instance and schema-level data. At the same time, users are increasingly interested in investigating or visualizing large collections of online data by performing complex analytic queries. The heterogeneity of Linked Data on the Web also poses new challenges to database systems. The capacity to store, track, and query provenance data is becoming a pivotal feature of Linked Data Management Systems. In this thesis, we tackle issues revolving around processing queries on big, unstructured, and heterogeneous Linked Data graphs.

In the first part of this work, we introduce a new hybrid storage model for Linked Data based on recurring graph patterns and graph partitioning to enable complex cloud computation on massive webs of data. We describe the architecture of our approach, its main data structures, and the new algorithms we use to partition and allocate data. Contrary to previous approaches, our techniques perform an analysis of both instance and schema information prior to partitioning the data. We extract graph patterns from the data and combine them with workload information in order to find effective ways of co-locating, partitioning and allocating data on clusters of commodity machines. Our approaches enable efficient and scalable distributed Linked Data management in the cloud, and support both transactional and analytic queries efficiently. We perform an extensive evaluation of our techniques showing that they are between 140 and 485 times faster than state-of-the-art approaches on standard workloads.

In the second part of this work, we extend our techniques to efficiently handle the problem of storing, tracking, and querying provenance in Linked Data. We implement two different storage models to physically co-locate lineage and instance data, and for each of them we implement various algorithms to efficiently track lineage of queries at two granularity levels. Additionally, we tackle the problem of efficiently executing queries tailored with provenance data. We introduce five different query execution strategies for queries that incorporate knowledge of provenance. We also present the results of a comprehensive empirical evaluation of our provenance-aware methods over two different datasets and workloads.

The techniques we develop over the course of this thesis are instrumental in deploying Linked Data Management Systems at large on clusters of commodity machines.

La prolifération des données sémantiques hétérogènes sur le Web exige que les systèmes de base de données liées améliorent constamment leur modularité et efficacité. Malgré les récents progrès dans la gestion des données complexes dans les systèmes distribués, le traitement de grandes quantités de données liées de manière évolutive est encore très difficile. En dépit de leurs modèles de données apparemment simples, elle encodent des graphes riches et complexes mélangeant les données d'instance et du schéma. En même temps, les utilisateurs sont de plus en plus intéressés à examiner et à visualiser des grandes collections de données en effectuant des requêtes analytiques complexes. L'hétérogénéité du Web de données pose des nouveaux défis pour les systèmes de base de données. La capacité de stocker, traquer et examiner la provenance des données devient une caractéristique essentielle des bases de données liées. Dans cette thèse, nous traitons des défis provenant de ces deux domaines: le traitement des requêtes sur de grands graphes non structurés et sur le Web de données hétérogènes.

Dans la première partie de ce travail, nous introduisons un nouveau modèle de stockage hybride pour les données liées qui se base sur les graphes récurrents et le partitionnement de graphe permettant d'effectuer des opérations complexes en nuage sur le Web de données. Nous décrivons l'architecture de nos techniques, leurs structures des données et les nouveaux algorithmes que nous utilisons pour partitionner et répartir les données. Contrairement aux techniques précédentes, nous effectuons des analyses des instances et du schéma avant le partitionnement des données. Nous extrayons des modèles des graphes récurrents des données et nous les combinons avec les informations concernant la charge de travail afin de trouver des moyens efficaces pour co-localiser, partitionner et répartir des données sur des grappes de serveurs. Nos approches permettent la gestion de données liées en nuage d'une manière très efficace et extensible.

Dans la deuxième partie de ce travail, nous étendons nos techniques pour traiter le problème du stockage, du traquage et de l'examen de la provenance des données liées de manière efficace. Nous avons mis en place deux différents modèles de stockage pour co-localiser les données des instances et de la provenance. Pour ces deux modèles, nous avons également mis en place différents algorithmes pour traquer la provenance des requêtes à deux niveaux de granularité. En outre, nous traitons le problème de l'exécution des requêtes adaptées avec des données de provenances. Nous introduisons les cinq stratégies différentes pour effectuer des requêtes qui intègrent la connaissance de la provenance.

Les techniques que nous établissons sont essentielles dans le déploiement de systèmes de gestion de données massives liées sur des grappes de serveurs.

Die wachsende Verbreitung von heterogenen semantischen Daten im Netz erfordert die konstante Verbesserung der Skalierbarkeit und der transaktionalen Effizienz von Linked Data. Trotz jüngster Entwicklungen im Gebiet des verteilten Linked Data ist das verarbeiten grosser Mengen von Linked Data äusserst anspruchsvoll.

Gerade wegen der scheinbar einfachen Datenmodelle, enkodieren schlussendlich umfangreiche und komplexe Graphen durch das Mischen von Instanzen und Schemata. Gleichzeitig sind Anwender zunehmend interessiert durch komplexe analytische Abfragen grosse Sammlungen von online Daten für Untersuchungen und Visualisierung zu nutzen. Die Heterogenität von Linked Data auf dem Netz bringen zusätzliche Anforderungen für Datenbanksysteme. Die Fähigkeit Herkunftsmetadaten (provenance data) zu speichern, verfolgen und abzufragen bekommt zunehmend ein zentrales Merkmal von modernen Triplestores. In dieser Doktorarbeit werden Herausforderung aus beiden Gebieten angegangen: das Ausführen von Abfragen auf grossen unstrukturierten Graphen und auch auf heterogenen Linked Data.

Im ersten Teil dieser Arbeit führen wir einen neuen hybrides Speichermodel für Linked Data ein. Dabei werden wiederholenden Graphmuster und Graphenpartionierungen genutzt um komplexe verteilte Berechnungen auf grossen Mengen von Datennetzen zu ermöglichen. Wir erläutern die Architektur unseres Ansatzes, die zentralen Datenstrukturen, und die neuen Algorithmen welche für die Partitionierung und das Allozieren von Daten zuständig sind. Im Gegensatz zu bestehenden Ansätzen basiert unsere Partitionierung auf der physiologischen Analyse der Instanz- und Schemainformationen. Wir extrahieren sich wiederholende Graphmuster aus den Daten und kombinieren diese mit Nutzlastinformation. Dies ermöglicht eine effektive Anordnung (Kollokation), Portionierung und Allokation der Daten auf einem verteilten System bestehend aus handelsüblicher Rechner. Unser Ansatz ermöglicht die effiziente und skalierbare Datenverwaltung von Linked Data in verteilten Systemen, wobei transaktionale und analytische Abfragen effizient unterstützt werden.

Im zweiten Teil erweitern wir unsere Techniken um das Speichern, Verfolgen und Abfragen von Herkunftsmetadaten (provenance data) effizient lösen zu können. Dazu haben wir zwei unterschiedliche Speichermodelle implementiert, für die Anordnung nach gemeinsamer Herkunft und für die Instanzdaten. Für beide haben wir verschiedenen Algorithmen um die Herkunft von Abfragen effizient zu verfolgen implementiert, dies auf zwei Ebenen der Granularität. Weiter zeigen wir auf wie Abfragen zugeschnitten anhand von Herkunftsmetadaten effizient ausgeführt werden können.

Diese Techniken entwickelt haben sind unabdingbar für den Einsatz von Datenverwaltung auf grossen verteilten Systemen bestehen aus handelsüblichen Rechnern.

Table of Contents

Chapter 1

Introduction

The nature of the World Wide Web has evolved from a web of linked documents to a web including Linked Data [20]. Traditionally, we were able to publish documents on the Web and create links between them. Those links however, allowed only to traverse the document space without understanding the relationships between the documents and without linking to particular pieces of information. Linked Data allows to create meaningful links between pieces of data on the Web [16]. The adoption of Linked Data technologies has shifted the Web from a space connecting documents to a global space where pieces of data from different domains are semantically linked and integrated to create a global Web of Data [20]. Linked Data enables operations to deliver integrated results as new data is added to the global space. This opens new opportunities for applications such as search engines, data browsers, and various domain-specific applications [20].

The Web of Linked Data is rapidly growing from a dozen data collections in 2007 to a space of hundreds data sources in April 2014 [12, 19, 103]. The number of linked datasets doubled between 2011 and 2014 [103], which shows an accelerating trend of data integration on the Web. The Web of Linked Data contains heterogeneous data coming from multiple sources, various contributors, produced using different methods, degrees of authoritativeness, and gathered automatically from independent and potentially unknown sources. Figure 1.1 shows the Linking Open Data cloud diagram created in April 2014; it depicts the scale and heterogeneity of Linked Data on the Web. Such data size and heterogeneity brings new challenges for Linked Data management systems (i.e., systems which allow to store and to query Linked Data). While small amounts of Linked Data can be handled in-memory or by standard relational database

systems, big Linked Data graphs, which we nowadays have to deal with, are very hard to manage. Modern Linked Data management systems have to face large amounts of heterogeneous, inconsistent, and schema-free data.

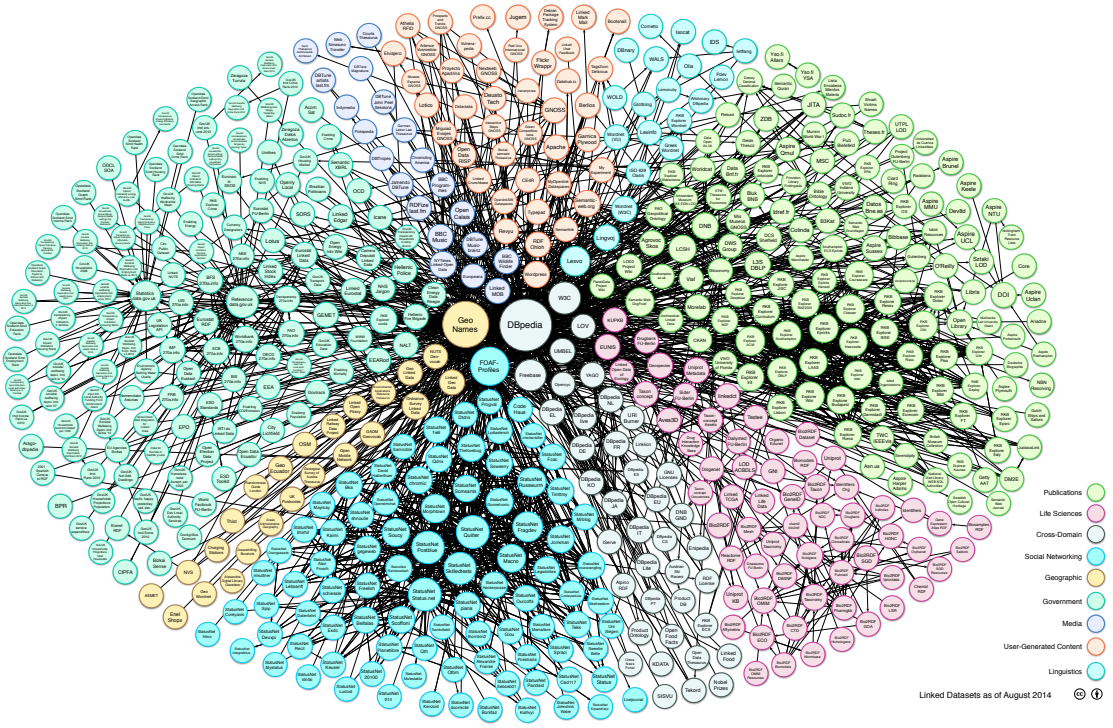


FIGURE 1.1: The diagram shows the interconnectedness of datasets (nodes in the graph) that have been published by heterogeneous contributors to the Linking Open Data community project. It is based on research conducted in April 2014.

The aim of this work is to propose a solution to manage Linked Data in an efficient way, with respect to consumption of resources and query execution performance. Our proposed approach is scalable; it is able to handle large datasets and it scales horizontally in the number of machines leveraged in the cloud environment. Our approach tackles the inconsistency and heterogeneity of Linked Data by adopting novel provenance-aware techniques.

In the remaining of this chapter, we introduce some background information, the research questions we tackled, and give an overview of our contributions. In Chapter 2 and 3, we analyze and evaluate in detail several well-known Linked Data management systems. We highlight their strong and weak points, discuss how they can be improved, and show that the current approaches are overall suboptimal. In Chapter 4, we propose our own approach to optimize the management of large amounts of Lined Data, which maintains an optimal balance between intraoperator parallelism and data co-location.

We describe and empirically evaluate our method for efficient and salable data management. Following that, we present, to the best of our knowledge, the first pragmatic solution to store, track, and query provenance in a Linked Data management system in Chapters 5 and 6. We provide a solution enabling to understand how the results of a query were produced, and which pieces of data were combined to derive the results. Moreover, we show how our approach allows to tailor query execution with respect to data provenance.

1.1 Background Information

In this section, we briefly introduce the basic concepts underpinning Linked Data technologies. We present a data model, vocabularies, and a data exchange format. Then, we introduce baseline approaches to trace provenance in query execution and to execute provenance-aware queries. A detailed presentation of current approaches to manage Linked Data is provided in Chapters 2 and 3. Nevertheless, we expect the reader to be familiar with a number of basic techniques from the Database Systems, Linked Data, and Provenance areas. We refer the reader to the following books for an introduction to the fields related to this work:

- "Readings in database systems." Hellerstein, Joseph M., and Michael Stonebraker. MIT Press, 2005. [69]
- "Database systems: the complete book." Garcia-Molina, Hector. Pearson Education India, 2008. [50]
- "Linked data: Evolving the web into a global data space." Heath, Tom, and Christian Bizer. Synthesis lectures on the semantic web: theory and technology 1.1 (2011): 1-136. [68]
- "Provenance: an introduction to PROV." Moreau, Luc, and Paul Groth. Synthesis Lectures on the Semantic Web: Theory and Technology 3.4 (2013): 1-129. [81]

1.1.1 Linked Data Concepts

Linked Data extends the principles of the World Wide Web from linking documents to linking pieces of data and create a Web of Data; it specifies data relationships and provides machine-processable data to the Internet. It is based on standard Web techniques

but extends them to provide data exchange and integration. The four main principles of the Web of Linked Data, as defined by Tim Berners-Lee [15], are:

1. Use URIs (Uniform Resource Identifier) [1] as names for things.
2. Use HTTP (Hypertext Transfer Protocol) [2] URIs so that people can look up those names.
3. When someone looks up a URI, provide useful information, using standards (Resource Description Framework [3], SPARQL Query Language [4]).
4. Include links to other URIs, so that they can discover more things.

Linked Data uses RDF, the Resource Description Framework, as basic data model. RDF provides means to describe resources in a semi-structured manner. The information expressed using RDF can be exchanged and processed by applications. The ability to exchange and interlink data on the Web means that it can be used by applications other than those for which it was originally created, and that it can be linked to further pieces of information to enrich existing data. It is a graph-based format, optionally defining a data schema, to represent information about resources. RDF allows to create statements in the form of triples consisting of *Subject, Predicate, Object*. A statement expresses a relationship (defined by a predicate) between resources (subject and object). The relationship is always from subject to object (it is directional). The same resource can be used in multiple triples playing the same or different roles, e.g., it can be used as the subject in one triple and as the object in another. This ability enables to define multiple connections between the triples, hence creating a connected graph of data. The graph can be represented as nodes representing resources and edges representing relationships between the nodes. Figures 1.2 and 1.3 depict simple examples of RDF graphs.

Elements appearing in the triples (subjects, predicates, objects) can be of one of he following types:

IRI (International Resource Identifier) identifies a resource. It provides a global identifier for a resource without implying its location or a way to access it. The identifier can be re-used by others to identify the same resource. IRI is a generalization

[1] `http://www.w3.org/Addressing/`
[2] `http://www.w3.org/Protocols/`
[3] `http://www.w3.org/RDF/`
[4] `http://www.w3.org/TR/sparql11-query/`

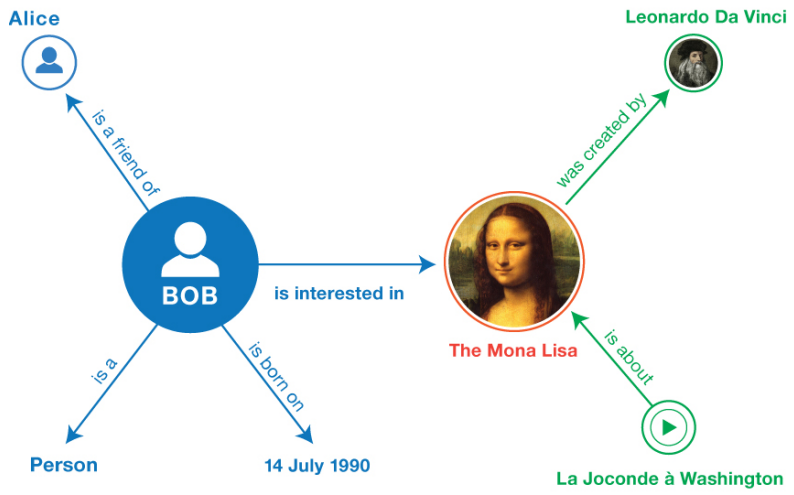


FIGURE 1.2: An exemplary graph of triples. [36]

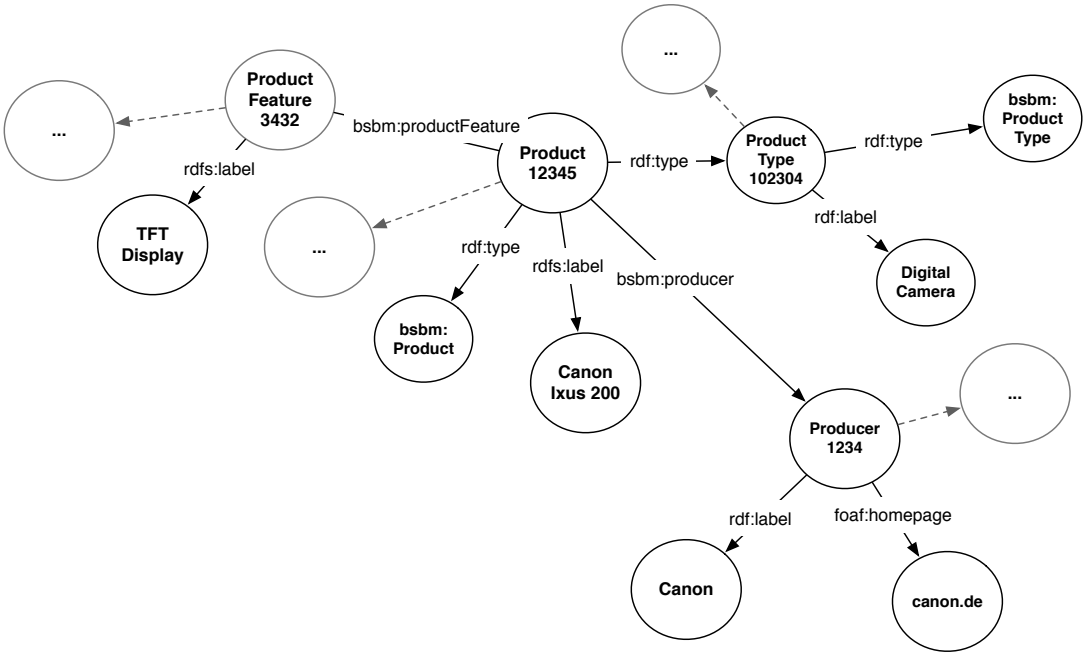


FIGURE 1.3: Example showing an RDF sub-graph using the subject, predicate, and object relations given by the sample data.

of URI (Uniform Resource Identifier) allowing non-ASCII characters to be used. IRI can appear at all three positions in a triple (subject, predicate, object).

Literal is a basic string value that is not an IRI. It can be associated with a datatype, thus can be parsed and correctly interpreted. It is allowed only as the object of a triple.

Blank node is used to denote a resource without assigning a global identifier with an IRI, it is a local unique identifier used within a specific RDF graph. It is allowed as the subject and the object in a triple.

The framework provides means to co-locate triples in a subset and to associate such subsets with an IRI [35]. A subset of triples constitutes an independent graph of data (named graph). In practice, it provides data managers with a mechanism to create a collection of triples. A dataset can consist of multiple named graphs and no more than one unnamed (default) graph.

Even though RDF does not require any naming convention for IRIs and does not impose any schema on data it can be used in combination with vocabularies provided by RDF Schema language [37]. RDFS is a semantic extension of RDF enabling to specify semantic characteristics of RDF data. It provides a data-modeling vocabulary for RDF data. It enables to state than an IRI is a property and that a subject and an object of the IRI have to be of a certain type. RDF schema allows to classify resources with categories, i.e. classes, types. Classes allow to regroup resources. Members of a class are called instances, while classes are also resources and can be described with triples. RDFS allows classes and properties to be hierarchical, as a class can be a sub-class of a more generic class. In the same way, properties can be defines as a specific property (sub-property) of a more generic one. RDFS enables also to specify a domain and a range of a predicate, i.e., types of resources allowed as subjects and objects. Properties are also resources that can be described by triples. An instance can be associated with several independent classes specifying different sets of properties. RDFS defines also a set of utility properties allowing to link pieces of data, e.g., *seeAlso* to indicate a resource providing additional information about the resource of a subject. Another interesting vocabulary set defined by RDFS is reification, which allows to write statements about statements.

Richer vocabularies or ontology languages (e.g., OWL) enable to express logical constraints on Web data. The OWL 2 Web Ontology Language [32] allows to define ontologies to give a semantic meaning to the data. An ontology provide classes, properties,

and data values. An ontology is exchanged along with the data as an RDF document, and defines vocabularies and relationships between terms, often covering a specific domain shared by a community. An ontology can also be seen as an RDF graph, where terms are represented by nodes and relationships between them are expressed by edges.

Linked Data in general is a static snapshot of information, though it can express events and temporal aspects of entities with specific vocabulary terms [34]. A snapshot of the state can be seen as a separate (named) RDF graph containing a current state of the universe. Changes in data typically concern relationships between resources, IRIs and Literals are constant and rarely change their value.

Linked Data allows to combine and process data from many sources [15]. The basic triple representation of pieces of data combined together results in large RDF graphs. Such large amounts of data are made available as Linked Data where datasets are interlinked and published in the Web.

Linked Data can be serialized in a number of formats that are logically equivalent. The data can be stored in the following formats:

N-Triples provides a simple, plain-text way to serialize Linked Data. Each line in a file represents a triple, the period at the end signals the end of a statement (triple). This format is often used to exchange large amount of Linked Data and for processing graphs with stream-oriented tools.

N-Quads is a simple extension of N-Triples. It allows to add a fourth optional element in a line denoting a named graph IRI, which the triple belongs to.

Turtle is an extension of N-Triples; it introduces a number of syntactic shortcuts, such as prefixes, lists, and shorthands for datatyped literals. It provides a trade-off between ease of writing, parsing, and readability. It does not support the notion of named graphs.

TriG extends Turtle to support multiple named graphs.

JSON-LD provides a JSON syntax for Linked Data. It can be used to transform JSON documents into Linked Data, and offers universal identifiers for JSON objects and a way in which a JSON document can link to an object in another document.

RDFa is a syntax used to embed Linked Data in HTML and XML documents. This enables to aggregate data from web pages and use it to enrich search results or presentation.

RDF/XML provides an XML syntax for Linked Data.

To facilitate querying and manipulating Linked Data on the Web, a semantic query language is needed. Such a language, named SPARQL Protocol and RDF Query Language, was introduced by the World Wide Web Consortium. SPARQL [33] can be used to formulate queries ranging from simple graph patterns to very complex analytic queries. Queries may include unions, optionals, filters, value aggregations, path expressions, subqueries, value assignment, etc. Apart from SELECT queries, the language also supports:

ASK queries to retrieve binary "yes/no" answer to a query,

CONSTRUCT queries to construct new RDF graphs from a query result.

All standards and Linked Data concepts are defined and explained in detail in documents published by the World Wide Web Consortium. We refer the reader to the following recommendations for further detail:

- RDF 1.1 Primer [36]
- RDF 1.1 Concepts and Abstract Syntax [34]
- RDF Schema 1.1 [37]
- RDF 1.1: On Semantics of RDF Datasets [35]
- OWL 2 Web Ontology Language [32]
- SPARQL 1.1 Overview [33]

1.1.2 Provenance

"Provenance is information about entities, activities, and people involved in producing a piece of data or thing, which can be used to form assessments about its quality, reliability or trustworthiness" [56].

Data provenance has been widely studied within the database, distributed systems, and Web communities. For a comprehensive review of the provenance literature, we refer readers to recent positions in the field [81, 86]. Likewise, Cheney et al. provide

a detailed review of provenance within the database community [29]. Broadly, one can categorize the work into three areas [55]: content, management, and use. Work in the content area has focused on representations and models of provenance. In management, the work has focused on collecting provenance in software ranging from scientific databases [38] to operating systems or large scale workflow systems as well as mechanisms for querying it. Finally, provenance is used for a variety of applications including debugging systems, calculating trust and checking compliance. Here, we briefly review the work on provenance with respect to the Web of Data. We also review recent results applying theoretical database results with respect to SPARQL.

Within the Web of Data community, one focus of work has been on designing models (i.e., ontologies) for provenance information [64]. The W3C Incubator Group on provenance mapped nine different models of provenance [102] to the Open Provenance Model [87]. Given the overlap in the concepts defined by these models, a W3C standardization activity was created that has led to the development of the W3C PROV recommendations for interchanging provenance [56]. This recommendation is being increasingly adopted by both applications and data set providers - there are over 60 implementations of PROV [72].

In practice, provenance is attached to Linked Data using either reification [67] or named graphs [26]. Widely used datasets such as YAGO [70] reify their entire structures to facilitate provenance annotations. Indeed, provenance is one reason for the inclusion of named graphs in the next version of RDF [112]. Both named graphs and reification lend to complex query structures especially as provenance becomes increasingly fined grained. Indeed, formally, it may be difficult to track provenance using named graphs under updates and RDFS reasoning [95].

To address these issues, a number of authors have adopted the notion of annotated RDF [47, 107]. This approach assigns annotations to each of the triples within a dataset and then tracks these annotations as they propagate through either the reasoning or query processing pipelines. Formally, these annotated relations can be represented by the algebraic structure of communicative semirings, which can take the form of polynomials with integer coefficients [54]. These polynomials represent how source tuples are combined through different relational algebra operators (e.g., UNION, JOINS). These relational approaches are now being applied to SPARQL [105].[5]

[5]Note, in terms of formalization, SPARQL poses difficulties because of the OPTIONAL operator, which implies negation.

As Damásio et al. have noted [40], many of the annotated RDF approaches do not expose how-provenance (i.e., how a query result was constructed). The most comprehensive implementations of these approaches are [107, 113]. However, they have only been applied to small datasets (around 10 million triples) and are not aimed at reporting provenance polynomials for SPARQL query results. Annotated approaches have also been used for propagating trust values [65]. Other recent work, e.g., [40, 51], has looked at expanding the theoretical aspects of applying such a semiring based approach to capturing SPARQL.

Miles defined the concept of *provenance query* [85] in order to only select a relevant subset of all possible results when looking up the provenance of an entity.

A number of authors have presented systems for specifically handling such provenance queries. Biton et al. showed how user views can be used to reduce the amount of information returned by provenance queries in a workflow system [18]. The MTCProv [49] and the RDFProv [28] systems focus on managing and enabling querying over provenance that results from scientific workflows. Similarly, the ProQL approach [74] defines a query language and proposes relational indexing techniques for speeding up provenance queries involving path traversals. Glavic and Alonso [52] presented the Perm provenance system, which was able of computing, storing and querying relational provenance data. Provenance was computed by using standard relational query rewriting techniques, e.g., using lazy and eager provenance computation models. Recently, Glavic with his team have built on this work to show the effectiveness of query rewriting for tracking provenance in database that support audit logs and time travel [7]. The approaches proposed in [18, 74] assume a strict relational schema whereas RDF data is by definition schema free.

The work on annotated RDF [107, 113] developed SPARQL query extensions for querying over annotation metadata (e.g. provenance). Halpin and Cheney have shown how to use SPARQL Update to track provenance within a triple store without modifications [60]. The theoretical foundations of using named graphs for provenance within the Semantic Web were established by Flouris et al. [47].

1.2 Research Questions

Efficient and scalable management of Big Data poses new challenges to the databases community [4]. Big Data is defined as data which "represents the progress of the human

cognitive processes, usually includes data sets with sizes beyond the ability of current technology, method and theory to capture, manage, and process the data within a tolerable elapsed time" [53]. The more recent and specific definition of Big Data, given by Gartner [6], specifies that "Big Data are high-volume, high-velocity, and/or high-variety information assets that require new forms of processing to enable enhanced decision making, insight discovery and process optimization" [17]. Big Data systems considered as appropriate for a specific set of application requirements [3] can be characterized by three dimensions, referred to as the 3Vs [80]:

Volume : the size of available data;

Velocity : the speed of data processing, how fast the data is streamed;

Variety : the number of types, structures, and sources of data, how unstructured and heterogeneous the data is.

In this thesis, we tackle a number of fundamental problems related to Linked Data management in the context of Big Data.

Tackling Big Data challenges, we intuitively begin with the volume issue. The size of Linked Data is steadily growing [103], thus a modern Linked Data management system has to be able to deal with increasing amounts of data. However, in the Linked Data context, variety is especially important. Since Linked Data is schema-free (i.e. the schema is not strict), standard databases techniques cannot be directly adopted to manage it. Even though organizing Linked Data in a form of a table is possible (see Section 2.1), querying such a giant triple table becomes very costly due to the multiple nested joins required. Moreover, Linked Data comes from multiple sources and can be produced in various ways for a specific scenario. Heterogeneous data can however incorporate knowledge on provenance, which can be further leveraged to provide users with a reliable and understandable description of the way the query was answered, that is, the way the answer was derived. Furthermore, it can enable a user to tailor queries with provenance data, including or excluding data of specific lineage (i.e., described in a systematic way).

We divide the problem we tackle into three sub-problems. Hence, we define three research questions to investigate:

[6] `http://www.gartner.com/`

(Q1) How to efficiently store and query vast amounts of Linked Data in the cloud?
The Linked Data community is still missing efficient and scalable data infrastructures. New kinds of data and queries (e.g., unstructured and heterogeneous data, graph and analytic queries) cannot be efficiently handles by existing systems. Small Linked Data graphs can be handled in-memory or by standard database systems. However, Big Linked Data with which we deal nowadays [103] are very hard to manage. Modern Linked Data management systems have to face vast amounts of heterogeneous, inconsistent, and schema-free data. In Chapters 2 and 3, we analyze in detail and evaluate several well-known Linked Data management systems. We describe their strong and weak points, and we show that the current approaches are overall suboptimal. Following that, we propose our own distributed Linked Data management system in Chapter 4.

(Q2) How to store and track provenance in Linked Data processing?
Within the Web community, there have been several efforts to develop models and syntaxes to interchange provenance, which resulted in the recent W3C PROV recommendation [56]. However, less attention has been given to the efficient handling of provenance data within Linked Data management systems. While some systems store quadruples or named graphs, to the best of our knowledge, no current high-performance triple store is able to automatically derive provenance data for the results it produces. We present our approaches to store and track provenance in Chapter 5.

(Q3) What is the most effective query execution strategy for provenance-enabled queries?
With the heterogeneity of Linked Data, users may want to tailor their queries based on the provenance, e.g., "find me all the information about Paris, but exclude all data coming from commercial websites". To support such use-cases, the most common mechanism used within Linked Data management systems is named graphs [26]. This mechanism was recently standardized in RDF 1.1. [97]. Named graphs associate a set of triples with a URI. Using this URI, metadata including provenance can be associated with the graph. While named graphs are often used for provenance, they are also used for other purposes, for example to track access control information. Thus, while Linked Data management systems (i.e., triple stores) support named graphs, there has only been a relatively small number of approaches specifically focusing on provenance within the triple store itself and much of it has been focused on theoretical aspects of the problem [40, 51]. We describe our methods and implementation to handle provenance-aware workload in Chapter 6.

1.3 Contributions

To answer the aforementioned research questions, we propose different techniques to store and process Linked Data. We divide them into three parts: storing and querying Linked Data in the cloud, storing and tracking provenance in Linked Data, and querying over provenance data.

The first part addresses the problem of efficient storage of Linked Data (Research Question **Q1**); we propose a novel hybrid storage model considering Linked Data both from a graph perspective (by storing molecules [7]) and from a "vertical" analytics perspective (by storing compact lists of literal values for a given attribute). Our molecule-based storage model allows to efficiently partition data in the cloud such as to minimize the number of expensive distributed operations (e.g., joins). We also propose efficient query execution strategies leveraging our compact storage model and taking advantage of advanced data co-location strategies enabling us to execute most of the operations fully in parallel. Specifically, we make the following contributions:

- a new data partitioning algorithm to efficiently and effectively partition the graph and co-locate related instances in the same partitions (Section 4.1);
- a new system architecture for handling fine-grained Linked Data partitions at scale (Section 4.2);
- novel data placement techniques to co-locate semantically related pieces of data (Section 4.3);
- new data loading and query execution strategies taking advantage of our system's data partitions and indices (Section 4.4);
- an extensive experimental evaluation showing that our methods are often two orders of magnitude faster than state-of-the-art systems on standard workloads (Section 4.5).

In the second part of this thesis, we present techniques supporting the transparent and automatic derivation of detailed provenance information for arbitrary queries (Research

[7] molecules [43] that are similar to property tables [111] and store, for each subject, the list or properties and objects related to that subject.

Question **Q2**). We introduce new physical models to store provenance data and several new query execution strategies to derive provenance information. We make the following contributions in that context:

- a new way to express the provenance of query results at two different granularity levels by leveraging the concept of provenance polynomials [8] (Section 5.2);
- two new storage models to represent provenance data in a native Linked Data store compactly, along with query execution strategies to derive the aforementioned provenance polynomials while executing the queries (Sections 5.3 and 5.4);
- a performance analysis of our techniques through a series of empirical experiments using two different Web-centric datasets and workloads (Section 5.5).

In the third part of this thesis, we investigate how Linked Data management systems can effectively support queries that specifically target provenance that it, provenance-enabled queries (Research Question **Q3**). To address this problem, we propose different provenance-aware query execution strategies and we test their performance with respect to our provenance-aware storage models and advanced co-location strategies. We make the following contributions in that context:

- a characterization of provenance-enabled queries (queries tailored with provenance data) (Section 6.1);
- five provenance-oriented query execution strategies (Section 6.2);
- storage model and indexing techniques extensions to handle provenance-aware query execution strategies (Section 6.3);
- an experimental evaluation of our query execution strategies and an extensive analysis of the datasets used for the experimental evaluation in the context of provenance data (Section 6.4).

[8] Provenance polynomials [54] are algebraic structures representing how the data is combined to derive the query answer using different relational algebra operators (e.g., UNION, JOINS).

1.3.1 List of Publications

The following list gives an overview of the main publications related to this thesis.

• *dipLODocus[RDF]: short and long-tail RDF analytics for massive webs of data*
Marcin Wylot, Jigé Pont, Mariusz Wisniewski, and Philippe Cudré-Mauroux
International Semantic Web Conference, 2011

This paper introduces a novel database system for RDF data management called dipLODocus[RDF], which supports both transactional and analytical queries efficiently. dipLODocus[RDF] takes advantage of a new hybrid storage model for RDF data based on recurring graph patterns. In this paper, we describe the general architecture of our system and compare its performance to state-of-the-art solutions for both transactional and analytic workloads.

• *DiploCloud: Efficient and Scalable Management of RDF Data in the Cloud*
Marcin Wylot and Philippe Cudré-Mauroux
Under Revision. IEEE Transactions on Knowledge and Data Engineering (TKDE), 2015

In this paper, we describe DiploCloud, an efficient and scalable distributed RDF data management system for the cloud. Contrary to previous approaches, DiploCloud runs an analysis of both instance and schema information prior to partitioning the data. It extracts recurring graph patterns from the data and combines them with workload information in order to find effective ways of partitioning and allocating data on clusters of commodity machines. In this paper, we describe the architecture of DiploCloud, its main data structures, as well as the new algorithms we use to partition and allocate data. We also present an extensive evaluation of DiploCloud showing that our system is between 140 and 485 times faster than state-of-the-art systems on standard workloads.

• *TripleProv: Efficient Processing of Lineage Queries in a Native RDF Store*
Marcin Wylot, Philippe Cudré-Mauroux, and Paul Groth
23rd International Conference on World Wide Web, 2014

This paper introduces TripleProv: a new system extending a native RDF store to efficiently handle such queries. TripleProv implements two different storage models to physically co-locate lineage and instance data, and for each of them implements algorithms for tracing provenance at two granularity levels. We present the overall architecture of our system, its different lineage storage models, and the various query execution

strategies we have implemented to efficiently answer provenance-enabled queries. In addition, we present the results of a comprehensive empirical evaluation of our system over two different datasets and workloads.

• *Executing Provenance-Enabled Queries over Web Data*

Marcin Wylot, Philippe Cudré-Mauroux, and Paul Groth

24th International Conference on World Wide Web, 2015

In this paper, we tackle the problem of efficiently executing provenance-enabled queries over RDF data. We propose, implement and empirically evaluate five different query execution strategies for RDF queries that incorporate knowledge of provenance. The evaluation is conducted on Web Data obtained from two different Web crawls (The Billion Triple Challenge, and the Web Data Commons). Our evaluation shows that using an adaptive query materialization execution strategy performs best in our context. Interestingly, we find that because provenance is prevalent within Web Data and is highly selective, it can be used to improve query processing performance. This is a counterintuitive result as provenance is often associated with additional overhead.

• *NoSQL Databases for RDF: An Empirical Evaluation*

Philippe Cudré-Mauroux, Iliya Enchev, Sever Fundatureanu, Paul Groth, Albert Haque, Andreas Harth, Felix Leif Keppmann, Daniel Miranker, Juan Sequeda, and Marcin Wylot

International Semantic Web Conference, 2013

This work is the first systematic attempt at characterizing and comparing NoSQL stores for RDF processing. We describe four different NoSQL stores and compare their key characteristics when running standard RDF benchmarks on a popular cloud infrastructure using both single-machine and distributed deployments.

• *BowlognaBench-Benchmarking RDF Analytics*

Gianluca Demartini, Iliya Enchev, Marcin Wylot, Joël Gapany, and Philippe Cudré-Mauroux

Data-Driven Process Discovery and Analysis, 2012

This paper introduces a novel benchmark for evaluating and comparing the efficiency of Semantic Web data management systems on analytic queries. Our benchmark models a real-world setting derived from the Bologna process and offers a broad set of queries reflecting a large panel of concrete, data-intensive user needs and it provides a way to evaluate systems over analytics and temporal queries.

• *A Comparison of Data Structures to Manage URIs on the Web of Data*
Ruslan Mavlyutov, Marcin Wylot, and Philippe Cudre-Mauroux
European Semantic Web Conference, 2015

The paper presents the first systematic comparison of the most common data structures used to encode URI data. We evaluate a series of data structures in term of their read-/write performance and memory consumption.

1.4 Outline

This thesis starts with an extensive analysis of current approaches to manage Linked Data, as well as an experimental evaluation of several of them. Afterwards, we introduce our approach to manage Linked Data in the cloud, our techniques to store and track provenance, and our algorithms to execute provenance-enabled queries. More specifically, the remaining chapters of the thesis are organized as follows:

Chapter 2 presents a detailed overview of multiple Linked Data storage systems and categorizes them into the following set of groups: native systems, massively parallel systems, and relational database-based systems.

Chapter 3 presents an empirical evaluation of four different approaches to process Linked Data regrouped under the NoSQL umbrella. To act as a reference point, we also measure the performance of 4store, a native triple store. The goal of this evaluation is to understand the current state of these systems. In particular, we are interested in: (i) determining if there are commonalities across the performance profiles of these systems in multiple configurations (data size, cluster size, query characteristics, etc.), (ii) characterizing the differences between NoSQL systems and native triple stores, (iii) providing guidance on where researchers and developers interested in Linked Data and NoSQL should focus their efforts, and (iv) providing an environment for replicable evaluation.

Chapter 4 describes our approaches to store Linked Data in a compact way. We introduce a new hybrid storage model considering Linked Data both from a graph and from an analytics perspective; we describe our template-based molecules; and a new data partitioning algorithm and data placement techniques to co-locate semantically related pieces of data in the cloud. We also present an implementation

of the aforementioned techniques in our triplestore, by describing a system architecture for handling fine-grained Linked Data partitions at scale. Finally, we experimentally evaluate our approaches showing that our methods are often two orders of magnitude faster than state-of-the-art systems on standard workloads.

Chapter 5 describes our approaches to efficiently store and track provenance in Linked Data. We start by presenting a new way to express the provenance of query results, where we leverage the concept of provenance polynomials. Later, we describe two new storage models to represent provenance data compactly, leveraging the aforementioned concept of molecules. We also present new query execution strategies to derive the provenance polynomials while executing the queries, and a performance analysis of our techniques through a series of empirical experiments using two different datasets and workloads.

Chapter 6 introduces our solution to efficiently execute queries over Linked Data including provenance data. We start by introducing a definition of provenance-enabled queries, then we describe our five provenance-oriented query execution strategies, and finally we perform an experimental evaluation of our query execution strategies and an extensive analysis of the datasets used for the experimental evaluation in the context of provenance data.

Chapter 7 summarizes our approaches and contributions; it provides conclusions drawn from our work and describes potential future work.

Chapter 2

Current Approaches to Manage Linked Data

In this chapter, we provide a detailed overview of current approaches to Linked Data management. Since Linked Data can be stored in many different formats, it is especially important to outline and classify these different approaches. In addition, we discuss the different advantages and drawbacks of the various technique using a common data model such that the different engines can be compared more easily.

Linked Data can be stored in a multiplicity of different storage engines. Some of these are more adapted to store Linked Dataa while others attempt to use general purpose database storage engines to persist Linked Dataa. We therefore look at a multiplicity of Linked Data storage systems in the following and try to categorize them into the following classes: native systems, massively parallel systems, and relational database engine systems.

In the context of this chapter, we describe native Linked Data management systems as systems which are originally designed to persist Linked Data primarily. This excludes systems which use for example relational databases and only transform input data and queries into Linked Data. The same is true for graph-oriented databases as they primarily store arbitrary graphs and not RDF graphs specifically. We define a non-native Linked Data storage engine as a specific engine that uses either traditional relational storage concepts or builds on these concepts to integrate storing and query execution for Linked Data. The biggest differentiation to native Linked Data storage solutions is hereby the translation of the Linked Data concepts into concepts that are native to the underlying engine instead of working directly with the Linked Data.

We discuss approaches using relational database systems in Section 2.1. Subsequently, in Section 2.2 we discuss Native Linked Data Management Systems, and finally in Section 2.3 we discuss the applicability of using massively parallel systems based on MapReduce and the Hadoop framework.

2.1 Storing Linked Data using Relational Databases

2.1.1 Statement Table

The general data structure that is represented by a set of RDF triples is an edge-labeled directed graph. Figure 1.3 shows a subset of nodes from a sample dataset inspired by the data model from the Berlin SPARQL Benchmark[22]. In this representation, subjects and objects are stored as nodes with an edge and an associated edge property assigned: $[S] - P \rightarrow [O]$. As stated by [6, 25], subjects and objects can be interchanged. In addition, all triples are unordered[6].

Since RDF does not describe any specific meta-model for the graph, there is no easy way to determine a set of partitioning or clustering criteria to derive a set of tables to store the information. In addition, there is no definite notion of schema stability, meaning that at any time the data schema might change, for example when adding a new subject-object edge to the overall graph.

A trivial way for for adopting a relational data structure to store RDF data is to store the input data as a linearized list of triples, storing them as ternary tuples. In [6], this approach is called the "generic" approach. The RDF specification states that the objects in the graphs can be either URIs, literals, or blank nodes. Properties (predicates) always are URI references. Subject nodes can only be URIs or blank nodes. This allows to specify the underlaying data types for storing subject and predicate values. For storing object values this becomes a little more complex since the data type of the object literal is defined by the XML schema that is referenced by the property. A common way is to store the object values using a common string representation and perform some type conversion whenever necessary. An example table showing the same data set as in 1.3 is shown in 2.1.

Subject	Predicate	Object
Product12345	rdf:type	bsbm:Product
Product12345	rdfs:label	Canon Ixus 2010
Product12345	bsbm:producer	bsbm-inst:Producer1234
...	...	...
Producer1234	rdf:label	Canon
Producer1234	foaf:homepage	http://www.canon.com
...	...	...

FIGURE 2.1: A simple RDF storage scheme using a linearized triple representation. The illustration uses schema elements from the Berlin SPARQL Benchmark[22]

An example on *Statement Table* approach is Jena1 [84] [1]. Jena1 for relational databases stores data in a statement table. The URI and String are encoded in ID and two separate dictionaries are maintained for literals and resources/URIs. To distinguish literals from URI in the statement table there are two columns. In Jena2 [110] the schema is denormalized and URIs and simple literals are directly stored in the statement table. The dictionary tables are used only to store long strings (exceeding a threshold). This allows to perform filters operation directly on the statement table, however it results also higher storage consumption, since string values are stored multiple times.

2.1.2 Optimizing Data Storage

Storing the triples as a linear list of triples is very simple and yet powerful, since it captures the complete essence of RDF data. However, the problem with this data structure is that additional information that is crucial for query processing needs to be analyzed at query run-time even though it is not likely to change. Examples of this issue are whether or not the object is a literal or a URI or if the edge is inferred or an original edge.

One disadvantage of storing the data inside a large triple table is that all fields in this table must be encoded as string with variable length. This generates additional overhead during data storage and data processing. The standard MySQL table storage format, for example, uses an 8 bit or if required 16 bit length identifier followed by the actual string. To process a set of fields it is not possible to perform a direct offset into the set of tuples,

[1] `https://jena.apache.org/`

and the database storage engine has to interpret the complete row, or requires additional data structure for pointers into the variable length fields.

One possible way to optimize the storage structure is to apply dictionary encoding on the resources and literal values. Dictionary encoding allows to replace the variable-length string representation of a literal or resource by a fixed-length integer value. There are several possible ways to generate such an integer values. In order to avoid issues with duplicates, Harris et al. propose in [61] to use a hash function that allows hashing of all literal and resource values. The actual values are replaced with the hash value and the hash and the value is stored in an additional table for later reference. 2.2 illustrates this scheme.

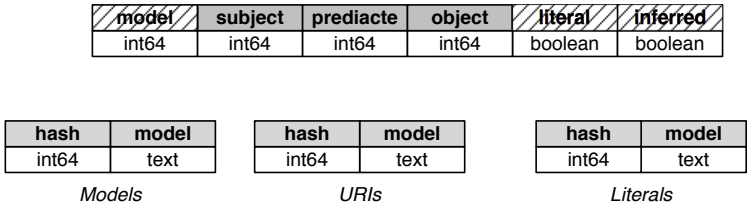
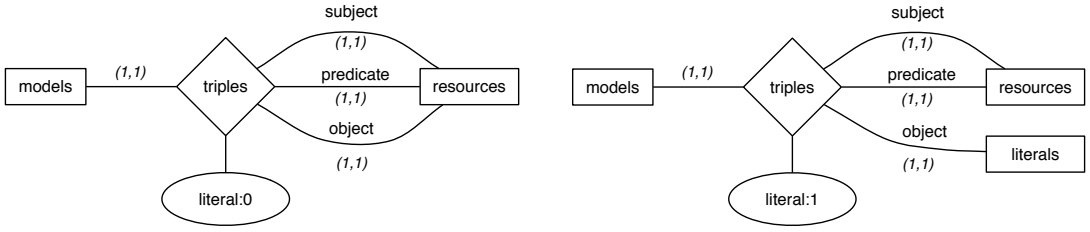


FIGURE 2.2: Logical database design of the triple table in 3store. Illustration after [61]

The downside of using a hash function to generate the encoded values lays within the properties of the hash function. Even though the probability of a hash collision can be low—depending on the actual hash implementation–they still can occur. As a consequence, the import system has to perform validity checks for all imported literals and resources to evaluate if the generated hash value generates a collision. While it is easy to handle such collisions during insertion time as described by Harris et al. it becomes more complicated to handle collisions at runtime when new triples are inferred based on the existing knowledge-base. If a collision happens, it is almost impossible to handle this without modifying the inferred value to generate a different hash value.

In addition, Harris et al. use a cryptographic hash function to calculate a hash key that has as few collisions as possible. The disadvantage of this approach is that computing a cryptographic has consumes more CPU cycles compared to a simpler hash-function. In their example, the calculation of the MD5 hash takes about 1'000 CPU cycles. This limits a single CPU core on a modern 2.5GHz CPU to calculating 2.5M hashes per second. Using such an expensive hash function can thus lead to CPU-bound behaviors even though the database does not operate at optimal speed.

To further reduce the probability of collisions, Harris et al. use two different buckets to store hashes and values. One bucket for literals and one bucket for resources. An E/R diagram showing the dependency for the two different triple types is shown in Figure 2.3. Another advantage of using the MD5 hash function to represent resources or literals is that during query processing the actual lookup of a value is not performed by joining the resources hash table with the triple table as the query processor can directly use the built-in MD5 hash function.



(A) ER diagram modeling a triple pattern with no literal object

(B) ER diagram modeling a triple pattern having a literal object

FIGURE 2.3: Dependency for the two different triple types [61].

2.1.3 Property Tables

Storing RDF triples in a single large statement table presents a number of disadvantages when it comes to query evaluation. In most cases, for each set of triple patterns that is evaluated in the query, a set of self-joins on the table is necessary to evaluate the graph traversal. Since the single statement table can become very large, this can have a negative effect on query execution.

While the horizontal storage of semantic data has been first introduced by Agrawal et al. in [5], the authors of Jena and Sesame propose different ways to alleviate this problem by introducing the concept of property tables in [25, 110]. Instead of building one large table for all occurrences of all properties, they propose two different strategies that can be distinguished into two different concepts: clustered and normalized property tables.

2.1.3.1 Clustered Property Tables

The goal of clustered property tables is to cluster commonly accessed nodes in the graph together in a single table to avoid expensive joins on the data. In [111], the use of clustered property tables is proposed for data that is stored using the Dublin Core

schema[2]. In the example of the dataset of the Berlin SPARQL benchmark shown in Figure 2.4, one property table for all products and a statement table for all other triples are considered. For efficiency reasons, a product record and all affected triples can only appear in the property table.

Product Property Table

Subject	Type	Label	NumericProperty1	aaa
Product12345	bsbm:Product	Canon Ixus 2010	NULL	...
...	...	...	...	...

Left-Over Triples

Subject	Predicate	Object
Producer1234	foaf:homepage	http://www.canon.com
...	...	...

FIGURE 2.4: Example illustrating clustered property tables. Frequently coaccessed attributes are stored together.

The advantage of this storage format is that querying the database using triple patterns that are materialized in the property can be evaluated using simple filter predicates instead of performing self-joins on the statement table. Given the query in Listing 2.1, the transformation to SQL would require two joins, one for each triple pattern. However, if the metadata define that all triples of the type `bsbm:Product` are stored in a property table, this can be translated into a simple predicate evaluation as shown in Listing 2.2.

```
SELECT ?a
WHERE (?a rdf:type bsbm:Product),
      (?a bsbm:NumericProperty1 10)
```

LISTING 2.1: Example SPARQL Query

```
SEELCT t.subject FROM clustered_products as t
WHERE t.NumericProperty1 = 10;
```

LISTING 2.2: Translation of the SPARQL Query in the Listing 2.1 to SQL using the clustered property table approach

The consequences of this approach are that the schema must be known in advance. If the properties for a materialized type change during runtime, this requires table alternations that are costly and often require explicit table-level locking. In addition, multi-valued attributes cannot be easily represented using a clustered property table. If multi-valued attributes must be considered, designer has to choose either to not materialize the path

[2] `http://dublincore.org/`

of the attribute or, if the sequence of the attribute is bounded, to include all possible occurrences in the materialized clustered property table.

Properties tables were also implemented in Jena2 [110] together with a statement table. In that context, multiple-values properties are clustered in a separate table. The system also allows to specify the type of the column in the underlying database system for the property value. This can be further leveraged for range queries and filtering. For example, the property *age* can be implemented as an integer, which can then be efficiently filtered.

2.1.3.2 Normalized Property Table

In this second approach of property tables, the database choses to store triples based on the occurrence of single properties. The RDF entailment rules *rdf1* [3] define that for each triple `s p o` a number of triples `s rdf:type rdf:Property` can be inferred. Based on this knowledge, the database can now select a subset of triples that will be materialized in these special normalized property tables. All other triples will be stored in a general statement table. Figure 2.5 shows an example for this pattern where all instances of `rdf:type` and `rdfs:label` are separated in distinct tables. Abadi et al. present in [2] an extension to this model where all distinct occurrences for a single property will be stored in a decomposed way in a property table.

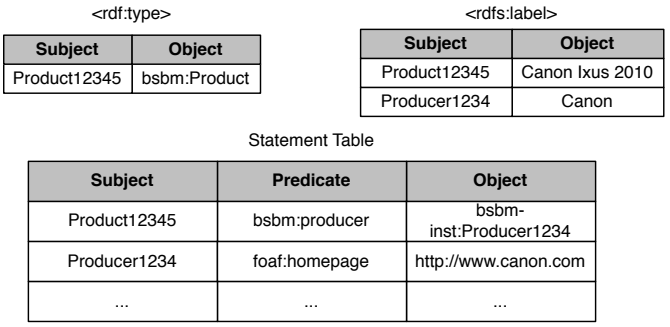


FIGURE 2.5: Example illustrating RDF property tables. For each existing predicate one subject-object table exists

Storing multi-valued attributes in a normalized property table can be achieved by adding one row per occurrence in the data set.

[3] `http://www.w3.org/TR/rdf-mt/#RDFRules`

Product Property Table

Subject	Type	Label	NumericProperty1	aaa
Product12345	bsbm:Product	Canon Ixus 2010	NULL	...
...	...	...	...	...

Left-Over Triples

Subject	Predicate	Object
Producer1234	foaf:homepage	http://www.canon.com
...	...	...

FIGURE 2.6: Example illustrating clustered property tables. In this example, only commonly used predicates are clustered in property tables.

2.1.4 Query Execution

Query execution in relational RDF engines pushes all computational logic of RDF query evaluation to the database to achieve the best performance and leverage available optimization strategies. In [31], Chong et al. present a system that builds on an Orcale relational database management system. Instead of supporting the complete syntax of an RDF query language like SPARQL or RDQL, they focus on the most important subset of these languages which is matching RDF triples. Therefore, they implement a table function[31] that allows to rewrite a set of RDF triple filters to SQL. Figure 2.7 shows an example of a translation of a simple triple pattern query into the matching SQL query that is then issued against the database system.

```
SELECT t.a age
FROM TABLE(RDF_MATCH(
   `(?r Age ?a)`,
    RDFModels('reviewer'),
    NULL)) t
WHERE t.a < 25;
```

```
SELECT t.a age
FROM (
  SELECT u1.UriValue a, u1.Type,
  FROM IdTriples t1, UriMap u1
  WHERE t1.PropertyID=29 AND
  t1.ModelId=1 AND
  u1.UriID = t1.SubjectID) t
WHERE t.a < 25;
```

FIGURE 2.7: The above listing shows a translation of the triple definition using the `RDF_MATCH()` table function into SQL.

The general processing schema using the `RDF_MATCH()` function is based on self-joins on the statement table. Since the runtime of queries increases with the size of this statement table, Chong et al. propose using the built-in materialized join-view functionality of the underlaying database management system. Therefore, they allow defining a set of materialized views in the form of `subject-subject`, `subject-property`, `subject-object`, `property-property`, `property-object`, and `object-object` as long as the storage requirements are met. In addition to these generic materialized join-views, they propose defining an additional set of subject- property matrix materialized join views, which are basically a adaptation of the previously-described clustered

property tables from Section 2.1.3.1. While this allows to increase the query performance, since for each materialized join in the matrix table one less self-join has to be performed, selecting the optimal properties to build the matrix materialized join view is non-trivial and heavily workload and data dependent.

The advantage of using database inherent materialized views is that they are fully integrated to the tuple life-time process and can thus be automatically dropped and rebuilt if required. Materialized views are as well independent of semantic schema changes, because they only have to be rebuilt in case the RDF model changes and can be considered as secondary storage.

Jena1 [84] simply rewrites SPARQL query to a single SQL query which is then executed over the statement table. In Jena2 [110], it is often impossible to construct one SQL query to satisfy all triple patterns over multiple tables (conjunction of statement and property tables), thus the system first generates a group of SQL queries, one for each set of patterns that can me evaluated with a singe table, and the second containing patterns that span tables. The two groups of queries are then joined in nested loops.

2.2 Native Linked Data Stores

In this section, we first describe native RDF storage and query execution strategies. As mentioned earlier, we define a native RDF systems as a system that was designed to store exclusively RDF data and thus that is fully optimized for persisting and querying this data. Naturally, traditional database design and architecture have had some impact on the design of native RDF stores and thus a number of well-known techniques for example from join processing were adjusted for querying RDF data as we explain in the following.

In the area of native RDF stores, we distinguish three main systems trends: Quadruple stores (Section 2.2.1), Index-Permuted stores (Section 2.2.2) and Graph stores (Section 2.2.3). We first examine systems that store the data in table-like structures, storing additional information per triple, thus maintaining quadruples that keeps information related to the specific sub-graph the triples belong to. Then, we analyze index-permuted storage systems, which use a multiplicity of indexes to support high-performance query execution for arbitrary queries. Finally, we move on to graph stores, which represent one of the most natural ways to represent RDF data. Nodes represent in this context

subjects and objects, while labeled edges are represented by predicates. In order to query a graph, an input query is translated into some graph pattern (see Figure 2.19) that is matched against the full graph. Even for simple patterns, this can result in full traversals of the complete graph and thus may require significant processing time. Due to the fact that graph matching, especially against large graphs, is a very complex and time-consuming task, all existing approached trying to deal with the problem in fact partition graphs into subgraphs. Although here are few different techniques for doing that, we can distinguish two main trends. The first one uses classic graph partitioning algorithms like GGGP [75] used by [23] or METIS used by [71].The second way to tackle graph partitioning is to try to discover recurring patterns/templates in the RDF graph to create subgraphs containing nodes describing certain topics within a defined scope, like what was proposed in [114]. All approaches proposes also a different way of indexing subgraphs, but the general trend is that there is one main index that allows to find certain subgraphs where the remaining part of the data can be found. More specific indices are also proposed for specific types of queries, like Dogma_ipd and Dogma_epd [24].

2.2.1 Quadruple Systems

The traditional way to persist RDF triples is to store triple statements directly in a table-like structure (see above). By exploiting semantic information from the complete RDF graph, additional data can by annotated per triple and stored as a fourth element for each input triple. To improve query execution performance on top of this structure, various indexes can be built. In this section, we present systems and architectures that deal with persisting RDF triples in a most direct way.

2.2.1.1 Data Storage and Partitioning

Virtuoso [44] by Erlin et al. stores data as RDF quads consisting of a graph element id, subject, predicate, and object. All the quads are persisted in one table. Each of the attributes can be indexed in different ways. From a high-level perspective, Virtuoso is comparable to a traditional relational database with enhanced RDF support. Virtuoso adds specific types (URIs, language and type-tagged strings) and indexes optimized for RDF. To partition the data in a clustered environment, Virtuoso uses hash-partitioning based on the subject of the GSPO(Graph/Subject/Predicate/Object) index. Since the

number of resulting partitions is significantly higher than the number of worker nodes in the cluster environment, one node might receive multiple partitions. The distribution of the individual partitions can either be simply round-robin or follow more elaborate models to account for different hardware capacities of the nodes. The system allows moving partitions between nodes and insures data consistency during the process. To provide fault tolerance, Virtuoso allows each logical partition to be placed on multiple nodes.

In [62], Harris et al. propose a system called 4store. The system applies a simple storage model: It stores quads of (model, subject, predicate, object). In 4store, the model attribute is equivalent to Virtuoso's graph. Data is partitioned as non-overlapping sets of records among segments sharing a subject. To distribute segments across the cluster, round-robin is used allowing each node of the cluster to store one or more segments. To cover failing nodes in the cluster, 4store allows to increase the replication of the partitions. The number of replicas in the cluster corresponds to the number of nodes which can fail without causing any significant issue.

2.2.1.2 Indexing

In Virtuoso, Erling et al. implement two indexes. The default index (set as a primary key) corresponds to GSPO (graph, subject, predicate, object). In addition, it provides an auxiliary bitmap index (OPGS). The former is used to deal with queries where the subject is known, the latter is applied to cases with known object and unknown subject. The indexes are stored in compressed form. As strings are the most common values in the database, for example in URIs, Virtuoso compresses these strings by eliminating common prefixes. The system does not precalculate optimization statistics; instead it samples data at query execution time. It also does not compute the exact statistics but just gets rough numbers of elements and estimates query cost to pick an pppoptimal execution plan.

Harris et al. propose to store each of the quads in three indexes; in addition, they store literal values separately. 4store maintains a hash table of graphs where each entry points to lists of triples in the graph (M-Index in Figure 2.8). Literals are indexed in a separate hash table (R Index in Figure 2.8) and they are represented as (S,P, O/Literal). Finally, they consider two predicate-based indexes, referred to as P-Indices in Figure 2.8. For each predicate, two radix tries are used where the key is either a subject or object, and respectively object or subject and graph are stored as entries. These indices can be used

to select all quads having a given predicate and their subject/object (they hence can be seen as traditional P:OS and P:SO indices).

2.2.1.3 Query Execution

In Virtuoso, Erling et al. build query execution plans as single lookups grouped in nested loop joins. They divide query execution into multiple steps where each step takes as input the output from the previous step. Their query execution plan can hence be seen as a pipeline of steps. Most of the steps are individually executed. Sometimes, steps can be joined and executed as a unit. Most queries use predicate indices (P-Indices) in order to merge elements:

```
{? a ub:Professor . ?x teacher_of <student> }
```

The query is executed as an intersection of elements from P-Indices (P:OS) a and $< teacher\_of >$; respectively, elements for $< Professor >$ are merged with elements related to $< student >$.

```
{ ?x a ub:Professor . ?x teaches_course ?c }
```

The second query is executed as a loop through Professor's (bitmap index); then, courses given by each professor are retrieved (from the main index).

```
select * from <lubm> where
{ ?x a ub:Professor . ?x ub:AdvisorOf ?y }
```

The full query is executed in four steps, one to translate URIs to IDs, a second one to get professors, a third one for students the professors advise, and a last one to translate results form IDs into strings.

```
select * from <lubm> where
{ ?x a ub:Professor ; ub:advisorOf ?y ; ub:telephone ?tel }
```

The last query is also executed in four steps since the two properties are retrieved at the same tame, they are co-located because they have the same subject (GSPO partitioned on subject).

2.2.2 Index Permuted Stores

The approach of index-permuted RDF storage exploits and optimizes traditional indexing techniques for storing RDF data. As most of the identifiers in RDF are URIs strings,

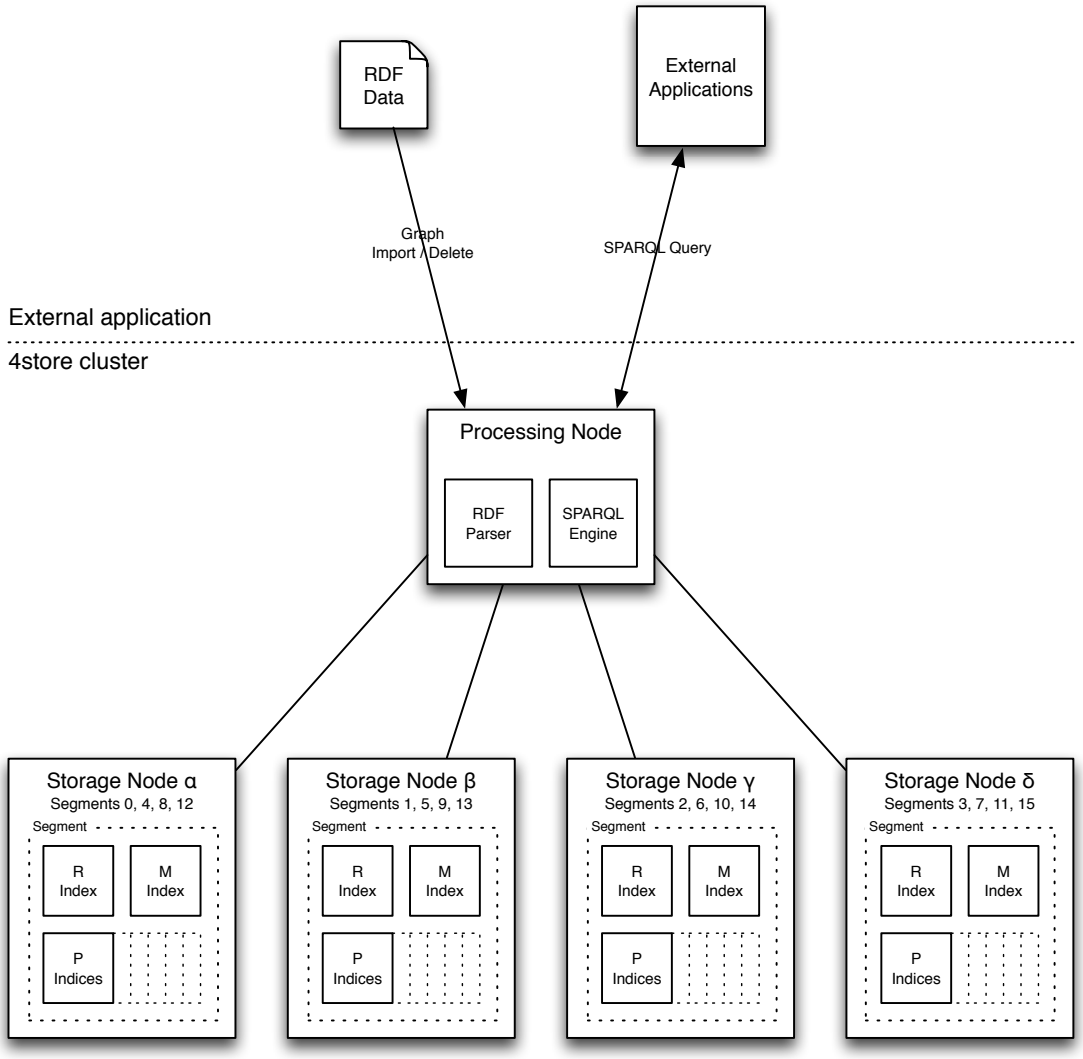


FIGURE 2.8: 4Store: System Architecture [62]

one optimizations is to replace these arbitrary long strings with unique integers. As the data is sparse and many URIs are repetitive, this technique, allows to save memory. To increase the resulting performance, the indexes are built based on shorter encoded values rather than the uncompressed values.

2.2.2.1 Indexing and Data Storage

One of the first approaches to exhaustive indexing was proposed by Harth et al. in [63] for a system called YARS. The authors take into consideration quads of (Subject, Predicate, Object, Context). Exhaustive indexing based on these attribute requires a total of 16 indexes. Harth et al. propose to use six indexes covering all major access patterns [SPOC, POC,OCS,CSP, CP,OS]. Their indexing approach leverages the property that

Subject	Predicate	Object
:the_matrix	:released_in	"1999"
:the_thirteenth_floor	:released_in	"1999"
:the_thirteenth_floor	:similar_plot_as	:the_matrix
:the_matrix	:is_a	:movie
:the_thirteenth_floor	:is_a	:movie

Distinct subjects: [:the_matrix, :the_thirteenth_floor]
Distinct predicates: [:released_in, :similar_plot_as, :is_a]
Distinct objects: [:the_matrix, "1999", :movie]

	:released_in			:similar_plot_as			:is_a		
:the_matrix	0	1	0	0	0	0	0	0	1
:the_thirteenth_floor	0	1	0	1	0	0	0	0	1

Note: Each bit sequence represents sequence of objects (:the_matrix, "1999", :movie)

FIGURE 2.9: BitMat: sample bit matrix [11]

B+tree indexes can be queried for range and prefix queries. If the key to such index is the full quad of subject, predicate, object and context, it becomes possible to query only a prefix of the key and use the remaining keys as values.

Atre et al. propose a system called BitMat [11] where they store data in compressed inverted index structures. They leverage the fact that RDF triples are fixed 3-dimensional entities. They propose a 3-dimensional bit-cube where each cell represents a unique triple and the cell value denotes the presence or absence of the triple. Figure 2.9 shows some sample RDF data and a corresponding bit matrix. The data is then compressed using D-gap compression[4] on each row level. In this approach first approach, the authors only store S-O matrices, however in their next work [10] they introduce also a transposed matrix of O-P. Furthermore, they also slice out rows along the S and O dimension and store also P-S and P-O matrices.

Janik et al. introduce a system called BRAHMS [73], whose storage model evolves around permuted indexes. They store data in three hash tables (S-OP,O-SP,P-SO). The hash tables are organized in a logically contiguous memory block which can be dumped and loaded from disk during startup and shutdown, though the system itself works in-memory.

[4] `http://bmagic.sourceforge.net/dGap.html`

While the previous section describes strictly relational storage schemas, it is possible to further increase the performance of query execution on RDF data by exhaustive indexing of the data. The challenges for relational storage engines can be summarized as follows: if the database uses a large statement table to store the triples for query evaluation, a large number of self-joins is required. Storing the data in property tables is not a fully contained solution to improve query performance. For queries with bounded predicate values, these patterns can be directly evaluated using filters on the property tables, but unbounded predicates in the triple patterns require expensive union and self-joins to evaluate the patterns.

To overcome these limitations, several pieces of work like [109] and [92] show that it is possible to use a new storage model that applies exhaustive indexing. The foundation for this approach is that any query on the stored data can be answered by a set of indices on the subject, predicates, and objects in different orders, namely all their permutations as shown in Chapter 2.10. In contrast to the concept of property tables where the table is only sorted by the subject[2], this allows fast access to all parts of the triples by sorted lists and fast merge-joins on the elements.

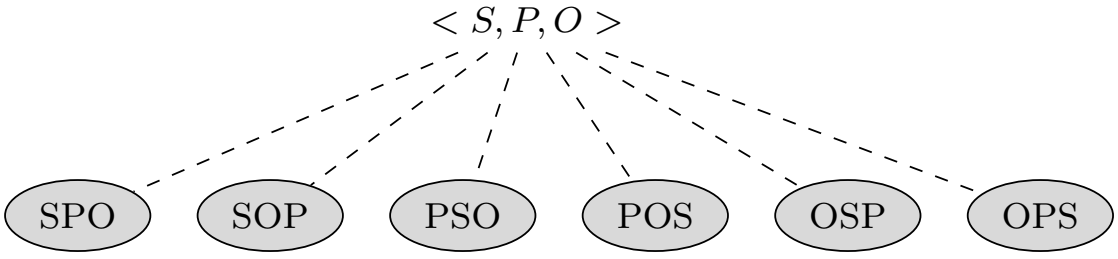


FIGURE 2.10: Exhaustive Indexing

The Hexastore index structure presented in [109] can be described as shown in Chapter 2.11. In this example, a `spo` index is described. The first level of the index is a sorted list of all subjects where each subject is associated to a list of sorted predicates. Each predicate links to a list of sorted objects. Queries that require many joins and unions in other storage systems can be answered directly by the index. In the case where the query requests a list of subjects that are related to two particular objects through any property, the answer can be computed by merging the subject lists of a `osp` index. Since the subject list of this `osp` index is sorted, this can be done in linear time.

The architectural drawback of this approach is the increase in memory consumption. Since every combination of possible query patterns is indexed, additional space is required due to the duplication of data. As the authors of [109] point out, less than a six-fold increase in memory consumption is required; the approach yields a worst-case

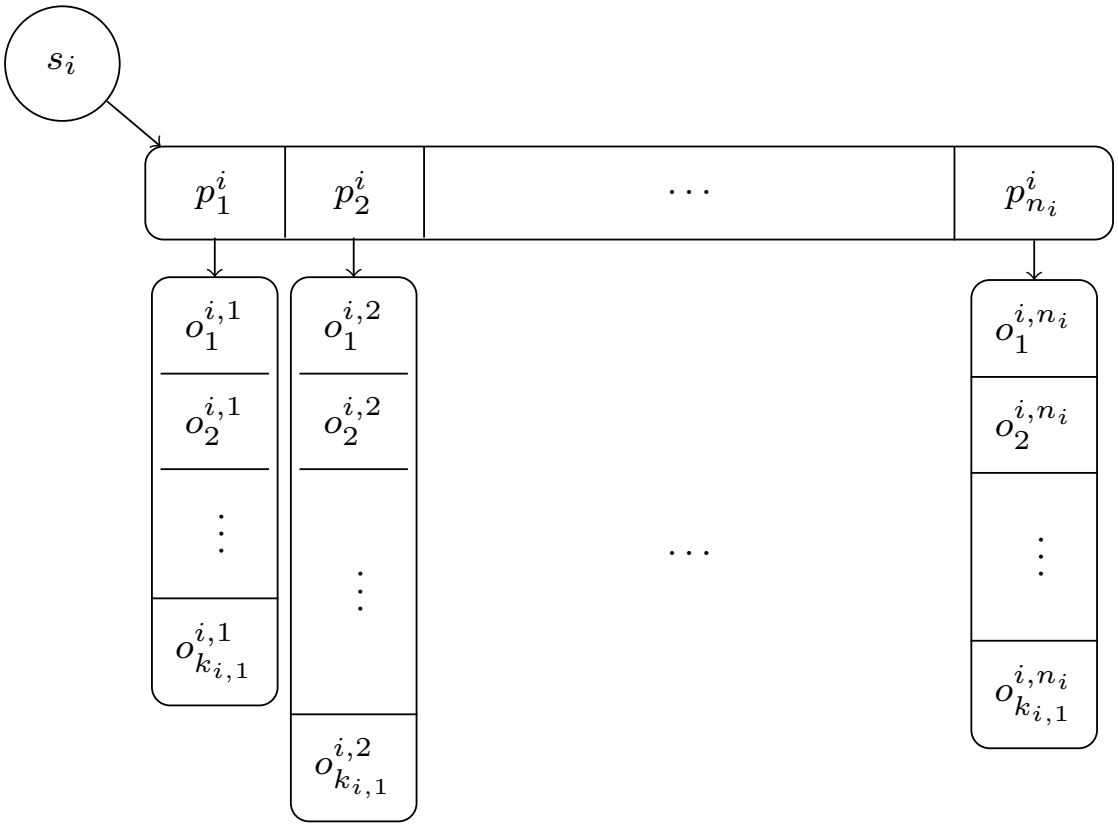


FIGURE 2.11: Hexastore Index Structure, Figure after[109]

five-fold increase since for the set of `spo`, `sop`, `osp`, `ops`, `pso`, `pos` indexes, one part can always be re-used: the initial sorted list of subjects, objects and predicates. Due to the replication of the data into the different index structures, updating and inserting into the index can become a second bottleneck.

Neumann et al. present RDF-3X[90] that relies on the same processing scheme with exhaustive indexing but further optimizes the data structures. As in Hexastore, they use dictionary encoding to replace variable-sized values by fixed integer IDs. In RDF-3X, the index data is stored in clustered B+ trees in lexicographic order.

The values inside the B+ tree are delta encoded to further reduce the required amount of main memory to persist all data. Each triple (in one of the previously defined orders of `spo`,`sop`,...) is stored as a block of maximum 13 bytes. Since the triples are sorted lexicographically, the expected delta is low. Now the header of the value block contains two pieces of information: First a flag that identifies if $value_1$ and $value_2$ are unchanged and the delta of $value_3$ is small enough to fit in the header block; second, if this flag is not set, it then identifies a case number of how many bytes are needed to decode the delta to the previous block. In Chapter 2.12, we illustrate this example. The upper part of the illustration shows the general block structure and the lower half an explicit case.

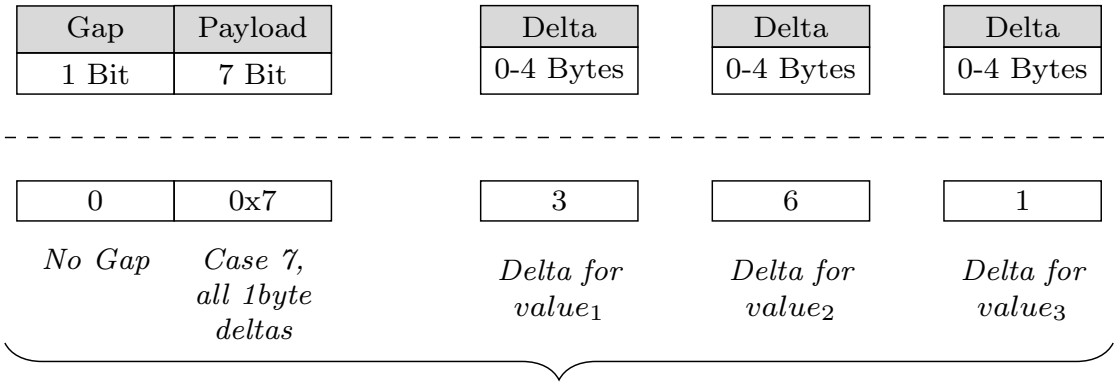


FIGURE 2.12: RDF-3X compression example [90].

Here, the flag is set to 0 meaning more than $value_3$ changed. Case 7 identifies that for $value_1$, $value_2$, and $value_3$ exactly one byte changed. Using this information, the deltas can be extracted and the actual value of the triple can be decoded. In addition to the full index, RDF-3X stores additional aggregated indices to maintain information about how often a relation between two values occurs. This can be used to increase the query performance of those queries where unbound variables are used in the triple pattern, but are not projected and therefore can be used as multipliers for the output of result patterns. These count-aggregated index structure add another nine indexes to the previous indexes, six indexes for all pairs of two values, and three indexes for all single values.

In [94] Owens et al. propose a new storage model for Jena [5] called TDB. The approach stores data in three B+-tree indexes. They use SPO, POS, and OSP permutations. Each index contains all elements of all triples. The string values are encoded as 64bit identifiers.

2.2.2.2 Query Execution

To execute simple queries, Harth et al. [63] evaluate which of the six indexes fits best to answer the query. Selecting the index depends on the access pattern; if a subject is specified in the query, the SPOC index should be used. If only a predicate is given in the query, the POC index should be used instead. More complex queries connected with logical operators require typical relational query optimization like reordering to efficiently execute all kinds of joins.

[5] `https://jena.apache.org/`

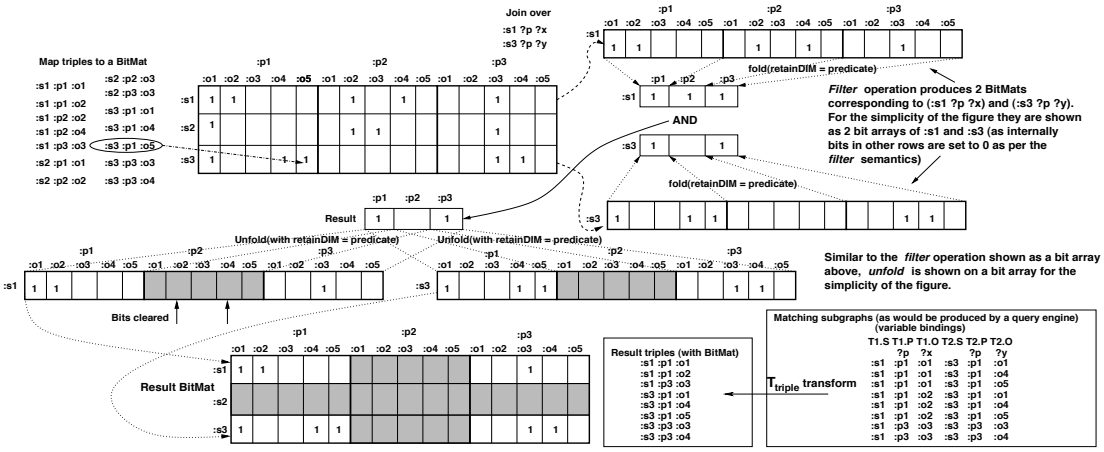


FIGURE 2.13: BitMat: Simple query execution [11]

To execute the queries in BitMat, Atre et al. [11] use bitwise AND and OR operators on rows in their bit matrices, which results in a binary intersection of elements. The operations are performed directly on compressed data. To perform a simple single join query, they first filter by subject rows from matrices containing only triples satisfying a given query patterns (*S1* and *S2* rows in Figure 2.13). The result rows are folded on objects, so that if any object is present for the *SP* pairs, the value is set to 1. In Figure 2.13 shows two rows for *S1* and *S3* (3 cells for each predicate *P1,P2,P3*) without any specified object. In the example (Figure 2.13) all cells have their value set to 1 for both subjects. Only the pair *S3* and *P2* are set to 0, because in the previously selected row for *S2*, for all objects related to *P2* we had value of 0, i.e., there is no triple (*S1,P2,X*). The following step performs AND operation on those two rows, which results in a row containing 1s for predicates which are present for both rows ("Result" in Figure 2.13, *P1* and *P3* are set to 1). Subsequently, the inverse operations are performed, i.e., between the initially selected rows for *S1* and *S3* and the result row from the previous step. An AND operation is applied on cells related to each predicate. This gives two rows for *S1* and *S3* containing the same values for predicates *P1* and *P3*, but cleared values (set to 0) for predicate *P2* and all objects related to *P2*. The two rows are combined it the same way as before with the initial matrix, which gives the final result. The rows for *S1* and *S3* are those which define the result, and since in the first step the *S2* row was not selected its values are cleared.

The authors also propose different algorithm to perform multiple-join operations, where first they create multi-join graphs capturing join variables. Then, for each join variable, they fold matrices associated to all possible triple patterns containing the variable. They

perform bitwise AND on bitarrays. The final result is unfolded and in the end a result BitMat is generated by OR operations on all matrices associated with the triple pattern.

Janik et al. focus in their work [73] on the semantic association discovery problem. The problem itself refers to finding a semantic connection between two objects. Tackling this issue, they had to overcome the fact that SPARQL does not fully support that kind of queries. It supports queries only with fixed distance, whereas to discover association one is interested in any association independently of a distance between objects (arbitrary transitive closures, which were not supported by SPARQL at the time). In BRAHMS, they mainly leveraged two graph algorithms to answer queries: depth-first search and breadth-first search.

TDB [94] divides a query into basic graph patterns [96] ,which are then matched onto the stored RDF data. Subsequently, the other operations are executed by replacing all known values for the variables. This is optimized by favoring elements that are expected to yield the fewest elements, based on statistics. Matching triple patterns is performed by choosing the most appropriate index. The system then performs a range scan of the index for finding particular elements.

2.2.3 Graph-Based Systems

RDF naturally forms graph structures, hence one way to store and process it is through graph-driven data structures and algorithms. Many graph algorithms are however known to be very computationally complex. In this section we present approaches trying to apply ideas from graph the graph processing world to efficiently handle RDF data.

2.2.3.1 Data Storage and Partitioning

In TripleT [46] Fletcher et al. introduce the term of atom around which triples are co-located. A key `k`, regardless of its role in the triples, is selected and then all triples where `k` occurs are co-located together it improve data locality. For example, for `k` three buckets are created. One containing pairs (p,o) where `k` is subject, one containing pairs (s,o) where `k` is predicate, one containing pairs (s,p) where `k` is object. All those pairs are sorted. The storage model itself has an index with keys corresponding to subjects and objects.

Prefix: y= http://en.wikipedia.org/wiki/

vID	vLabel	adjList {(eLabel, nLabel)+}
001	y:Abraham_Lincoln	(hasName, “Abraham Lincoln”) (BornOnDate, “1809-02-12”), (DiedOnDate, “1865-04-15”) (DiedIn, y:Washington_D.C)
002	y:Washington_D.C	(hasName, “Washington D.C.”) (FoundYear , “1790”) (rdf:type, y:city)
003	y:United_States	(hasName, “United States”) (hasCapital,y:Washington_D.C) (rdf:type, y:country)
004	y:Reese_Witherspoon	(hasName, “ReeseWitherspoon”) (BornOnDate, “1976-03-22”) (hasCapital, y:New_Orleans,_Louisiana) (rdf:type, y:Actor)
005	y:New_Orleans,_Louisiana	(FoundYear, “1718”), (locatedIn, y:United_States) (rdf:type, y:city)

FIGURE 2.14: gStore: Adjacency List Table [114]

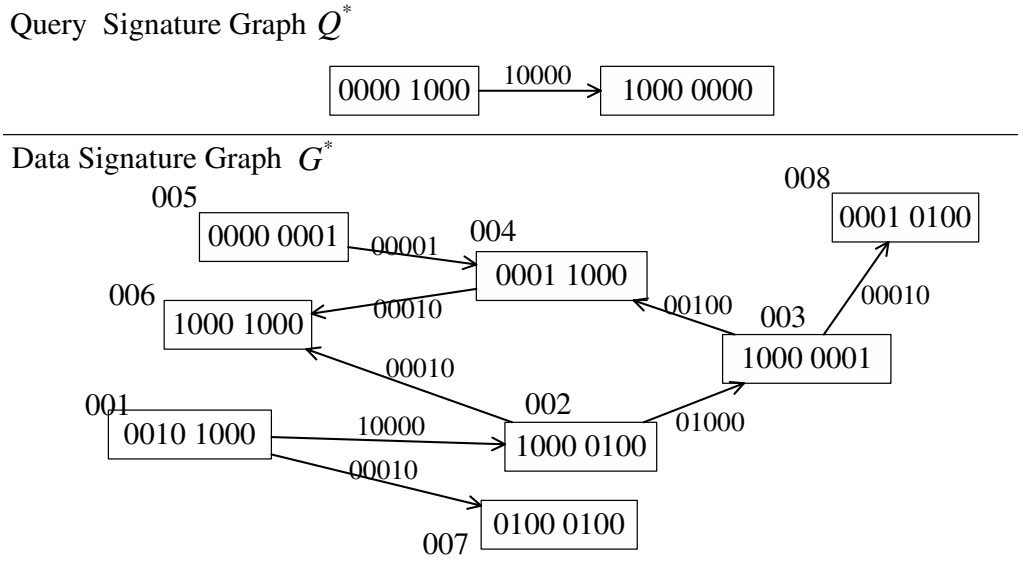


FIGURE 2.15: gStore: Signature Graphs [114]

Das et al., in their system called gStore [114], organize data in adjacency list tables. Each vertex is represented as an entry in the table with a list of outgoing edges and neighbors (Figure 2.14). The entries take the following form [vID,vLabel,adjList], where vID is the ID of the vertex, vLabel is an URI and adjList is a list of outgoing edges and neighbors, which in fact results in shapes that are similar to the atoms proposed in TripleT. As a following step, a bitstring signature is assigned to each vertex (seeFigure 2.15).

Brocheler et al. [24] propose a system called DOGMA. Their approach is based on a balanced binary tree where each node is located on one disk page. Since the page size

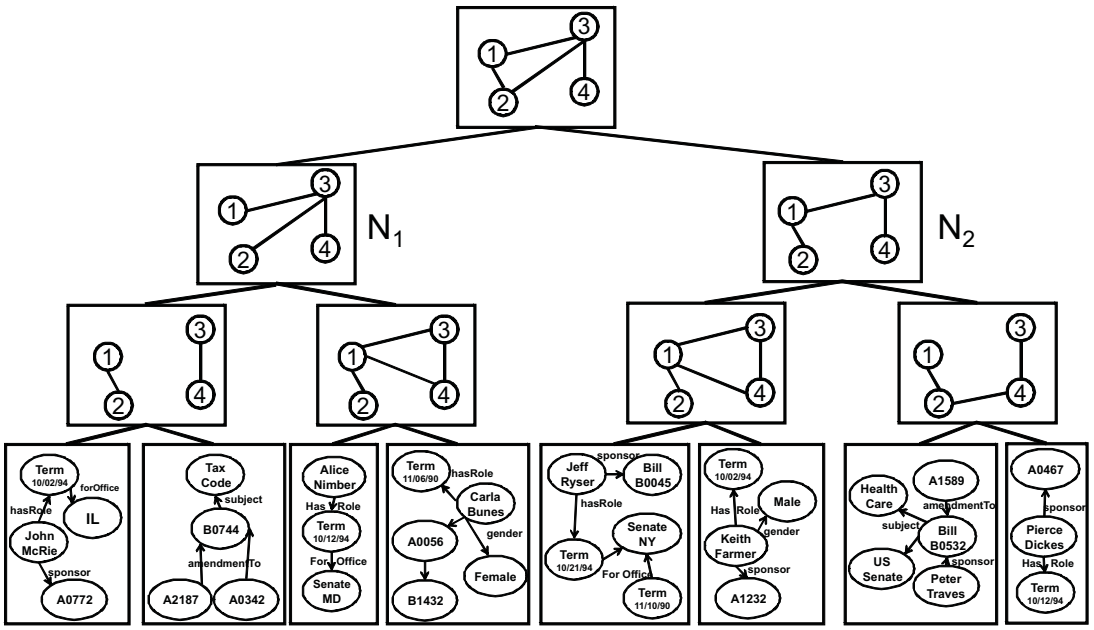


FIGURE 2.16: DOGMA: Index [24]

is fixed, the size of the subgraph located on a page is limited also. There can be many different indexes built for the same RDF database; the authors of DOGMA focuse on minimizing potential cross edges between subgraphs located on different pages. Supposing we have two nodes N1 and N2, the fewer cross edges we have, the more independent the nodes become. When answering a query, we will most probably not have to open/read both of those subgraphs (see Figure 2.16).

To partition graphs, the authors use the algorithm proposed in [75]. Their algorithm takes as input a weighted graph and partitions it in the way that the total weight of cross edges (between subgraphs) is minimized and the sum of the weights in each subgraph is approximately equal. They start by assigning a weight of 1 to each vertex and edge in an RDF graph and then coarsen the graph into a subgraph so that the latter contains about half of the vertices from the former, and so on with each of the subgraphs until each of the subgraphs has no more than the predefined number of vertices (i.e., to fit into a disk page). The coarsening algorithm randomly picks one vertex (v), then selects a maximally-weighted node (m). It merges the neighbors of the node m and m itself into one node, updates the weights and removes v. Edges from m to its neighbors are also removed. This process is rerun until the subgraph contains half or less vertices as the initial graph. Then, the index is built for all subgraphs.

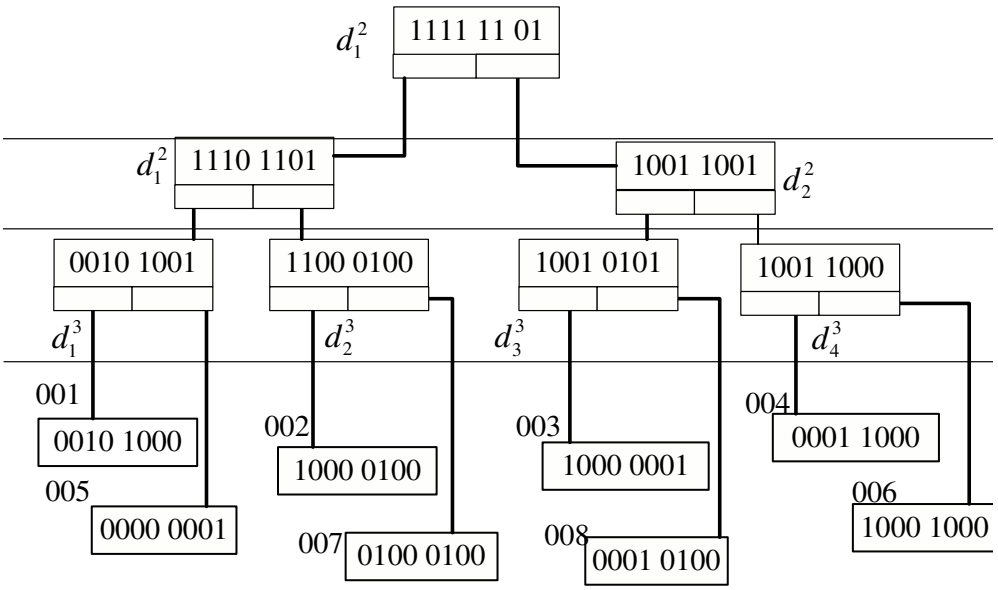


FIGURE 2.17: gStore: S-tree [114]

2.2.3.2 Indexing

Das et al. build an S-tree for all vertices in their adjacency list table to reduce the search space (Figure 2.17). The tree leafs correspond to vertices from the initial graph (G*), and each intermediate (parent) node is formed by superimposing all children signatures (by performing bitwise OR operation). However, S-trees cannot support multi-way join processing; to solve this issue, the authors propose a VS-tree extension. Given an S-tree, leafs are linked according to the initial graph, and new edges are introduced depending on whether certain leafs are connected in G*. Specifically, two leafs in S-tree (001 and 002 in Figure 2.17) are linked if there is an edge in G* between vertices corresponding to them. On the upper level in S-tree, super-edges are introduced between nodes if there is at least one connection between the children of those nodes. In other words, if there is a link between two leafs which does not share a parent, a link between their parents is then created. Bitwise "O" operations over connecting edge labels of the children are performed to assign labels to such super-edges (see Figure 2.18).

In the approach proposed by Brocheler et al., the storage model itself is an index mostly, though the authors also propose two additional indexes to help pruning the result candidates. The DOGMA internal partition distance (IPD) index stores, for each vertex v in node N, the distance to edge of the subgraph corresponding to N. During query

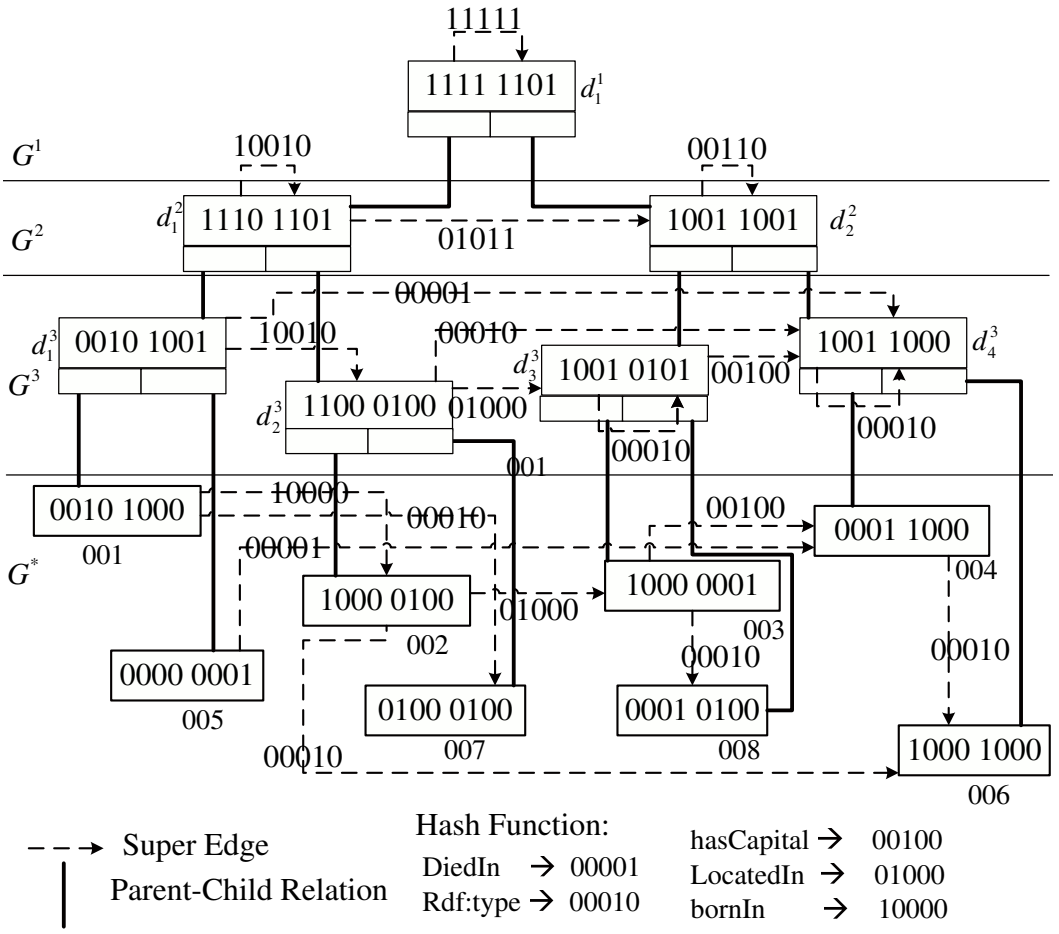


FIGURE 2.18: gStore: VS-tree [114]

execution, for two vertices (v, u) the algorithm looks for nodes for which the vertices belong (N!=M). N and M are at the same level of the tree and closest to the root. If such nodes do not exist, because the vertices are in the same leaf node of the tree, then the distance between them is set to 0, otherwise it is set to the maximal distance from each of them to the border of the subgraph the vertex belongs to (formally: $d(u, v) = max(ipd(v, N), ipd(u, M))$). The idea behind the DOGMA external partition distance (EPD) index is to maintain distances to other subgraphs. For each lowest-level subgraph, a color is assigned. For each vertex and color, the shortest distance from v to a subgraph colored with c is stored. In Figure 2.19 we illustrate how this method can be used to further prune result candidates. Basically, if the distance to the subgraph where a candidate lies is bigger than the distance between constant and variable vertices, the candidate can be pruned.

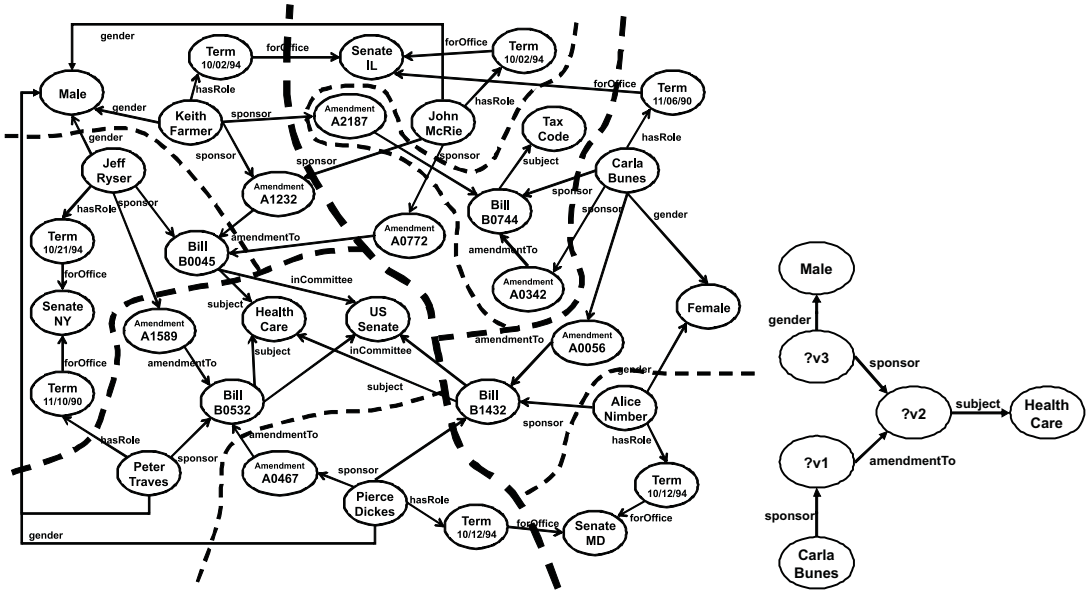


FIGURE 2.19: DOGMA: Example RDF graph (a) and query (b) [24]

2.2.3.3 Query Execution

In TripleT, [46] Fletcher et al. try to minimize complex subject-object joins which in table-oriented systems involve many self-join operations. Thanks to their indexing scheme, they can perform a join as a single look-up on a common join variable (same value for subject and object) and then merge values related to the subjects and the objects.

To answer queries, gStore employs a top-down strategy over a VS-tree to find the match of a query (Q*) over a graph (G*). First, the system finds the top-matches of Q* in the VS-tree, and queues those matches. Then, it pops up one match from the queue and expands it to its children (all descendant and the node) and for each of them checks if it matches Q*. All valid matches are queued back again. This process is iterated until reaching the leaf entries of the VS-tree. Finally, the system finds matches of Q* over leaf entries in VS-tree, i.e., matches of Q* over G*.

To answer a query Brocheler et al. in DOGMA first retrieve for all variable vertices in Q* a set of result candidates w.r.t the vertices (see Figure 2.20). The sets are initialized with vertices that are connected to a defined vertex with a defined predicate. Then, for the vertex with the lowest cardinality of result candidates, each candidate is set as a value of the vertex, such that there is a new constant vertex. The algorithm can be

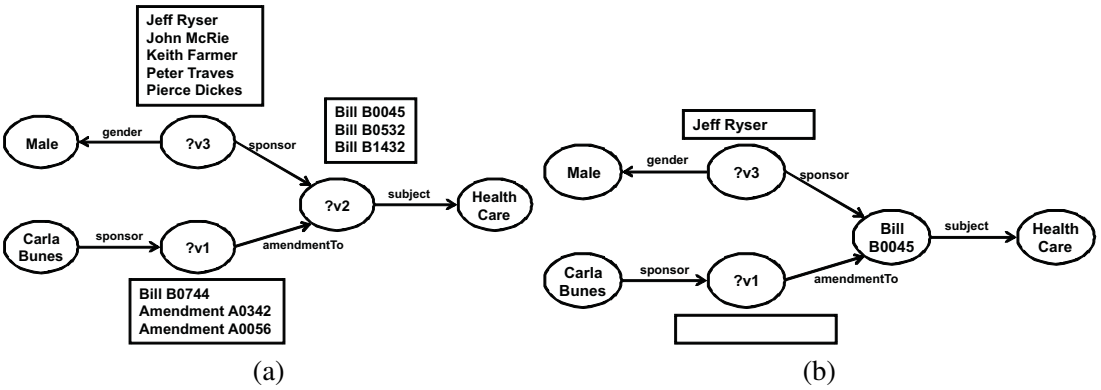


FIGURE 2.20: DOGMA: Execution of DOGMA_basic [24]

rerun to prune result candidates for other vertices, and so on until the final result is found. The basic algorithm presented above is efficient enough for simple queries on neighboring vertices. Considering vertices located in different nodes, additional indices to help prune the result candidates would be needed. The authors propose a second algorithm, which verifies if two vertices are "in range". Let v be a variable vertex with set of result candidates, and c a constant vertex with a long range dependency on v. Then any result candidate of v must not be further away from c than the distance between c and v in the query (more formally $d(v, c) >= d(T(v), s)$). Any other candidate can be pruned. While the result candidates are initialized, the algorithm ensures that each element satisfies this constraint. To efficiently look up for a $d(v, c)$, a distance index is introduced through two lower-bound distance indexes (see Section 2.2.3.2).

2.3 Massively Parallel Processing for Linked Data

With ever larger data sets, distributing RDF data across multiple nodes becomes an important requirement. Instead of designing and implementing custom distributed RDF storage systems, one approach is to reuse existing infrastructure like Hadoop MapReduce and the Hadoop File System.

MapRedude is specifically designed to process large amounts of data. Processing RDF data with MapReduce based on a relational table-like storage model can be very demanding due to possibly high numbers of joins in RDF queries. If the joins produce large intermediate results, these must be distributed across the executor nodes requiring additional storage and network traffic. However, the advantage of Hadoop MapReduce

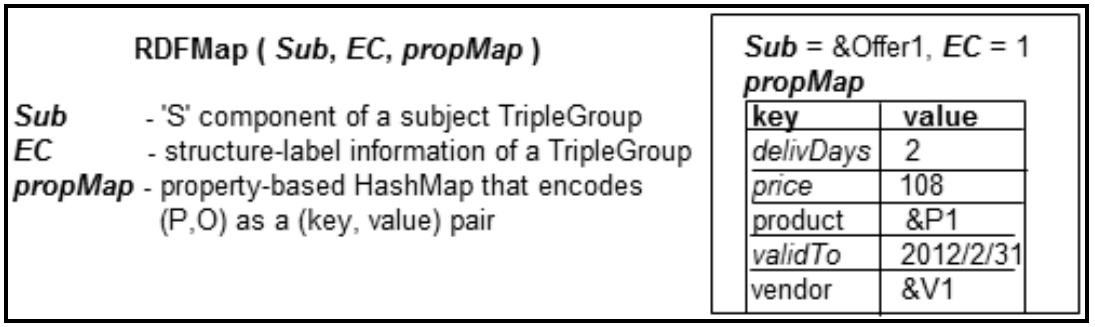


FIGURE 2.21: RAPID: RDFMap representing a TripleGroup [101]

and HDFS is that both systems are established on proven infrastructure systems being able to scale to thousands of nodes and almost arbitrary dataset sizes. As a consequence, optimizing data storage and query execution becomes a challenging and interesting aspect of native RDF database systems. The goal of this section is to present systems that leverage MapReduce and HDFS for large scale RDF storage and query execution.

2.3.1 Data Storage and Partitioning

Rohloff et al. in their work [101] propose a system called SHARD. While they do not introduce any novel storage model, they nevertheless expect data to be stored in a specific format (not ordinary triples). In the datafile, they expect each line to correspond to a star-like shape centering around a subject and all edges from this node. The files containing all the data is stored directly on HDFS without any specific partitioning scheme, by exploiting the replication factor of the underlying distributed file system.

```
Kurt owns car0 livesIn Cambridge
car0 a car madeBy Ford madeIn Detroit
Detroit a city
Cambridge a city
```

The example above represents the following triples:

```
Kurt owns car0
Kurt livesIn Cambridge
car0 a car
car0 madeBy Ford
car0 madeIn Detroit
Detroit a city
Cambridge a city
```

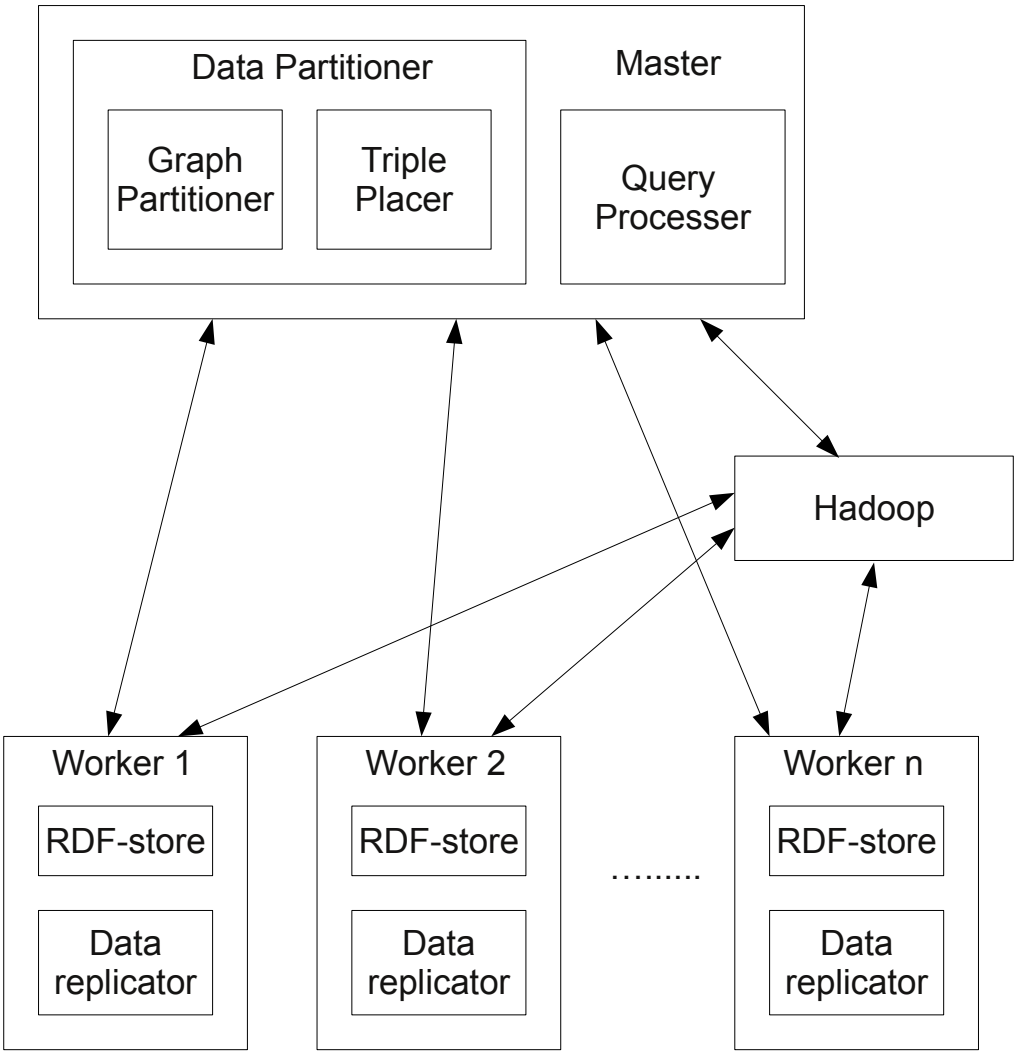


FIGURE 2.22: MapReduce + RDF-3X: System Architecture[71]

Ravindra et al. implement their system (RAPID+) [100] on top of Apache Pig [6]. They leverage a nested HashMap called RDFMap. Data is grouped in TripleGroup (implemented using a native bag data structure from Pig) around a subject which is a first-level key in the map, i.e., the data is co-located for a shared subject which is a hash value in the map. The nested element (i.e., the value from the previous map) (`propMap`) is a hash map with predicates as keys and objects as values. Figure 2.21 shows an example RDFMap. In fact, it forms a star-like substructures around subjects. They are in addition indexed on the first level by subject and on the second level by predicate.

[6] `https://pig.apache.org/`

Huang et al. [71] propose a hybrid solution combining a single node RDF-store (RDF-3X, see above) and Hadoop MapReduce to provide horizontal scalability. To distribute triples across nodes, they leverage the METIS graph partitioning system[7]. Hence, they co-locate triples forming a subgraph (star-like structure) on a particular node. This enables to maximize the number of operations performed in parallel on separate processing nodes avoiding expensive centralized cross-nodes joins. All this allows reducing the amount of data that is transferred over the network for intermediate results. Figure 2.22 shows the architecture of the system.

Data is loaded and partitioned on the master node while triples are distributed among workers. On each node in the Hadoop cluster, there is an installation of the native RDF store which receives and loads subsets of triples. The authors partition graph vertexes so that each worked receives a subset of those vertexes that are close to each other in the graph. Having all vertexes partitioned, the system assigns triples to worker in the way that the triple is placed on the machine if its subject is among vertexes owned by the worker. The process consist in two steps. First, the system divides vertices into disjoint subsets. Then, it assigns triples to workers. Before partitioning vertices, the system removes all triples where the predicate is `rdf:type`. Following this step, the system prepares an input list of edges and vertices (an undirected graph) for METIS. As an output from METIS, the system receives partitions of vertexes that are disjoint. Having all vertexes partitioned, the system starts placing triples on nodes in a cluster. The basic idea is to place a triple on a partition if its subjectis among the vertices assigned to the partition; this forms 1-hop star-like subgraph. This can be extended to further hops so that objects of triples are extended with triples considering them as subjects. The triple placement can also be performed on an undirected graph such that triples containing the vertex assigned to a partition as an object are also placed in it. Both of these extensions are trade-offs between duplicating data on worker nodes and query execution performance (the more extended the sub-graphs are, the less joins have to be performed in the final step).

2.3.2 Query Execution

Ravindra et al. propose an intermediate algebra called Nested Triple Group Algebra (NTGA) to optimize their query execution process [100]. This approach minimizes the number of MapReduce cycles to answer the query. It also introduces algorithms to

[7] `http://glaros.dtc.umn.edu/gkhome/views/metis`

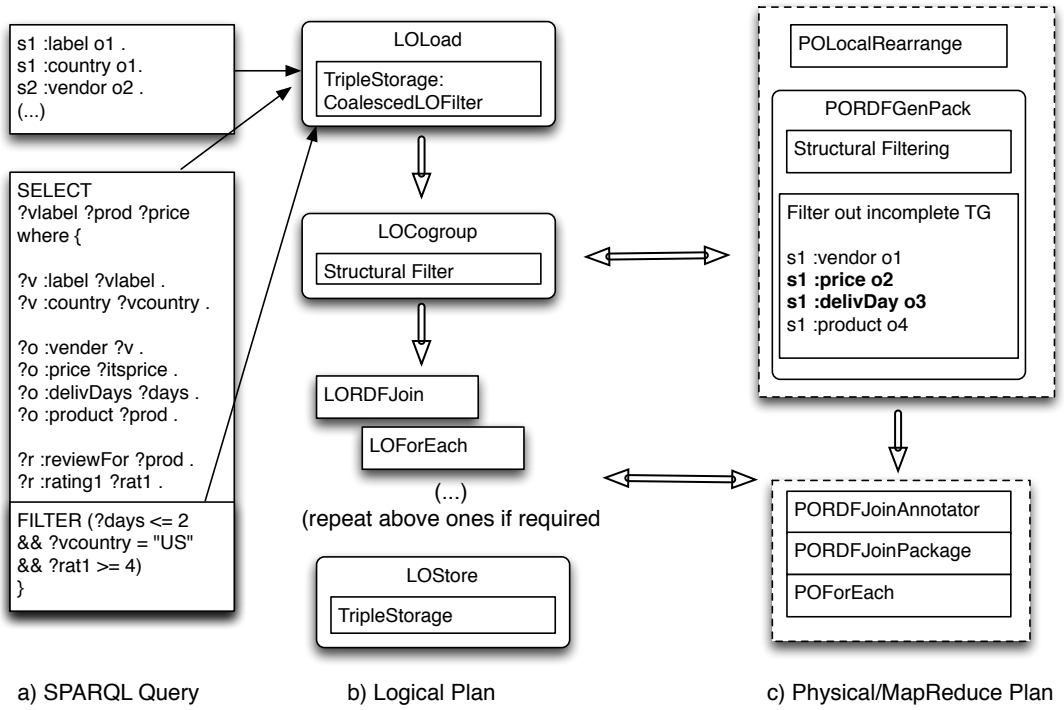


FIGURE 2.23: RAPID+: Query execution in [76]

postpone the decompression of intermediate results so they they can be kept in compact form, thus reducing the number of I/O operations. The fundamental concept of NTGA is a TripleGroup [99], which is a group of triples sharing the same subject or object (star-like structure). Within one MapReduce operation, they pre-compute all possible star substructures, thus materializing all possible first-hop joins. Having computed all star-like structures, the system filters-out those stars that do not fulfill query constrains. In the next step, if necessary, the system joins stars. Figure 2.23 shows an example query and its execution plan. The first step (LOLoad) loads all data and at the same time also applies value-based filters on the data to avoid processing irrelevant triples. Then, during one Reduce operation, the LOCogroup operator groups triples and applies constrains on the groups, such that all irrelevant "stars" are filtered out. The last step in the flow is joining stars based on subjects or objects, which is achieved with the LORDFJoin operator.

Rohloff et al. introduce, in their SHARD system, a clause iteration algorithm [101] the main idea of which is to iterate over all clauses and incrementally bind variables and satisfy constrains (Figure 2.24). During the first step, they identify all edges matching to a clause and remove duplicates. The output collection consists of keys (which are variable bindings from the clause) and NULL values. The following step identifies

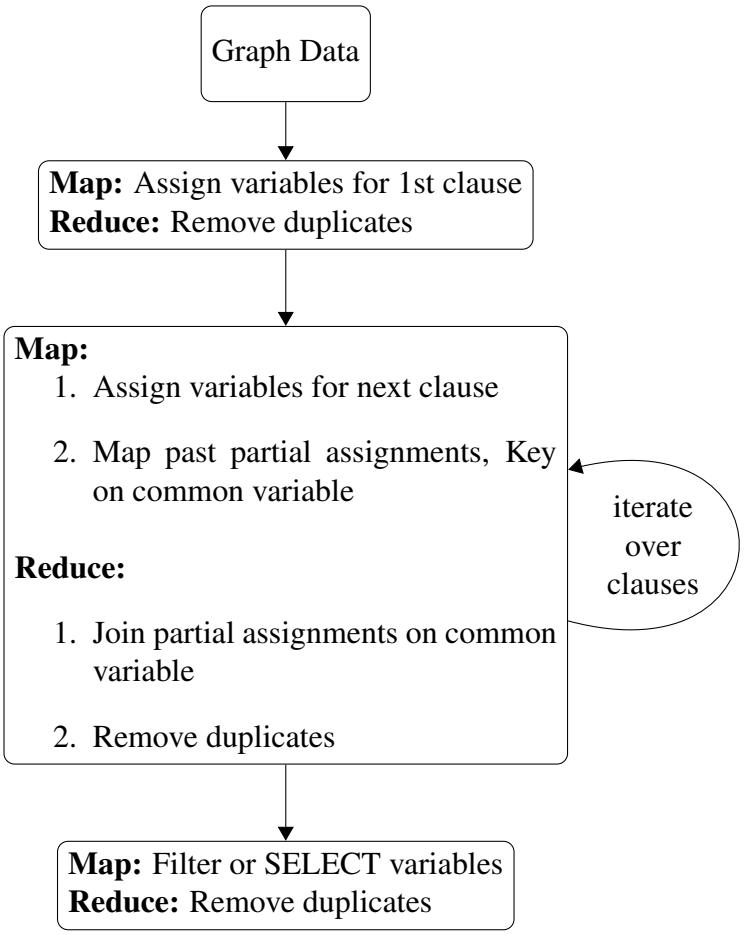


FIGURE 2.24: SHARD: A schema of the clause iteration algorithm [101]

edges matching to the remaining clauses (in the same way as previously). It also joins them with sub-graphs corresponding to the previously matched edges. The final step filters variables to obtain those requested in the SELECT clause. Algorithm 1 runs a `firstClauseMapReduce` MapReduce job to perform the first step. As an output, it returns sets of possible assignments to the variables of the first clause. `boundVars` tracks variables that were bound during this step. For the following example query:

```
SELECT ?person WHERE {
        ?person :owns ?car .
        ?car :a :car .
        ?car :madeIn :Detroit .}
```

for example, variables `?person` and `?car` are bound and set to `boundVars`. The iterating step runs the `intermediateClauseMapReduce` MapReduce job to perform the second step. It identifies triples matching to each clause (one by one) and then performs joins over intermediate results of this step and all previous steps. For instance, after the very first step, the system gets a set of bound variables `<(?car`

Algorithm 1 SHARD: Iteration algorithm [101].

Require: triples, query

1: $mrOutput \longleftarrow NULL$
2: $mrInput \longleftarrow triples$
3: firstClauseMapReduce(mrInput, mrOutput, query.clause(0))
4: $boundVars \longleftarrow query.clause(0).getVars()$
5: **for** $i \leftarrow 1$ **to** $query.numClauses - 1$ **do**
6: $\quad mrInput \longleftarrow union(triples, mrOutput)$
7: $\quad curVars \longleftarrow query.clause(i).getVars()$
8: $\quad comVars \longleftarrow intersection(boundVars, curVars)$
9: $\quad$ intermediateClauseMapReduce(mrInput,mrOutput, query.clause(i), comVars)
10: **end for**
11: $mrInput \longleftarrow mrOutput$
12: selectMapReduce(mrInput, mrOutput, query.select())
13: **return** $mrOutput$

`car0), null>`; after the first iteration, it gets a map of variables bound during the second and first steps `<(?car car0), (?person, Kurt)>`. The reduce phase combines those two and returns `<(?car car0 ?person, Kurt)>`.

Huang et al. [71] take advantage of their partitioning scheme and of their backend triplestore. Queries are decomposed into chunks executed in parallel and then reconstructed with MapReduce. They push as much of query processing as possible to the triplestore while the remaining part is consolidated by Hadoop. The system divides queries into two kinds. First, those that can be executed on a node, meaning that each node has sufficient data to generate complete result tuples. The second kind of query has to be decomposed into sub-queries executed on nodes, whose results are finally collected and joined at the master node.

Chapter 3

An Empirical Evaluation of NoSQL Systems to Manage Linked Data

In the previous chapter we presented in details the most significant scientific approaches to manage Linked Data. In this chapter we show the results of an empirical evaluation of different approaches to process Linked Data regrouped under the NoSQL umbrella, they were not specifically tailored to handle Linked Data in the first place, though they were adapted. Our choice of systems was based on two factors: (i) Developing and optimizing a full-fledged Linked Data management layer on top of a NoSQL system is a very time-consuming task. For our work, we selected systems that were already in development; and (ii) we chose systems that represent a variety of NoSQL system types: document databases (CouchDB), key-value/column stores (Cassandra, HBase), and query compilation for Hadoop (Hive). In addition, we also provide results for 4store, which is a well-known and native RDF store. We use the notation (spo) or SPO to refer to the subject, predicate, and object of the RDF model. Question marks denote variables. All the systems evaluated in this chapter are available online [1].

3.1 Systems

We now turn to brief descriptions of the five systems used in our tests, focusing on the modifications and additions needed to support RDF.

[1] `http://ribs.csres.utexas.edu/nosqlrdf/`

3.1.1 4store

We use 4store[2] as a baseline, native, and distributed RDF DBMS. 4store stores RDF data as quads of (model, subject, predicate, object), where a model is analogous to a SPARQL graph. URIs, literals, and blank nodes are all encoded using a cryptographic hash function. The system defines two types of computational nodes in a distributed setting: (i) storage nodes, which store the actual data, and (ii) processing nodes, which are responsible for parsing the incoming queries and handling all distributed communications with the storage nodes during query processing. 4store partitions the data into non-overlapping segments and distributes the quads based on a hash-partitioning of their subject.

Schema
Data in 4store is organized as property tables [62]. Two radix-tree indices (called P indices) are created for each predicate: one based on the subject of the quad and one based on the object. These indices can be used to efficiently select all quads having a given predicate and subject/object (they hence can be seen as traditional P:OS and P:SO indices). In case the predicate is unknown, the system defaults to looking up all predicate indices for a given subject/object.

4store considers two auxiliary indices in addition to P indices: the lexical index, called R index, stores for each encoded hash value its corresponding lexical (string) representation, while the M index gives the list of triples corresponding to each model. Further details can be found in [62].

Querying
4store's query tool (4s-query) was used to run the benchmark queries.

3.1.2 Jena+HBase

Apache HBase[3] is an open source, horizontally scalable, row consistent, low latency, random access data store inspired by Google's BigTable [27]. It relies on the Hadoop Filesystem (HDFS)[4] as a storage back-end and on Apache Zookeeper[5] to provide support for coordination tasks and fault tolerance. Its data model is a column oriented,

[2] `http://4store.org/`
[3] `http://hbase.apache.org/`
[4] `http://hadoop.apache.org/hdfs`
[5] `http://zookeeper.apache.org/`

sparse, multi-dimensional sorted map. *Columns* are grouped into *column families* and timestamps add an additional dimension to each cell. A key distinction is that *column families* have to be specified at schema design time, while columns can be dynamically added.

There are a number of benefits in using HBase for storing RDF. First, HBase has a proven track-record for scaling out to clusters containing roughly 1000 nodes.[6] Second, it provides considerable flexibility in schema design. Finally, HBase is well integrated with Hadoop, a large scale MapReduce computational framework. This can be leveraged for efficiently bulk-loading data into the system and for running large-scale inference algorithms [108].

Schema

The HBase schema employed is based on the optimized index structure for quads presented by Harth et al. [63] and is described in detail in [48]. In this evaluation, we use only triples so we build 3 index tables: *SPO*, *POS* and *OSP*. We map RDF URIs and most literals to 8-byte ids and use the same table structure for all indices: the row key is built from the concatenation of the 8-byte ids, while column qualifiers and cell values are left empty. This schema leverages lexicographical sorting of the row keys, covering multiple triple patterns with the same table. For example, the table *SPO* can be used to cover the two triple patterns: subject position bound i.e. *(s ? ?)*, subject and predicate positions bound i.e. *(s p ?)*. Additionally, this compact representation reduces network and disk I/O, so it has the potential for fast joins. As an optimization, we do not map numerical literals, instead we use a number's Java representation directly in the index. This can be leveraged by pushing down SPARQL filters and reading only the targeted information from the index. Two dictionary tables are used to keep the mappings to and from 8-byte ids.

Querying

We use Jena as the SPARQL query engine over HBase. Jena represents a query plan through a tree of iterators. The iterators, corresponding to the tree's leafs, use our HBase data layer for resolving triple patterns e.g. *(s ? ?)*, which make up a Basic Graph Pattern (BGP). For joins, we use the strategy provided by Jena, which is indexed nested loop joins. As optimizations, we pushed down simple numerical SPARQL filters i.e. filters which compare a variable with a number, translating them into HBase prefix filters on the index tables. We used these filters, together with selectivity heuristics [106], to reorder subqueries within a BGP. In addition, we enabled joins based on ids, leaving

[6] see e.g.,`http://www.youtube.com/watch?v=byXGqhz2N5M`

the materialization of ids after the evaluation of a BGP. Finally, we added a mini LRU cache in Jena's engine, to prevent the problem of redundantly resolving the same triple pattern against HBase. We were careful to disable this mini-cache in benchmarks with fixed queries i.e. DBpedia, so that HBase is accessed even after the warmup runs.

3.1.3 Hive+HBase

The second HBase implementation uses Apache Hive[7], a SQL-like data warehousing tool that allows for querying using MapReduce.

Schema

A property table is employed as the HBase schema. For each row, the RDF subject is compressed and used as the row key. Each column is a predicate and all columns reside in a single HBase column family. The RDF object value is stored in the matching row and column. Property tables are known to have several issues when storing RDF data [2]. However, these issues do not arise in our HBase implementation. We distinguish multi-valued attributes from one another by their HBase timestamp. These multi-valued attributes are accessed via Hive's array data type.

Querying

At the query layer, we use Jena ARQ to parse and convert a SPARQL query into HiveQL. The process consists of four steps. Firstly, an initial pass of the SPARQL query identifies unique subjects in the query's BGP. Each unique subject is then mapped onto its requested predicates. For each unique subject, a Hive table is temporarily created. It is important to note that an additional Hive table does not duplicate the data on disk. It simply provides a mapping from Hive to HBase columns. Then, the join conditions are identified. A join condition is defined by two triple patterns in the SPARQL `WHERE` clause, $(s_1\ p_1\ s_2)$ and $(s_2\ p_2\ s_3)$, where $s_1 \neq s_2$. This requires two Hive tables to be joined. Finally, the SPARQL query is converted into a Hive query based on the subject-predicate mapping from the first step and executed using MapReduce.

[7] `http://hive.apache.org/query`

3.1.4 CumulusRDF: Cassandra+Sesame

CumulusRDF[8] is an RDF store which provides triple pattern lookups, a linked data server and proxy capabilities, bulk loading, and querying via SPARQL. The storage back-end of CumulusRDF is Apache Cassandra, a NoSQL database management system originally developed by Facebook [79]. Cassandra provides decentralized data storage and failure tolerance based on replication and failover.

Schema

Cassandra's data model consists of nestable distributed hash tables. Each hash in the table is the hashed key of a row and every node in a Cassandra cluster is responsible for the storage of rows in a particular range of hash keys. The data model provides two more features used by CumulusRDF: super columns, which act as a layer between row keys and column keys, and secondary indices that provide value-key mappings for columns.

The index schema of CumulusRDF consists of four indices (SPO, PSO, OSP, CSPO) to support a complete index on triples and lookups on named graphs (contexts). Only the three triple indices are used for the benchmarks. The indices provide fast lockup for all variants of RDF triple patterns. The indices are stored in a "flat layout" utilizing the standard key-value model of Cassandra [78]. CumulusRDF does not use dictionaries to map RDF terms but instead stores the original data as column keys and values. Thereby, each index provides a hash based lookup of the row key, a sorted lookup on column keys and values, thus enabling prefix lookups.

Querying

CumulusRDF uses the Sesame query processor[9] to provide SPARQL query functionality. A stock Sesame query processor translates SPARQL queries to index lookups on the distributed Cassandra indices; Sesame processes joins and filter operations on a dedicated query node.

3.1.5 Couchbase

Couchbase is a document-oriented, schema-less distributed NoSQL database system, with native support for JSON documents. Couchbase is intended to run in-memory

[8] `http://code.google.com/p/cumulusrdf/`
[9] `http://www.openrdf.org/`

mostly, and on as many nodes as needed to hold the whole dataset in RAM. It has a built-in object-managed cache to speed-up random reads and writes. Updates to documents are first made in the in-memory cache, and are only later persisted to disk using an eventual consistency paradigm.

Schema

We tried to follow the document-oriented philosophy of Couchbase when implementing our approach. To load RDF data into the system, we map RDF triples onto JSON documents. For the primary copy of the data, we put all triples sharing the same subject in one document (i.e., creating RDF molecules), and use the subject as the key of that document. The document consists of two JSON arrays containing the predicates and objects. To load RDF data, we parse the incoming triples one by one and create new documents or append triples to existing documents based on the triples' subject.

Querying

For distributed querying, Couchbase provides MapReduce views on top of the stored JSON documents. The JavaScript Map function runs for every stored document and produces 0, 1 or more key-value pairs, where the values can be null (if there is no need for further aggregation). The reduce function aggregates the values provided by the Map function to produce results. Our query execution implementation is based on the Jena SPARQL engine to create triple indices similar to the HBase approach described above. We implement Jena's Graph interface to execute queries and hence provide methods to retrieve results based on triple patterns. We cover all triple pattern possibilities with only three Couchbase views, on $(?p?)$ $(??o)$ and $(?po)$. For every pattern that includes the subject, we retrieve the entire JSON document (molecule), parse it, and provide results at the Java layer. For query optimization, similar to the HBase approach above, selectivity heuristics are used.

3.2 Experimental Setting

We now describe the benchmarks, computational environment, and system setting used in our evaluation.

3.2.1 Benchmarks

3.2.1.1 Berlin SPARQL Benchmark (BSBM)

The Berlin SPARQL Benchmark[21] is built around an e-commerce use-case in which a set of products is offered by different vendors and consumers are posting reviews about products. The benchmark query mix emulates the search and navigation patterns of a consumer looking for a given product. Three datasets were generated for this benchmark:

- 10 million: 10,225,034 triples (Scale Factor: 28,850)
- 100 million: 100,000,748 triples (Scale Factor: 284,826)
- 1 billion: 1,008,396,956 triples (Scale Factor: 2,878,260)

3.2.1.2 DBpedia SPARQL Benchmark (DBPSB)

The DBpedia SPARQL Benchmark[88] is based on queries that were actually issued by humans and applications against DBpedia. We used an existing dataset provided on the benchmark website.[10] The dataset was generated from the original DBpedia 3.5.1 with a scale factor of 100% and consisted of 153,737,783 triples.

3.2.2 Computational Environment

All experiments were performed on the Amazon EC2 Elastic Compute Cloud infrastructure[11]. For the instance type, we used m1.large instances with 7.5 GiB of memory, 4 EC2 Compute Units (2 virtual cores with 2 EC2 Compute Units each), 850 GB of local instance storage, and 64-bit platforms.

To aid in reproducibility and comparability, we ran Hadoop's TeraSort [93] on a cluster consisting of 16 m1.large EC2 nodes (17 including the master). Using TeraGen, 1 TB of data was generated in 3,933 seconds (1.09 hours). The data consisted of 10 billion, 100 byte records. The TeraSort benchmark completed in 11,234 seconds (3.12 hours).

[10] `http://aksw.org/Projects/DBPSB.html`
[11] `http://aws.amazon.com/`

Our basic scenario was to test each system against benchmarks on environments composed of 1, 2, 4, 8 and 16 nodes. In addition, one master node was set up as a zookeeper/coordinator to run the benchmark. The loading timeout was set to 24 hours and the individual query execution timeout was set to 1 hour. Systems that were unable to load data within the 24 hour timeout limit were not allowed to run the benchmark on that cluster configuration.

For each test, we performed two warm-up runs and ten workload runs. We considered two key metrics: the arithmetic mean and the geometric mean. The former is sensitive to outliers whereas the effect of outliers is dampened in the latter.

3.2.3 System Settings

3.2.3.1 4store

We used 4store revision v1.1.4. To set the number of segments, we followed the rule of thumb proposed by the authors, i.e., power of 2 close to twice as many segments as there are physical CPU cores on the system. This led to four segments per node. To benchmark against BSBM, we used the SPARQL endpoint server provided by 4store, and disabled its soft limit. For the DBpedia benchmark, we used the standard 4store client (4s-query), also with the soft limit disabled. 4store uses an Avahi daemon to discover nodes, which requires network multicasting. As multicasts is not supported in EC2, we built a virtual network between the nodes by running an openvpn infrastructure for node discovery.

3.2.3.2 Jena+HBase

We used Hadoop 1.0.3, HBase 0.92, and Hive 0.8.1. One zookeeper instance was running on the master for all cases. We provided 5GB of RAM for the region servers, while the rest was given to Hadoop. All nodes were located in the North Virginia and North Oregon region. The parameters used for HBase are available on our website which is listed in Section 3.3. When configuring each HBase table, we took into account the access patterns. As a result, for the two dictionary tables with random reads, we used an 8 KB block size so that lookups are faster. For indices, we use the default 64 KB block size such that range scans are more efficient. We enable block caching for all tables, but we favor caching of the *Id2Value* table by enabling the *in-memory* option. We also

enable compression for the *Id2Value* table in order to reduce I/O when transferring the verbose RDF data.

For loading data into this system, we first run two MapReduce jobs which generate the 8-byte ids and convert numerical literals to binary representations. Then, for each table, we run a MapReduce job which sorts the elements by row key and outputs files in the format expected by HBase. Finally, we run the HBase bulk-loader tool which actually adopts the previously generated files into the store.

3.2.3.3 Hive+HBase

We used the identical setup of HBase like for Jena+HBase. Before creating the HBase table, we identify the split keys such that the dataset is roughly balanced when stored across the cluster. This is done using Hadoop's `InputSampler.RandomSampler`. We use a frequency of 0.1, the number of samples as 1 million, and the maximum sampled splits as 50% the number of original dataset partitions on HDFS. Once the HBase table has been generated, we run a MapReduce job to convert the input file into the HFile format. We likewise run the HBase bulk-loader to load the data in the store. Jena 2.7.4 was used for the query layer.

3.2.3.4 CumulusRDF (Cassandra+Sesame)

For CumulusRDF, we ran Ubuntu 13.04 loaded from Amazon's Official Machine Image. The cluster consisted of one node running Apache Tomcat with CumulusRDF and a set of nodes with Cassandra instances that were configured as one distributed Cassandra cluster. Depending on the particular benchmark settings, the size of the Cassandra cluster varied.

Cassandra nodes were equipped with Apache Cassandra 1.2.4 and a slightly modified configuration: a uniform cluster name and appropriate IP configuration were set per node, the location of directories for data, commit logs, and caches were moved to the local instance storage. All Cassandra instances equally held the maximum of 256 index tokens since all nodes ran on the same hardware configuration. The configuration of CumulusRDF was adjusted to fit the Cassandra cluster and keyspace depending on the particular benchmark settings. CumulusRDF's bulk loader was used to load the benchmark data into the system. A SPARQL endpoint of the local CumulusRDF instances was used to run the benchmark.

3.2.3.5 Couchbase

Couchbase Enterprise Edition 64 bit 2.0.1 was used with default settings and 6.28 GB allocated per node. The Couchbase java client version was 1.1.0. The NxParser version 1.2.3 was used to parse N-Triples and json-simple 1.1 to parse JSON. The Jena ARQ version was 2.9.4.

3.3 Performance Evaluation

Figure 3.1 and 3.2 show a selected set of evaluation results for the various systems. Query execution times were computed using a geometric average. For a more detailed list of all cluster, dataset, and system configurations, we refer the reader to our website.[12] This website contains all results, as well as our source code, how-to guides, and EC2 images to rerun our experiments. We now discuss the results with respect to each system and then make broader statements about the overall experiment in the conclusion.

Table 3.1 shows a comparison between the total costs incurred on Amazon for loading and running the benchmark for the BSBM 100 million, 8 nodes configuration. The costs are computed using the formula:

$$(1+8)nodes * \$0.240/hour * (loading\_time + benchmark\_time)$$

where the loading and benchmark time are in hours. All values are in U.S. dollars and prices are listed as of May 2013. Exact costs may vary due to hourly pricing of the EC2 instances.

TABLE 3.1: Total Cost – BSBM 100 million on 8 nodes

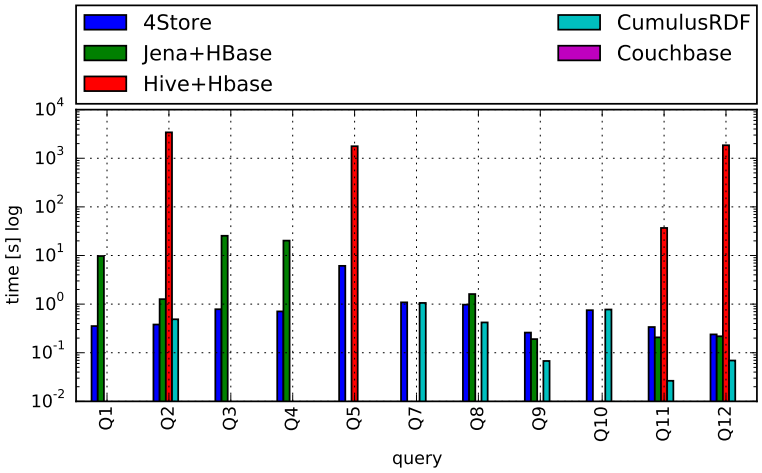
4store	Jena+HBase	Hive+Hbase	CumulusRDF	Couchbase
$1.16	$35.80	$81.55	$105.15	$86.44

3.3.1 4store

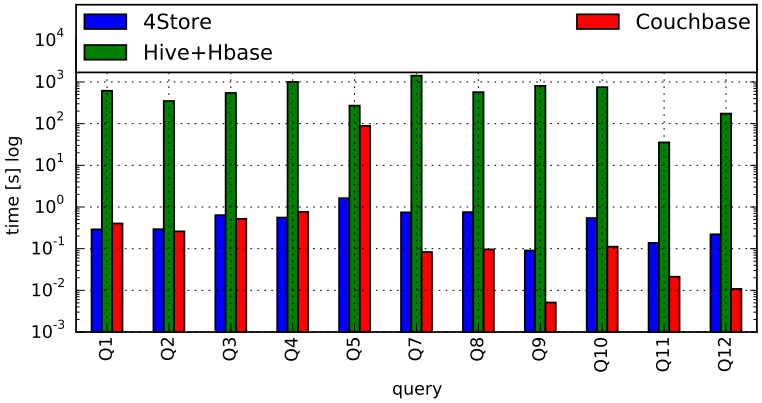
4store achieved sub-second response times for BSBM queries on 4, 8, and 16 nodes with 10 and 100 million triples. The notable exception is Q5, which touches a lot of data and contains a complex FILTER clause. Results for BSBM 1 billion are close to

[12] `http://ribs.csres.utexas.edu/nosqlrdf`

Berlin Sparql Benchmark (BSBM), 1 billion triples, 16 nodes



Berlin Sparql Benchmark (BSBM), 100 million triples, 16 nodes



Berlin Sparql Benchmark (BSBM), 100 million triples, 1 node

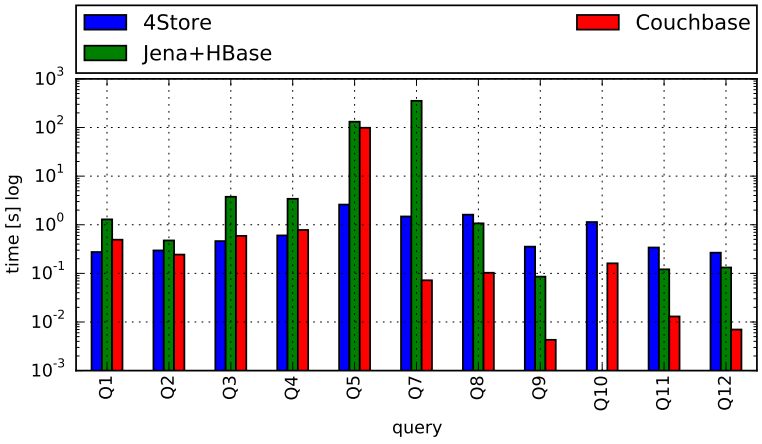
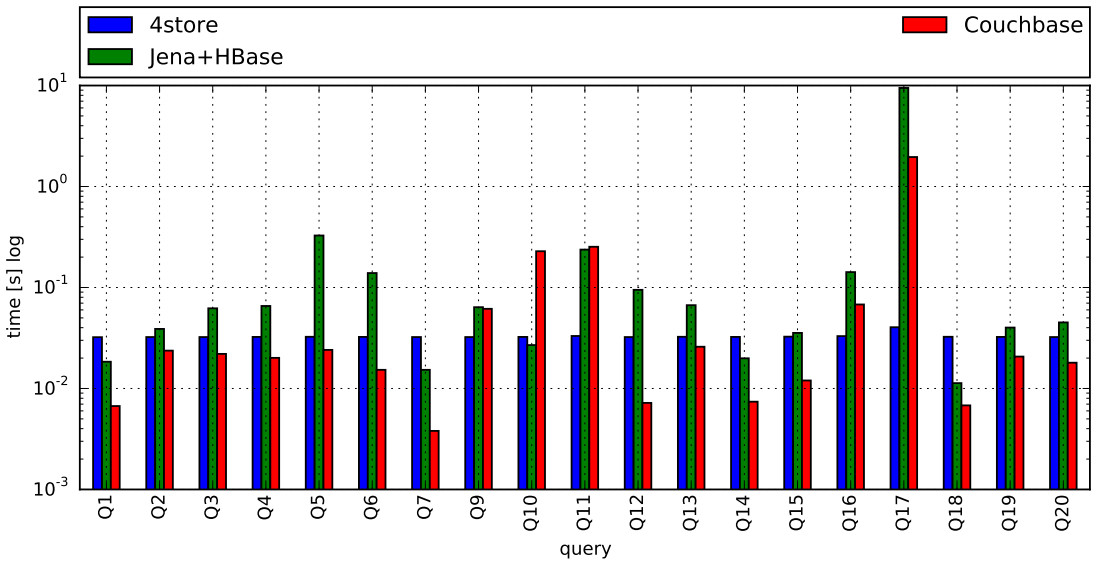


FIGURE 3.1: Results for BSBM showing 1 billion and 100 million triples datasets run on a 16 node cluster. Results for the 100 million dataset on a single node are also shown to illustrate the effect of the cluster size.



Loading Time for DBpedia SPARQL Benchmark, 150 million triples

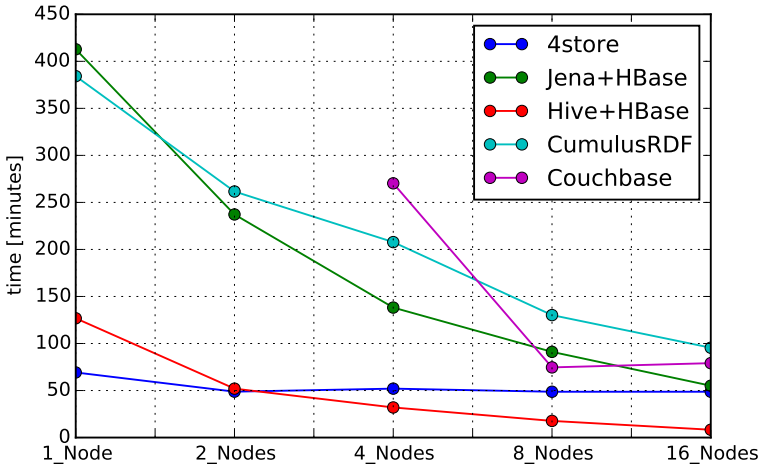


FIGURE 3.2: Results for the DBpedia SPARQL Benchmark and loading times.

1 second, except again for Q5 which takes between 6 seconds (16 nodes) and 53 seconds (4 nodes). Overall, the system scales for BSBM as query response times steadily decrease as the number of machines grow. Loading takes a few minutes, except for the 1 billion dataset which took 5.5 hours on 16 nodes and 14.9 hours on 8 nodes. Note: 4store actually times out when loading 1 billion for 4 nodes but we still include the results to have a coherent baseline.

Results for the DBpedia SPARQL Benchmark are all in the same ballpark, with a median around 11 seconds when running on 4 nodes, 19 seconds when running on 8, and 32 seconds when running on 16 nodes. We observe that the system is *not* scalable in

this case, probably due to the high complexity of the dataset and an increase in network delays caused by the excessive fragmentation of DBpedia data stored as property tables on multiple machines.

3.3.2 Jena+HBase

This implementation achieves sub-second query response times up to a billion triples on a majority of the highly selective query mixes for BSBM (Q2, Q8, Q9, Q11, and Q12). This includes those queries that contain a variable predicate. It is not relevant whether we have an inner or an outer join, instead results are greatly influenced by selectivity. For low selectivity queries (Q1, Q3, Q10), we see that leveraging HBase features is critical to even answer queries. For Q1 and Q3, we provide results for all dataset sizes. These two queries make use of numerical SPARQL filters which are pushed down as HBase prefix filters, whereas with Q10 we are unable to return results as it contains a date comparison filter which has not been pushed down. The importance of optimizing filters is also shown when a query touches a lot of data such as Q5, Q7 – both of which contain complex or date specific filters.

In terms of the DBpedia SPARQL Benchmark, we see sub-second response times for almost all queries. One reason is that our loading process eliminates duplicates, which resulted in 60 million triples from the initial dataset being stored. In addition, the queries tend to be much simpler than the BSBM queries. The one slower query (Q17) is again due to SPARQL filters on strings that could were not implemented as HBase filters. With filters pushed into HBase, the system approaches the performance of specially designed triple stores on a majority of queries. Still, we believe there is space for improvement in the join strategies as currently the “off-the-shelf” Jena strategy is used.

3.3.3 Hive+HBase

The implementation of Hive atop HBase introduces various sources of overhead for query execution. As a result, query execution times are in the minute range. However, as more nodes are added to the cluster, query times are reduced. Additionally, initial load times tend to be fast.

For most queries, the MapReduce shuffle stage dominates the running time. Q7 is the slowest because it contains a 5-way join and requires 4 MapReduce passes from Hive.

Currently, the system does not implement a query optimizer and it uses the naive Hive join algorithm[13]. For the DBpedia SPARQL Benchmark, we observed faster query execution times than on BSBM. The dataset itself is more sparse than BSBM and the DBpedia queries are simpler; most queries do not involve a join. Due to the sparse nature of the dataset, bloom filters allow us to scan less HDFS blocks. The simplicity of the queries reduce network traffic and also reduce time spent in the shuffle and reduce phases. However, queries with language filters (e.g., Q3, Q9, Q11) perform slower since Hive performs a substring search for the requested SPARQL language identifier).

3.3.4 CumulusRDF: Cassandra+Sesame

For this implementation, the BSBM 1 billion dataset was only run on a 16 node cluster. The loading time was 22 hours. All other cluster configurations exceeded the loading timeout. For the 100 million dataset, the 2 node cluster began throwing exceptions mid-way in the benchmark. We observed that the system not only slowed down as more data was added but also as the cluster size increased. This could be attributed to the increased network communication required by Cassandra in larger clusters. With parameter tuning, it may be possible to reduce this. As expected, the loading time decreased as the cluster size increased.

BSBM queries 1, 3, 4, and 5 were challenging for the system. Some queries exceeded the one hour time limit. For the 1 billion dataset, these same queries timed out while most other queries executed in the sub-second range.

The DBpedia SPARQL Benchmark revealed three outliers: Q2, Q3, and Q20. Query 3 timed out for all cluster sizes. As the cluster size increased, the execution time of query 2 and 20 increased as well. All other queries executed in the lower millisecond range with minor increases in execution time as the cluster size increased.

One hypothesis for the slow performance of the above queries is that, as opposed to other systems, CumulusRDF does not use a dictionary encoding for RDF constants. Therefore, joins require equality comparisons which are more expensive than numerical identifiers.

[13] `https://cwiki.apache.org/Hive/languagemanual-joins.html`

3.3.5 Couchbase

Couchbase encountered problems while loading the largest dataset, BSBM 1 billion, which timed out on all cluster sizes. While the loading time for 8 and 16 nodes is close to 24 hours, index generation in this case is very slow, hampered by frequent node failures. Generating indices during loading was considerably slower with smaller cluster sizes, where only part of the data can be held in main memory (the index generation process took from less than an hour up to more than 10 hours). The other two BSBM data sets, 10 million and 100 million, were loaded on every cluster configuration with times ranging from 12 minutes (BSBM 10 million, 8 nodes) to over 3.5 hours (100 million, 1 node). In the case of DBpedia, there are many duplicate triples spread across the data set, which cause many read operations and thus slower loading times. Also, the uneven size of molecules and triples caused frequent node failures during index creation. For this reason, we only report results with 4 clusters and more, where the loading times range from 74 min for 8 nodes to 4.5 hours for 4 nodes.

Overall, query execution is relatively fast. Queries take between 4 ms (Q9) and 104 seconds (Q5) for BSBM 100 million. As noted above, Q5 touches a lot of data and contains a complex FILTER clause, which leads to a very high number of database accesses in our case, since none of the triple patterns of the query is very restrictive. As the cluster size increases, the query execution times remain relatively constant. Results for DBpedia benchmark exhibit similar trends.

3.4 Conclusions

This work represents, to the best of our knowledge, the first systematic attempt at characterizing and comparing NoSQL stores for Linked Data processing. The systems we have evaluated above all exhibit their own strengths and weaknesses. Overall, we can make a number of key observations:

1. Distributed NoSQL systems can be competitive against distributed and native RDF stores (such as 4store) with respect to query times. Relatively simple SPARQL queries such as distributed lookups, in particular, can be executed very efficiently on such systems, even for larger clusters and datasets. For example, on BSBM 1 billion triples, 16 nodes, Q9, Q11, and Q12 are processed more efficiently on Cassandra and Jena+HBase than on 4store.

2. Loading times for RDF data varies depending on the NoSQL system and indexing approach used. However, we observe that most of the NoSQL systems scale more gracefully than the native RDF store when loading data in parallel.

3. More complex SPARQL queries involving several joins, touching a lot of data, or containing complex filters perform, generally-speaking, poorly on NoSQL systems. Take the following queries for example: BSBM Q3 contains a negation, Q4 contains a union, and Q5 touches a lot of data and has a complex filter. These queries run considerably slower on NoSQL systems than on 4store.

4. Following the comment above, we observe that classical query optimization techniques borrowed from relational databases generally work well on NoSQL RDF systems. Jena+HBase, for example, performs better than other systems on many join and filter queries since it reorders the joins, pushes down the selections and filters in its query plans.

5. MapReduce-like operations introduce a higher latency for distributed view maintenance and query execution times. For instance, Hive+HBase and Couchbase (on larger clusters) introduce large amounts of overhead resulting in slower runtimes.

In conclusion, NoSQL systems represent an alternative to distributed and native RDF stores for simple workloads. Considering the encouraging results from this study, the very large user base of NoSQL systems, and the fact that there is still ample room for query optimization techniques, we are confident that NoSQL databases will present an ever growing opportunity to store and manage Linked Data in the cloud. Nevertheless, native Linked Data systems begin tailored to handle Linked Data and workload have high potential to significantly outperform non-native (e.g. NoSQL) systems on workload of medium and high complexity. Even for queries of medium complexity NoSQL approaches remain sub-optimal and for highly complex queries they cannot provide results. For those reasons in the next chapter we introduce our own native techniques to efficiently handle Linked Data and workload.

Chapter 4

Storing and Querying Linked Data in the Cloud

We showed in the previous chapters that despite recent advances in distributed Linked Data management, processing large-amounts of Linked Data in the cloud is still very challenging. In spite of its seemingly simple data model, Linked Data actually encodes rich and complex graphs mixing both instance and schema-level data. Sharding such data using classical techniques or partitioning the graph using traditional min-cut algorithms leads to very inefficient distributed operations and to a high number of joins. In this chapter, we describe our methods to manage Linked Data in efficient and scalable way in the cloud. Contrary to previous approaches, out algorithm runs a physiological analysis of both instance and schema information prior to partitioning the data. In this chapter, we describe the data structures we use to compactly co-locate data, the new algorithms we use to partition and distribute data, and out query execution strategies to efficiently derive answers of queries in a distributed environment. We also present an extensive evaluation of our methods showing that they are often two orders of magnitude faster than the state-of-the-art approaches on standard workloads. All presented techniques were implemented in the system named DiploCloud.

4.1 Storage Model

Our storage system in DiploCloud can be seen as a hybrid structure extending several of the ideas from above. Our system is built on three main structures: RDF molecule

clusters (which can be seen as hybrid structures borrowing both from property tables and RDF subgraphs), template lists (storing literals in compact lists as in a column-oriented database system) and an efficient key index indexing URIs and literals based on the clusters they belong to.

Figure 4.1 gives a simple example of a few molecule clusters—storing information about students—and of a template list—compactly storing lists of student IDs. Molecules can be seen as *horizontal* structures storing information about a given object instance in the database (like rows in relational systems). Template lists, on the other hand, store *vertical* lists of values corresponding to one *type* of object (like columns in a relational system). Hence, we say that DiploCloud is a *hybrid* system, following the terminology used for approaches such as Fractured Mirrors [98] or our own recent Hyrise system [58].

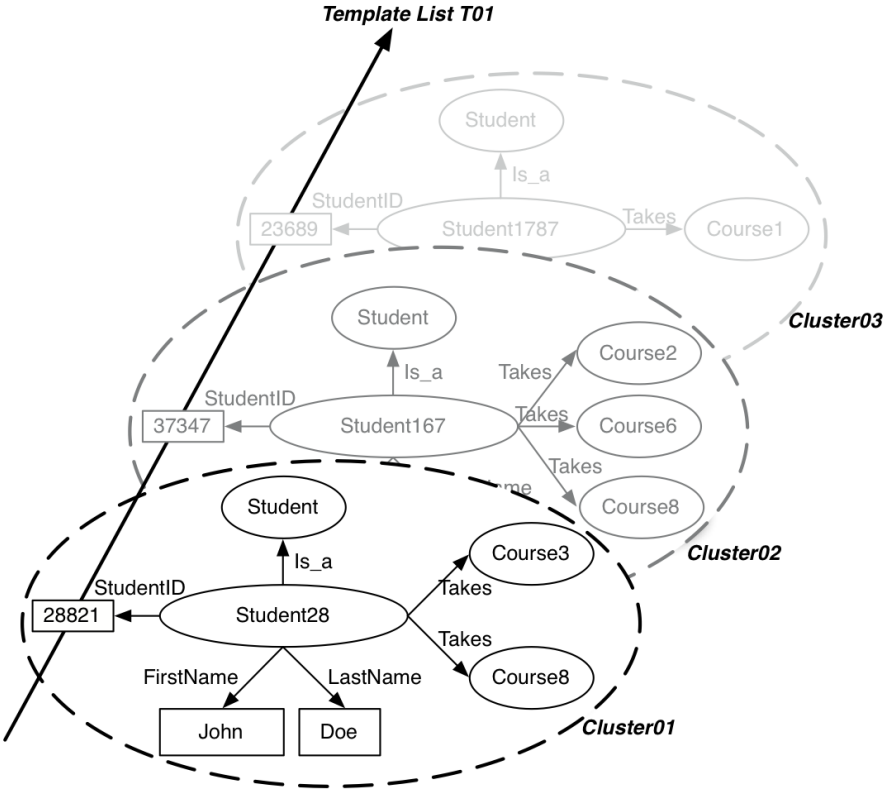


FIGURE 4.1: The two main data structures in DiploCloud: molecule clusters, storing in this case RDF subgraphs about students, and a template list, storing a list of literal values corresponding to student IDs.

Molecule clusters are used in two ways in our system: to logically group sets of related URIs and literals in the hash-table (thus, pre-computing joins), and to physically co-locate information relating to a given object on disk and in main-memory to reduce disk

and CPU cache latencies. Template lists are mainly used for analytics and aggregate queries, as they allow to process long lists of literals efficiently.

4.1.1 Key Index

The Key Index is the central index in DiploCloud; it uses a lexicographical tree to parse each incoming URI or literal and assign it a unique numeric key value. It then stores, for every key and every template ID, an ordered list of all the clusters IDs containing the key (e.g., "key 10011, corresponding to a Course object [template ID 17], appears in clusters 1011, 1100 and 1101"; see also Figure 4.2 for another example). This may sound like a pretty peculiar way of indexing values, but we show below that this actually allows us to execute many queries very efficiently simply by reading or intersecting such lists in the hash-table directly.

The key index is responsible for encoding all URIs and literals appearing in the triples into a unique system id (key), and back. We use a tailored lexicographic tree to parse URIs and literals and assign them a unique numeric ID. The lexicographic tree we use is basically a prefix tree splitting the URIs or literals based on their common prefixes (since many URIs share the same prefixes) such that each substring prefix is stored once and only once in the tree. A key ID is stored at every leaf, which is composed of a type prefix (encoding the type of the element, e.g., $Student$ or $xsd : date$) and of an auto-incremented instance identifier. This prefix trees allow us to completely avoid potential collisions (caused for instance when applying hash functions on very large datasets), and also let us compactly co-locate both type and instance ids into one compact key. A second structure translates the keys back into their original form. It is composed of a set of inverted indices (one per type), each relating an instance ID to its corresponding URI / literal in the lexicographic tree in order to enable efficient key look-ups.

4.1.2 Templates

One of the key innovations of DiploCloud revolves around the use of *declarative storage patterns* [39] to efficiently co-locate large collections of related values on disk and in main-memory. When setting-up a new database, the database administrator may give DiploCloud a few hints as to how to store the data on disk: the administrator can give a list of triple patterns to specify the *root nodes*, both for the template lists and the

molecule clusters (see for instance Figure 4.1, where "Student" is the root node of the molecule, and "StudentID" is the root node for the template list). Cluster roots are used to determine which clusters to create: a new cluster is created for each instance of a root node in the database. The clusters contain all triples departing from the root node when traversing the graph, until another instance of a root node is crossed (thus, one can join clusters based on their root nodes). Template roots are used to determine which literals to store in template lists.

Based on the storage patterns, the system handles two main operations in our system: i) it maintains a schema of triple templates in main-memory and ii) it manages template lists. Whenever a new triples enters the system, it associates template IDs corresponding to the triple by considering the type of the subject, the predicate, and the type of the object. Each distinct list of "(subject-type, predicate, object-type)" defines a new triple template. The triple templates play the role of an instance-based RDF schema in our system. We don't rely on the explicit RDF schema to define the templates, since a large proportions of constraints (e.g., domains, ranges) are often omitted in the schema (as it is for example the case for the data we consider in our experiments, see Section 4.5). In case a new template is detected (e.g., a new predicate is used), then the template manager updates its in-memory triple template schema and inserts new template IDs to reflect the new pattern it discovered. Figure 4.2 gives an example of a template. In case of very inhomogeneous data sets containing millions of different triple templates, wildcards can be used to regroup similar templates (e.g., "Student - likes - *"). Note that this is very rare in practice, since all the datasets we encountered so far (even those in the LOD cloud) typically consider a few thousands triple templates at most.

Afterwards, the system inserts the triple in one or several molecules. If the triple's object corresponds to a root template list, the object is also inserted into the template list corresponding to its template ID. Templates lists store literal values along with the key of their corresponding cluster root. They are stored compactly and segmented in sublists, both on disk and in main-memory. Template lists are typically sorted by considering a lexical order on their literal values—though other orders can be specified by the database administrator when he declares the template roots. In that sense, template lists are reminiscent of *segments* in a column-oriented database system.

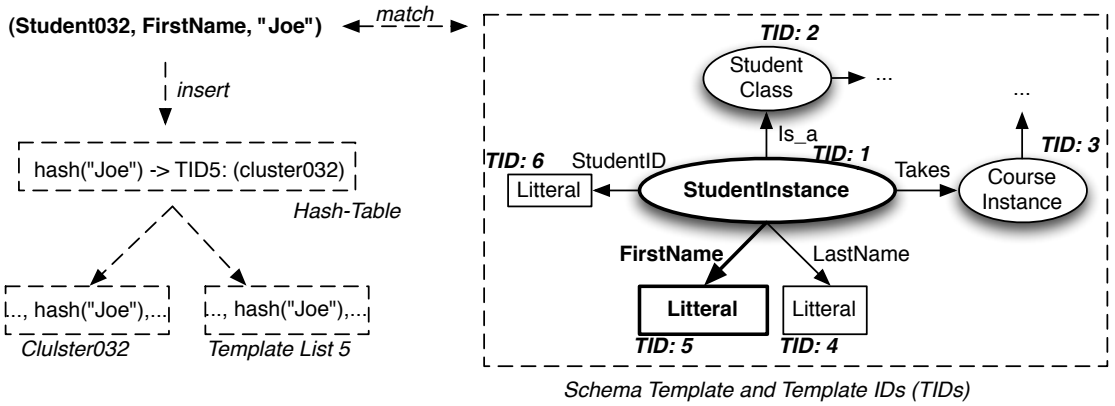


FIGURE 4.2: An insert using templates: an incoming triple (left) is matched to the current RDF template of the database (right), and inserted into the hash-table, a cluster, and a template list.

4.1.3 Molecules

DiploCloud uses physiological RDF partitioning and molecule patterns to efficiently co-locate RDF data in distributed settings. Figure 4.3 (ii) gives an example of molecule. Molecules have three key advantages in our context:

- Molecules represent the *ideal* tradeoff between co-location and degree of parallelism when partitioning RDF data. Partitioning RDF data at the triple-level is suboptimal because of the many joins it generates; Large graph partitions (such as those defined in [71]) are suboptimal as well, since in that case too many related triples are co-located, thus inhibiting parallel processing (see Section 4.5).
- All molecules are template-based, and hence store data extremely compactly;
- Finally, the molecules are defined in order to *materialize* frequent joins, for example between an entity and its corresponding values (e.g., between a student and his/her firstname), or between two semantically related entities (e.g., between a student and his/her advisor) that are frequently co-accessed.

When receiving a new triple the system inserts it in the corresponding molecule(s). In case the corresponding molecule does not exist yet, the system creates a new molecule cluster, inserts the triple in the molecule, and inserts the cluster in the list of clusters it maintains. Figures 4.3 gives a template example that co-locates information relating to Student instances along with an instance of a molecule for Student123.

Similarly to the template lists, the molecule clusters are serialized in a very compact form, both on disk and in main-memory. Each cluster is composed of two parts: a list

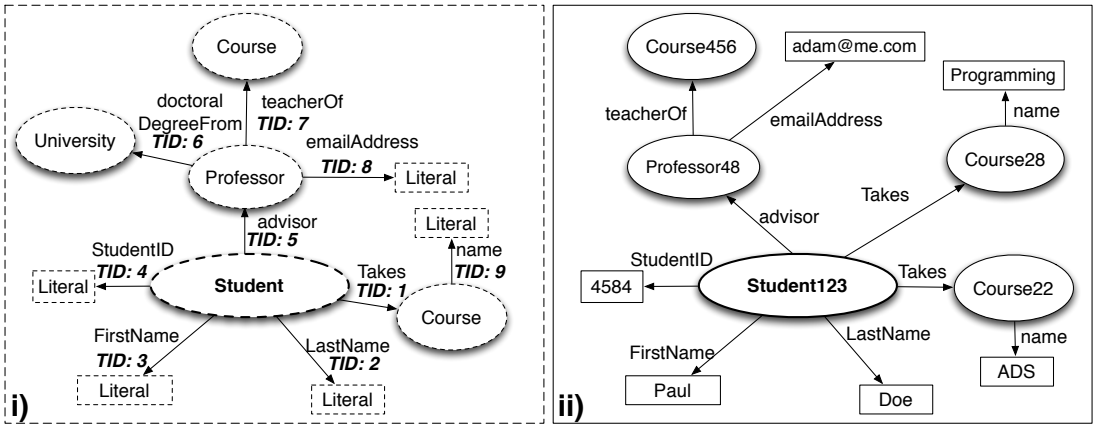


FIGURE 4.3: A molecule template (i) along with one of its RDF molecules (ii)

of offsets, containing for each template ID in the molecule the offset at which the keys corresponding for the template ID are stored, and the list of keys themselves. Thus, the size of a molecule, both on-disk and in main-memory, is $\#TEMPLATES + (KEY\_SIZE * \#TRIPLES)$, where KEY_SIZE is the size of a key (in bytes), $\#TEMPLATES$ is the number of templates IDs in the molecule, and $\#TRIPLES$ is the number of triples in the molecule (we note that this storage structure is much more compact than a standard list of triples). To retrieve a given information in a molecule, the system first determines the position of the template ID corresponding to the information sought in the molecule (e.g., "FirstName" is the sixth template ID for the "Student" molecule above in Figure 4.2). It then jumps to the offset corresponding to that position (e.g., 5^{th} offset in our example), reads that offset and the offset of the following template ID, and finally retrieves all the keys/values between those two offsets to get all the values corresponding to that template ID in the molecule. The molecule depicted in Figures 4.3 (ii), for instance, contains 15 triples (including type information), and would hence require 45 URIs/literals to be encoded using a standard triple-based serialization. Our molecule, on the other hand, only requires to store 10 keys to be correctly defined, yielding a compression ratio of $1 : 4.5$.

4.1.4 Auxiliary Indexes

While creating molecule templates and molecules identifiers, our system also take cares of two additional data gathering and analysis tasks. First, it inspects both the schema and instance data to determine all subsumption (subclass) relations between the classes, and maintains this information in a compact *type hierarchy*. We assign to every key the

most specific type possible in order to avoid having to materialize the type hierarchy for every instance, and handle type inference at query time by looking-up types in the type hierarchy. In case two unrelated types are assigned to a given instance, the partition manager creates a new virtual type composed of the two types and assigns it to the instance. Finally, we maintain *statistics* on each templates, counting the number of instances for each vertex (instance / literal) and edge (property) in the templates.

For each type, DiploCloud also maintains a list of the keys belonging to that type (*type index*). In addition, it maintains a *molecule index* storing for each key the list of molecules storing that key (e.g., "key 15123 [Course12] is stored in molecule 23521 [root:Student543]"). This index is particularly useful to answer triple-pattern queries as we explain below in Section 4.4.

4.2 System Overview

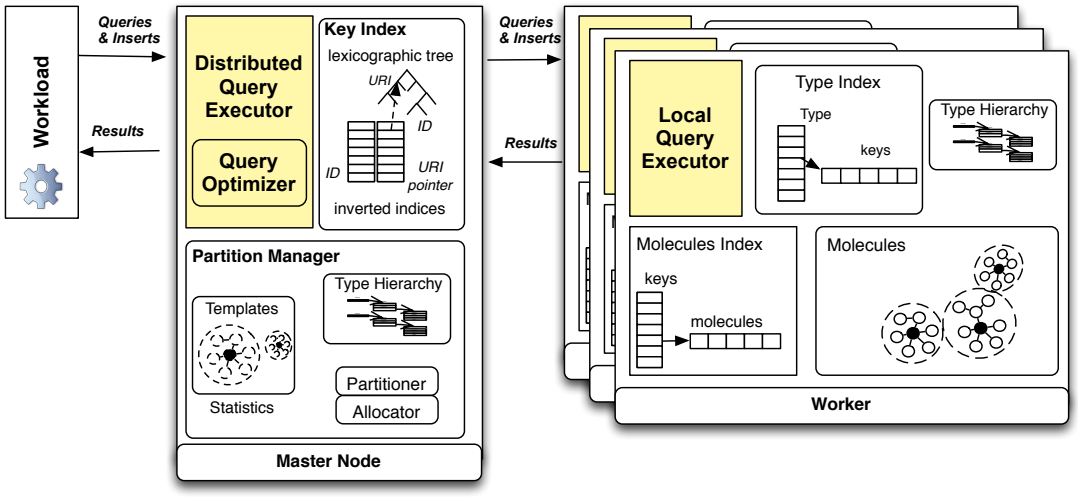


FIGURE 4.4: The architecture of DiploCloud.

The figure 4.4 gives a simplified architecture of DiploCloud. DiploCloud is a native, RDF database system. It was designed to run on clusters of commodity machines in order to *scale out* gracefully when handling bigger RDF datasets. In a cloud environment our system defines two distinct kinds of nodes running on the machines: the Master node—which is responsible for parsing the incoming queries, translating all URIs and values appearing in the RDF triples into system keys, delegating parts of the queries

to further nodes, and gathering and returning the answers—and the Worker nodes—that hold (part of) the RDF data and execute the query subplans they receive from the Master. We describe those two components in more detail in the following.

Our system design follows the architecture of many modern cloud-based distributed systems (e.g., Google's BigTable [27]), where one (Master) node is responsible for interacting with the clients and orchestrating the operations performed by the other (Worker) nodes.

4.2.1 Master Node

The Master node is composed of three main subcomponents: a key index (c.f. Section 4.1.1), in charge of encoding URIs and literals into compact system identifiers and of translating them back, a partition manager, responsible for partitioning the RDF data into recurring subgraphs, and a distributed query executor, responsible for parsing the incoming query, rewriting the query plans for the Workers, collecting and finally returning the results to the client. Note that the Master node can be replicated whenever necessary to insure proper query load-balancing and fault-tolerance. The Master can also be duplicated to scale out the key index for extremely large datasets, or to replicate the dataset on the Workers using different partitioning schemes (in that case, each new instance of the Master is responsible for one partitioning scheme).

Partition Manager: The partition manager has a number of duties all related to type management and fine-grained data partitions (*molecules*). It is the first component called when loading triples (see Parallel Loading in Section 4.4). When ingesting series of new triples comes, the Partition Manager first identifies RDF subgraphs following a given RDF partitioning scheme. DiploCloud supports several partitioning schemes (see Section 4.3.1). The partition manager analyzes the incoming data and builds molecule *templates* first, which act as data prototypes to create RDF molecules. Once the templates have been defined, the partition manager starts creating molecule identifiers based on the molecule roots (i.e., central molecule node) it identifies in the incoming data, and assigns each new molecule to a given Worker in the cluster using a given data allocation algorithm (see Section 4.3.2). The Workers then build and index the molecules on their own in a parallel way (see below).

Distributed Query Executor: The Master node also hosts the Distributed Query Executor. The Distributed Query Executor handles connections from clients, optimizes

incoming queries, and creates query sub-plans that are sent to the Workers. It also takes care of all intermediate communications with the workers, collecting and post-processing intermediate results, and finally preparing and sending the final results to the client. More information on distributed query processing is given in Section 4.4.3.

4.2.2 Worker Nodes

The Worker nodes hold the partitioned data and its corresponding local indices, and are responsible for running subqueries and sending results back to the Master node. Conceptually, the Workers are much simpler than the Master node and are built on three main data structures: i) a type index, clustering all keys based on their types ii) a series of RDF *molecules*, storing RDF data as very compact subgraphs, and iii) a molecule index, storing for each key the list of molecules where the key can be found.

Type Hierarchy & Type Index: Each Worker stores a copy of the type hierarchy (c.f. Section 4.1.4) locally. The Workers also store a local Type Index, which is partitioned, in the sense that each Worker only indexes the keys it is locally responsible for.

RDF Molecules: Workers store the primary copy of the RDF data as RDF molecules. They store, for every template defined by the template manager, a compact list of objects connected to the root of the molecule. Molecules are partitioned across the Workers, following the algorithm (e.g., round robin) specified by the partition manager.

Molecule Index: In addition to the molecules themselves, the Workers also maintain a Molecule Index for molecules located on the Worker.

4.3 Data Partitioning & Allocation

Triple-table and property-table hash-partitionings are currently the most common partitioning schemes for distributed RDF systems. While simple, such hash-partitionings almost systematically implies some distributed coordination overhead (e.g., to execute joins / path traversals on the RDF graph), thus making it inappropriate for most large-scale clusters and cloud computing environments exhibiting high network latencies. The other two standard relational partitioning techniques, (tuple) round-robin and range partitioning, are similarly flawed for the data and setting we consider, since they would partition triples either at random or based on the subject URI / type, hence seriously

limiting the parallelism of most operators (e.g., since many instances sharing the same type would end up on the same node).

Partitioning RDF data based on standard graph partitioning techniques (similarly to what [71] proposes) is also from our perspective inappropriate in a cloud context, for three main reasons:

Loss of semantics: standard graph partitioning tools (such as METIS[1], which was used in [71]) consider unlabeled graphs mostly, and hence are totally agnostic to the richness of an RDF graph including classes of nodes and edges.

Loss of parallelism: partitioning an RDF graph based, for instance, on a min-cut algorithm will lead to very *coarse* partitions where a high number of related instances (for instance linked to the same type or sharing links to the same objects) will be co-located, thus drastically limiting the degree of parallelism of many operators (e.g., projections or selections on certain types of instances).

Limited scalability: finally, attempting to partition very large RDF graphs is unrealistic in cloud environments, given that state-of-the-art graph partitioning techniques are inherently centralized and data/CPU intensive (as an anecdotal evidence, we had to borrow a powerful server and let it run for several hours to partition the largest dataset we use in Section 4.5 using METIS).

DiploCloud has been conceived from the ground up to support distributed data partitioning and co-location schemes in an efficient and flexible way. DiploCloud adopts an intermediate solution between tuple-partitioning and graph-partitioning by opting for a recurring, fine-grained graph-partitioning technique taking advantage of molecule templates. DiploCloud's molecule templates capture recurring patterns occurring in the RDF data naturally, by inspecting both the instance-level (*physical*) and the schema-level (*logical*) data, hence the expression *physiological*[2] partitioning.

4.3.1 Physiological Data Partitioning

We now define the three main molecule-based data partitioning techniques supported by our system:

[1] `http://glaros.dtc.umn.edu/gkhome/views/metis`

[2] *physiological* characterizes in our context a process that work both on the physical and logical layers of the database, as the classical Aries recovery algorithm

Scope-k Molecules: the simplest method is to define a number of template types (by default the system considers all types) serving as root nodes for the molecules, and then to co-locate all further nodes that are directly or indirectly connected to the roots, up to given scope k. Scope-1 molecules, for example, co-locate in the molecules all root nodes with their direct neighbors (instances or literals) as defined by the templates. Scope-2 or 3 molecules concatenate compatible templates from the root node (e.g., $(student, takes, course)$ and $(course, hasid, xsd : integer)$) recursively up to depth k to materialize the joins around each root, at the expense of rapidly increasing storage costs since much data is typically replicated in that case (see Section 4.5).

Manual Partitioning: Root nodes and the way to concatenate the various templates can also be specified by hand by the database administrator, who just has to write a configuration file specifying the roots and the way templates should be concatenated to define the generic shape of each molecule type. This is typically the best solution for relatively stable datasets and workloads whose main features are well-known.

Adaptive Partitioning: Finally, DiploCloud's most flexible partitioning algorithm starts by defining scope-1 molecules by default, and then adapts the templates following the query workload. The system maintains a sliding-window w tracking the recent history of the workload, as well as related statistics about the number of joins that had to be performed and the incriminating edges (e.g., missing co-location between students and courses causing a large number of joins). Then at each time epoch ϵ, the system: i) expands one molecule template by selectively concatenating the edges (rules) that are responsible for the most joins up to a given threshold for their maximal depth and ii) decreases (up to scope-1) all extended molecules whose extensions were not queried during the last epoch. In that way, our system slowly adapts to the workload and materializes frequent paths in the RDF graph while keeping the overall size of the molecules small.

4.3.2 Distributed Data Allocation

Once the physiological partitions are defined, DiploCloud still faces the choice of how to distribute the concrete partitions (i.e, the actual RDF molecules defined from the molecule templates) across the physical nodes. Data allocation in distributed RDF systems is delicate, since a given allocation scheme has to find a good tradeoff between

perfect load-balancing and data co-location. Our template manager implements three main allocation techniques:

Round-Robin: The round-robin allocation simply takes each new molecule it defines and assigns it to the next worker. This scheme favors load-balancing mostly.

Coarse Allocation: Coarse allocation splits the incoming data in W parts, where W is the number of workers, and assigns each part to a given worker. This allocation scheme favors data co-location mostly.

Semantic Co-location: The third allocation tries to achieve a tradeoff between load-balancing and co-location by grouping a small number of molecule instances (typically 10) that are semantically related through some connection (i.e., predicate), and then by allocating such groups in a round-robing fashion.

4.4 Common Operations

We now turn to describing how our system handles typical operations in distributed environments.

4.4.1 Bulk Load

Loading RDF data is generally speaking a rather expensive operation in DiploCloud but can be executed in a fairly efficient way when considered in bulk. We basically trade relatively complex instance data examination and complex local co-location for faster query execution. We are willing to make this tradeoff in order to speed-up complex queries using our various data partitioning and allocation schemes, especially in a Semantic Web or LOD context where isolated inserts or updates are from our experience rather infrequent.

We assume that the data to be loaded is available in a shared space on the cloud (e.g., typically in a S3 bucket on AWS). Bulk loading is a hybrid process involving both the Master—whose task is to encode all incoming data, to identify potential molecule roots from the instances, and to assign them to the Workers using some allocation scheme—and all the Workers—which build, store and index their respective molecules in parallel based on the molecule templates defined.

On the worker nodes, building the molecule is an n-pass algorithm (where n is the deepest level of the molecule, see Section 4.1) in DiploCloud, since we need to construct the RDF molecules in the clusters (i.e., we need to materialize triple joins to form the clusters). In a first pass, we identify all root nodes and their corresponding template IDs, and create all clusters. The subsequent passes are used to join triples to the root nodes (hence, the student clusters depicted in the Figure 4.1 are built in two phases, one for the Student root node, and one for the triples directly connected to the Student). During this operation, we also update the template lists and the key index incrementally. Bulk inserts have been highly optimized in DiploCloud, and use an efficient page-manager to execute inserts for large datasets that cannot be kept in main-memory.

This division of work and the fact that the most expensive operation (molecule construction) is performed completely in parallel enables DiploCloud to bulk load efficiently as we experimentally show in Section 4.5.

4.4.2 Updates

As for other hybrid or analytic systems, updates can be relatively complex to handle in DiploCloud, since they might lead to a partial rewrite of the key index and molecule indices, and to a reorganization of the physical structures of several molecules. To handle them efficiently, we adopt a lazy rewrite strategy, similarly to many modern read-optimized system (e.g., CStore or BigTable). All updates are performed on write-optimized log-structures in main-memory. At query time, both the primary (read-optimized) and log-structured (write-optimized) data stores are tapped in order to return the correct results.

We distinguish between two kinds of updates: in-place and complex updates. In-place updates are punctual updates on literal values; they can be processed directly in our system by updating the key index, the corresponding cluster, and the template lists if necessary. Complex updates are updates modifying object properties in the molecules. They are more complex to handle than in-place updates, since they might require a rewrite of a list of clusters in the key index, and a rewrite of a list of keys in the molecule clusters. To allow for efficient operations, complex updates are treated like updates in a column-store (see [104]): the corresponding structures are flagged in the key index, and new structures are maintained in write-optimized structures in main-memory. Periodically, the write-optimized structures are merged with the main data structures in an offline fashion.

4.4.3 Query Processing

Query processing in DiploCloud is very different from previous approaches to execute queries on RDF data, because of the three peculiar data structures in our system: a key index associating URIs and literals to template IDs and cluster lists, clusters storing RDF molecules in a very compact fashion, and template lists storing compact lists of literals. All queries composed of one Basic Graph Pattern (star-like queries) are executed totally in parallel, independently on all Workers without any central coordination thanks to the molecules and their indices.

For queries that still require some degree of distributed coordination—typically to handle distributed joins—we resort to adaptive query execution strategies. We mainly have two ways of executing distributed joins: whenever the intermediate result set is small (i.e., up to a few hundred tuples according to our Statistics components), we ship all results to the Master, which finalizes the join centrally. Otherwise, we fall back to a distributed hash-join by distributing the smallest result set among the Workers. Distributed joins can be avoided in many cases by resorting to the distributed data partitioning and data co-location schemes described above.

We describe below how a few common queries are handled in DiploCloud.

4.4.3.1 Basic Graph Patterns

Basic Graph Patterns are relatively simple in DiploCloud: they are usually resolved by looking for a bound-variable (URI) in the key index or molecules index, retrieving the corresponding molecules numbers, and finally retrieving values from the molecules when necessary. Conjunctions and disjunctions of triples patterns can be resolved very efficiently in our system. Since the RDF nodes are logically grouped by molecules in the key index, it is typically sufficient to read the corresponding list of molecules in the molecules index. No join operation is needed since joins are implicitly materialized in molecules. The following query (query # 1 in Section 4.5), for instance:

```
?X a :GraduateStudent .
?X :takesCourse <GraduateCourse0> .
```

is first optimized by the Master based on the statistics it collected; a query plan is then sent to all Workers asking them to first look-up all molecules containing $GraduateCourse0$ (since it is the most selective pattern in the query) using their local molecule index.

Each Worker can then contribute to the results independently and in parallel, by retrieving the molecule ids, filtering them based on the $GraduateStudent$ type (by simply inspecting the ids) and returning the resulting ids to the master node. If the template ID of $GraduateCourse0$ in the molecule is ambiguous (for example when a $GraduateStudent$ can both teach and take courses), then an additional filtering step is carried out locally at the end of the query plan by looking up molecules and filtering them based on their predicate (e.g., predicate linking $GraduateStudent$ to $GraduateCourse0$).

4.4.3.2 Molecule Queries

Molecule queries or queries retrieving many values/instances around a given instance (for example for visualization purposes) are also extremely efficient in our system. In most cases, the key index is invoked to find the corresponding molecule, which contains then all the corresponding values. For bigger scopes (such as the ones we consider in our experimental evaluation below), our system can efficiently join clusters based on the various root nodes they contain.

4.4.3.3 Aggregates and Analytics

Aggregate and analytic queries can also be efficiently resolved by our system. Many analytic queries can be solved by first intersecting lists of clusters in the molecule index, and then looking up values in the remaining molecule clusters. Large analytic or aggregate queries on literals (such as our third analytic query below, returning the names of all graduate students) can be extremely efficiently resolved by taking advantage of template lists (containing compact and sorted lists of literal values for a given template ID), or by filtering template lists based on lists of molecule IDs retrieved from the key index.

4.4.3.4 Distributed Join

As a more complete example of query processing, we consider the following LUBM [59] query:

```
?Z is_a :Department .
?Y is_a :University .
?X is_a :GraduateStudent .
?Z :subOrganizationOf ?Y .      <-- 1st
```

```
?X :undergraduateDegreeFrom ?Y .<-- 2nd
?X :memberOf ?Z .               <-- 3rd
```

We briefly discuss three possible strategies for dealing with this query below.

For the simplest and the most generic one (Algorithm 2), we prepare intermediate results on each node; we then send them to the Master node where we perform the final join. In that way we retrieve elements meeting the 1st constraint (*Department subOrganizationOf University*), then the 2nd constraint (*GraduateStudent undergraduateDegreeFrom University*), and the 3rd constraint (*GraduateStudent memberOf Department*). Finally, we perform hash-joins for all those intermediate results on the Master node.

For the second method, we prepare intermediate results for the 1st constraint, and we distribute them across the cluster, since in every molecule of type $GraduateStudent$, we have all information about the object instance (i.e. $undergraduateDegreeFrom$ and $memberOf$) for each $GraduateStudent$; having distributed intermediate results corresponding to the 1st constraint, we can perform the joint for the 2nd and 3rd constraints completely in parallel.

The third and most efficient strategy would be to increase the size of the considered molecules, so that in every molecule, besides all information about the root ($GraduateStudent$), we would also store all information about $Department$ related to the root, and further $University$ related to the $Department$. To answer the query, we just need to retrieve data about the 2nd and the 3rd constraints in this case, and perform a check on the molecule to validate that a given $University$ from the 2nd constraint is the same as the one related to the $Department$ from the 3rd constraint, which indicates that the 1st constraint is met.

4.5 Performance Evaluation

We have implemented a prototype of DiploCloud following the architecture and techniques described above. The following experiments were conducted for two scenarios: centralized and distributed. For each of them we evaluated the performance of DiploCloud and we compared it with the state-of-the-art systems and techniques.

Algorithm 2 Query Execution Algorithm with Join on the Master Node

```
1:  procedure EXECUTEQUERY(a, b)
2:      for all BGP in QUERY do                          ▷ BGP - Basig Graph Pattern
3:          if BGP.subject then
4:              molecules ← GetMolecule(subject)
5:          else if BGP.object then
6:              molecules ← GetMolecules(object)
7:          end if
8:          for all molecules do
9:                                          ▷ check if the molecule matches the BGP
10:             for all TP in BGP do                          ▷ TP - Triple Pattern
11:                 if TP.subject != molecule.subject then
12:                     nextMolecule
13:                 end if
14:                 if TP.predicate != molecule.predicate then
15:                     nextMolecule
16:                 end if
17:                 if TP.object != molecule.object then
18:                     nextMolecule
19:                 end if
20:             end for
21:                         ▷ the molecule matches the BGP, so we can retrieve entities
22:             resultBGP ← GetEntities(molecule,BGP)
23:         end for
24:         results ← resultBGP
25:     end for
26:     SendToMasterNode(results)
27: end procedure
28:                                          ▷ On the Master do Hash Join
```

4.5.1 Datasets and Workloads

To compare the various systems, we used three different benchmarks.

- the Lehigh University Benchmark (LUBM) [59]
- the BowlognaBench Benchmark [41]
- the DBPedia dataset with five queries [14]

LUBM is one of the oldest and most popular benchmarks for Semantic Web data. It provides an ontology describing universities together with a data generator and fourteen queries. We generated the following datasets:

- 10 universities: 1'272'814 triples [226 MB]
- 100 universities: 13'876'209 triples [2.4 GB]
- 400 universities: 55 035 263 triples [9.4 GB]
- 800 universities: 110 128 171 triples [19 GB]
- 1600 universities: 220 416 262 triples [38 GB]

We compared the runtime execution for LUBM queries and for three analytic queries inspired by BowlognaBench [41]. LUBM queries are criticized by some for their reasoning coverage; this was not an issue in our case, since we focused on RDF DB query processing rather than on reasoning capabilities. We keep an in-memory representation of subsumption trees in DiploCloud and rewrite queries automatically to support subclass inference for the LUBM queries. We manually rewrote inference queries for the systems that do not support such functionalities.

The three additional analytic/aggregate queries that we considered are as follows: 1) a query returning the professor who supervises the most students 2) a query returning a big molecule containing all triples within a scope of 2 of Student0 and 3) a query returning all graduate students.

For BowlognaBench, we used two different datasets generated with the BowlognaBench Instance Generator:

- 1 departments: 1.2 million triples [273MB]
- 10 departments: 12 millions triples [2.7GB]

For both datasets we set 4 fields per department and 15 semesters. We run the 13 queries of BowlognaBench to compare the query execution time for RDF systems.

Additionally, we also used a dataset extracted from DBPedia (which is interesting in our context as it is much more *noisy* than the LUBM and BowlognaBench data) with five queries [14]. From the original DBpedia 3.5.1, we extracted a subset of:

- 73 731 354 triples [9.3 GB]

All inference queries were implemented by rewriting the query plans for DiploCloud and the systems that did not support such queries.

4.5.2 Methodology

As for other benchmarks (e.g., tpc-x[3] or our own OLTP-Benchmark [42]) we include a warm-up phase before measuring the execution time of the queries. We first run all the queries in sequence once to warm-up the systems, and then repeat the process ten times (i.e., we run in total 11 batches containing all the queries in sequence for each system). We report the mean values for each query and each system below. We assumed that the maximum time for each query should not exceed 2 hours (we stopped the tests if one query took more than two hours to be executed). We compared the output of all queries running on all systems to ensure that all results were correct.

We tried to do a reasonable optimization job for each system, by following the recommendations given in the installation guides for each system. We did not try to optimize the systems any further, however. We performed no fine-tuning or optimization for DiploCloud.

We avoided the artifact of connecting to the server, initializing the DB from files and printing results for all systems; we measured instead the query execution times only.

4.5.3 Systems

We chose those systems to have different comparison points, and because they were either freely available on the Web, or possible to implement with relatively little effort. We give a few details about each system below.

AllegroGraph [1] We used AllegroGraph RDFStore 4.2.1 AllegroGraph unfortunately poses some limits on the number of triples that can be stored for the free edition, such that we couldn't load the big data set. For AllegroGraph, we prepared a SPARQL Python script using libraries provided by the vendor.

BigOWLIM [77] We used BigOWLIM 3.5.3436. OWLIM provides us with a java application to run the LUBM benchmark, so we used it directly for our tests.

Jena [94] We used Jena-2.6.4 and the TDB-0.8.10 storage component. We created the database by using the "tdbloader" provided by Jena. We created a Java application to run and measure the execution time of each query.

[3] *http://www.tpc.org/*

Virtuoso [45] We used Virtuoso Open-Source Edition 6.1.3. Virtuoso supports ODBC connections, and we prepared a Python script using the PyODBC library for our queries.

RDF-3X [90] We used RDF-3X 0.3.5. We slightly modified the system to measure the execution time of the queries only, without taking into account the initialization of the database and turning off the print-outs.

4store [62] is a well-known distributed, native RDF system based on property tables and distributing triples (or quads, actually) based on a hash-partitioning of their subject. We used 4store revision v1.1.4., with eight segments per node, and the provided tools to load and query.

SHARD [101] stores RDF triples directly in HDFS and takes advantage of Hadoop for all distributed processes. We slightly modified the system in order to measure the execution time of the queries only, without taking into account the initialization of the database and by turning off the print-outs.

RDF-3X GraphPartitioning : we re-implemented the base approach described in [71] by using RDF-3X and by partitioning the RDF data using METIS. Rather than using Hadoop for the distributed coordination, we implemented all distributed joins in Java, following the same design as for our own prototype.

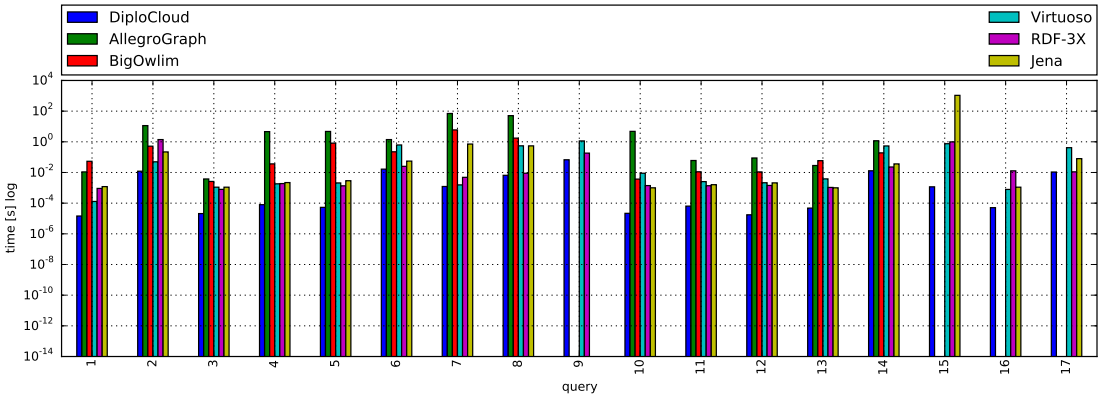
4.5.4 Centralized Environment

4.5.4.1 Hardware Platform

All experiments were run on a HP ProLiant DL360 G7 server with two Quad-Core Intel Xeon Processor E5640, 6GB of DDR3 RAM and running Linux Ubuntu 10.10 (Maverick Meerkat). All data were stored on recent 1.4 TB Serial ATA disk.

4.5.4.2 Results

Relative execution times for all queries and all systems are given below, in the Figure 4.5 (log-scale) for 10 universities and in the Figure 4.6 (log-scale) for 100 universities. The Tables 4.1 and 4.2 shows the loading time in seconds and the storage consumption in MB for respectively 10 and 100 universities of the LUBM benchmark.



q #	DiploCloud	AllegroGraph	BigOwlim	Virtuoso	RDF-3X	Jena
1	0.000014	0.01090	0.0537	0.000129	0.000914	0.0012
2	0.012100	11.40000	0.5190	0.049600	1.400000	0.2190
3	0.000021	0.00378	0.0026	0.001100	0.000791	0.0011
4	0.000080	4.62000	0.0363	0.001820	0.001890	0.0022
5	0.000053	4.74000	0.8190	0.002080	0.001350	0.0029
6	0.016500	1.40000	0.2230	0.622000	0.025100	0.0552
7	0.001220	70.30000	5.9600	0.001550	0.004820	0.7140
8	0.006540	50.90000	1.7400	0.547000	0.008940	0.5430
9	0.067400	NaN	NaN	1.140000	0.183000	NaN
10	0.000022	4.80000	0.0037	0.008930	0.001400	0.0010
11	0.000064	0.06040	0.0111	0.002540	0.001370	0.0016
12	0.000017	0.08810	0.0109	0.002140	0.001430	0.0021
13	0.000048	0.02850	0.0589	0.003820	0.001060	0.0010
14	0.012900	1.17000	0.1900	0.537000	0.022800	0.0362
15	0.001160	NaN	NaN	0.750000	0.993000	1070.0000
16	0.000051	NaN	NaN	0.000785	0.012800	0.0011
17	0.010700	NaN	NaN	0.413000	0.011000	0.0801

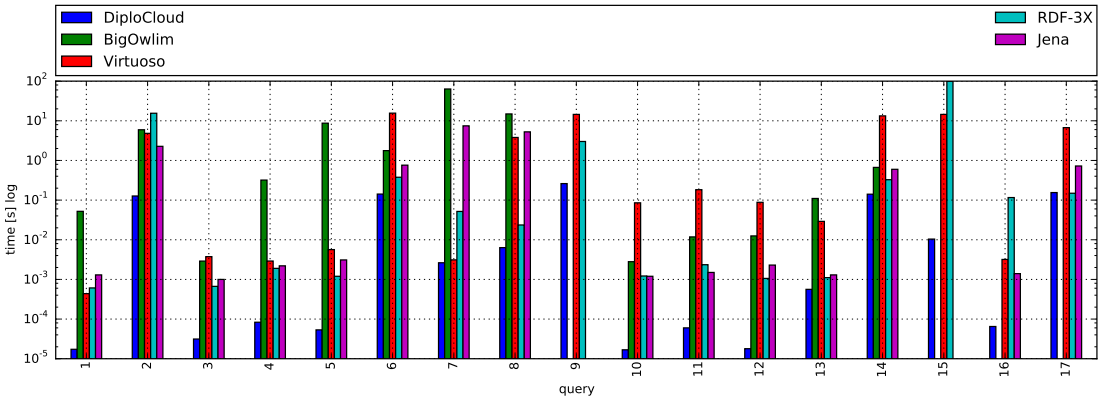
FIGURE 4.5: Query execution time for the 10 universities LUBM data set

	DiploCloud	AllegroGraph	BigOwlim	Virtuoso	RDF-3X	Jena
Load Time [s]	31	13	50	88	16	98
size [MB]	87	696	209	140	66	118

TABLE 4.1: Load times and size of the databases for the 10 universities LUBM data set.

	DiploCloud	BigOwlim	Virtuoso	RDF-3X	Jena
Load Time [s]	427	748	914	214	1146
size [MB]	913	2012	772	694	1245

TABLE 4.2: Load times and size of the databases for the 100 universities LUBM data set.



q #	DiploCloud	BigOwlim	Virtuoso	RDF-3X	Jena
1	0.000017	0.0521	0.000438	0.000610	0.0013
2	0.127000	5.9400	4.830000	15.500000	2.2700
3	0.000031	0.0029	0.003750	0.000668	0.0010
4	0.000083	0.3200	0.002910	0.001900	0.0022
5	0.000053	8.7100	0.005650	0.001200	0.0031
6	0.142000	1.7700	15.600000	0.377000	0.7610
7	0.002630	63.4000	0.003110	0.051800	7.4600
8	0.006340	14.9000	3.800000	0.023600	5.2600
9	0.261000	NaN	14.500000	3.010000	NaN
10	0.000017	0.0028	0.085400	0.001220	0.0012

q #	DiploCloud	BigOwlim	Virtuoso	RDF-3X	Jena
11	0.000060	0.0118	0.183000	0.002350	0.0015
12	0.000018	0.0125	0.088100	0.001060	0.0023
13	0.000562	0.1100	0.029100	0.001110	0.0013
14	0.141000	0.6680	13.300000	0.327000	0.5980
15	0.010400	NaN	14.500000	99.300000	NaN
16	0.000065	NaN	0.003190	0.116000	0.0014
17	0.155000	NaN	6.720000	0.149000	0.7250

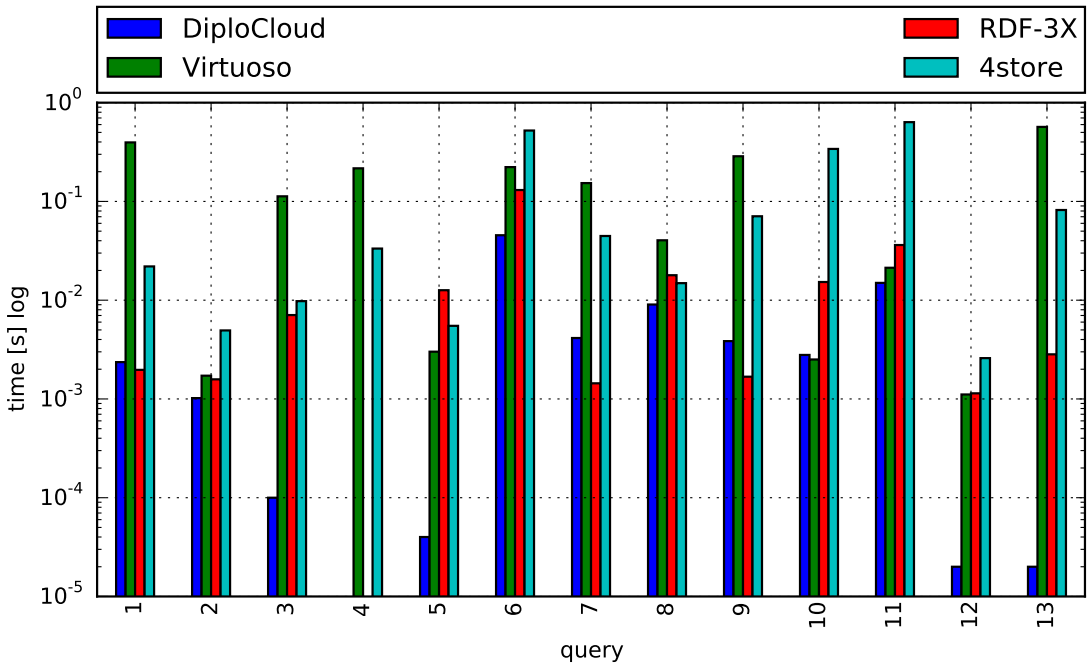
FIGURE 4.6: Query execution time for the 100 universities LUBM data set

	DiploCloud	Virtuoso	RDF-3X	4store
Load Time [s]	18.3503	31.71	11.94	6.25
size [MB]	92.0000	108.00	60.00	192.00

TABLE 4.3: Load times and size of the databases for the 1 department BowlognaBench data set.

We observe that DiploCloud is generally speaking very fast, both for the bulk inserts, for the LUBM queries and especially for the analytic queries. DiploCloud is not the fastest system for inserts, and produces slightly larger databases on disk than some other systems (like RDF-3X), but performs overall very-well for all the queries. Our system is on average 30 times faster than the fastest RDF data management system we have considered (i.e., RDF-3X) for the LUBM queries, and on average 350 times faster than the fastest system (Virtuoso) on the analytic queries. Is is also very scalable (both the bulk insert and the query processing scale gracefully from 10 to 100 universities). We can see (Tables 4.3 and 4.4) that Virtuoso takes more time to load and index the dataset but the size of the indices scales better than for the other systems. The fastest system is 4Store which also has the biggest indices. Both RDF-3X and Virtuoso achieve a good compression.

The Figures 4.7 (log-scale) and 4.8 (log-scale) report the experimental results for the BowlognaBench datasets consisting of 1 and 10 departments respectively. The values

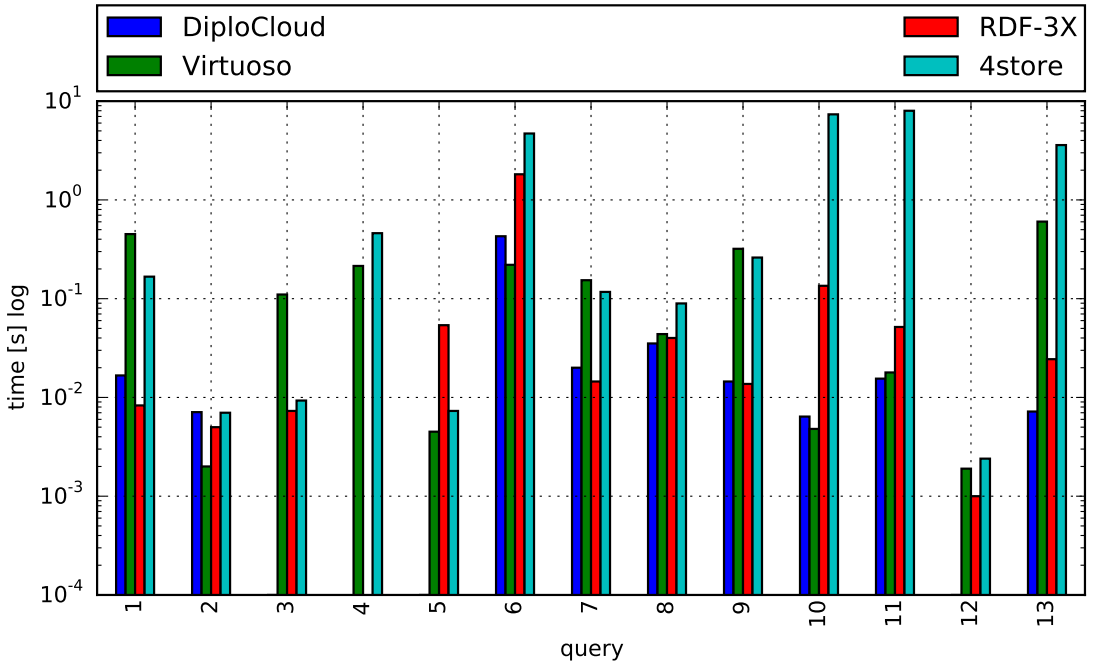


q #	DiploCloud	Virtuoso	RDF-3X	4store
1	0.00236	0.39582	0.00197	0.02197
2	0.00102	0.00172	0.00158	0.00494
3	0.00010	0.11265	0.00707	0.00978
4	NaN	0.21641	NaN	0.03332
5	0.00004	0.00301	0.01262	0.00550
6	0.04547	0.22265	0.13043	0.52229
7	0.00414	0.15336	0.00144	0.04467
8	0.00904	0.04041	0.01790	0.01488
9	0.00385	0.28660	0.00168	0.07075
10	0.00279	0.00251	0.01528	0.34008
11	0.01497	0.02130	0.03619	0.63500
12	0.00002	0.00111	0.00114	0.00259
13	0.00002	0.56823	0.00283	0.08187

FIGURE 4.7: Query execution time for the 1 department BowlognaBench data set.

	DiploCloud	Virtuoso	RDF-3X	4store
Load Time [s]	526.652	363.24	139.55	69.65
size [MB]	920.000	616.00	618.00	1752.00

TABLE 4.4: Load times and size of the databases for the 10 department BowlognaBench data set.



q #	DiploCloud	Virtuoso	RDF-3X	4store
1	0.0167	0.4508	0.0083	0.1671
2	0.0071	0.0020	0.0050	0.0070
3	0.0001	0.1102	0.0073	0.0093
4	NaN	0.2145	NaN	0.4604
5	0.0001	0.0045	0.0539	0.0073
6	0.4286	0.2202	1.8182	4.7038
7	0.0200	0.1538	0.0145	0.1171
8	0.0352	0.0438	0.0400	0.0895
9	0.0145	0.3199	0.0137	0.2609
10	0.0064	0.0048	0.1350	7.3526
11	0.0155	0.0179	0.0517	7.9934
12	0.0001	0.0019	0.0010	0.0024
13	0.0072	0.6033	0.0244	3.5917

FIGURE 4.8: Query execution time for the 10 department BowlognaBench data set.

indicate query execution times for each query of the BowlognaBench benchmark. We note that query 4 could not be run on RDF-3X and DiploCloud as they do not provide support for pattern matching. The Tables 4.3 and 4.4 shows the loading time in seconds and the storage consumption in MB for respectively 1 and 10 departments.

As we can see, the query execution time for the BowlognaBench analytic queries strongly vary for different systems. DiploCloud is slightly slower for the queries 1 and 7 than RDF-3X, and it is outperformed by Virtuoso for the queries 2 and 10. We can observe the slower performance of 4Store for 10 out of 13 queries as compared with the other systems: for some queries (e.g. 10) the execution times took more than 7 seconds.

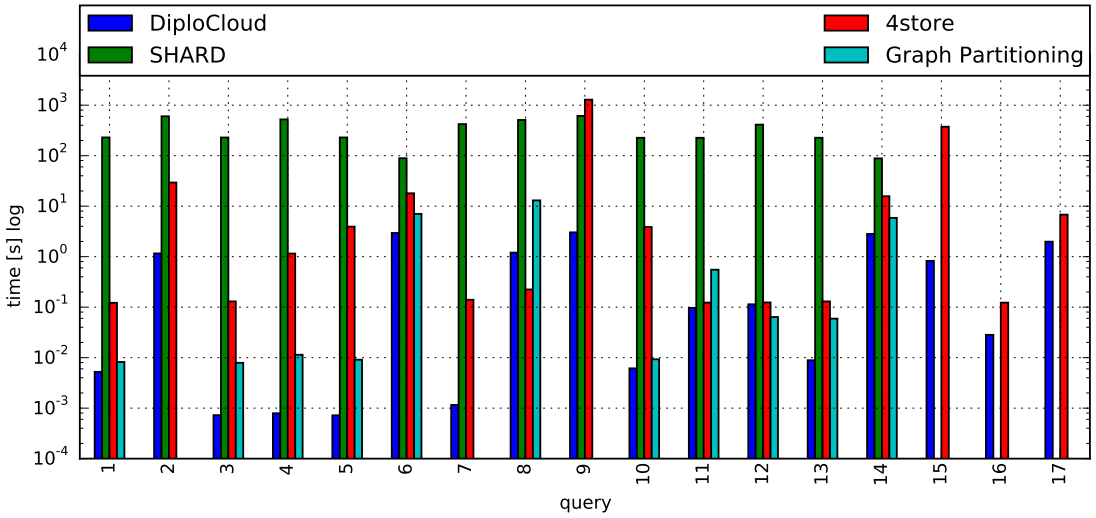
Specifically, longest query executions can be observed for the queries 6, 10, and 11. The slowest is the path query which involves several joins. For all those queries DiploCloud performs very well. We can see that the query 8 is not easy to be efficiently answered for all the systems. The queries 3 and 11 are also challenging because of the several joins involved, though DiploCloud handles them without any problem (especially the query 3). Instead, the count queries (i.e., 1 and 2) can be performed quite efficiently. One difference that we can observe for the bigger dataset of BowlognaBench as compared with the smaller dataset is the good result of Virtuoso: it performed faster than RDF-3X on 10 out of 13 queries. We can also observe that DiploCloud scales very well, whilst the competitors for some cases have issues handling the big dataset (e.g. 4store query 8, RDF-3X query 6). In general, we can again observe that DiploCloud outperforms the competitors for most of the queries for the both datasets and that it scales gracefully.

Over the course of those experiments, we observed that a significant part of query execution times can be consumed by encoding and decoding IDs assigned to URIS back and forth. For this reason, we conducted in [83] to the best of our knowledge the first systematic comparison of the most common data structures and hash functions used to encode URI data. As the result of our work we decided to change the structures used in DiploCloud following those comparison. As we need in our context to favor fast insertions (both for ordered and unordered datasets), fast look-ups and relatively compact structures with no collision, we decided to replace our prefix tree (LexicographicTree) with the HAT-trie [8]. We gained both in terms of memory consumption and efficient look-ups compared to our previous structure; We believe that this new choice will considerably speed-query execution times and improve the scalability of our system.

4.5.5 Distributed Environment

4.5.5.1 Hardware Platform

All experiments (except the EC2 experiments) were run in three cloud configurations of 4, 8, and 16 nodes. Worker nodes were commodity machines with Quad-Core Intel i7-2600 CPUs @ 3.40GHz, 8GB of DDR3-1600 RAM, 500GB Serial ATA HDD, running Ubuntu 12.04.2 LTS. The Master node was similar, but with 16GB RAM.



q #	DiploCloud	SHARD	4store	Graph Partitioning
1	0.004530	244.5374	0.191	0.0134
2	1.285735	727.6800	62.328	NaN
3	0.000719	272.1212	0.194	0.0085
4	0.000859	612.8159	3.517	0.0164
5	0.000810	258.6905	13.014	0.0142
6	7.027607	103.3408	33.480	10.9248
7	0.001422	527.1044	0.201	NaN
8	1.191533	594.6692	0.277	21.6869
9	3.411679	781.3004	2889.510	NaN

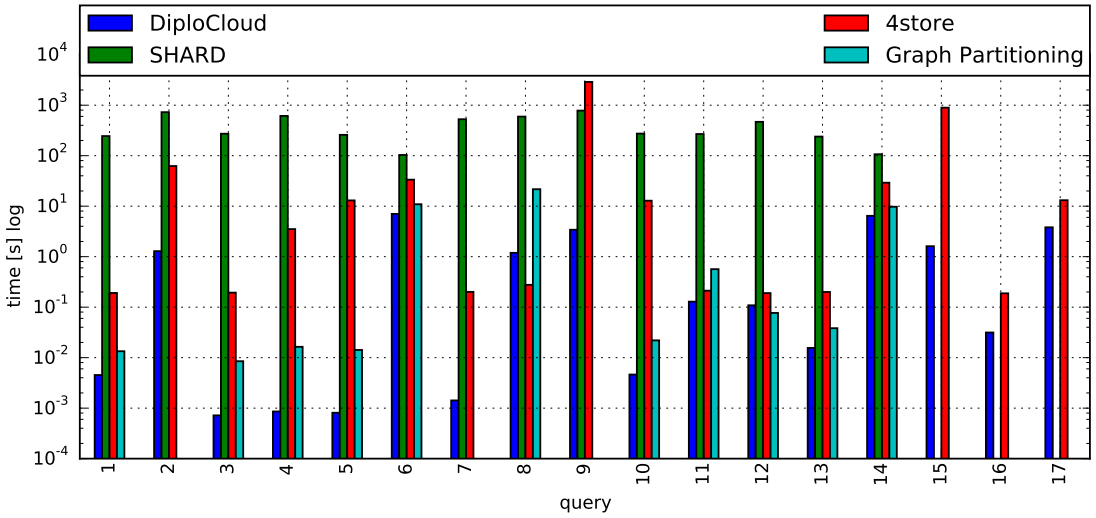
q #	DiploCloud	SHARD	4store	Graph Partitioning
10	0.004629	274.0291	12.842	0.0220
11	0.128569	268.5251	0.212	0.5655
12	0.108854	468.1124	0.190	0.0768
13	0.015579	239.0662	0.200	0.0383
14	6.437258	106.7680	29.176	9.7465
15	1.610378	NaN	893.725	NaN
16	0.031483	NaN	0.188	NaN
17	3.813508	NaN	13.120	NaN

FIGURE 4.9: Query execution time for 4 nodes and 400 universities LUBM data set

4.5.5.2 Results

We start by comparing the query execution times for DiploCloud deployed in its simplest configuration i.e., partitioning with Scope-1 molecules, and allocating molecules in a round-robin fashion.

The Figures 4.9, 4.10, and 4.11 (log-scale) give the results for the LUBM datasets for 400, 800, and 1600 universities executed respectively on 4, 8, and 16 servers. Note that several queries timed-out for GraphPartitioning (2, 7, 9, 15, 16, 17) (mostly due to the very large number of generated intermediate results, and due to the subsequent distributed joins). On the biggest deployment, DiploCloud is on average 140 times faster than 4store, 244 times faster than SHARD, and 485 times faster than the graph partitioning approach using RDF-3X (including the time-out values for the timed-out queries). The Figures 4.12, 4.13, and 4.14 (log-scale) give the results for the DPBedia dataset. DiploCloud achieves sub-second latencies on most queries, and is particularly efficient when deployed on larger clusters. We explain some of those results in more detail below.



q #	DiploCloud	SHARD	4store	Graph Partitioning	q #	DiploCloud	SHARD	4store	Graph Partitioning
1	0.004530	244.5374	0.191	0.0134	10	0.004629	274.0291	12.842	0.0220
2	1.285735	727.6800	62.328	NaN	11	0.128569	268.5251	0.212	0.5655
3	0.000719	272.1212	0.194	0.0085	12	0.108854	468.1124	0.190	0.0768
4	0.000859	612.8159	3.517	0.0164	13	0.015579	239.0662	0.200	0.0383
5	0.000810	258.6905	13.014	0.0142	14	6.437258	106.7680	29.176	9.7465
6	7.027607	103.3408	33.480	10.9248	15	1.610378	NaN	893.725	NaN
7	0.001422	527.1044	0.201	NaN	16	0.031483	NaN	0.188	NaN
8	1.191533	594.6692	0.277	21.6869	17	3.813508	NaN	13.120	NaN
9	3.411679	781.3004	2889.510	NaN					

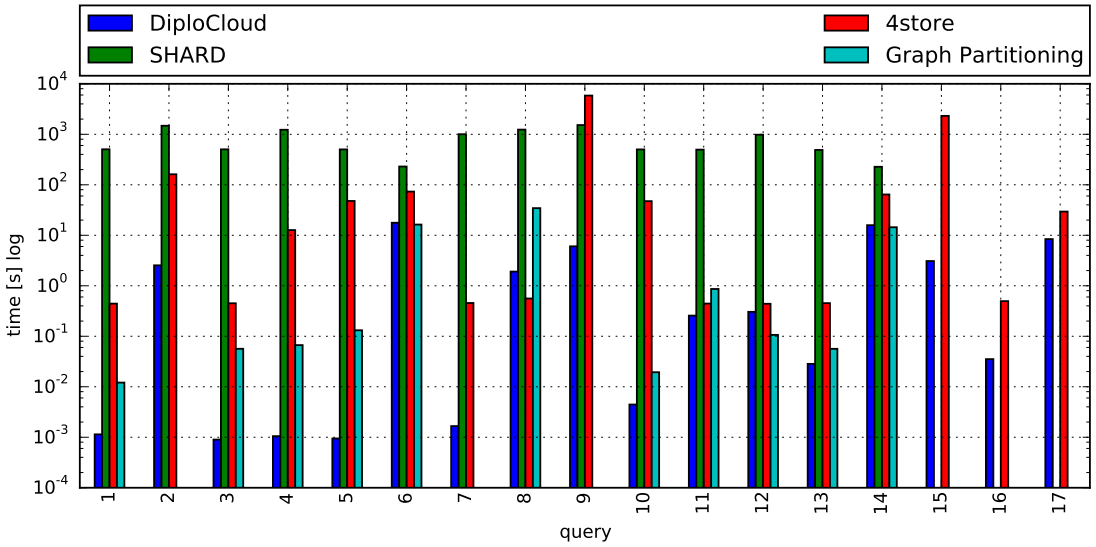
FIGURE 4.10: Query execution time for 8 nodes and 800 universities LUBM data set

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
DiploCloud	0	1/0	0	0	0	0	1	1/0	1/0	0	1/0	1	0	0	0	1/0	0
4store	1	5	1	5	2	1	7	5	11	2	2	2	2	0	3	2	1
RDF3X Part.	0+1	2+5	0+1	0+5	0+2	0+1	2+5	1+5	2+9	0+2	1+2	1+3	0+2	0	-	-	-

TABLE 4.5: Joins analysis for several system on the LUBM workload (Distributed Environment). For DiploCloud scope-1/adaptive molecules.

Data Partitioning & Allocation: We now turn to our adaptive partitioning approach. We implemented our adaptive partitioning approach, keeping all the queries in the history, considering a max-depth of 2, and switching to a new time epoch after each query batch. The results are available on the Figures 4.15, 4.16, and 4.17 (log-scale) for respectively 4, 8, and 16 nodes. Only the deepest (in terms of RDF paths) LUBM queries are shown on the graphs (the other queries behave the same for both partitioning schemes). By co-locating all frequently queried elements, the query execution using the adaptive partitioning is on average more than 3 times faster than the simple partitioning for those queries. Note that scope-2 molecules would behave like the adaptive scheme in that case, but take much more space (see Table 4.6).

Join Analysis: In order to better understand the above results, we made a small query



q #	DiploCloud	SHARD	4store	Graph Partitioning
1	0.001140	504.8053	0.443	0.0121
2	2.539618	1475.7045	161.499	NaN
3	0.000899	503.9658	0.450	0.0565
4	0.001051	1229.6080	12.672	0.0668
5	0.000946	502.0908	47.819	0.1314
6	17.726164	230.1689	73.384	16.3098
7	0.001662	1003.6873	0.456	NaN
8	1.909516	1238.7459	0.560	34.4615
9	6.041795	1523.6669	5850.255	NaN

q #	DiploCloud	SHARD	4store	Graph Partitioning
10	0.004448	502.0709	47.422	0.0194
11	0.256411	497.1197	0.444	0.8637
12	0.304655	982.2692	0.440	0.1065
13	0.028383	489.6785	0.453	0.0564
14	15.849092	227.3002	64.291	14.4278
15	3.087212	NaN	2316.523	NaN
16	0.035231	NaN	0.496	NaN
17	8.404547	NaN	29.481	NaN

FIGURE 4.11: Query execution time for 16 nodes and 1600 universities LUBM data set

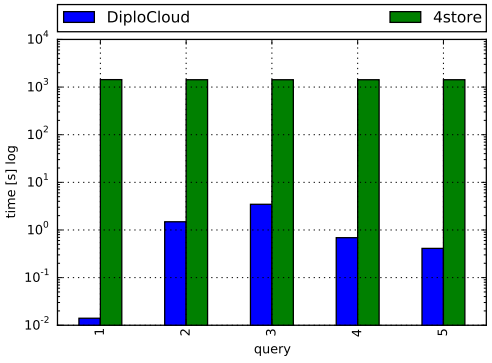


FIGURE 4.12: Query execution time for DBPedia running on 4 nodes

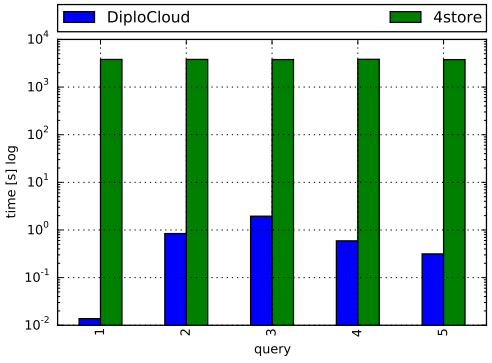
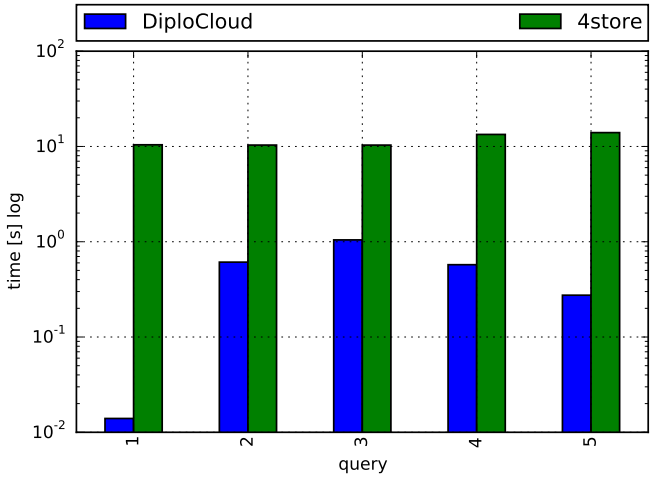


FIGURE 4.13: Query execution time for DBPedia running on 8 nodes

execution analysis (see Table 4.5) on the LUBM workload, counting the number of joins for DiploCloud (counting the number of joins between molecules for scope-1 / adaptive molecules), 4store (by inspecting the query plans given by the system), and RDF-3X GraphPartitioning (using EXPLAINs). For the RDF-3X GraphPartitioning approach, we report both distributed joins (first number) and local joins (second number). We



q #	DiploCloud	4store
1	0.013963	10.413
2	0.610787	10.340
3	1.045317	10.335
4	0.574045	13.370
5	0.274768	13.972

FIGURE 4.14: Query execution time for DBPedia running on 16 nodes

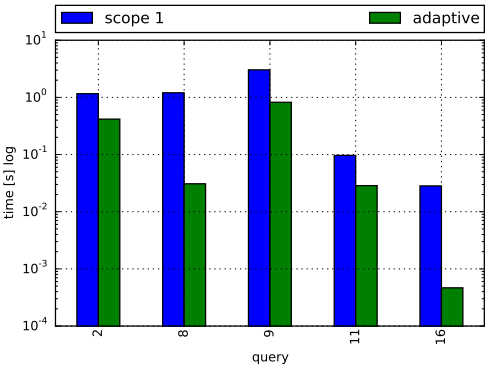


FIGURE 4.15: Scope-1 and adaptive partitioning on the most complex LUBM queries for 4 nodes.

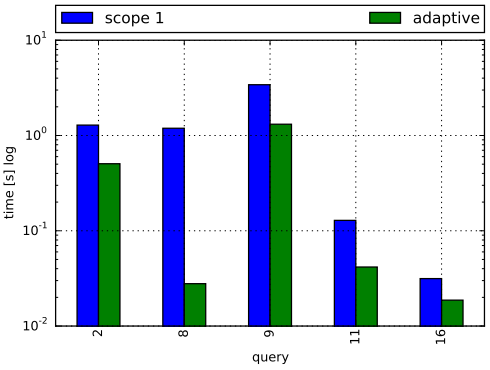


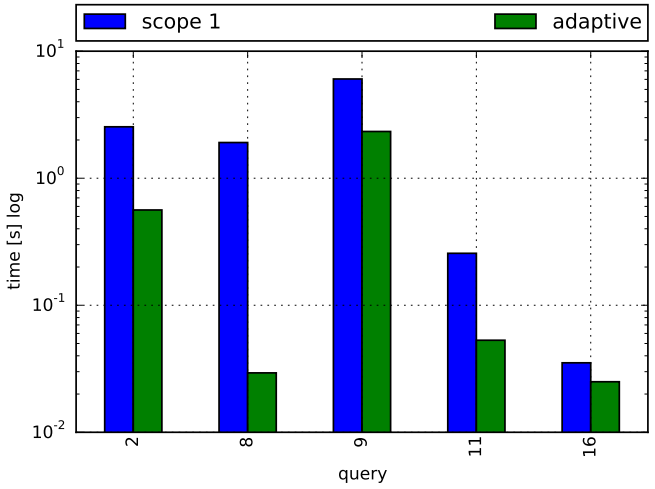
FIGURE 4.16: Scope-1 and adaptive partitioning on the most complex LUBM queries for 8 nodes.

observe that DiploCloud avoids almost all joins even for complex queries.

Queries and Results Analysis: The queries in the Table 4.5 can be classified into three main categories:

- relatively simple queries with a small output, which do not exhibit any significant difference when changing the kind of partitioning (e.g., queries 1,3,10,13); for those kinds of queries DiploCloud significantly outperforms other solutions because of our template and indexing strategies. Those queries are executed on Workers independently, fully in parallel, and results are sent to the Master.

- queries generating a big result set, where the main factor then revolves around transferring data to the master node (e.g., queries 6,14,17); for those queries,



q #	scope 1	adaptive
2	2.539618	0.562298
8	1.909516	0.029346
9	6.041795	2.332641
11	0.256411	0.053035
16	0.035231	0.024990

FIGURE 4.17: Scope-1 and adaptive partitioning on the most complex LUBM queries for 16 nodes.

DiploCloud is often closer to the other systems and suffers from the (potentially) high network latency associated with cloud environments.

- queries which typically require a distributed join, and for which the partitioning plays a significant role; DiploCloud performs very well on those queries (since most joins can be pre-computed in our molecules), with the exception of query 8, which is also characterized with a big output. For such queries, we differentiate two kinds of joins as briefly evoked above:
  - distributed joins (where we distribute intermediate results among the Workers and then process local joins in parallel); for that kind of queries the influence of the partitioning is not significant, though the co-location of molecules on the same node speedups the exchange of intermediate results, and hence the resulting query execution times
  - centralized joins; when a distributed join is too costly, the intermediate results are shipped to the master node where the final join is performed. We note that for queries 11 and 12, which are based on molecules indirectly related through one particular object, that for the coarse partitioning, all work is done by one node, where the particular object is located; that is the reason why this partitioning performs slower for those queries.

As presented above, DiploCloud often outperforms the other solutions in terms of the query execution time, mainly thanks to the fact that related pieces of data are already co-located in the molecules. For example for the query 2, DiploCloud has to perform

			4 workers			8 workers			16 workers		
	molecules configuration		scope-1	scope-2	adaptive	scope-1	scope-2	adaptive	scope-1	scope-2	adaptive
DiploCloud	master	memory (GB)	3.1	3.1	3.1	6.2	6.2	6.2	12.4	12.4	12.4
		loading time (sec)	157	154.8	158	372	374	371.83	786	796	784
	per worker	memory (GB)	2.32	6.06	3.35	2.41	6.27	3.42	2.7	6.45	4
		loading time (sec)	11.72	43	26.38	12	66	37.5	39	115	85
4store		loading time (sec)	226			449			893		

TABLE 4.6: Load times and size of the databases for the LUBM data set (Distributed Environment).

			4 workers	8 workers	16 workers
DiploCloud	master	memory (GB)	3.2	3.2	3.2
		loading time (sec)	1285	296	296
	per worker	memory (GB)	3.1	1.6	0.82
		loading time (sec)	28	14	7
4store		loading time (sec)	537	1284	1313

TABLE 4.7: Load times and size of the databases for the DBPedia data set (Distributed Environment).

only one join (or zero if we adapt the molecules) since all data related to the elements queried (e.g. GraduateStudent or Department) are located on one worker and are in addition directly co-located in memory; The only thing DiploCloud has to do in this case is to retrieve the list of elements on each Worker and to send it back to the Master, where it either performs a distributed hash-join (if we have molecules of scope-1), or it directly takes the result as is (if molecules are adapted). We have similar situations for the queries 8, 9, 11, and 16. For the query 7, we cannot take advantage of the pre-computed joins since we currently store RDF data as a directed graph and this particular query traverses the graph in the opposite direction (this is typically one kind of query DiploCloud is not optimized for at this stage). For the remaining queries, we do not require to perform any join at all, and can process the queries completely in parallel on the Workers and send back results to the Master, while the other systems have to take into account the intermediate joins (either locally or in a distributed fashion). Another group of queries for which DiploCloud should be further optimized are queries with high numbers of returned records, like the queries 6 or 14. In some cases we still outperform other systems for those queries, but the difference is not as significant.

Data Loading: The Table 4.6 gives the loading times for 4store and DiploCloud using the LUBM datasets and different partitioning strategies. We observe that the size taken by the deeper molecules (scope 2) rapidly grows, though the adaptive molecules strike a good balance between depth and size (we loaded the data according to the final version of the adaptive partitioning in that case in order to have comparable results for all variants). Using our parallel batch-loading strategies and adaptive partitioning, DiploCloud

is more than 10 times faster than 4store at loading data for the biggest deployment. The Table 4.7 reports the corresponding numbers for the DBPedia dataset.

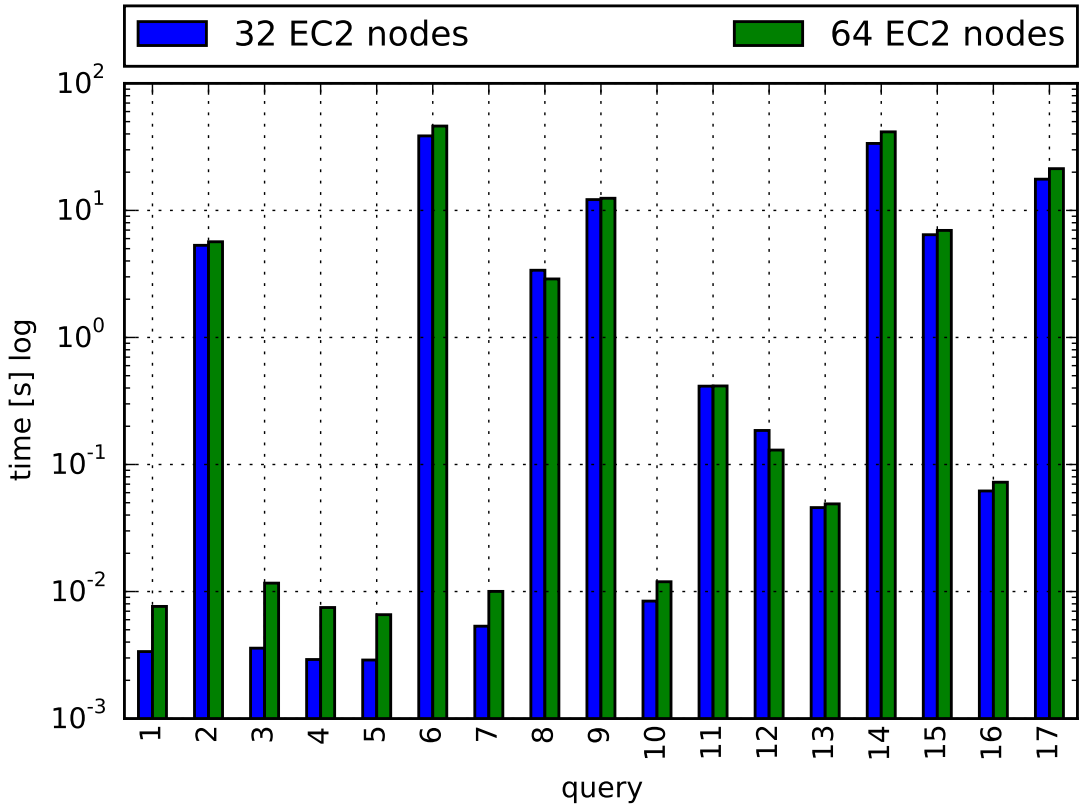
EC2 Deployment: Finally, to evaluate how DiploCloud performs in bigger cloud environments, we deployed it on Amazon EC2 instances[4]. We picked an M3 Extra Large Instance for the Master, and M1 Large Instances for the Workers, and load the LUBM 1600 dataset on 32 and 64 nodes. The results (see Figures 4.18) are comparable to those obtained on our own cluster, though slower, due to the larger network latency on EC2 (hence emphasizing once more the importance of minimizing distributed operations in the cloud, as DiploCloud does).

We also tested out adaptive partitioning approach on the EC2 infrastructure. The results are available on the Figures 4.19 and 4.20 (log-scale). Here again we show that by co-locating all frequently queried elements we can significantlly increase the performance. Co-location is especially important in environments where the network is not reliable so that we can minimize the amount of transferred data. We performed a small analysis of the network latency in that case. We measured the time spent on the Workers and Master for pure query execution and discovered that the network overhead is between 40% and 70% of the query execution time.

4.6 Conclusions

DiploCloud implements our techniques to efficient and scalable management of Linked Data in the cloud. From our perspective, it strikes an optimal balance between intra-operator parallelism and data co-location by considering recurring, fine-grained physiological Linked Data partitions and distributed data allocation schemes, leading however to potentially bigger data (redundancy introduced by higher scopes or adaptive molecules) and to more complex inserts and updates. Out methods are particularly suited to clusters of commodity machines and cloud environments where network latencies can be high, since they systematically tries to avoid all complex and distributed operations for query execution. Our experimental evaluation showed that our implementation of presented methods very favorably compare to the state-of-the-art systems in such environments.

[4]http://aws.amazon.com/ec2/instance-types/



q #	32 EC2 nodes	64 EC2 nodes	q #	32 EC2 nodes	64 EC2 nodes
1	0.003368	0.007636	10	0.008409	0.011940
2	5.312672	5.666275	11	0.413627	0.415163
3	0.003582	0.011645	12	0.185264	0.129909
4	0.002916	0.007484	13	0.045790	0.048970
5	0.002890	0.006573	14	33.673436	41.534871
6	38.603498	46.140534	15	6.437457	6.960480
7	0.005336	0.010024	16	0.061883	0.072513
8	3.380344	2.887932	17	17.615777	21.314872
9	12.166524	12.463822			

FIGURE 4.18: Query execution time on Amazon EC2 for 1600 Universities from LUBM dataset.

In the next chapter we extend our molecule-based storage mode to include provenance data in a very compact way. We present and compare several ways of storing provenance in Linked Data. We also present a way to track provenance of query execution to provide information how exactly the query results were derived, i.e. which pieces of data and how were combined it obtain the results.

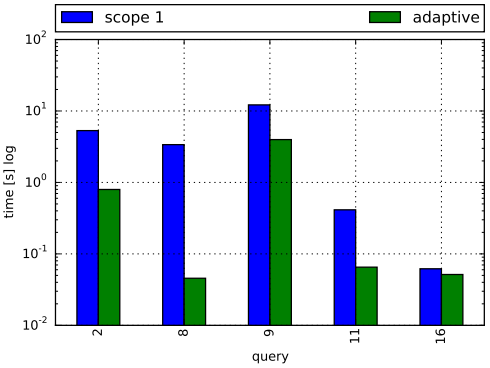


FIGURE 4.19: Scope-1 and adaptive partitioning on Amazon EC2 (32 Nodes) for 1600 Universities from LUBM dataset.

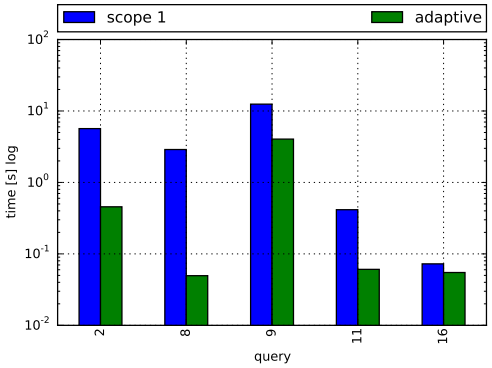


FIGURE 4.20: Scope-1 and adaptive partitioning on Amazon EC2 (64 Nodes) for 1600 Universities from LUBM dataset.

Chapter 5

Storing and Tracing Provenance in a Linked Data Management System

In the previous chapter we presented efficient techniques to store and query Big Linked Data in the cloud. Given the heterogeneity of the data one can find on the Linked Data cloud, being able to trace back the provenance of query results is rapidly becoming a must-have feature of Linked Data management systems. While provenance models have been extensively discussed in recent years, little attention has been given to the efficient implementation of provenance-enabled queries inside data stores. This chapter extend the techniques presented in Chapter 4 to efficiently handle such queries. We present two different storage models to physically co-locate lineage and instance data, and for each of them describe algorithms for tracing provenance at two granularity levels. In addition, we present the results of a comprehensive empirical evaluation of our methods over two different datasets and workloads. We present our implementation of this approach in TripleProv.

5.1 System Overview

In the following, we give a high-level overview of TripleProv, a native RDF store supporting the efficient generation of provenance polynomials during query execution. TripleProv is based on our storing and querying techniques described in details in Chapter 4.

Figure 5.1 gives an overview of the architecture of our system, composed of a series of subcomponents:

a query executor responsible for parsing the incoming query, rewriting the query plans, collecting and finally returning the results along with the provenance polynomials to the client;

a key index in charge of encoding URIs and literals into compact system identifiers and of translating them back;

a type index clustering all keys based on their types;

a series of RDF *molecules* storing RDF data as very compact subgraphs;

a molecule index storing for each key the list of molecules where the key can be found.

We give below an overview of the three most important subcomponents of our system in a provenance context, i.e., the key index, the molecules, and the molecule index.

The key index is responsible for encoding all URIs and literals appearing in the triples into a unique system id (key), and back. We use a tailored lexicographic tree to parse URIs and literals and assign them a unique numeric ID. The lexicographic tree we use is essentially a prefix tree splitting the URIs or literals based on their common prefixes (since many URIs share the same prefixes), such that each substring prefix is stored once and only once in the tree. A key ID is stored at every leaf, which is composed of a type prefix (encoding the type of the element, e.g., $Student$ or $xsd:date$) and of an auto-incremented instance identifier. This prefix tree allows us to completely avoid potential collisions (caused for instance when applying hash functions on very large datasets), and also lets us compactly co-locate both type and instance ids into one compact key. A second structure translates the keys back into their original form. It is composed of a set of inverted indices (one per type), each relating an instance ID to its corresponding URI / literal in the lexicographic tree in order to enable efficient key look-ups.

In their simplest form, RDF molecules [43] are similar to property tables [111] and store, for each subject, the list or properties and objects related to that subject. Molecule clusters are used in two ways: to logically group sets of relates URIs and literals (thus, pre-computing joins), and to physically co-locate information related to a given object on disk and in main-memory to reduce disk and CPU cache latency. TripleProv stores

such lists of molecules very compactly on disk or in main memory, thus making query resolution fast in many contexts.

In addition to the molecules themselves, the system also maintains a molecule index storing for each key the list of local molecules storing that key (e.g., "key 15123 [Course12] is stored in molecules 23521 [root:Student543] and 23522 [root:Student544]"). This index is particularly useful to answer triple-pattern queries as we explain below in Section 5.4.

TripleProv extends our molecule storage (cf. Chapter 4) in two important ways: i) it introduces new storage structures to store lineage data directly co-located to the instance data and ii) it supports the efficient generation of provenance polynomials during query execution. Figure 5.1 gives an overview of our system in action, taking as input a SPARQL query (and optionally a provenance granularity level), and returning as output the results of the query along with the provenance polynomials derived during query execution.

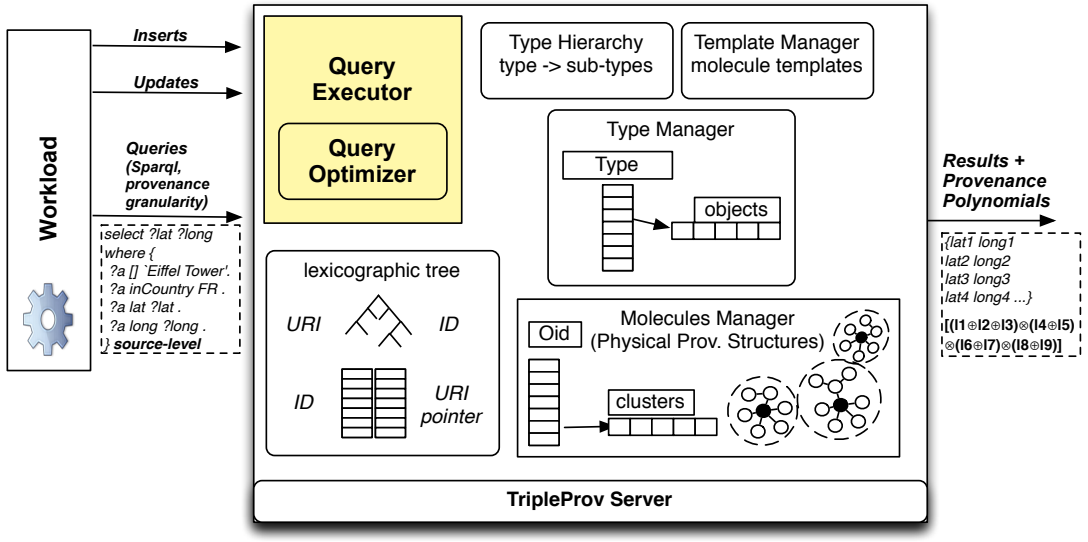


FIGURE 5.1: The architecture of TripleProv; the system takes as input queries (and optionally a provenance granularity level), and produces as output query results along with their corresponding provenance polynomials.

5.2 Provenance Polynomials

The first question we tackle is how to represent provenance information that we want to return to the user in addition to the results themselves. Beyond listing the various sources involved in the query, we want to be able to characterize the specific ways in which each source contributed to the query results. There has been quite a bit of work on provenance models and languages recently. Here, we leverage the notion of *provenance polynomials*. However, in contrast to the many recent pieces of work in this space, which tackled more theoretical issues, we focus on the practical realization of this model within a high performance triple store to answer queries seen as useful in practice. Specifically, we focus on two key requirements:

1. the capability to pinpoint, for each query result, the exact source from which the result was selected;
2. the capability to trace back, for each query result, the complete list of sources and how they were combined to deliver a result.

Hence, we support two different provenance operators at the physical level, one called $pProjection$, meeting the first requirement and pinpointing to the exact sources from which the result was drawn, and a second one called $pConstraint$, tracing back the full lineage of the results.

At the logical level, we use two basic operators to express the provenance polynomials. The first one ($\oplus$) to represent unions of sources, and the second ($\otimes$) to represent joins between sources.

Unions are used in two cases when generating the polynomials. First, they are used when a constraint or a projection can be satisfied with triples coming from multiple sources (meaning that there are more than one instance of a particular triple which is used for a particular operation). The following polynomial:

$$l1 \oplus l2 \oplus l3$$

for instance, encodes the fact that a given result can originate from three different sources ($l1$, $l2$, or $l3$, see below Section 5.2.1 for a more formal definition of the *sources*). Second, unions are also used when multiple entities satisfy a set of constraints or projections (like the collection 'provenanceGlobal' in 5.4.1).

As for the join operator, it can also be used in two ways: to express the fact that sources were joined to handle a constraint or a projection, or to handle object-subject or object-object joins between a few sets of constraints. The following polynomial:

$$(l1 \oplus l2) \otimes (l3 \oplus l4)$$

for example, encodes the fact that sources $l1$ or $l2$ were joined with sources $l3$ or $l4$ to produce results.

5.2.1 Provenance Granularity Levels

One can model RDF data provenance at different granularity levels. Current approaches, typically, return a list of named graphs from which the answer was computed. Our system, besides generating polynomials summarizing the complete provenance of results, also supports two levels of granularity. First, a lineage l_i (i.e., an element appearing in a polynomial) can represent the source of a triple, (e.g., the fourth element in a quadruple). We call this granularity level *source-level*. Alternatively, a lineage can represent a quadruple (i.e., a triple plus its corresponding source). This second type of lineage produces polynomials consisting of all the pieces of data (i.e., quadruples) that were used to answer the query, including all intermediate results. We call this level of granularity *triple-level*.

In addition to those two provenance granularity levels, TripleProv also supports two levels of aggregation to output the results. The default level aggregates the polynomials for all results, i.e., it gives an overview of all triples/sources used during query execution. The second level provides full provenance details, explaining—for each single result—the way (polynomial) through which this particular result was constructed. Both aggregation levels typically perform similarly (since one is basically derived from the other), hence, we mainly focus on aggregated polynomial results in the following.

5.3 Storage Models

We now discuss the RDF storage model of TripleProv, based on our previous techniques (cf. Chapter 4), and extended with new physical storage structures to store provenance.

5.3.1 Native Storage Model

RDF Templates When ingesting new triples, TripleProv first identifies RDF subgraphs. It analyzes the incoming data and builds what are termed molecule *templates*. These templates act as data prototypes to create RDF molecules. Figures 5.2 i) gives a template example that co-locates information relating to Student instances. Once the templates have been defined, the system starts creating molecule identifiers based on the molecule roots (i.e., central molecule nodes) that it identifies in the incoming data.

While creating molecule templates and molecule identifiers, the system takes care of two additional data gathering and analysis tasks. First, it inspects both the schema and instance data to determine all subsumption (subclass) relations between the classes, and maintains this information in a compact *type hierarchy*. In case two unrelated types are assigned to a given instance, the system creates a new virtual type composed of the two types and assigns it to the instance.

RDF Molecules TripleProv stores the primary copy of the RDF data as RDF molecules, which can be seen as hybrid data structures borrowing both from property tables and from RDF subgraphs. They store, for every template defined by the template manager, a compact list of objects connected to the root of the molecule. Figure 5.2 (ii) gives an example of a molecule. Molecules co-locate data and are template-based, hence can store data extremely compactly. The molecule depicted in Figures 5.2 (ii), for instance, contains 15 triples (including type information), and would hence require 45 URIs/literals to be encoded using a standard triple-based serialization. Our molecule, on the other hand, only requires the storage 10 keys to be correctly defined, yielding a compression ratio of $1 : 4.5$.

Data (or workload) inspection algorithms can be exploited in order to *materialize* frequent joins though molecules. In addition to materializing the joins between an entity and its corresponding values (e.g., between a student and his/her firstname), one can hence materialize the joins between two semantically related entities (e.g., between a student and his/her advisor) that are frequently co-accessed by co-locating them in the same molecule.

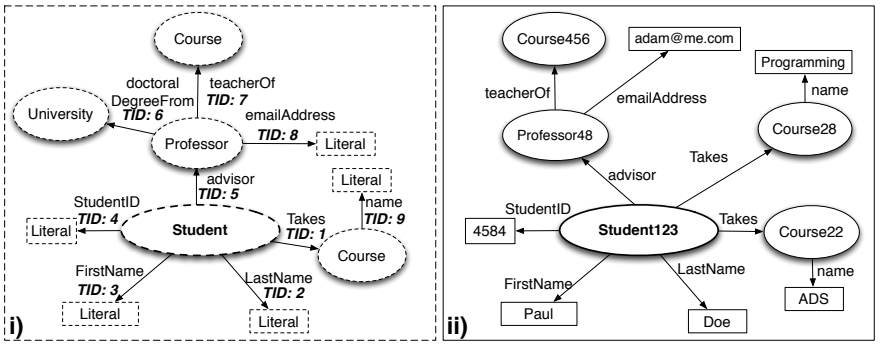


FIGURE 5.2: A molecule template (i) along with one of its RDF molecules (ii).

5.3.2 Storage Model Variants for Provenance

We now turn to the problem of extending the physical data structures of TripleProv to support provenance queries. There are a number of ways one could implement this in our system. A first way of storing provenance data would be to simply annotate every object in the database with its corresponding source. This produces quadruple physical data structures ($SPOL$, where S is the subject of the quadruple, P its predicate, O its object, and L its source), as illustrated in Figure 5.3, $SPOL$). The main advantage of this variant is its ease of implementation (e.g., one simply has to extend the data structure storing the object to also store the source data). Its main disadvantage, however, is memory consumption since the source data has to be repeated for each triple.

One can try to physically co-locate the source and the triples differently, which results in a different memory consumption profile. One extreme option would be to regroup molecules in clusters based on their source ($LSPO$ clustering). This, however, has the negative effect of splitting our original molecules into several structures (one new structure per new source using a given subject), thus braking pre-computed joins and defeating the whole purpose of storing RDF as molecules in the first place. The situation would be even worse for deeper molecules (i.e., molecules storing data at a wider scope, or considering large RDF subgraphs). On the other hand, such structures would be quite appropriate to resolve vertical provenance queries, i.e., queries that explicitly specify which sources to consider during query execution (however, our goal is not to optimize for such provenance queries in the present work).

The last two options for co-locating source data and triples are $SLPO$ and $SPLO$. $SLPO$ co-locates the source data with the predicates, making it technically speaking compelling, since it avoids the duplication of the same source inside a molecule, while at the same time still co-locating all data about a given subject in one structure. $SPLO$,

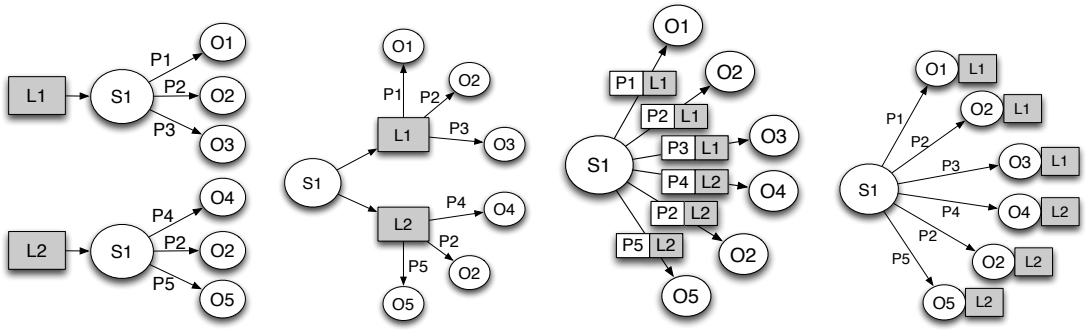
finally, co-locates the source data with the predicates in the molecules. In practice, this last physical structure is very similar to $SPOL$ in terms of storage requirements, since it rarely happens that a given source uses the same predicates with many values. Compared to $SPOL$, it also has the disadvantage of considering a relatively complex structure (PO) in the middle of the physical storage structure (e.g., as the key of a hash-table mapping to the objects).

These different ways of co-locating data naturally result in different memory overheads. The exact overhead, however, is highly dependent on the dataset considered, its structure, and the homogeneity / heterogeneity of the sources involved for the different subjects. Whenever the data related to a given subject comes from many different sources (e.g., when the objects related to a given predicate come from a wide variety of sources), the overhead caused by repeating the predicate in the $SLPO$ might not be compensated by the advantage of co-location. In such cases, models like $SPLO$ or $SPOL$ might be more appropriate. If, on the other hand, a large portion of the different objects attached to the predicates come from the same sources, then the $SPLO$ model might pay off (see also Section 5.5 for a discussion on those points). From this analysis, it is evident that no single provenance storage model is overall best—since the performance of such models is somewhat dependent on the queries, of course, but also on the homogeneity / heterogeneity of the datasets considered.

For the reasons described above, we focus below on two very different storage variants in our implementation: $SLPO$, which we refer to as *data grouped by source* in the following (since the data is regrouped by source inside each molecule), and $SPOL$, which we refer to as *annotated provenance* since the source data is placed like an annotation next to the last part of the triple (object). We note that implementing such variants at the physical layer of the database system is a significant effort, since all higher-level calls (i.e., all operators) directly depend on how the data is laid-out on disk and in memory.

5.4 Query Execution

We now turn to the way we take advantage of the source information stored in the molecules to produce provenance polynomials. We have implemented specific query execution strategies in TripleProv that allow to return a complete record of how the results were produced (including detailed information of key operations like unions and joins) in addition to the results themselves. The provenance polynomials our system



(A) LSPO Storage (B) SLPO Storage (C) SPLO Storage (D) SPOL Storage

FIGURE 5.3: The four different physical storage models identified for co-locating source information (L) with the triples (SPO) inside RDF molecules.

produce can be generated at source-level or at triple-level, and both for detailed provenance records and for aggregated provenance records.

5.4.1 General Query Answering Algorithm

Algorithm 3 gives a simplified view on how simple star-like queries are answered in TripleProv. Given a SPARQL query, our system first analyzes the query to produce a physical query plan, i.e., a tree of operators that are then called iteratively to retrieve molecules susceptible of containing data relevant to the query. The molecules are retrieved by taking advantage of the lexicographic tree to translate any unbound variables in the query into keys, and then by using the molecule index to locate all molecules containing those keys (see Chapter 4 for details).

In parallel to the classical query execution process, TripleProv keeps track of the various triples and sources that have been instrumental in producing results for the query. For each molecule inspected, our system keeps track of the provenance of any triple matching the current pattern being handled (*checkIfTripleExists*). In a similar fashion, it keeps track of the provenance of all entities being retrieved in the projections (*getEntity*). In case multiple molecules are used to construct the final results, the system keeps track of the local provenance of the molecules by performing a union of the local provenance data using a global provenance structure (provenanceGlobal.union). To illustrate such operations and their results, we describe below the execution of two sample queries.

Algorithm 3 Simplified algorithm for provenance polynomials generation

Require: SPARQL query q

```
 1: results ← NULL
 2: provenanceGlobal ← NULL
 3: getMolecules ← q.getPhysicalPlan
 4: constraints ← q.getConstraints
 5: projections ← q.getProjections

 6: for all getMolecules do
 7:     provenanceLocal ← NULL
 8:     for all constrains do
 9:         if checkIfTripleExists then
10:             provenanceLocal.join
11:         else
12:             nextMolecule
13:         end if
14:     end for

15:     for all projections do
16:         entity = getEntity(for particular projection)
17:         if entity is NOT EMPTY then
18:             results.add(entity)
19:             provenanceLocal.join
20:         else
21:             nextMolecule
22:         end if
23:     end for

24:     if allConstrainsSatisfied AND allProjectionsAvailable then
25:         provenanceGlobal.union
26:     end if
27: end for
```

5.4.2 Example Queries

The first example query we consider is a simple star query, i.e., a query defining a series of triple patterns, all joined on an entity that has to be identified:

```
select ?lat ?long
where {
 ?a [] ``Eiffel Tower''. (<- 1st constraint)
 ?a inCountry FR .     (<- 2nd constraint)
 ?a lat ?lat .       (<- 1st projection)
 ?a long ?long .    (<- 2nd projection)
}
```

To build the corresponding provenance polynomial, TripleProv first identifies the constraints and projections from the query (see the annotated listing above). The query executor chooses the most selective pattern to start looking up molecules (in this case the first pattern), translates the bound variable ("Eiffel Tower") into a key, and retrieves all molecules containing that key. Each molecule is then inspected in turn to determine whenever both i) the various constraints can be met (*checkIfTripleExists* in the algorithm) and ii) the projections can be correctly processed (*getEntity* in the algorithm). Our system keeps track of the provenance of each result, by joining the local provenance information of each triple used during query execution to identify the result.

Finally, a provenance polynomial such as the following is issued:

$$[(l1 \oplus l2 \oplus l3) \otimes (l4 \oplus l5) \otimes (l6 \oplus l7) \otimes (l8 \oplus l9)].$$

This particular polynomial indicates that the first constraint has been satisfied with lineage $l1$, $l2$ or $l3$, while the second has been satisfied with $l4$ or $l5$. It also indicates that the first projection was processed with elements having a lineage of $l6$ or $l7$, while the second one was processed with elements from $l8$ or $l9$. The triples involved were joined on variable $?a$, which is expressed by the join operation ($\otimes$) in the polynomial. Such a polynomial can contain lineage elements either at the source level or at the triple level, and can be returned both in an aggregate or detailed form.

The second example we examine is slightly more involved, as it contains two sets of constraints and projections with an upper-level join to bind them:

```
select ?l ?long ?lat
where {
  (-- first set)
  ?p name ``Krebs, Emil'' .
  ?p deathPlace ?l .

  (-- second set)
  ?c [] ?l .
  ?c featureClass P .
  ?c inCountry DE .
  ?c long ?long .
  ?c lat ?lat .
}
```

The query execution starts similarly as for the first sample query. After resolving the first two patterns, the second set of patterns is processed by replacing variable $?l$ with the results derived from the first set, and by joining the corresponding lineage elements.

Processing the query in TripleProv automatically generates provenance polynomials such as the following:

$$[(l1 \oplus l2 \oplus l3) \otimes (l4 \oplus l5)] \bigotimes [(l6 \oplus l7) \otimes (l8) \otimes (l9 \oplus l10) \otimes (l11 \oplus l12) \otimes (l13)]$$

where an upper-level join ($\bigotimes$) is performed across the lineage elements resulting from both sets. More complex queries are solved similarly, by starting with the most selective patterns and iteratively joining the results and the provenance information across molecules.

5.5 Performance Evaluation

To empirically evaluate our approach, we implemented the storage models and query execution strategies described above. Specifically, we implemented two different storage models: SPOL and SLOP. For each model, we support two different levels of provenance granularity: source granularity and triple granularity. Our system does not parse SPARQL queries at this stage (adapting a SPARQL parser is currently in progress), but offers a similar, high-level and declarative API to encode queries using triple patterns. Each query is then encoded into a logical physical plan (a tree of operators), which is then optimized into a physical query plan as for any standard database system. In that sense, we follow the algorithms described above in Section 5.4.

In the following, we experimentally compare the vanilla version of TripleProv, i.e., the bare-metal system without provenance storage and provenance polynomials generation, to both SPOL and SLOP on two different datasets and workloads. For each provenance storage model, we report results both for generating polynomials at the source and at the triple granularity levels. We also compare our system to 4store[1], where we take advantage of 4store's quadruple storage to encode provenance data as named graphs and manually rewrite queries to return some provenance information to the user (as discussed below, such an approach cannot produce valid polynomials, but is interesting anyhow to illustrate the fundamental differences between TripleProv and standard RDF stores when it comes to provenance).

[1] `http://4store.org/`

We note that the RDF storage system that TripleProv extends (i.e., the vanilla version of TripleProv) in Chapter 4 has already been compared to a number of other well-known database systems, including Postgres, AllegroGraph, BigOWLIM, Jena, Virtuoso, 4store, and RDF 3X. The system is on average 30 times faster than the fastest RDF data management system we have considered (RDF-3X) for LUBM queries, and on average 350 times faster than the fastest system we have considered (Virtuoso) on more complex analytics.

5.5.1 Hardware Platform

All experiments were run on a HP ProLiant DL385 G7 server with an AMD Opteron Processor 6180 SE (24 cores, 2 chips, 12 cores/chip), 64GB of DDR3 RAM and running Ubuntu 12.04.3 LTS (Precise Pangolin). All data were stored on a recent 3 TB Serial ATA disk.

5.5.2 Datasets

We used two different sources for our data: the Billion Triples Challenge (BTC)[2] and the Web Data Commons (WDC) [89].[3] Both datasets are collections of RDF data gathered from the Web. They represent two very different kinds of RDF data. The Billion Triple Challenge dataset was crawled based on datasets provided by Falcon-S, Sindice, Swoogle, SWSE, and Watson using the MultiCrawler/SWSE framework. The Web Data Commons project extracts all Microformat, Microdata and RDFa data from the Common Crawl Web corpus, the largest and most up-to-data Web corpus that is currently available to the public, and provides the extracted data for download in the form of RDF-quads and also in the form of CSV-tables for common entity types (e.g., products, organizations, locations, etc.).

Both datasets represent typical collections of data gathered from multiple sources, thus tracking provenance for them seems to precisely address the problem we focus on. We consider around 115 million triples for each dataset (around 25GB). To sample the data, we first pre-selected quadruples satisfying the set of considered queries. Then, we

[2] `http://km.aifb.kit.edu/projects/btc-2009/`
[3] `http://webdatacommons.org/`

randomly sampled additional data up to 25GB. Both datasets are available for download on our website[4].

5.5.3 Workloads

We consider two different workloads. For BTC, we use eight existing queries originally proposed in [91]. In addition, we added two queries with UNION and OPTIONAL clauses, which we thought were missing in the original set of queries. Based on the queries used for the BTC dataset, we wrote 7 new queries for the WDC dataset, encompassing different kinds of typical query patterns for RDF, including star-queries of different sizes and up to 5 joins, object-object joins, object-subject joins, and triangular joins. In addition, we included two queries with UNION and OPTIONAL clauses. As for the data, the workloads we consdiered are available on our website.

5.5.4 Experimental Methodology

As is typical for benchmarking database systems (e.g., for tpc-x[5]), we include a warm-up phase before measuring the execution time of the queries in order to measure query execution times in a steady-state mode. We first run all the queries in sequence once to warm-up the systems, and then repeat the process ten times (i.e., we run for each system we benchmark a total of 11 batches, each containing all the queries we consider in sequence). We report the mean values for each query. In addition, we avoided the artifacts of connecting from the client to the server, of initializing the database from files, and of printing results; We measured instead the query execution times inside the database system only.

5.5.5 Variants Considered

As stated above, we implemented two storage models (grouped/co-located and annotated) in TripleProv and for each model we considered two granularity levels for tracking provenance (source and triple). This gives us four different variants to compare against the vanilla version of our system. Our goal is in that sense to understand the

[4] `http://exascale.info/tripleprov`
[5] *http://www.tpc.org/*

various trade-offs of the approaches and to assess the performance penalty caused by enabling provenance. We use the following abbreviations to refer to the different variants in the following:

V: the vanilla version of our system (i.e., the version where provenance is neither stored nor looked up during query execution);

SG: source-level granularity, provenance data grouped by source;

SA: source-level granularity, annotated provenance data;

TG: triple-level granularity, provenance data grouped by source;

TA: triple-level granularity, annotated provenance data.

5.5.6 Comparison to 4Store

First, we start by an informal comparison with 4Store to highlight the fundamental differences between our provenance-enabled system and a quad-store supporting named graphs.

While 4Store storage takes into account quads (and thus, source data can be explicitly stored), the system does not support the generation of detailed provenance polynomials tracing back the lineage of the results. Typically, 4Store simply returns standard query results as any other RDF store. However, one can try to simulate some basic provenance capabilities by leveraging the *graph* construct in SPARQL and extensively rewriting the queries by inserting this construct for each query pattern.

As an example, rewriting the first sample query we consider above in Section 5.4.1 would result in the following:

```
select ?lat ?long ?g1 ?g2 ?g3 ?g4
where {
  graph ?g1 {?a [] "Eiffel Tower" . }
  graph ?g2 {?a inCountry FR .  }
  graph ?g3 {?a lat ?lat .      }
  graph ?g4 {?a long ?long .    }
}
```

However, such a query processed in 4store would obviously not produce full-fledged provenance polynomials. Rather, a simple list of concatenated sources would be returned, whether or not they were in the end instrumental to derive the final results of the query, as follows:

```
lat long l1 l2 l4 l4, lat long l1 l2 l4 l5,
lat long l1 l2 l5 l4, lat long l1 l2 l5 l5,
lat long l1 l3 l4 l4, lat long l1 l3 l4 l5,
lat long l1 l3 l5 l4, lat long l1 l3 l5 l5,

lat long l2 l2 l4 l4, lat long l2 l2 l4 l5,
lat long l2 l2 l5 l4, lat long l2 l2 l5 l5,
lat long l2 l3 l4 l4, lat long l2 l3 l4 l5,
lat long l2 l3 l5 l4, lat long l2 l3 l5 l5,

lat long l3 l2 l4 l4, lat long l3 l2 l4 l5,
lat long l3 l2 l5 l4, lat long l3 l2 l5 l5,
lat long l3 l3 l4 l4, lat long l3 l3 l4 l5,
lat long l3 l3 l5 l4, lat long l3 l3 l5 l5.
```

The listing above consists of all permutations of values bound to variables referring to data used to answer the original query ($?lat$, $?long$). Additionally, all named graphs used to resolve the triple patterns from the query (relating to variables $?g1$, $?g2$, $?g3$, and $?g4$) are also integrated. Obviously, this type of outcome is insufficient for correctly tracing back the provenance of the results.

Whereas in TripleProv the answer to the original query (without the graph clauses) would be as follows:

```
lat long
```

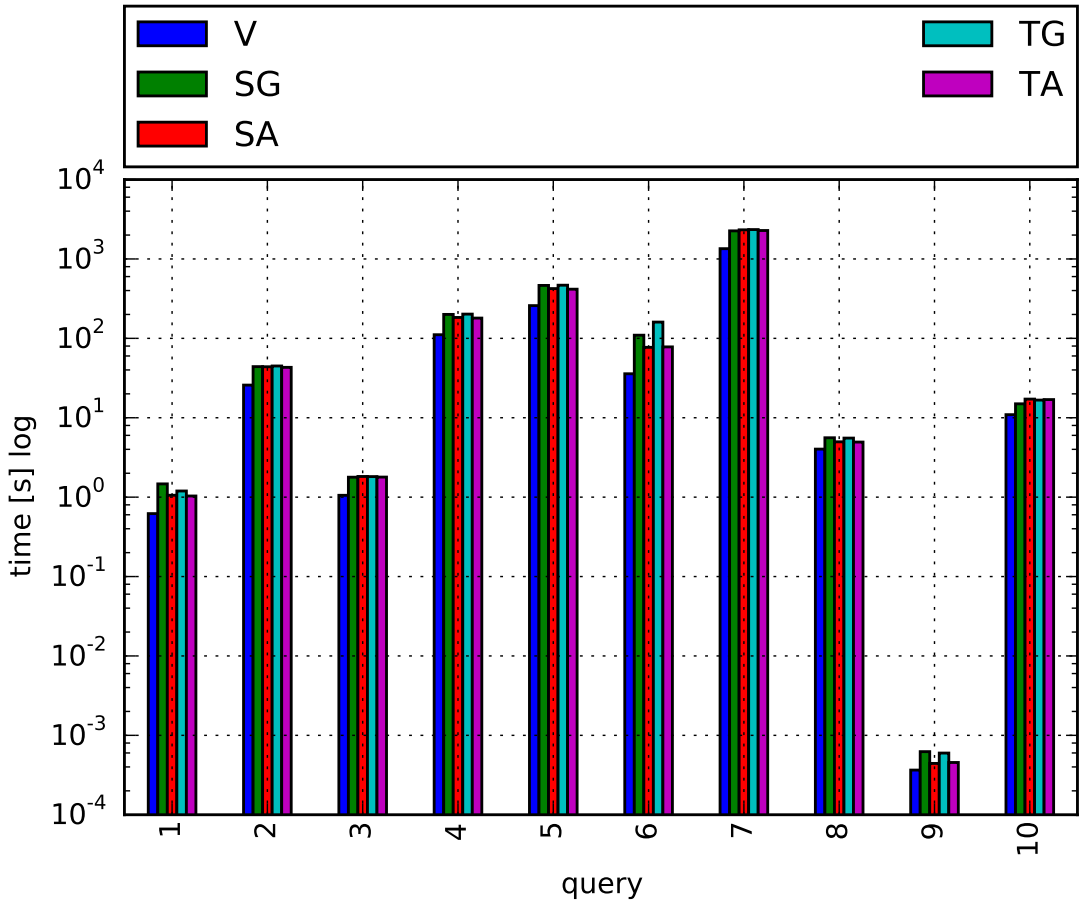
with, in addition, the following compact provenance polynomial:

$$[(l1 \oplus l2 \oplus l3) \otimes (l2 \oplus l3) \otimes (l4 \oplus l5) \otimes (l4 \oplus l5)].$$

5.5.7 Query Execution Times

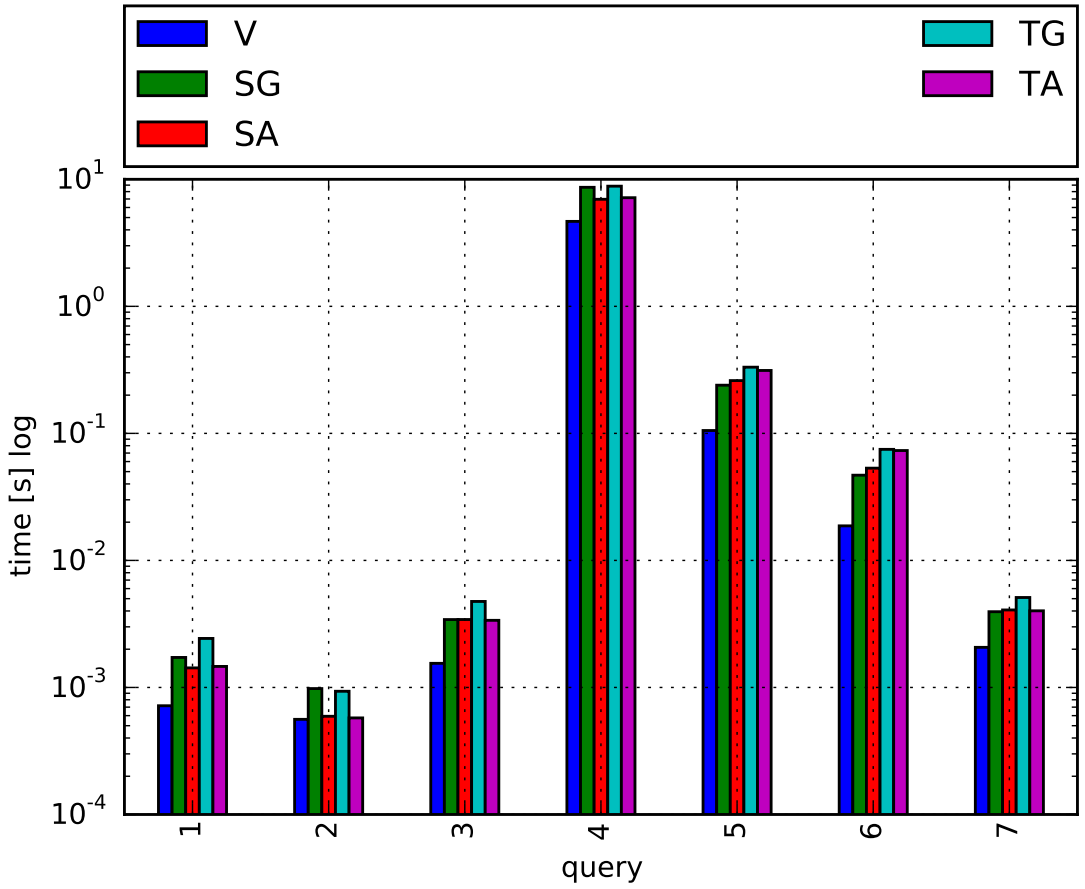
The Figure 5.4 gives the query execution times for the BTC dataset, while the Figure 5.5 presents similar results for WDC. We also explicitly give the overhead generated by our various approaches compared to the non-provenance-enabled (vanilla) version, in the Figure 5.6 for BTC, and in the Figure 5.7 for WDC, respectively.

Overall, the performance penalty created by tracking provenance in TripleProv ranges from a few percents to almost 350%. Clearly, we observe a significant difference between the two main provenance storage models implemented (SG vs SA and TG vs TA). Retrieving data from co-located structures takes about 10%-20% more time than from simply annotated graph nodes. We experimented with various physical structures for



query #	V	SG	SA	TG	TA
1	0.620354	1.468967	1.056384	1.195556	1.034092
2	25.781301	44.043982	43.874489	44.866396	43.136169
3	1.055016	1.783890	1.821844	1.814778	1.789879
4	111.107712	200.284815	183.341964	201.988128	180.039427
5	258.409811	464.092544	423.462056	467.121539	416.138520
6	35.804843	109.601699	77.090780	160.291805	78.071724
7	1347.435725	2258.412791	2327.513576	2344.101949	2281.881660
8	4.027142	5.597102	4.976826	5.544529	4.940878
9	0.000365	0.000623	0.000443	0.000598	0.000455
10	10.934833	14.980501	17.178909	16.689216	16.935428

FIGURE 5.4: Query execution times (in seconds) for the BTC dataset (logarithmic scale)

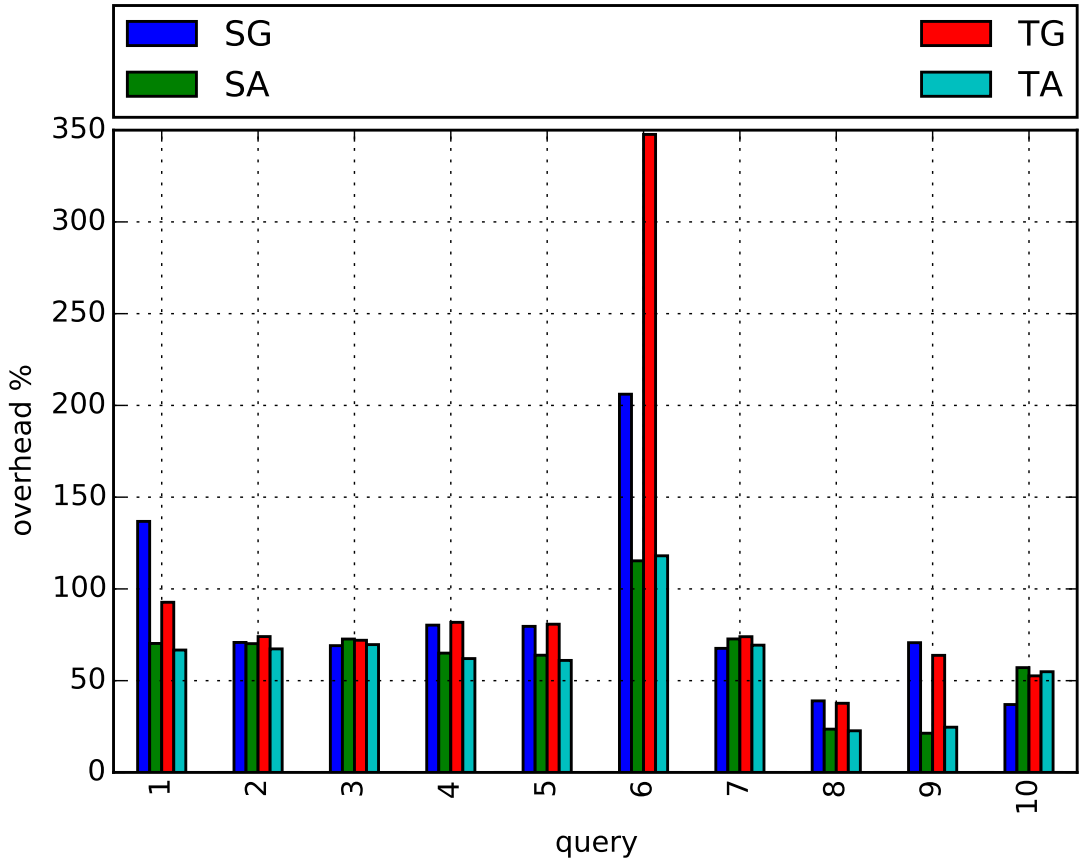


query #	V	SG	SA	TG	TA
1	0.000718	0.001723	0.001426	0.002431	0.001464
2	0.000561	0.000980	0.000593	0.000933	0.000576
3	0.001548	0.003423	0.003426	0.004753	0.003376
4	4.663725	8.649200	6.961700	8.829900	7.164200
5	0.105323	0.239502	0.260361	0.331954	0.312802
6	0.018709	0.046867	0.053244	0.074857	0.073286
7	0.002069	0.003947	0.004074	0.005106	0.004011

FIGURE 5.5: Query execution times (in seconds) for the WDC dataset (logarithmic scale)

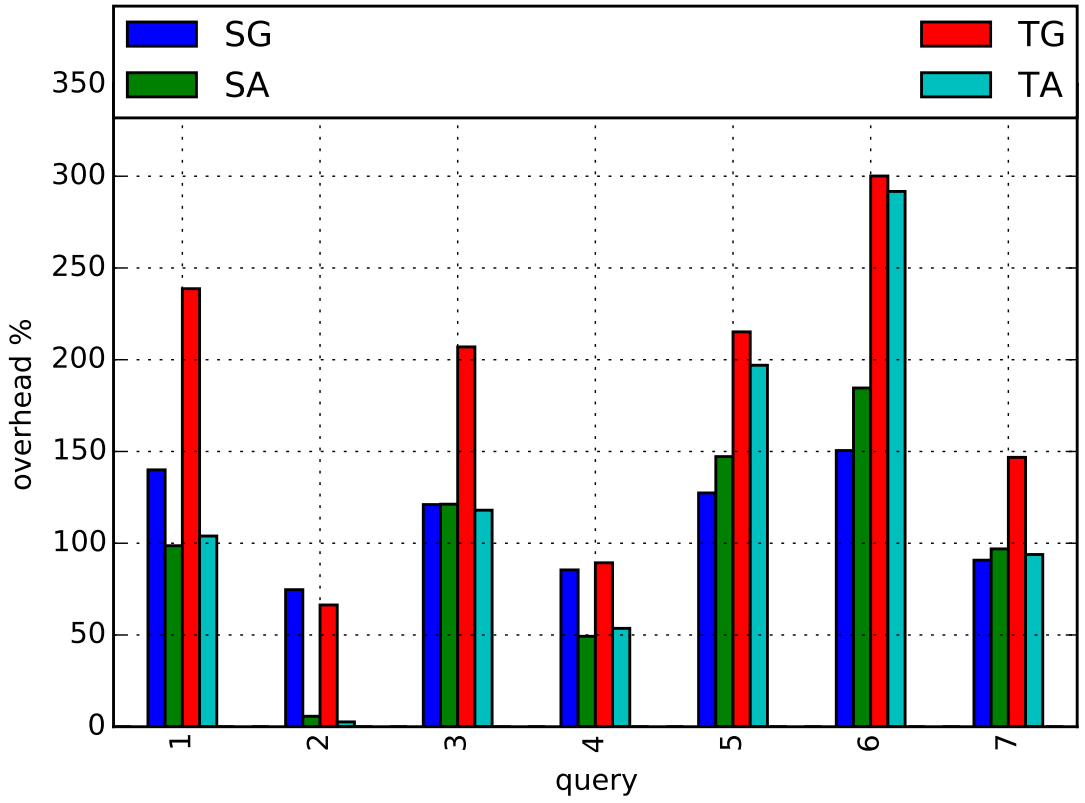
SG and TG, but could not significantly reduce this overhead, caused by the additional look-ups and loops that have to be considered when reading from extra physical data containers.

We also notice considerable difference between the two granularity levels (SG vs TG and SA vs TA). Clearly, the more detailed triple-level provenance granularity requires more time for query execution than the simpler source-level, because of the more complete physical structures that need to be created and updated while collecting the intermediate results sets.



query #	SG	SA	TG	TA
1	136.794915	70.287265	92.721448	66.693802
2	70.836925	70.179499	74.026890	67.315719
3	69.086452	72.683932	72.014179	69.654121
4	80.261849	65.012816	81.794877	62.040442
5	79.595559	63.872283	80.767726	61.038205
6	206.108591	115.308249	347.681914	118.047948
7	67.608202	72.736520	73.967626	69.349945
8	38.984467	23.582071	37.678985	22.689424
9	70.700986	21.358160	63.800657	24.644031
10	36.997982	57.102612	52.624334	54.875957

FIGURE 5.6: Overhead of tracking provenance compared to the vanilla version of the system for the BTC dataset



query #	SG	SA	TG	TA
1	139.983282	98.634717	238.701588	103.928671
2	74.688057	5.704100	66.345811	2.638146
3	121.066908	121.273573	206.974942	118.005683
4	85.456904	49.273381	89.331489	53.615404
5	127.398298	147.203164	215.178319	196.993808
6	150.509920	184.592277	300.117592	291.718336
7	90.768487	96.906718	146.805220	93.852102

FIGURE 5.7: Overhead of tracking provenance compared to the vanilla version of the system for the WDC dataset

Also, we observe some important differences between the query execution times from the two datasets we used, even for very similar queries (01-05 map directly from one dataset onto the other; 09BTC maps to 06WDC and 10BTC maps to 07WDC). Clearly, the efficiency of our provenance polynomial generation on a given query depends upon underlying data characteristics. One important dimension in that context is the heterogeneity—in terms of number of sources providing the data—of the dataset. The more heterogeneous the data, the better the annotated storage model performs, since this model makes no attempt at co-locating data w.r.t. the sources and hence avoids additional look-ups when many sources are involved. On the other hand, the more structured the data, the better the co-located models perform.

	V	G	A
Loading Time [s]	23.32	27.9	26.8
Memory Consumption [GB]	36.26	53.62	39.54

FIGURE 5.8: Loading times and memory consumption for the BTC dataset

	V	G	A
Loading Time [s]	27.46	67.78	30.56
Memory Consumption [GB]	42.53	66.22	50.29

FIGURE 5.9: Loading times and memory consumption for the WDC dataset

Finally, we briefly discuss two peculiar results appearing in Figure 5.6 fore queries 01 and 06. For query 01, the reason behind the large disparity in performance has to do with the very short execution times (at the level of 10^{-3} second), which cannot be measured more precisely and thus introduces some noise. The performance overheard for query 06 is caused by a very large provenance record on one hand, and a high heterogeneity in terms of sources for the elements that are used to answer the query.

5.5.8 Loading Times & Memory Consumption

Finally, we discuss the loading times and memory consumption for the various approaches. Figure 5.8 reports results for the BTC dataset, while Figure 5.9 provides similar figures for the WDC dataset.

Referring to loading times, the more complex co-located storage model requires more computations to load the data than the simpler annotation model, which obviously increases the time needed to load data. In terms of memory consumption, the experimental results confirm our analysis from Section 5.3; The datasets used for our experiments are crawled from the Web, and hence consider data collated from a wide variety of sources, which results in a high diversification of the sources for each subject. As we explained in Section 5.3, storage structures such as $SPLO$ or $SPOL$ are more appropriate in such a case.

5.6 Conclusions

In this chapter, we described our approach for managing Linked Data while also tracking provenance. To the best of our knowledge, this is the first work that translates theoretical insights from the database provenance literature into a high-performance triple store. We not only present simple tracing of sources for query answers, but also consider fine-grained multilevel provenance. In this chapter, we described two possible storage models for supporting provenance in Linked Data management systems. Our experimental evaluation shows that the overhead of provenance, even though considerable, is acceptable for the resulting provision of a detailed provenance trace. We note that both our query algorithms and storage models can be reused by other databases (e.g., considering property tables or subgraph storage structures) with only small modifications. As we integrate a myriad of datasets from the Web, provenance becomes a critical aspect in ascertaining trust and establishing transparency [57]. Our implementation of the presented techniques provides the infrastructure needed for exposing and working with fine-grained provenance in Linked Data oriented environments.

In the next chapter we extend the presented techniques enabling to tailor queries over Linked Data with provenance data. The methods we present allow to specify which pieces of data should be used to answer the query. We introduce the concept of provenance-enabled queries and we present our five provenance-oriented query execution strategies.

Chapter 6

Executing Provenance-Enabled Queries over Linked Data

As we mentioned above the proliferation of heterogeneous Linked Data on the Web poses new challenges to database systems. In particular, because of this heterogeneity, the capacity to store, track, and query provenance data is becoming a pivotal feature of modern triple stores. In this chapter, we tackle the problem of efficiently executing provenance-enabled queries over Linked Data. We propose, implement and empirically evaluate five different query execution strategies for queries over Linked Data that incorporate knowledge of provenance. To develop those strategies we leverage the techniques we presented in Chapter 5. The evaluation is conducted on Linked Data obtained from two different Web crawls (The Billion Triple Challenge, and the Web Data Commons). Our evaluation shows that using an adaptive query materialization execution strategy performs best in our context. Interestingly, we find that because provenance is prevalent within Linked Data and is highly selective, it can be used to improve query processing performance. This is a counterintuitive result as provenance is often associated with additional overhead.

6.1 Provenance-Enabled Queries

"Provenance is information about entities, activities, and people involved in producing a piece of data or thing, which can be used to form assessments about its quality, reliability or trustworthiness" [56]. The W3C PROV Family of Documents[1] defines a model, corresponding serializations and other supporting definitions to enable the interoperable interchange of provenance information in heterogeneous environments such as the Web. In the work, we adopt the view proposed in those specifications. We also adopt the terminology of Cyganiak's original NQuads specification[2], where the *context value* refers to the provenance or source of the triple. We note that context values often are used to refer the named graph to which a triple belongs. Based on this background, we introduce the following terminology used within this thesis:

Definition 6.1. A **Workload Query** is a query producing results a user is interested in. These results are referred to as *workload query results*.

Definition 6.2. A **Provenance Query** is a query that selects a set of data from which some workload query results should originate. Specifically, a *Provenance Query* returns a set of *context values* whose triples will be considered during the execution of a *Workload Query*.

Definition 6.3. A **Provenance-Enabled Query** is a pair consisting of a *Workload Query* and a *Provenance Query*, producing results a user is interested in (as specified by the *Workload Query*) and originating only from data pre-selected by the *Provenance Query*.

As mentioned above, provenance data can be taken into account during query execution through the use of named graphs. Those solutions are however not optimized for provenance, and require rewriting all workload queries with respect to a provenance query. Our approach aims to keep workload queries unchanged and introduce provenance-driven optimization on the database system level.

We assume a strict separation of the workload query on one hand and the provenance query on the other (as illustrated in Figure 6.1)[3]. Provenance and workload results are joined to produce a final result. A consequence of our design is that workload queries

[1] `http://www.w3.org/TR/prov-overview/`

[2] `http://sw.deri.org/2008/07/n-quads/`

[3] We note that including the provenance predicates directly in the query itself is also possible, and that the execution strategies and models we develop in the rest of this thesis would work similarly in that case.

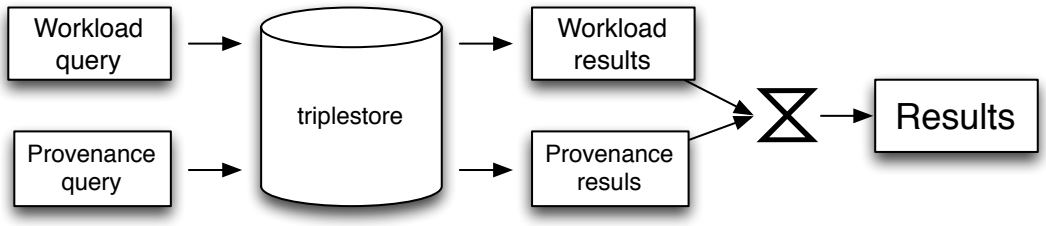


FIGURE 6.1: Executing provenance-enabled queries; both a workload and a provenance query are given as input to a triplestore, which produces results for both queries and then combine them to obtain the final results.

can remain unchanged, while the whole process of applying provenance filtering takes place during query execution. Both provenance and workload queries are to be delivered in the same way, preferably using the SPARQL language or a high-level API that offers similar functionality. The body of the provenance query specifies the set of context values that identify which triples will be used when executing the workload queries.

To further illustrate our approach, we present a few provenance-enabled queries that are simplified versions of use cases found in the literature. In the examples below, *context values* are denoted as *?ctx*.

Provenance-enabled queries can be used in various ways. A common case is to ensure that the data used to produce the answer comes from a set of trusted sources [81]. Given a workload query that retrieves titles of articles about "Obama":

```
SELECT  ?t WHERE  {
  ?a <type> <article> .
  ?a <tag> <Obama> .
  ?a <title> ?t .
}
```

One may want to ensure that the articles retrieved come from sources attributed to the government:

```
SELECT ?ctx WHERE {
  ?ctx  prov:wasAttributedTo <government> .
}
```

As per the W3C definition, provenance is not only about the source of data but is also about the manner in which the data was produced. Thus, one may want to ensure that the articles in question were edited by somebody who is a "SeniorEditor" and that articles where checked by a "Manager". Thus, we could apply the following provenance query while keeping the same "Obama" workload query:

```
SELECT ?ctx WHERE {
  ?ctx prov:wasGeneratedBy <articleProd>.
  <articleProd> prov:wasAssociatedWith ?ed .
  ?ed rdf:type <SeniorEdior>  .
  <articleProd> prov:wasAssociatedWith ?m .
  ?m rdf:type <Manager> .
}
```

A similar example to the one above, albeit for a curated protein database, is described in detail in [30].

Another way to apply provenance-enabled queries is for scenarios in which data is integrated from multiple sources. For example, we may want to aggregate the chemical properties of a drug (e.g., its potency) provided by one database with information on whether it has regulatory approval provided by another:

```
SELECT ?potency ?approval WHERE {
  ?drug <name> ''Sorafenib'' .
  ?drug ?link ?chem.
  ?chem <potency> ?potency .
  ?drug <approvalStatus> ?approval
}
```

Here, we may like to select not only the particular sources that the workload query should be answered over but also the software or approach used in establishing the links between those sources. For instance, we may want to use links generated manually or for a broader scope those generated through the use of any type of string similarity. Such a use-case is described in detail in [13]. Below is an example of how such a provenance query could be written:

```
SELECT ?ctx WHERE {
{
 ?ctx prov:wasGeneratedBy ?linkingActivity.
 ?linkingActivity rdf:type <StringSimilarity>
 }
UNION {
 ?ctx prov:wasDerivedFrom <ChemDB>}
UNION {
 ?ctx prov:wasDerivedFrom <DrugDB>}
}
```

In the following, we discuss approaches to processing these types of queries.

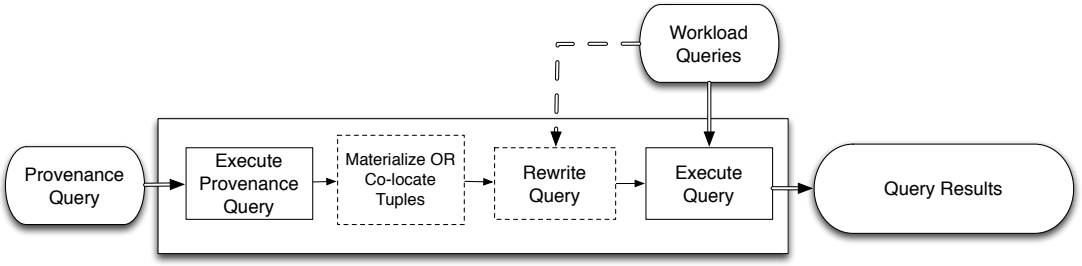


FIGURE 6.2: Generic provenance-enabled query execution pipeline, where both the workload queries and the provenance query get executed in order to produce the final results

6.2 Provenance in Query Processing

There are several possible ways to execute provenance-enabled queries in a triple store. The simplest way is to execute both the RDF query and the provenance query independently, and to join both result sets based on context values. One also has the option of pre-materializing some of the data based on the provenance specification. Another way to execute a provenance-enabled query is through dynamic query rewriting; in that case, the workload query is rewritten using the provenance query (or some of its results) and only then is the query executed. The query execution strategies presented in this section can be implemented in practically any triplestore—provided that it offers some support for storing and handling context values. We discuss our own implementation based on TripleProv in Section 6.3.

6.2.1 Query Execution Pipeline

Figure 6.2 gives a high-level perspective on the query execution process. The provenance and workload queries are provided as input; the query execution process can vary depending on the exact strategy chosen, but typically starts by executing the provenance query and optionally pre-materializing or co-locating data; the workload queries are then optionally rewritten—by taking into account some results of the provenance query—and finally get executed. The process returns as output the workload query results, restricted to those which are following the specification expressed in the provenance query. We give more detail on this execution process below.

Algorithm 4 Generic executing algorithm for provenance-enabled queries

```
1: ctxSet = ExecuteQuery(ProvenanceQuery)
2: materializedTuples = MaterializeTuples (ctxSet) OPTIONAL
3: collocatedTuples = fromProvIdx(ctxSet) OPTIONAL
4: for all workload queries do
5:     ExecuteQuery(queryN, ctxSet)
6: end for
```

6.2.2 Generic Query Execution Algorithm

Algorithm 4 gives a simplified, generic version of the provenance-enabled query execution algorithm. We start by executing the provenance query, which is processed like an ordinary query (*ExecuteQuery*) but *always* returns sets on context values as an output. Subsequently, the system optionally materializes or adaptively co-locates selected tuples[4] containing data related to the provenance query. We then execute workload queries taking into account the context values returned from the previous step. The execution starts as a standard query execution, but optionally includes a dynamic query rewriting step to dynamically prune early in the query plan those tuples that cannot produce valid results given their provenance.

6.2.3 Query Execution Strategies

From the generic algorithm presented above, we now introduce five different strategies for executing provenance-enabled queries and we describe how they could be implemented in different triplestores.

Post-Filtering: this is the baseline strategy, which executes both the workload and the provenance query independently. The provenance and workload queries can be executed in any order (or concurrently) in this case. When both the provenance query and the workload query have been executed, the results from the provenance query (i.e., a set of context values) are used to filter *a posteriori* the results of the workload query based on their provenance (see Algorithm 5). In addition to retrieving the results, the database system needs in this case to track the lineage of all results produced by the workload query. More specifically, the system needs to keep track of the context values of all triples that were involved in

[4]We use *tuples* in a generic way here to remain system-agnostic; tuples can take the form of atomic pieces of data, triples, quads, small sub-graphs, n-ary lists/sets or RDF molecules (cf. Chapters 4 and 5) depending on the database system used

producing a valid result. We discussed how to come up and how to compactly represent such lineage using *provenance polynomials* in Chapter 5. Tracking lineage during query execution is however non-trivial for the systems which, unlike TripleProv, are not provenance-aware. For quadstores, for instance, it involves extensively rewriting the queries, leading to more complex query processing and to an explosion of the number of results retrieved, as we discussed in detail in Section sec:ComparisonTo4Store of Chapter 5.

Query Rewriting: the second strategy we introduce executes the provenance query upfront; then, it uses the set of context values returned by the provenance query to filter out all tuples that do not conform to the provenance results. This can be carried out logically by rewriting the query plans of the workload queries to add provenance constraints (see Algorithm 6, *is present in ctxSet*). This solution is efficient from the provenance query execution side, though it can be suboptimal from the workload query execution side (see Section 6.4). It can be implemented in two ways by the triplestores, either by modifying the query execution process, or by rewriting the workload queries in order to include constraints on the named graphs. We note that the query rewriting is very different than for the case discussed above (for post-filtering, the queries may have to be rewritten to keep track of the lineage of the results; in this case, we know what context values we should filter on during query execution, which makes the rewriting much simpler.)

Full Materialization: this is a two-step strategy where the provenance query is first executed on the entire database (or any relevant subset of it), and then materializes all tuples whose context values satisfy the provenance query. The workload queries are then simply executed on the resulting materialized view, which only contains tuples that are compatible with the provenance specification. This strategy will outperform all other strategies when executing the workload queries, since they are executed *as is* on the relevant subset of the data. However, materializing all potential tuples based on the provenance query can be prohibitively expensive, both in terms of storage space and latency. Implementing this strategy requires either to manually materialize the relevant tuples and modify the workload queries accordingly, or to use a triplestore supporting materialized views.

Pre-Filtering: this strategy takes advantages of a dedicated *provenance index* co-locating, for each context values, the ids (or hashes) of all tuples belonging to this context.

Algorithm 5 Algorithm for the *Post-Filtering* strategy.

```
Require: WorkloadQuery
Require: ProvenanceQuery
1: (ctxSet) = ExecuteQuery(ProvenanceQuery)
2: (results, polynomial) = ExecuteQuery(WorkloadQuery)   ▷ independent execution
   of ProvenanceQuery and WorkloadQuery
3: for all results do
4:     if ( polynomial[result].ContextValues ⊈ ctxSet ) then
5:         remove result
6:     else
7:         keep result
8:     end if
9: end for
```

This index should typically be created upfront when the data is loaded. After the provenance query is executed, the provenance index can be looked up to retrieve the lists of tuple ids that are compatible with the provenance specification. Those lists can then be used to filter out early the intermediate and final results of the workload queries (see Algorithm 7). This strategy requires to create a new index structure in the system (see Section 6.3 for more detail on this), and to modify both the loading and the query execution processes.

Adaptive Partial Materialization: this strategy introduces a tradeoff between the performance of the provenance query and that of the workload queries. The provenance query is executed first. While executing the provenance query, the system also builds a temporary structure (e.g., a hash-table) maintaining the ids of *all* tuples belonging to the context values returned by the provenance query. When executing the workload query, the system can then dynamically (and efficiently) look-up all tuples appearing as intermediate or final results, and can filter them out early in case they do not appear in the temporary structure. Further processing is similar to the *Query Rewriting* strategy, that is, we include individual checks of context values inside the tuples. However those checks, joins, and further query processing operations can then be executed faster on a reduced number of elements. This strategy can achieve performance close to the *Full Materialization* strategy while avoiding to replicate the data, at the expense of creating and maintaining temporary data structures. The implementation of the strategy requires the introduction of an additional data structure at the core of the system, and the adjustment of the query execution process in order to use it.

Algorithm 6 Algorithm for the *Rewriting* strategy.

```
Require: query: workload query
Require: ctxSet: context values; results of provenance query
tuples =q.getPhysicalPlan (FROM materializedTuples for materializes scenario)
for all tuples do
    for all entities do
        if ( entity.ContextValues ⊈ ctxSet) then
            nextEntity
        else
            inspect entity
        end if
    end for
end for
```

Algorithm 7 Algorithm for the *Pre-Filtering* strategy.

```
Require: query: workload query
Require: ctxSet: context values; results of provenance query
tuples =q.getPhysicalPlan
for all tuples do
    for all ctxSet do
        ctxTuples = getTuplesFromProvIdx(ctx)
        if ( tuple ⊈ ctxTuples) then
            nextTuple
        end if
    end for
    for all entities do
        if ( entity.ContextValues ⊈ ctxSet) then
            nextEntity
        else
            inspect entity
        end if
    end for
end for
```

6.3 Storage Model and Indexing

We implemented all the provenance-enabled query execution strategies introduced in Section 6.2 in TripleProv, our own triplestore supporting different storage models to handle provenance data. TripleProv is available as an open-source package[5], the extension implemented for this work is also available online[6]. In the following, we briefly present the implementation of provenance-oriented data structures and indices we used to evaluate the query execution strategies described above. We note that it would be

[5] http://exascale.info/tripleprov

[6] http://exascale.info/provqueries

Algorithm 8 Algorithm for the *Partial Materialization* strategy.

Require: query: workload query
Require: ctxSet: context values; results of provenance query
Require: collocatedTuples: collection of hash values of tuples related to the result of the provenance query (ctxSet)

```
 1: tuples =q.getPhysicalPlan
 2: for all tuples do
 3:     if ( tuple ⊈ collocatedTuples) then
 4:         nextTuple
 5:     end if
 6:     for all entities do
 7:         if ( entity.ContextValues ⊈ ctxSet) then
 8:             nextEntity
 9:         else
10:             inspect entity
11:         end if
12:     end for
13: end for
```

possible to implement our strategies in other systems, using the exact same techniques. The effort to do so, however, is beyond the scope of the thesis as our techniques requires one to revise the loading and/or the query execution processes, to create new indices, to add support for materialized views, and to add support for tracking lineage.

6.3.1 Provenance Storage Model

We use the most basic storage structure we presented in Chapters 4 and 5 in the following: 1-scope RDF molecules (cf. Chapter 4), which co-locate objects related to a given subject and which in are equivalent to property tables (a very popular storage strategy for RDF systems). In that sense, any *tuple* we consider is composed of a subject, and a series of predicate and object related to that subject.

TripleProv supports different models to store provenance information. We compared those models in Chapter 5. For this work, we consider the "SLPO" storage model, which co-locates the context values with the predicate-object pairs, and which offers good overall performance in practice. This avoids the duplication of the same context value, while at the same time co-locating all data about a given subject in one structure. The resulting storage model is illustrated in Figure 6.3. In the rest of this section, we briefly introduce the secondary storage structures we implemented to support the query execution strategies of Section 6.2.

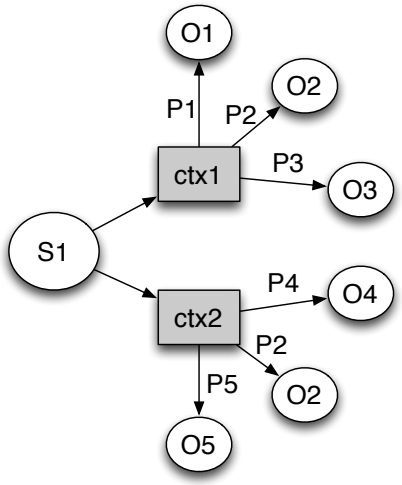


FIGURE 6.3: Our physical storage model for co-locating context values (ctx) with the predicates and objects (PO) inside an RDF molecule.

6.3.2 Provenance Index

Our base system support a number of vertical and horizontal data co-location structures. Here, we propose one more way to co-locate molecules, based on the context values. This gives us the possibility to prune molecules during query execution as explained above. Figure 6.4 illustrates this index, which boils down, in our implementation, to lists of co-located molecule identifiers, indexed by a hash-table whose keys are the context values the triples stored in the molecules belong to. Note, a given molecule can appear multiple times in this index. This index is updated upfront, e.g., at loading time.

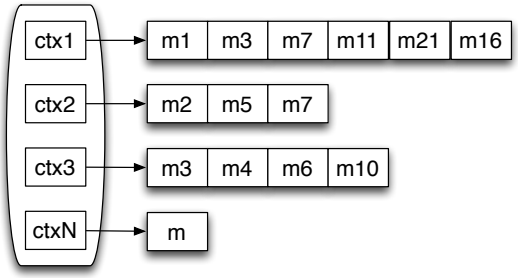


FIGURE 6.4: Provenance-driven indexing schema

6.3.3 Provenance-Driven Full Materialization

To support the provenance-driven materialization scheme introduced in Section 6.2.3, we implemented some basic view creation, update and querying mechanisms in TripleProv.

These mechanisms allow us to project, materialize and utilize as a secondary structure the portions of the molecules that are following the provenance specification (see Figure 6.5 for a simple illustration.)

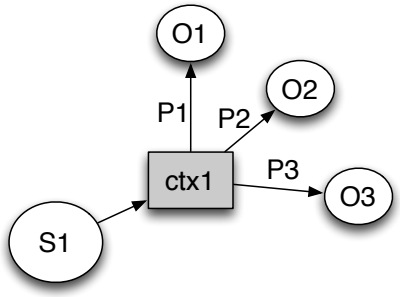


FIGURE 6.5: The molecule after materialization, driven by a provenance query returning only one context value (ctx1).

6.3.4 Adaptive Partial Materialization

Finally, we also implement a new, dedicated structure for the adaptive materialization strategy. In that case, we co-locate all molecule identifiers that are following the provenance specification (i.e., that contain *at least* one context value compatible with the provenance query). We explored several options for this structure and in the end implemented it through a *hashset*, which gives us constant time performance to both insert molecules when executing the provenance query and to query for molecules when executing workload queries.

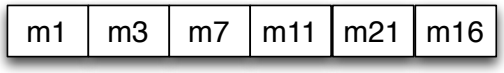


FIGURE 6.6: Set of molecules which contain at least some data related to a provenance query.

6.4 Experiments

To empirically evaluate the query execution strategies discussed above in Section 6.2.3, we implemented them all in TripleProv. The source code of our system is available online[7]. In the following, we experimentally compare a baseline version of our system

[7] http://exascale.info/provqueries

that does not support provenance queries to our five strategies executing provenance-enabled queries. We perform the evaluation on two different datasets and workloads.

Within TripleProv, queries are specified as triple patterns using a high-level declarative API that offers similar functionality to SPARQL.[8] The queries are then encoded into a logical plan (a tree of operators), which is then optimized into a physical query plan as in any standard database system. The system supports all basic SPARQL operations, including "UNION" and "OPTIONAL"; at this point, it does not support "FILTER", however.

6.4.1 Implementations Considered

Our goal is to understand the various tradeoffs of the query execution strategies we proposed in Section 6.2.3 and to assess the performance penalty (or eventual speed-up) caused by provenance queries. We use the following abbreviations to refer to the different implementations we compare:

TripleProv: the vanilla version of (cf. Chapter 5), without provenance queries; this version stores provenance data, tracks the lineage of the results, and generates provenance polynomials , but does not support provenance queries;

Post-Filtering: implements our post-filtering approach; after a workload query execution gets executed, its results are filtered based on the results from the provenance query;

Rewriting: our query execution strategy based on query rewriting; it rewrites the workload query by adding provenance constraints in order to filter out the results;

Full Materialization: creates a materialized view based on the provenance query, and executes the workload queries over that view;

Pre-Filtering: uses a dedicated provenance index to pre-filter tuples during query execution;

Adaptive Materialization: implements a provenance-driven data co-location scheme to co-locate molecule ids that are relevant given the provenance query.

[8] We note that our current system does not parse full SPARQL queries at this stage. Adapting a SPARQL parser is currently in progress.

6.4.2 Experimental Environment

Hardware Platform: All experiments were run on a HP ProLiant DL385 G7 server with an AMD Opteron Processor 6180 SE (24 cores, 2 chips, 12 cores/chip), 64GB of DDR3 RAM, running Ubuntu 12.04.3 LTS (Precise Pangolin). All data were stored on a recent 3 TB Serial ATA disk.

Datasets: We used two different datasets for our experiments: the Billion Triples Challenge (BTC)[9] and the Web Data Commons (WDC) [89][10]. Both datasets are collections of RDF data gathered from the Web. They represent two very different kinds of RDF data. The Billion Triple Challenge dataset was created based on datasets provided by Falcon-S, Sindice, Swoogle, SWSE, and Watson using the MultiCrawler/SWSE framework. The Web Data Commons project extracts all Microformat, Microdata and RDFa data from the Common Crawl Web corpus—the largest and most up-to-data Web corpus that is currently available to the public—and provides the extracted data for download in the form of RDF-quads or CSV-tables for common entity types (e.g., products, organizations, locations, etc.).

Both datasets represent typical collections of data gathered from multiple and heterogeneous online sources, hence applying some provenance query on them seems to precisely address the problem we focus on. We consider around 40 million triples for each dataset (around 10GB). To sample the data, we first pre-selected quadruples satisfying the set of considered workload and provenance queries. Then, we randomly sampled additional data up to 10GB.

For both datasets, we added provenance specific triples (184 for WDC and 360 for BTC) so that the provenance queries we use for all experiments do not modify the result sets of the workload queries, i.e., the workload query results are always the same. We decided to implement this to remove a potential bias when comparing the strategies and the vanilla version of the system (in this way, in all cases all queries have exactly the same input and output). We note that this scenario represents in fact a worst-case scenario for our provenance-enabled approaches, since the provenance query gets executed but does not filter out any result. Therefore, we also performed experiments on the original data (see Section 6.4.3.4), where we use the dataset as is and where the provenance query modifies the output of the workload queries. They show that the performance

[9] `http://km.aifb.kit.edu/projects/btc-2009/`
[10] `http://webdatacommons.org/`

gain for all provenance-enabled strategies is even higher in more realistic scenario and they confirm the what is shown in the main experiments.

Workloads: We consider two different workloads. For BTC, we use eight existing queries originally proposed in [91]. In addition, we added two queries with UNION and OPTIONAL clauses, which we thought were missing in the original set of queries. Based on the queries used for the BTC dataset, we wrote 7 new queries for the WDC dataset, encompassing different kinds of typical query patterns for RDF, including star-queries of different sizes and up to 5 joins, object-object joins, object-subject joins, and triangular joins. We also included two queries with UNION and OPTIONAL clauses. In addition, for each workload we prepared a complex provenance query, which is conceptually similar to those presented in the Section 6.1.

The datasets, query workloads and provenance-queries presented above are all are available on-line[11].

Experimental Methodology: As is typical for benchmarking database systems (e.g., for tpc-x[12] or our own OLTP-Benchmark [42]), we include a warm-up phase before measuring the execution time of the queries in order to measure query execution times in a steady-state mode. We first run all the queries in sequence once to warm-up the system, and then repeat the process ten times (i.e., we run 11 query batches for each variant we benchmark, each containing all the queries we consider in sequence). We report the average execution time of the last 10 runs for each query. In addition, we avoided the artifacts of connecting from the client to the server, of initializing the database from files, and of printing results; we measured instead the query execution times inside the database system only.

6.4.3 Results

In this section, we present the results of the empirical evaluation. We note that our original RDF back-end (the system TripleProv extends) has already been compared to a number of other well-known triple stores. We refer the reader to Chapter 4 for a comparison to non-provenance-enabled triple stores. We have also performed an evaluation of TripleProv and different physical models for storing provenance information in Chapter 5. In this chapter, we focus on a different topic and discuss results for the query

[11] `http://exascale.info/provqueries`
[12] *http://www.tpc.org/*

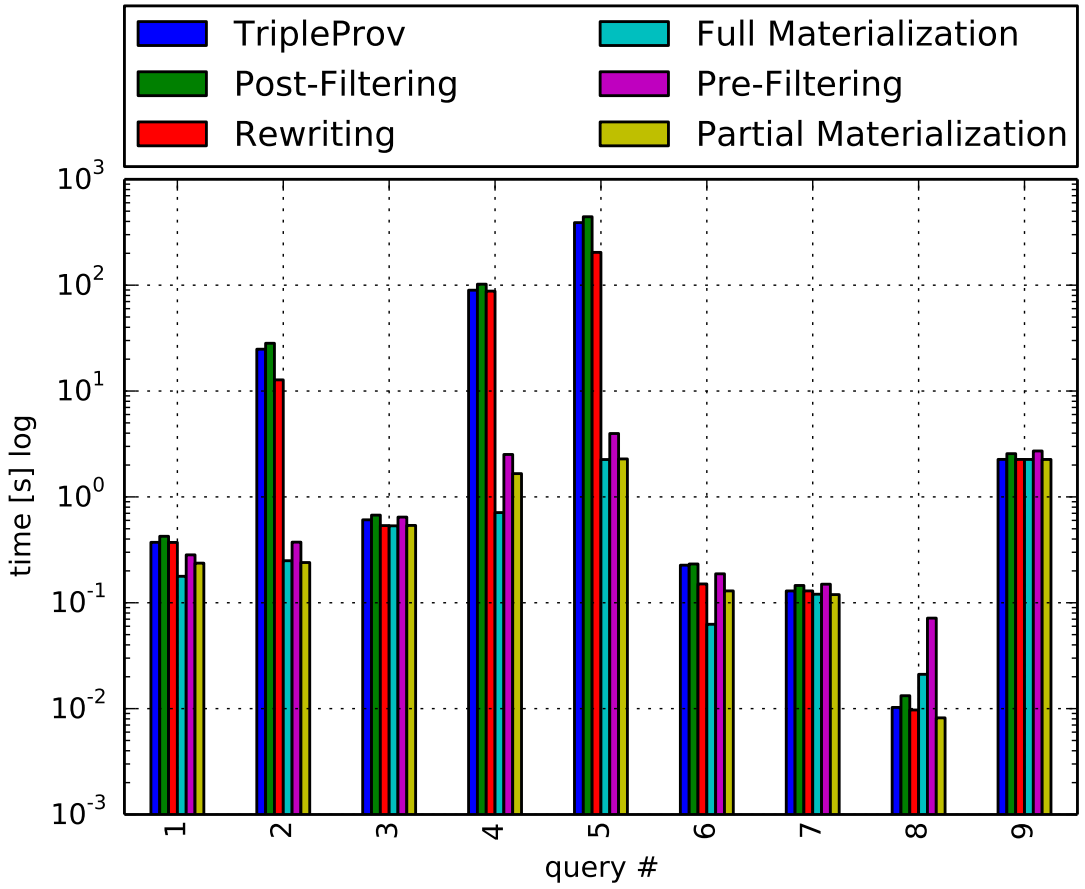


FIGURE 6.7: Query execution times for the BTC dataset (logarithmic scale)

execution strategies for *Provenance-Enabled Queries*. Figure 6.7 reports the query execution times for the BTC dataset, while Figure 6.8 shows similar results for the WDC dataset. We analyze those results below.

6.4.3.1 Datasets Analysis

To better understand the influence of provenance queries on the performance, we start by taking a look at the dataset, provenance distribution, workload, cardinality of intermediate results, number of molecules inspected, and number of basic operations for all query execution strategies. The analysis detailed below was done for the BTC dataset and workload.

First, we analyze the distribution of context values among triples. There are 6'819'826 unique context values in the dataset. Figure 6.9 shows the distribution of the number of triples given the context values (i.e., how many context values refer to how many triples). We observe that there are only a handful of context values that are widespread

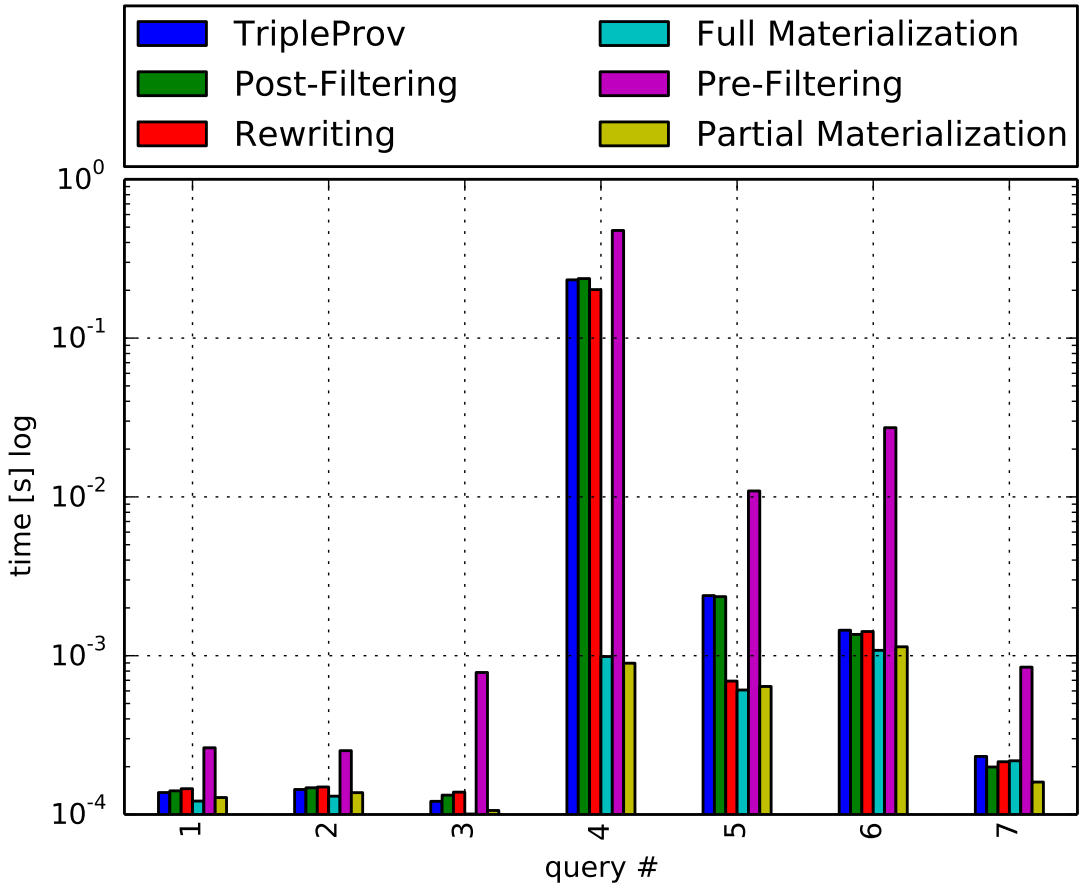


FIGURE 6.8: Query execution times for the WDC dataset (logarithmic scale).

(left-hand side of the figure) and that the vast majority of the context values are highly selective. On average, each context values is related to about 5.8 triples. Co-locating data inside molecules further increases the selectivity of the context values, we have on average 2.3 molecules per context value. We leverage those properties during query execution, as some of our strategies prune molecules early in the query plan based on their context values.

6.4.3.2 Discussion

Our implementations supporting provenance-enabled queries overall outperform in the vanilla TripleProv. This is unsurprising, since as we showed before the selectivity of provenance data in the datasets allows us to avoid unnecessary operations on tuples which do not add to the result.

The *Full Materialization* strategy, where we pre-materialize all relevant subsets of the molecules, makes the query execution on average 44 times faster than the vanilla version

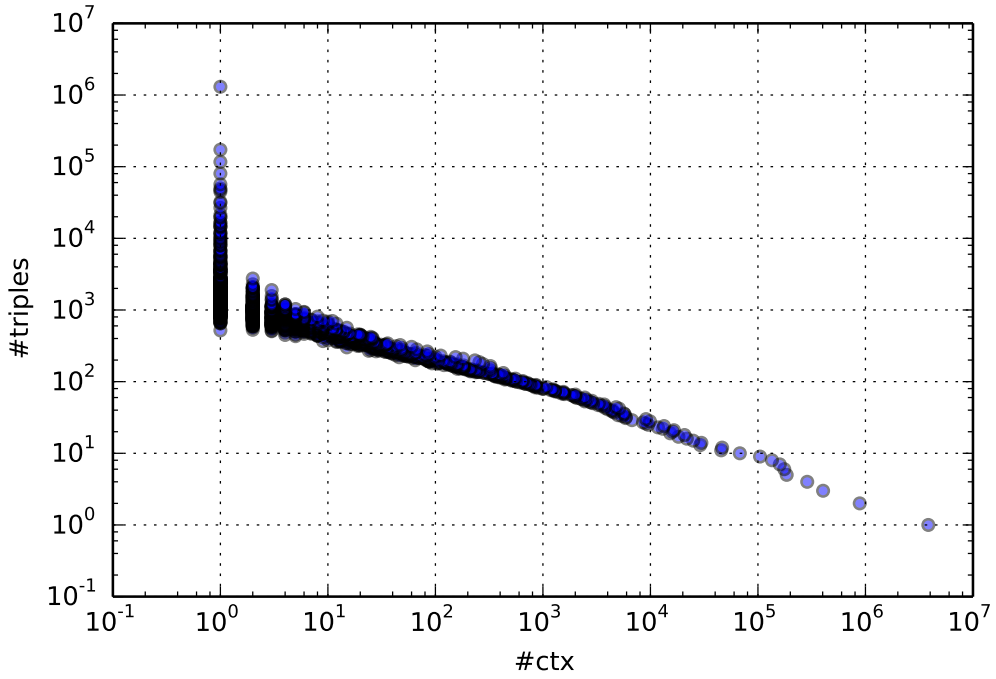


FIGURE 6.9: Distribution of number of triples for number of context values for the BTC dataset.

for the BTC dataset. The speedup ranges from a few percents to more than 200x (queries 2 and 5 of BTC) over the vanilla version. The price for the performance improvement is the time we have to spend to materialize molecules, in our experiments for the BTC it was 95 seconds (the time increases with data size), which can however be amortized by executing enough workload queries (see Section 6.4.4). This strategy consumed about 2% more memory for handling the materialized data.

The *Pre-Filtering* strategy performs on average 23 times faster than the vanilla version for the BTC dataset, but the *Adaptive Partial Materialization* strategy performs on average 35 times faster for the BTC dataset. The advantage over the *Full Materialization* strategy is that for *Adaptive Partial Materialization* time to execute a provenance query and materialize data is 475 times lower and takes only 0.2 second.

The *Query Rewriting* strategy performs significantly slower than the strategies mentioned above for the BTC dataset, since here we have to perform additional checking over provenance data for each query. However, even in this case, for some queries we can observe some performance improvement over the vanilla version of the system; when the provenance query significantly limits the number of tuples inspected

during query execution (see Section 6.2), we can compensate the time taken by additional checks to improve the overall query execution time—see queries 2 and 5 for BTC. Those queries can be executed up to 95% faster than the vanilla version as they requires the highest number of tuple inspections, which can significantly limit other strategies (see Section 6.4.3.3).

We note that the *Post-Filtering* strategy performs in all cases slightly worse than TripleProv (on average 12%) which is expected since there is no early pruning of tuples; queries are executed in the same way as in TripleProv, and in addition the post-processing phase takes place to filter the results set.

For the WDC dataset we have significantly higher cardinality of context values set (10 times more elements), which results in significantly worse performance of *Pre-Filtering* strategy, since this strategy performs a loop over the set of context values. The provenance overhead here is not compensated on workload query execution since they are already executed very fast (below 10^-2 second for most cases) for this dataset. For this scenario the time consumed for *Full Materialization* was 60 seconds and for *Adaptive Partial Materialization* only 0.002 seconds. The *Adaptive Partial Materialization* strategy outperforms other strategies even more clearly on the WDC dataset.

The WDC workload shows even higher predominance of *Adaptive Partial Materialization* strategy over other strategies.

6.4.3.3 Query Performance Analysis

We now examine the reasons behind the performance differences for the different strategies, focusing on the BTC dataset. Thanks to materialization and co-location, we limit the number of molecule look-ups we require to answer the workload queries. The tables below explain the reasons behind the difference in performance. We analyze the number of inspected molecules, the number of molecules after filtering by provenance, the cardinality of intermediate results, and the number of context values used to answer the query.

#r - number of results

#m - total number of molecules used to answer the query, before checking against context values

#mf - total number of molecules after pruning with provenance data

#prov - total number of provenance context values used to answer the query (to generate a polynomial)

#im - intermediate number of molecules used to answer the query, before checking against context values

#imf - intermediate number of molecules after pruning with provenance data

#i - number of intermediate results, used to perform joins

#ec - number of basic operation executed on statements containing only constrains in a query

#er - number of basic operation executed on statements containing projections in a query

The total number of executed basic operations (*#bos*) equals $\#ec + \#er$.

We prepared the provenance query to ensure that the results for all variants are constant, therefore we avoid the bias of having different result sets.

query #	#r	#m	#mf	#prov	#im	#imf	#i	#ec	#er
1	2	4	4	2	0	0	0	84039	470
2	9	203	203	4	0	0	0	3698911	8392
3	13	32	32	7	0	0	0	18537	5580
4	5	1335	1335	5	1	1	1	44941143	4048
5	5	3054	3053	8	3052	3052	3	79050305	37040
6	2	137	133	6	136	132	374	22110	8365
7	2	20	6	5	2	2	18	438	7239
8	237	267	251	287	0	0	0	752	0
9	17	32	32	8	0	0	0	18537	101420

TABLE 6.1: Query execution analysis for TripleProv and the *Post-Filtering* strategy.

Table 6.1 shows the baseline statistics for the vanilla version, TripleProv.

Table 6.2 give statistics for the *Rewriting*. We observe at this level already, we inspect data from on average 50x less molecules, which results on average in a 30% boost in performance. However, executing the provenance query also has its price, which balances this gain in performance for simpler queries (e.g., 7-9).

Table 6.3 gives statistics for our second variant (*Full Materialization*). The total number of molecules initially available is in this case reduced by 22x. Thanks to this, the total number of molecules used to answer the query ('#m') decreases on average 63x; we also

query #	#r	#m	#mf	#prov	#im	#imf	#i	#ec	#er
1	2	4	2	2	0	0	0	5438	470
2	9	203	1	4	0	0	0	832980	6176
3	13	32	32	6	0	0	0	9715	3990
4	5	1335	22	5	1	1	1	1666409	3304
5	5	3054	18	8	3052	17	3	2163812	8008
6	2	137	98	6	136	97	6	13434	5506
7	2	20	2	5	2	1	18	399	7211
8	237	267	237	287	0	0	0	580	0
9	17	32	32	7	0	0	0	9715	52220

TABLE 6.2: Query execution analysis for the *Rewriting* strategy.

query #	#r	#m	#mf	#prov	#im	#imf	#i	#ec	#er
1	2	1	1	2	0	0	0	4660	466
2	9	1	1	4	0	0	0	832426	4144
3	13	31	31	6	0	0	0	2801	2826
4	5	8	8	5	1	1	1	87716	2386
5	5	16	15	8	14	14	3	1865699	4662
6	2	102	98	6	101	97	6	10279	4513
7	2	15	2	5	1	1	14	284	7102
8	237	237	237	287	0	0	0	435	0
9	17	31	31	7	0	0	0	2801	5114

TABLE 6.3: Query execution analysis for the *Full Materialization* strategy.

reduce the number of molecules inspected after pruning with provenance data ('#mf') by 33% compared to the baseline version. This results in a performance improvement of 29x on average. For some queries (3, 7 and 9), the number of used and inspected molecules remains almost unchanged, since the workload query itself is very selective and since there is no room for further pruning molecules before inspecting them. Those queries perform similarly as for the baseline version. For queries 2, 4, and 5, we observe that the reduction in terms of the number of molecules used is 200x, 166x, and 190x, respectively, which significantly impacts the final performance. The price to pay for these impressive speedups is the time spent on the upfront materialization, which was 95 seconds for the dataset considered.

query #	#r	#m	#mf	#prov	#im	#imf	#i	#ec	#er
1	2	2	2	2	0	0	0	5436	470
2	9	1	1	4	0	0	0	832680	6176
3	13	32	32	6	0	0	0	9715	3990
4	5	22	22	5	1	1	1	1663384	3304
5	5	19	18	8	17	17	3	2159510	8008
6	2	102	98	6	101	97	6	13353	5506
7	2	15	2	5	1	1	18	393	7211
8	237	237	237	287	0	0	0	537	0
9	17	32	32	7	0	0	0	9715	52220

TABLE 6.4: Query execution analysis for the *Pre-Filtering* and *Adaptive Partial Materialization* strategies.

Table 6.4 gives statistics for our last two implementations using *Pre-Filtering* and *Adaptive Partial Materialization*. The statistics are similar for both cases (though the structures used to answer the queries and the query execution strategies vary, as explained in Sections 6.2 and 6.3). Here the cardinality of the molecule sets remains unchanged with respect to the vanilla version, and the total number of molecules used to answer the query is identical to molecules after provenance filtering for the naive version, but all molecules we inspect contain data related to the provenance query ('#m' and '#mf' are equal for each query). In fact, we inspect a number of molecules similar to *Full Materialization*, which yields performance of a similar level, on average 14x (*Pre-Filtering*) and 22x (*Adaptive Partial Materialization*) faster than the *Rewriting* strategy. The cost of materialization for *Adaptive Partial Materialization* is much lower than for *Full Materialization*, however, as the strategy only requires 0.2 extra second in order to dynamically co-locate molecules containing data relevant for the provenance query.

6.4.3.4 Representative Scenario

As we mentioned above, our experiments so far aimed at fairly comparing the execution times for different strategies, thus we prepared an experimental scenario where the final output remains unchanged for all implementations (including vanilla TripleProv). In this section, we present a micro-benchmark depicting a representative scenario run on the original data, where the output changes due to constraints imposed on the workload by a provenance query.

Table 6.5 shows the query execution analysis. The number of results is in this case smaller for many queries as results are filtered out based on their context values.

query #	#r	#m	#mf	#prov	#im	#imf	#i	#ec	#er
1	2	1	1	1	0	0	0	2166	222
2	8	1	1	2	0	0	0	604489	5064
3	10	4	4	4	0	0	0	2002	2970
4	5	8	8	3	1	1	1	82609	2768
5	3	5	5	4	4	4	1	1381357	6364
6	1	4	4	4	3	3	1	5601	2523
7	1	15	2	3	1	1	18	297	4079
8	5	5	5	4	0	0	0	5	0
9	10	4	4	4	0	0	0	2002	2970

TABLE 6.5: Query execution analysis for the Pre-Filtering and Partial Materialization strategies for the Representative Scenario.

Figure 6.10 shows performance results for the BTC dataset and workload and a provenance query modifying the output.

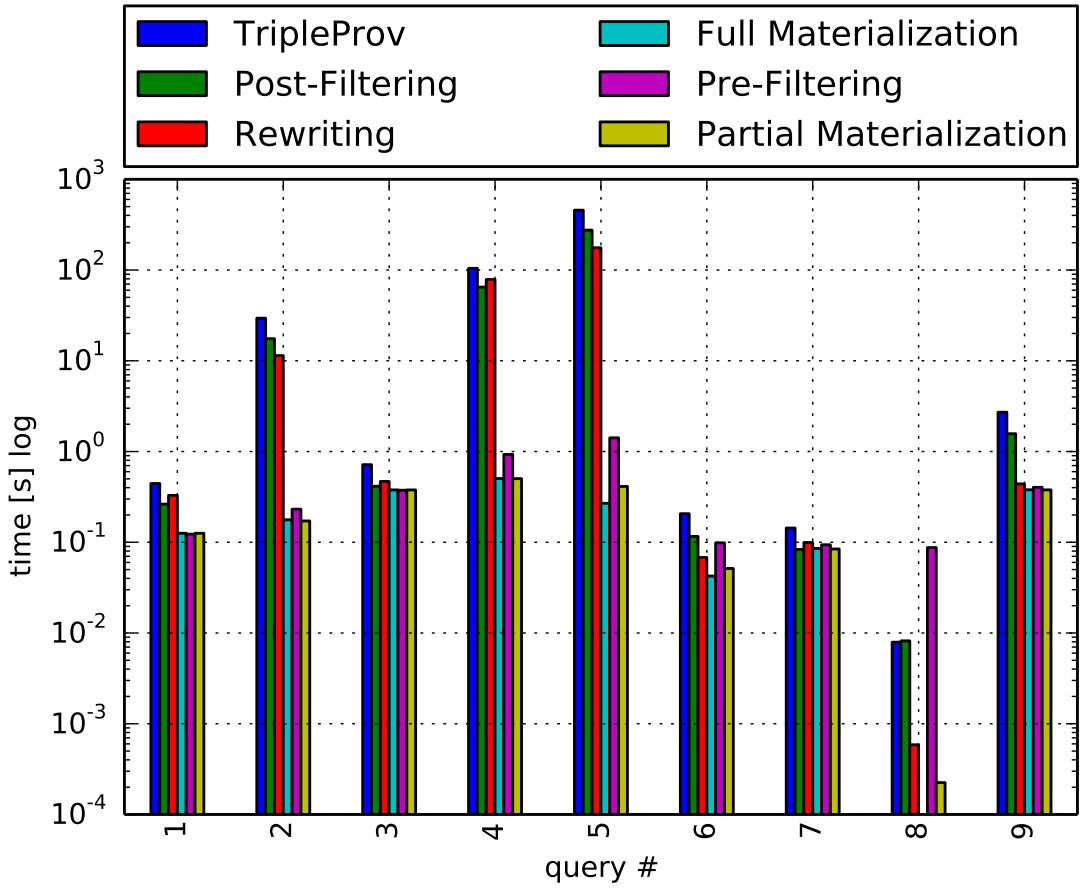


FIGURE 6.10: Query execution times for the BTC dataset (logarithmic scale), Representative Scenario.

As shown on Figure 6.10, the performance gain for all provenance-enabled strategies is higher in more realistic scenario where we did not modify the original data. This speedup is caused by smaller number of *basic operations* ($\#ec + \#er$), which results from a fewer number of intermediate results. For queries for which the results remains the same (2 and 4), the improvement is directly related to the smaller number of *basic operations* performed, caused by a limited number of context values resulting from the provenance query.

6.4.4 End-to-End Workload Optimization

Having several query execution strategies, it is interesting to know which one performs better under what circumstances. Specifically, when it pays off to use a strategy which has a higher cost for executing the provenance query and when it is not beneficial. Ideally, the time consumed on the execution of the provenance query (including

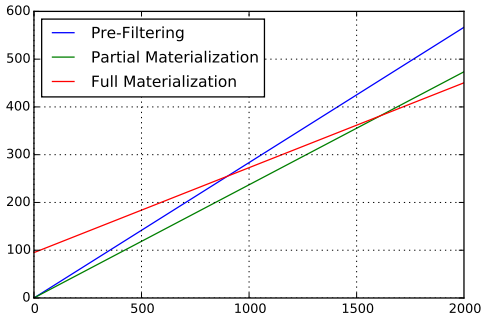


FIGURE 6.11: Cumulative query execution time including time of materialization for 2'000 repetitions of query 1 for BTC.

some potential pre-materialization) should be compensated when executing the workload queries. Let i and j denote two different query execution strategies and P and W denote the time taken to execute the provenance and the workload queries, respectively. If: $P_i + W_i < P_j + W_j$, then strategy i should be chosen since it yields an overall loser cost for running the entire provenance-enabled workload.

As an illustration, Figure 6.11 shows the cumulative query execution time for query 1 of BTC including the time overhead for the provenance query execution and data materialization. We observe that the *Partial Materialization* strategy compensates the overhead of running the provenance query and of materialization after a few repetitions of the query already, compared with the *Pre-Filtering*, which has a lower cost from a provenance query execution perspective, but which executes workload queries slower. For the case of *Full Materialization*, which has a significantly higher materialization overhead, it takes about 900 workload query repetitions to amortize the cost of running the provenance query and pre-materializing data and to beat the *Pre-Filtering* strategy. The *Full Materialization* strategy outperforms the *Partial Materialization* strategy only after more than 1'500 repetitions of the query.

In the end, the optimal strategy depends on the data, on the exact mixture of (provenance and workload) queries, and of their frequencies. Given those three parameters, one can pick the optimal execution strategies using several techniques. If the provenance and the workload queries are known in advance or do not vary much, one can run a sample of the queries using different strategies (similarly to what we did above) and pick the best-performing one. If the queries vary a lot, then one has to resort to an approximate model of query execution in order to pick the best strategy, as it is customary in traditional query optimization. Different models can by used in this context, like the

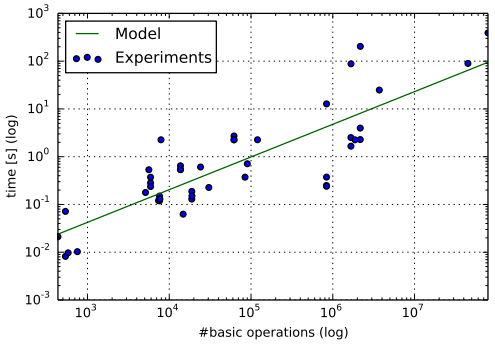


FIGURE 6.12: Query execution time vs. number of basic operations from experimental results from our model, where the model parameters a and b were fit to 0.85 and -9.85, respectively.

very detailed main-memory model we proposed in [58], or the system-agnostic model recently proposed in [66].

As observed above, however, the performance of our various strategies are strongly correlated (with a correlation coefficient of 95%) to the number of basic operations (e.g., molecule look-ups) performed—at least as run in our system. Hence, we propose a simple though effective model in our context based on this observation. We fit a model based on experimental data giving the time to execute a varying number of basic operations. In our setting, the best model turns out to be $e^{(a*\ln bos+b)}$ (the logarithm comes from the cost of preparing a query, such as translating strings into identifiers and building the query plan, which gets amortized with a higher number of subsequent basic operations). Figure 6.12 shows the performance of this model in practice. Using this model and statistics about the predicates in the queries, we can successfully predict the winning strategy, i.e., *Partial Materialization* for the scenarios discussed above.

6.5 Conclusions

In this chapter, we considered the following research question: "What is the most effective query execution strategy for provenance-enabled queries"? The ultimate answer to this question depends on the exact data and queries used, though based on our empirical analysis above, we believe that a, adaptive materialization strategy provides the best trade-off for Linked Data in general. Our performance results show that both on the Web of Data Commons and on the Billion Triple Challenge datasets, this strategy performs best when taking into account the costs of materialization. A key reason for

this result is the selectivity of provenance on the Web of Linked Data. Hence, by leveraging knowledge of provenance, one can execute many types of queries roughly 30x faster than a baseline store.

In order to answer the research question above, this chapter made the following contributions: a characterization of provenance-enabled queries, definitions for five query execution strategies, an implementation of these strategies in TripleProv, as well as a detailed performance and dataset analysis.

Chapter 7

Conclusions

In this thesis, we dealt with the problem of Linked Data management, which can be characterized as high in volume, variety, velocity, and heterogeneity (see Chapter 1). Because of those characteristics, storing and querying graph-oriented Big Linked Data becomes a very complex task; moreover, storing, tracking, and querying provenance data is becoming a pivotal feature of modern triple stores.

To better understand this problem, we started with a detailed analysis of existing approaches to manage Linked Data; in Chapter 2, we presented a series of current approaches in the field, and analyzed storage models, indexing and querying strategies. Subsequently, we evaluated in Chapter 3 a set of Linked Data Management Systems regrouped under the NoSQL umbrella; we showed that in spite of the fact that existing infrastructures can handle simple Linked Data workloads, the performance of more complex queries is still unsatisfying using such systems.

Following those conclusions, we proposed in Chapter 4 our own methods to efficient store and query big amounts of Linked Data (**Research Question Q1**). We presented a novel hybrid storage model considering Linked Data both from a graph and from an analytics perspective. Our molecule-based storage model allows us to efficiently partition data in the cloud to minimize the number of expensive distributed operations. We also proposed a series of efficient query execution strategies leveraging our compact storage model and taking advantage of advanced data co-location strategies, enabling us to execute most of the operations fully in parallel. Our techniques to query Linked Data in the cloud strike an optimal balance between intra-operator parallelism and data co-location by considering recurring, fine-grained physiological Linked Data partitions and

distributed data allocation schemes, leading however to potentially bigger data (redundancy introduced by higher scopes or adaptive molecules) and to more complex insert and update operations. Our methods systematically avoid all complex and distributed operations for query execution.

Subsequently, we tackled in Chapter 5 the problem of storing and tracking provenance during query execution (**Research Question Q2**). We leveraged our molecule-based storage model to store provenance data. We co-located data based on provenance, thus maintaining provenance information in a very compact form. We described two possible storage models for supporting provenance in Linked Data Management Systems. We also presented algorithms to track the provenance of query answering. Our provenance-aware techniques not only present simple tracing of sources for query answers, but also consider fine-grained multilevel provenance.

Finally, we described in Chapter 6 our techniques to tailor queries over Linked Data with provenance data (**Research Question Q3**). We extended our provenance-aware techniques with five query execution strategies to enable specifying constrains of data used to derive the answer. We introduced new provenance-based indexing strategies and materialization algorithms to better handle such workloads.

Our experimental evaluations of the presented techniques showed that our molecule-based storage model represents an optimal way of co-locating Linked Data in a very compact manner, resulting in excellent performance when executing queries in the cloud. Moreover, when extended to store provenance data, it remains efficient from a storage consumption perspective, and allows time-efficient tracing of the queries' lineage. Furthermore, evaluating the presented provenance-aware techniques, we showed that because provenance is prevalent within Linked Data and is highly selective, it can be used to improve query processing performance.

7.1 Future Work

The presented work could be extended in several directions; we elaborate on a series of possible avenues for future work below.

In Chapter 4, we leveraged templates defined by the type of the various resources and identified an interesting problem in automatic templates discovery based on frequent

patterns for untyped elements. Despite the fact that Linked Data is generally schema-free, it tends to exhibit frequent patterns which allow to reconstruct a reliable schema for the considered data. Such emerging schema templates could be afterwards leveraged to cluster the molecules. The clustering would be done only among elements which belong to templates that are related to each other. The main idea is to first cluster templates into sub-groups of elements having strong relations. In the second step, we would rank relations between molecules regrouped under the previously clustered templates sub-groups. We can distinguish four sub-problems to be solved in this context: discovering basic templates, merging templates, co-locating templates, and co-locating molecules.

In addition, we plan to investigate dynamic storage models to enable further optimization in memory consumption and query execution. In this context, we can leverage techniques developed around graph theory, machine learning, and rankings to decide which templates could be extended to higher scopes in order to improve performance. The next possible way to improve performance in this area would be to use data affinity techniques to decide if and which molecules could be replicated among multiple nodes in the cloud to further eliminate expensive distributed operations. Both those avenues can also be explored for dynamically incoming data (streams) and workload-driven optimization techniques. The template structures and replication can be dynamically adjusted using the aforementioned methods with incoming data and workload to optimize data organization.

In our techniques, we used highly efficient data structures to organize resources in memory. Those data structures however tend to be suboptimal in terms of in-memory and disk consumption. This suggest a trade-off between in-memory computations and query execution performance which can pose issues when storing data on low-capacity storage devices. We plan to investigate a low-level compression mechanism to optimize storage consumption (e.g., like those from RDF-3X [90] or HDT [82]). One possible way would to compute deltas between element in molecules to minimize the number of bytes consumed by each value; in Section 4.1.3, we describe our current serialization strategy, which could be further extended to store buckets (subsets) of deltas instead of full IDs. Another possible way would be to consider bit maps for our molecules, similarly to those presented in BitMat [9]. Such compressed elements can also be used to transfer data trough the network in distributed environments, which, as we show in Section 4.5.5, represents a significant part of query execution time.

In terms of provenance handling, we plan to extend our current implementation to output PROV, which would open the door to queries over the provenance of the query

results and the data itself – merging both internal and external provenance. Such an approach would facilitate trust computation over provenance that takes into account the history of the original data as well as how it was processed within the database. Also, we plan to add some support for comparing queries (e.g., *diffs*) based on their results and on provenance information.

Another interesting question worth investigating is whether provenance can be leveraged to partition Linked Data in the cloud e.g., if molecules sharing the same provenance should be co-located on one node to eliminate distributed operations. Also, tracing provenance in a distributed environment can introduce interesting challenges e.g., whether introducing information about the nodes providing results can help to evaluate the validity of the results or to discover bottlenecks in the underlying physical infrastructure.

All presented techniques can also be combined with new developments such as the Internet of Everything, the Physical Web, or Machine to Machine communication technologies. Such applications will definitely impose interesting constraints and generate novel and challenging workloads for future Linked Data Management Systems.

List of Figures

List of Tables

Bibliography

[1] Jans Aasman. Allegro graph: RDF triple database. Technical report, Technical report. Franz Incorporated, 2006. ur l: http://www. franz. com/agraph/allegrograph/(visited on 10/14/2013)(cited on pp. 52, 54), 2006.

[2] Daniel J. Abadi, Adam Marcus, Samuel Madden, and Katherine J. Hollenbach. Scalable Semantic Web Data Management Using Vertical Partitioning. In *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, pages 411–422. ACM, 2007.

[3] Divyakant Agrawal, Sudipto Das, and Amr El Abbadi. Big data and cloud computing: new wine or just new bottles? *Proceedings of the VLDB Endowment*, 3(1-2):1647–1648, 2010.

[4] Divyakant Agrawal, Sudipto Das, and Amr El Abbadi. Big data and cloud computing: current state and future opportunities. In *Proceedings of the 14th International Conference on Extending Database Technology*, pages 530–533. ACM, 2011.

[5] Rakesh Agrawal, Amit Somani, and Yirong Xu. Storage and Querying of E-Commerce Data. In *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*, pages 149–158. Morgan Kaufmann, 2001.

[6] Sofia Alexaki, Vassilis Christophides, Gregory Karvounarakis, and Dimitris Plexousakis. On Storing Voluminous RDF Descriptions: The Case of Web Portal Catalogs. In *WebDB*, pages 43–48, 2001.

[7] Bahareh Arab, Dieter Gawlick, Venkatesh Radhakrishnan, Hao Guo, and Boris Glavic. A generic provenance middleware for queries, updates, and transactions. In *6th USENIX Workshop on the Theory and Practice of Provenance (TaPP 2014)*, Cologne, June 2014. USENIX Association.

[8] Nikolas Askitis and Ranjan Sinha. Hat-trie: a cache-conscious trie-based data structure for strings. In *Proceedings of the thirtieth Australasian conference on Computer science-Volume 62*, pages 97–105. Australian Computer Society, Inc., 2007.

[9] Medha Atre, Vineet Chaoji, Jesse Weaver, and Gregory Williamss. Bitmat: An in-core rdf graph store for join query processing. In *Rensselaer Polytechnic Institute Technical Report*, 2009.

[10] Medha Atre, Vineet Chaoji, Mohammed J. Zaki, and James A. Hendler. Matrix "Bit" loaded: a scalable lightweight join query processor for RDF data. In *Proceedings of the 19th International Conference on World Wide Web, WWW 2010, Raleigh, North Carolina, USA, April 26-30, 2010*, pages 41–50. ACM, 2010.

[11] Medha Atre and James A Hendler. BitMat: a main memory bit-matrix of RDF triples. In *The 5th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2009)*, page 33. Citeseer, 2009.

[12] Sören Auer, Jan Demter, Michael Martin, and Jens Lehmann. Lodstats–an extensible framework for high-performance dataset analytics. In *Knowledge Engineering and Knowledge Management*, pages 353–362. Springer, 2012.

[13] Colin R. Batchelor, Christian Y. A. Brenninkmeijer, Christine Chichester, Mark Davies, Daniela Digles, Ian Dunlop, Chris T. A. Evelo, Anna Gaulton, Carole A. Goble, Alasdair J. G. Gray, Paul T. Groth, Lee Harland, Karen Karapetyan, Antonis Loizou, John P. Overington, Steve Pettifer, Jon Steele, Robert Stevens, Valery Tkachenko, Andra Waagmeester, Antony J. Williams, and Egon L. Willighagen. Scientific lenses to support multiple views over linked chemistry data. In *ISWC 2014 - 13th International Semantic Web Conference, Riva del Garda, Italy, October 19-23, 2014. Proceedings, Part I*, pages 98–113, October 2014.

[14] Christian Becker. RDF store benchmarks with DBpedia, 2008. http://wifo5-03.informatik.uni-mannheim.de/benchmarks-200801/.

[15] Tim Berners-Lee. Linked data-design issues, 2006.

[16] Tim Berners-Lee, James Hendler, Ora Lassila, et al. The semantic web. *Scientific american*, 284(5):28–37, 2001.

[17] Mark A Beyer and Douglas Laney. The importance of 'big data': a definition. *Stamford, CT: Gartner*, 2012.

[18] Olivier Biton, Sarah Cohen-Boulakia, and Susan B. Davidson. Zoom*userviews: Querying relevant provenance in workflow systems. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 1366–1369. VLDB Endowment, 2007.

[19] Chris Bizer, Anja Jentzsch, and Richard Cyganiak. State of the lod cloud. *Version 0.3 (September 2011)*, 1803, 2011.

[20] Christian Bizer, Tom Heath, and Tim Berners-Lee. Linked data-the story so far, 2009.

[21] Christian Bizer and Andreas Schultz. The berlin sparql benchmark. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 5(2):1–24, 2009.

[22] Christian Bizer and Andreas Schultz. The Berlin SPARQL Benchmark. *Int. J. Semantic Web Inf. Syst.*, 5(2):1–24, 2009.

[23] Matthias Bröcheler, Andrea Pugliese, and V. S. Subrahmanian. DOGMA: A Disk-Oriented Graph Matching Algorithm for RDF Databases. In *The Semantic Web - ISWC 2009, 8th International Semantic Web Conference, ISWC 2009, Chantilly, VA, USA, October 25-29, 2009. Proceedings*, pages 97–113. Springer, 2009.

[24] Matthias Bröcheler, Andrea Pugliese, and VS Subrahmanian. Dogma: A disk-oriented graph matching algorithm for rdf databases. In *The Semantic Web-ISWC 2009*, pages 97–113. Springer, 2009.

[25] Jeen Broekstra, Arjohn Kampman, and Frank van Harmelen. Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In *The Semantic Web - ISWC 2002, First International Semantic Web Conference, Sardinia, Italy, June 9-12, 2002, Proceedings*, pages 54–68. Springer, 2002.

[26] Jeremy J Carroll, Christian Bizer, Pat Hayes, and Patrick Stickler. Named graphs, provenance and trust. In *Proceedings of the 14th international conference on World Wide Web*, pages 613–622. ACM, 2005.

[27] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: a distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation -*

Volume 7, OSDI '06, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.

[28] Artem Chebotko, Shiyong Lu, Xubo Fei, and Farshad Fotouhi. Rdfprov: A relational rdf store for querying and managing scientific workflow provenance. *Data Knowl. Eng.*, 69(8):836–865, August 2010.

[29] James Cheney, Laura Chiticariu, and Wang-Chiew Tan. *Provenance in databases: Why, how, and where*. Now Publishers Inc, 2009.

[30] Christine Chichester, Pascale Gaudet, Oliver Karch, Paul Groth, Lydie Lane, Amos Bairoch, Barend Mons, and Antonis Loizou. Querying nextprot nanopublications and their value for insights on sequence variants and tissue expression. *Web Semantics: Science, Services and Agents on the World Wide Web*, pages –, 2014.

[31] Eugene Inseok Chong, Souripriya Das, George Eadon, and Jagannathan Srinivasan. An Efficient SQL-based RDF Querying Scheme. In *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*, pages 1216–1227. ACM, 2005.

[32] World Wide Web Consortium. OWL 2 Web Ontology Language, 2012.

[33] World Wide Web Consortium. SPARQL 1.1 Overview, 2013.

[34] World Wide Web Consortium. RDF 1.1 Concepts and Abstract Syntax, 2014.

[35] World Wide Web Consortium. RDF 1.1: On Semantics of RDF Datasets, 2014.

[36] World Wide Web Consortium. RDF 1.1 Primer, 2014.

[37] World Wide Web Consortium. RDF Schema 1.1, 2014.

[38] P. Cudré-Mauroux, K.T. Lim, R. Simakov, E. Soroush, P. Velikhov, D. L. Wang, M. Balazinska, J. Becla, D. DeWitt, B. Heath, D. Maier, S. Madden, J. M. Patel, M. Stonebraker, and S. Zdonik. A Demonstration of SciDB: A Science-Oriented DBMS. *Proceedings of the VLDB Endowment (PVLDB)*, 2(2):1534–1537, 2009.

[39] P. Cudré-Mauroux, E. Wu, and S. Madden. The Case for RodentStore, an Adaptive, Declarative Storage System. In *Biennial Conference on Innovative Data Systems Research (CIDR)*, 2009.

[40] Carlos Viegas Damásio, Anastasia Analyti, and Grigoris Antoniou. Provenance for sparql queries. In *Proceedings of the 11th international conference on The Semantic Web - Volume Part I*, ISWC'12, pages 625–640, Berlin, Heidelberg, 2012. Springer-Verlag.

[41] Gianluca Demartini, Iliya Enchev, Marcin Wylot, Joel Gapany, and Philippe Cudre-Mauroux. Bowlognabench - benchmarking rdf analytics. In Karl Aberer, Ernesto Damiani, and Tharam Dillon, editors, *Data-Driven Process Discovery and Analysis*, volume 116 of *Lecture Notes in Business Information Processing*, pages 82–102. Springer Berlin Heidelberg, 2012.

[42] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *PVLDB*, 7(4):277–288, 2013.

[43] Li Ding, Yun Peng, Paulo Pinheiro da Silva, and Deborah L. McGuinness. Tracking RDF Graph Provenance using RDF Molecules. In *International Semantic Web Conference*, 2005.

[44] Orri Erling and Ivan Mikhailov. Towards web scale rdf. *Proc. SSWS*, 2008.

[45] Orri Erling and Ivan Mikhailov. RDF Support in the Virtuoso DBMS. In *Networked Knowledge-Networked Media*, pages 7–24. Springer, 2009.

[46] George H. L. Fletcher and Peter W. Beck. Scalable indexing of RDF graphs for efficient join processing. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management, CIKM 2009, Hong Kong, China, November 2-6, 2009*, pages 1513–1516. ACM, 2009.

[47] Giorgos Flouris, Irini Fundulaki, Panagiotis Pediaditis, Yannis Theoharis, and Vassilis Christophides. Coloring rdf triples to capture provenance. In *Proceedings of the 8th International Semantic Web Conference*, ISWC '09, pages 196–212, Berlin, Heidelberg, 2009. Springer-Verlag.

[48] Sever Fundatureanu. A scalable rdf store based on hbase. Master's thesis, Vrije University, 2012. http://archive.org/details/ScalableRDFStoreOverHBase.

[49] Luiz M. Gadelha, Jr., Michael Wilde, Marta Mattoso, and Ian Foster. Mtcprov: A practical provenance query framework for many-task scientific computing. *Distrib. Parallel Databases*, 30(5-6):351–370, October 2012.

[50] Hector Garcia-Molina. *Database systems: the complete book*. Pearson Education India, 2008.

[51] Floris Geerts, Grigoris Karvounarakis, Vassilis Christophides, and Irini Fundulaki. Algebraic structures for capturing the provenance of sparql queries. In *Proceedings of the 16th International Conference on Database Theory*, ICDT '13, pages 153–164, New York, NY, USA, 2013. ACM.

[52] Boris Glavic and Gustavo Alonso. The perm provenance management system in action. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 1055–1058, New York, NY, USA, 2009. ACM.

[53] D Graham-Rowe, D Goldston, C Doctorow, M Waldrop, C Lynch, F Frankel, R Reid, S Nelson, D Howe, SY Rhee, et al. Big data: science in the petabyte era. *Nature*, 455(7209):8–9, 2008.

[54] Todd J Green, Grigoris Karvounarakis, and Val Tannen. Provenance semirings. In *Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 31–40. ACM, 2007.

[55] Paul Groth, Yolanda Gil, James Cheney, and Simon Miles. Requirements for provenance on the web. *International Journal of Digital Curation*, 7(1), 2012.

[56] Paul Groth and Luc Moreau (eds.). PROV-Overview. An Overview of the PROV Family of Documents. W3C Working Group Note NOTE-prov-overview-20130430, World Wide Web Consortium, April 2013.

[57] Paul T. Groth. Transparency and reliability in the data supply chain. *IEEE Internet Computing*, 17(2):69–71, 2013.

[58] Martin Grund, Jens Krüger, Hasso Plattner, Alexander Zeier, Philippe Cudré-Mauroux, and Samuel Madden. Hyrise - a main memory hybrid storage engine. *PVLDB*, 4(2):105–116, 2010.

[59] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. Lubm: A benchmark for owl knowledge base systems. *Web Semant.*, 3:158–182, October 2005.

[60] Harry Halpin and James Cheney. Dynamic provenance for sparql updates. In Peter Mika, Tania Tudorache, Abraham Bernstein, Chris Welty, Craig Knoblock, Denny Vrandečić, Paul Groth, Natasha Noy, Krzysztof Janowicz, and Carole

Goble, editors, *The Semantic Web – ISWC 2014*, volume 8796 of *Lecture Notes in Computer Science*, pages 425–440. Springer International Publishing, 2014.

[61] Stephen Harris and Nicholas Gibbins. 3store: Efficient Bulk RDF Storage. In *PSSS1 - Practical and Scalable Semantic Systems, Proceedings of the First International Workshop on Practical and Scalable Semantic Systems, Sanibel Island, Florida, USA, October 20, 2003*. CEUR-WS.org, 2003.

[62] Steve Harris, Nick Lamb, and Nigel Shadbolt. 4store: The design and implementation of a clustered rdf store. In *5th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2009)*, pages 94–109, 2009.

[63] Andreas Harth and Stefan Decker. Optimized Index Structures for Querying RDF from the Web. In *IEEE LA-WEB*, pages 71–80, 2005.

[64] O. Hartig. Provenance information in the web of data. In *Proceedings of the 2nd Workshop on Linked Data on the Web (LDOW2009)*, 2009.

[65] Olaf Hartig. Querying trust in rdf data with tsparql. In *Proceedings of the 6th European Semantic Web Conference on The Semantic Web: Research and Applications*, ESWC 2009 Heraklion, pages 5–20, Berlin, Heidelberg, 2009. Springer-Verlag.

[66] Rakebul Hasan and Fabien Gandon. Predicting SPARQL query performance. In *The Semantic Web: ESWC 2014 Satellite Events - ESWC 2014 Satellite Events, Anissaras, Crete, Greece, May 25-29, 2014, Revised Selected Papers*, pages 222–225, 2014.

[67] Patrick Hayes and Brian McBride. Rdf semantics. W3C Recommendation, February 2004.

[68] Tom Heath and Christian Bizer. Linked data: Evolving the web into a global data space. *Synthesis lectures on the semantic web: theory and technology*, 1(1):1–136, 2011.

[69] Joseph M Hellerstein and Michael Stonebraker. *Readings in database systems*. MIT Press, 2005.

[70] Johannes Hoffart, Fabian M. Suchanek, Klaus Berberich, and Gerhard Weikum. Yago2: A spatially and temporally enhanced knowledge base from wikipedia. *Artificial Intelligence*, 194(0):28–61, 2013. Artificial Intelligence, Wikipedia and Semi-Structured Resources.

[71] Jiewen Huang, Daniel J. Abadi, and Kun Ren. Scalable SPARQL Querying of Large RDF Graphs. *PVLDB*, 4(11):1123–1134, 2011.

[72] Trung Dong Huynh, Paul Groth, and Stephan Zednik (eds.). PROV Implementation Report. W3C Working Group Note NOTE-prov-implementations-20130430, World Wide Web Consortium, April 2013.

[73] Maciej Janik and Krys Kochut. BRAHMS: A WorkBench RDF Store and High Performance Memory System for Semantic Association Discovery. In *The Semantic Web - ISWC 2005, 4th International Semantic Web Conference, ISWC 2005, Galway, Ireland, November 6-10, 2005, Proceedings*, pages 431–445. Springer, 2005.

[74] Grigoris Karvounarakis, Zachary G Ives, and Val Tannen. Querying data provenance. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 951–962. ACM, 2010.

[75] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 20(1):359–392, 1998.

[76] HyeongSik Kim, Padmashree Ravindra, and Kemafor Anyanwu. From sparql to mapreduce: The journey using a nested triplegroup algebra. *PVLDB*, 4(12):1426–1429, 2011.

[77] Atanas Kiryakov, Damyan Ognyanov, and Dimitar Manov. OWLIM–a pragmatic semantic repository for OWL. In *Web Information Systems Engineering–WISE 2005 Workshops*, pages 182–192. Springer, 2005.

[78] Günter Ladwig and Andreas Harth. CumulusRDF: Linked data management on nested key-value stores. In *The 7th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS 2011)*, page 30, 2011.

[79] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[80] Doug Laney. 3d data management: Controlling data volume, velocity and variety. *META Group Research Note*, 6:70, 2001.

[81] luc Moreau and Groth Paul. *Provenance: An Introduction to PROV*. Morgan and Claypool, September 2013.

[82] Miguel A. Martínez-Prieto, Mario Arias, and Javier D. Fernandez. Exchange and Consumption of Huge RDF Data. In *The Semantic Web: Research and Applications*, pages 437–452. Springer, 2012.

[83] Ruslan Mavlyutov, Marcin Wylot, and Philippe Cudre-Mauroux. A comparison of data structures to manage uris on the web of data. In *The Semantic Web: Trends and Challenges*. Springer, 2015.

[84] Brian McBride. Jena: A semantic web toolkit. *IEEE Internet computing*, 6(6):55–59, 2002.

[85] Simon Miles. Electronically querying for the provenance of entities. In Luc Moreau and Ian Foster, editors, *Provenance and Annotation of Data*, volume 4145 of *Lecture Notes in Computer Science*, pages 184–192. Springer Berlin Heidelberg, 2006.

[86] Luc Moreau. The foundations for provenance on the web. *Foundations and Trends in Web Science*, 2(2–3):99–241, November 2010.

[87] Luc Moreau, Ben Clifford, Juliana Freire, Joe Futrelle, Yolanda Gil, Paul Groth, Natalia Kwasnikowska, Simon Miles, Paolo Missier, Jim Myers, Beth Plale, Yogesh Simmhan, Eric Stephan, and Jan Van den Bussche. The open provenance model core specification (v1.1). *Future Generation Computer Systems*, 27(6):743–756, June 2011.

[88] Mohamed Morsey, Jens Lehmann, Sören Auer, and Axel-Cyrille Ngonga Ngomo. Dbpedia sparql benchmark–performance assessment with real queries on real data. In *The Semantic Web–ISWC 2011*, pages 454–469. Springer, 2011.

[89] Hannes Mühleisen and Christian Bizer. Web data commons - extracting structured data from two large web corpora. In Christian Bizer, Tom Heath, Tim Berners-Lee, and Michael Hausenblas, editors, *LDOW*, volume 937 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2012.

[90] Thomas Neumann and Gerhard Weikum. RDF-3X: a RISC-style engine for RDF. *Proceedings of the VLDB Endowment (PVLDB)*, 1(1):647–659, 2008.

[91] Thomas Neumann and Gerhard Weikum. Scalable join processing on very large rdf graphs. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 627–640. ACM, 2009.

[92] Thomas Neumann and Gerhard Weikum. The RDF-3X engine for scalable management of RDF data. *VLDB J.*, 19(1):91–113, 2010.

[93] Own O'Malley. Terabyte sort on apache hadoop, 2008.

[94] Alisdair Owens, Andy Seaborne, Nick Gibbins, et al. Clustered tdb: a clustered triple store for jena, 2008.

[95] Panagiotis Pediaditis, Giorgos Flouris, Irini Fundulaki, and Vassilis Christophides. On explicit provenance management in rdf/s graphs. In *Workshop on the Theory and Practice of Provenance*, 2009.

[96] Eric Prud'Hommeaux, Andy Seaborne, et al. Sparql query language for rdf. *W3C recommendation*, 15, 2008.

[97] D. Wood R. Cyganiak and M. Lanthaler (Ed.). RDF 1.1 Concepts and Abstract Syntax. W3C Recommendation, February 2014. http://www.w3.org/TR/rdf11-concepts/.

[98] Ravishankar Ramamurthy, David J. DeWitt, and Qi Su. A case for fractured mirrors. In *Proceedings of the 28th international conference on Very Large Data Bases*, VLDB '02, pages 430–441. VLDB Endowment, 2002.

[99] Padmashree Ravindra, Vikas V Deshpande, and Kemafor Anyanwu. Towards scalable rdf graph analytics on mapreduce. In *Proceedings of the 2010 Workshop on Massive Data Analytics on the Cloud*, page 5. ACM, 2010.

[100] Padmashree Ravindra, HyeongSik Kim, and Kemafor Anyanwu. An Intermediate Algebra for Optimizing RDF Graph Pattern Matching on MapReduce. In *The Semanic Web: Research and Applications - 8th Extended Semantic Web Conference, ESWC 2011, Heraklion, Crete, Greece, May 29 - June 2, 2011, Proceedings, Part II*, pages 46–61. Springer, 2011.

[101] Kurt Rohloff and Richard E Schantz. Clause-iteration with mapreduce to scalably query datagraphs in the shard graph-store. In *Proceedings of the fourth international workshop on Data-intensive distributed computing*, pages 35–44. ACM, 2011.

[102] Satya Sahoo, Paul Groth, Olaf Hartig, Simon Miles, Sam Coppens, James Myers, Yolanda Gil, Luc Moreau, Jun Zhao, Michael Panzer, et al. Provenance vocabulary mappings. Technical report, W3C, 2010.

[103] Max Schmachtenberg, Christian Bizer, and Heiko Paulheim. Adoption of the linked data best practices in different topical domains. In *The Semantic Web–ISWC 2014*, pages 245–260. Springer, 2014.

[104] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. R. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-Store: A Column Oriented DBMS. In *International Conference on Very Large Data Bases (VLDB)*, 2005.

[105] Yannis Theoharis, Irini Fundulaki, Grigoris Karvounarakis, and Vassilis Christophides. On provenance of queries on semantic web data. *IEEE Internet Computing*, 15(1):31–39, January 2011.

[106] Petros Tsialiamanis, Lefteris Sidirourgos, Irini Fundulaki, Vassilis Christophides, and Peter Boncz. Heuristics-based query optimisation for sparql. In *Proceedings of the 15th International Conference on Extending Database Technology*, 2012.

[107] Octavian Udrea, Diego Reforgiato Recupero, and VS Subrahmanian. Annotated RDF. *ACM Transactions on Computational Logic (TOCL)*, 11(2):10, 2010.

[108] Jacopo Urbani, Spyros Kotoulas, Jason Maassen, Niels Drost, Frank Seinstra, Frank Van Harmelen, and Henri Bal. H.: Webpie: A web-scale parallel inference engine. In *In: Third IEEE International Scalable Computing Challenge (SCALE2010), held in conjunction with the 10th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2010.

[109] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. Hexastore: sextuple indexing for semantic web data management. *Proceeding of the VLDB Endowment (PVLDB)*, 1(1):1008–1019, 2008.

[110] Kevin Wilkinson, Craig Sayers, Harumi A. Kuno, and Dave Reynolds. Efficient rdf storage and retrieval in jena2. In *SWDB'03*, pages 131–150, 2003.

[111] Kevin Wilkinson and Kevin Wilkinson. Jena property table implementation. In *International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS)*, 2006.

[112] Jun Zhao, Christian Bizer, Yolanda Gil, Paolo Missier, and Satya Sahoo. Provenance requirements for the next version of rdf. In *W3C Workshop RDF Next Steps*, 2010.

[113] Antoine Zimmermann, Nuno Lopes, Axel Polleres, and Umberto Straccia. A general framework for representing, reasoning and querying with annotated semantic web data. *Web Semant.*, 11:72–95, March 2012.

[114] Lei Zou, Jinghui Mo, Lei Chen, M. Tamer Oezsu, and Dongyan Zhao. gstore: Answering sparql queries via subgraph matching. *PVLDB*, 4(8), 2011.

Marcin Wylot *September 18, 1982* marcin.wylot@unifr.ch

Experience

University of Fribourg — FRIBOURG, SWITZERLAND
PhD Student, Researcher — *March 2011 – August 2015*

Vrije Universiteit Amsterdam — AMSTERDAM, THE NETHERLANDS
Visiting Researcher — *June 2013 – August 2013*

Asseco Poland — WARSAW, POLAND
Information Technology Specialist — *December 2007 – January 2011*

Warsaw School of Information Technology — WARSAW, POLAND
System Administrator — *January 2006 – June 2007*

Prokom Software — WARSAW, POLAND
Database Specialist — *September 2005 – November 2005*

MasterFilm LTD. — WARSAW, POLAND
System Administrator and Software Engineer — *September 2004 – May 2005*

Education

University of Fribourg — FRIBOURG, SWITZERLAND
Doctor of Philosophy (Ph.D.) in Computer Science — *March 2011 – June 2015*

Claude Bernard University (Lyon I) — LYON, FRANCE
Research Internship — *February 2009 – June 2009*

University of Lodz — LODZ, POLAND
Master's Degree in Computer Science — *October 2007 – April 2010*

Warsaw School of Information Technology — WARSAW, POLAND
Bachelor's Degree in Computer Science — *October 2002 – October 2006*