Scalable algorithms for cloud-based Semantic Web data management

Stamatis Zampetakis

UNIVERSITÉ PARIS-SUD

ÉCOLE DOCTORALE INFORMATIQUE DE PARIS-SUD (ED 427)

INRIA Saclay & Laboratoire de Recherche en Informatique (LRI)

***DISCIPLINE** INFORMATIQUE*

THÈSE DE DOCTORAT

Soutenue le 21 Septembre 2015 par

Stamatis ZAMPETAKIS

Scalable algorithms for cloud-based Semantic Web data management

Thèse dirigée par :

Ioana Manolescu	INRIA & Université Paris-Sud
François Goasdoué	Université de Rennes 1 & INRIA

Rapporteurs :

Bernd Amann	Université Pierre & Marie Curie
Tamer Özsu	Université de Waterloo, Canada

Examinateurs :

Serge Abiteboul	INRIA & ENS Cachan
Christine Froidevaux	Université Paris-Sud
Patrick Valduriez	INRIA & Université de Montpellier

Abstract

“Scalable algorithms for cloud-based Semantic Web data management”

Stamatis Zampetakis

In order to build smart systems, where machines are able to reason exactly like humans, data with semantics is a major requirement. This need led to the advent of the Semantic Web, proposing standard ways for representing and querying data with semantics. RDF is the prevalent data model used to describe web resources, and SPARQL is the query language that allows expressing queries over RDF data. Being able to store and query data with semantics triggered the development of many RDF data management systems.

The rapid evolution of the Semantic Web provoked the shift from centralized data management systems to distributed ones. The first systems to appear relied on P2P and client-server architectures, while recently the focus moved to cloud computing.

Cloud computing environments have strongly impacted research and development in distributed software platforms. Cloud providers offer distributed, shared-nothing infrastructures, that may be used for data storage and processing. The main features of cloud computing involve scalability, fault-tolerance, and elastic allocation of computing and storage resources following the needs of the users.

This thesis investigates the design and implementation of scalable algorithms and systems for cloud-based Semantic Web data management. In particular, we study the performance and cost of exploiting commercial cloud infrastructures to build Semantic Web data repositories, and the optimization of SPARQL queries for massively parallel frameworks.

First, we introduce the basic concepts around Semantic Web and the main components of cloud-based systems. In addition, we provide an extended overview of existing RDF data management systems in the centralized and distributed settings, emphasizing on the critical concepts of storage, indexing, query optimization, and infrastructure.

Second, we present AMADA, an architecture for RDF data management using public cloud infrastructures. We follow the Software as a Service (SaaS) model, where the complete platform is running in the cloud and appropriate APIs are provided to the end-users for storing and retrieving RDF data. We explore various storage and querying strategies revealing pros and cons with respect to performance and also to monetary cost, which is a important new dimension to consider in public cloud services.

Finally, we present CliqueSquare, a distributed RDF data management system built on top of Hadoop, incorporating a novel optimization algorithm that is able to produce massively parallel plans for SPARQL queries. We present a family of optimization algorithms, relying on n-ary (star) equality joins to build flat plans, and compare their ability to find the flattest possibles. Inspired by existing partitioning and indexing techniques we present a generic storage strategy suitable for storing RDF data in HDFS (Hadoop’s Distributed File System). Our experimental results validate the efficiency and effectiveness of the optimization algorithm demonstrating also the overall performance of the system.

Keywords: Semantic Web, RDF, commercial cloud services, indexing strategies, distributed systems, distributed storage, query processing, query optimization, query parallelization, MapReduce, Hadoop, HDFS, CliqueSquare, AMADA, RDF data management, n-ary joins, flat plans.

Résumé

“Algorithmes passant à l’échelle pour la gestion de données du Web sémantique sur les platformes cloud”

Stamatis Zampetakis

Afin de construire des systèmes intelligents, où les machines sont capables de raisonner exactement comme les humains, les données avec sémantique sont une exigence majeure. Ce besoin a conduit à l’apparition du Web sémantique, qui propose des technologies standards pour représenter et interroger les données avec sémantique. RDF est le modèle répandu destiné à décrire de façon formelle les ressources Web, et SPARQL est le langage de requête qui permet de rechercher, d’ajouter, de modifier ou de supprimer des données RDF. Être capable de stocker et de rechercher des données avec sémantique a engendré le développement des nombreux systèmes de gestion des données RDF.

L’évolution rapide du Web sémantique a provoqué le passage de systèmes de gestion des données centralisées à ceux distribués. Les premiers systèmes étaient fondés sur les architectures pair-à-pair et client-serveur, alors que récemment l’attention se porte sur le cloud computing.

Les environnements de cloud computing ont fortement impacté la recherche et développement dans les systèmes distribués. Les fournisseurs de cloud offrent des infrastructures distribuées autonomes pouvant être utilisées pour le stockage et le traitement des données. Les principales caractéristiques du cloud computing impliquent l’évolutivité, la tolérance aux pannes et l’allocation élastique des ressources informatiques et de stockage en fonction des besoins des utilisateurs.

Cette thèse étudie la conception et la mise en œuvre d’algorithmes et de systèmes passant à l’échelle pour la gestion des données du Web sémantique sur des platformes cloud. Plus particulièrement, nous étudions la performance et le coût d’exploitation des services de cloud computing pour construire des entrepôts de données du Web sémantique, ainsi que l’optimisation de requêtes SPARQL pour les cadres massivement parallèles.

Tout d’abord, nous introduisons les concepts de base concernant le Web sémantique et les principaux composants des systèmes fondés sur le cloud. En outre, nous présentons un aperçu des systèmes de gestion des données RDF (centralisés et distribués), en mettant l’accent sur les concepts critiques de stockage, d’indexation, d’optimisation des requêtes et d’infrastructure.

Ensuite, nous présentons AMADA, une architecture de gestion de données RDF utilisant les infrastructures de cloud public. Nous adoptons le modèle de logiciel en tant que service (software as a service - SaaS), où la plateforme réside dans le cloud et des APIs appropriées sont mises à disposition des utilisateurs, afin qu’ils soient capables de stocker et de récupérer des données RDF. Nous explorons diverses stratégies de stockage et d’interrogation, et nous étudions leurs avantages et inconvénients au regard de la performance et du coût monétaire, qui est une nouvelle dimension importante à considérer dans les services de cloud public.

Enfin, nous présentons CliqueSquare, un système distribué de gestion des données

RDF basé sur Hadoop. CliqueSquare intègre un nouvel algorithme d'optimisation qui est capable de produire des plans massivement parallèles pour des requêtes SPARQL. Nous présentons une famille d'algorithmes d'optimisation, s'appuyant sur les équijointures n-aires pour générer des plans plats, et nous comparons leur capacité à trouver les plans les plus plats possibles. Inspirés par des techniques de partitionnement et d'indexation existantes, nous présentons une stratégie de stockage générique appropriée au stockage de données RDF dans HDFS (Hadoop Distributed File System). Nos résultats expérimentaux valident l'effectivité et l'efficacité de l'algorithme d'optimisation démontrant également la performance globale du système.

Mot-clés: Web sémantique, RDF, plateformes commerciales de cloud computing, stratégies d'indexation, systèmes distribués, stockage distribué, traitement des requêtes, optimisation des requêtes, parallélisation de l'exécution de requêtes, MapReduce, Hadoop, HDFS, CliqueSquare, AMADA, gestion des données RDF, jointures n-aires, plans plats.

Acknowledgments

In this section, I express my sincere gratitude to all people who have helped in the completion of this thesis.

First and foremost, I would like to thank my advisors Ioana Manolescu and François Goasdoué. Despite my passion for exploring new ideas and solving interesting problems, I had never thought of starting a PhD before meeting them. It took me only a few weeks as a master intern to realize that I wanted to be part of this team and part of the long journey that is associated with conducting a PhD. From the beginning of my internship until the completion of this thesis they were abundantly helpful and offered invaluable assistance, support and guidance. Despite how hard a PhD can be, with Ioana and François, I would even consider starting a second one! Every meeting was a pleasant experience when I had the opportunity to learn more and more things each time. They are the role models to whom I will look up, if some day I supervise my own students.

I am grateful to Bernd Amann and Tamer Özsu for thoroughly reading my thesis and for their valuable feedback. Further, I would like to thank Serge Abiteboul, Christine Froidevaux, and Patrick Valduriez for being part of my examining committee; I am very honored.

Furthermore, I owe a lot to my colleagues Julien Leblay, Konstantinos Karanasos, Yannis Katsis, Zoi Kaoudi, Jorge Quiané-Ruiz, Jesús Camacho Rodríguez, Francesca Bugiotti, and Benjamin Djahandideh, with whom I was involved in very interesting projects. They are all contributors to this thesis in various ways even with writing code and doing French translations like Benjamin!

During my time within OAK, I was fortunate enough to meet great people and share many experiences with them. I would like to thank Damian, Asterios, Jesus, André, Juan, Sejla, Raphael, Soudip, Paul, Benoit, Andrés, Alexandra, Tushar, Guillaume, Fede, and Gianluca. Apart from colleagues you were great friends and you helped me enjoy every day that I was coming to the office.

Being also a member of LaHDAK, I had the opportunity to collaborate and socialize with many people, whom I thank for being a part of my everyday life. In particular, I am grateful to Chantal Reynaud for trusting me to be a teaching assistant at her course despite my basic knowledge in French. Moreover, I would like to thank Nicole Bidoit, not only for taking care of Katerina, but also for teaching me a lot of things (through transitive relations).

Further, I wish to offer my regards and blessings to all of my best friends for helping me get through the difficult times, and for all the emotional support, camaraderie, entertainment, and caring they provided during this thesis. Boulouki, Danai, Dimitri, Fab, Foivo, Ioana, Mairidio, Mathiou, Mitsara, Nelly, Xaze (Marie) thank you for everything.

Especially, I would like to thank Katerina for always being next to me since the very beginning (and before the beginning) of this thesis. At the hard times, she was the first to listen to my complaints, and disappointments, being also the person who could get me back on track. I am glad that she was there with me to celebrate every little or big success during this PhD. Apart from the emotional support our scientific discussions and her productive comments are incorporated in various places inside my thesis, presentations, and papers.

Last, but not least, I wish to thank my brother, Ilias, and my parents, Giorgos and Antonia. They raised me, supported me, taught me, and loved me. To them I dedicate this thesis.

Contents

Chapter 1

Introduction

The World Wide Web (or WWW, in short) dates back in the early '90s when Tim Berners-Lee had a vision of a more effective communication system for CERN. The Web is defined as a system of interlinked hypertext documents that are accessed through the Internet. Hypertext documents are documents containing text, that is annotated using a specific markup language (HTML), so that individual machines can render the content correctly. A particularly crucial feature of HTML is the ability to represent links between documents; this is a major ingredient of the Web as a content and interaction platform, allowing users to navigate from one document to another.

Looking back to the first web site to ever go online [1] in November of 1992, and comparing it with some of the most well known websites (for example Youtube [2]) today, the differences are very striking. It is not only the technology that evolved but also the philosophy behind them. The first web sites are characterized by static content, which corresponds to considering the users purely as consumers. The web site is provided to the user following the paradigm of a newspaper, where users can flip pages, respectively, navigate back and forth to gather information, but the interaction is limited. This static incarnation of the Web is associated with the term *Web 1.0.* On the other hand, a web site like Youtube is fundamentally built by the users, whose role has changed from passive consumers to active producers of information. This mentality change in the way web pages are built, is usually referred to using the term *Web 2.0.*

1.1 Semantic Web

Currently, information production and consumption on the Web is moving toward the so-called Web 3.0, also known as the Semantic Web. Consider for instance the query *"What are the most common bugs?"* To answer it, Youtube proposes a list of videos about spiders, cockroaches, mosquitoes, and other type of insects. For non-programmers, this may be the correct answer, however for a tech-oriented user, the desired answers should have been videos about bugs in computer programs. Humans can very easily reason using

1. http://www.w3.org/History/19921103-hypertext/hypertext/WWW/TheProject.html
2. https://www.youtube.com/

context to infer the correct answer, yet this is still not completely handled by computers, which either miss the relevant context or are not able to take advantage of it. The Semantic Web describes an extension of the Web where computers will be able to process efficiently the data and reason about them exactly as human would do. Tim Berners-Lee stated the following back in 2000 [BF00]:

> *"I have a dream for the Web [in which computers] become capable of analyzing all the data on the Web – the content, links, and transactions between people and computers. A "Semantic Web", which makes this possible, has yet to emerge, but when it does, the day-to-day mechanisms of trade, bureaucracy and our daily lives will be handled by machines talking to machines."*

In order for computers to be able to analyze all the data, these have to be under a common data model. The World Wide Web Consortium (W3C) introduced the Resource Description Framework (RDF) [W3C14b] for this purpose. RDF is a flexible data model that permits to express statements about available resources. The statements (a.k.a. triples) are of the form $(s\ p\ o)$ where s stands for the subject, p for the property, and o for the object. This requires that all resources are uniquely identified, i.e., all resources have a unique resource identifier (URI). Currently, every web page has a URI, but the vision of the Semantic Web is not restricted to documents (web pages) but anything inside the documents, or, more generally, any resource of the physical world as well.

If a famous soccer player, e.g., Cristiano Ronaldo, appears inside a document, then he has a URI, and all other entities (players, teams, stadiums, matches, etc.) refer to him by this URI. These connections between different resources make RDF a *graph* data model. A vivid example of the use of RDF in the Web is the site for the World Football Cup of 2010 created by BBC[3]. The power of RDF comes from the ability to link entities even if they do not belong to the same dataset. For example, BBC expresses, using RDF, the information relevant to the World Cup. It produces statements like the number of goals Cristiano Ronaldo achieved in the tournament, the number of yellow and red cards he received, etc. Then, in another dataset, e.g., in DBPedia[4] (the semantic counterpart of Wikipedia[5]) there is further information about Ronaldo, like his age, his nationality, information about his family, and many more things. DBPedia appeared before 2010, thus Cristiano Ronaldo was already assigned a URI by the time that the World Cup started. BBC uses the same URI for expressing the new statements regarding the performance of Ronaldo in the World Cup benefiting from the existing information, and contributing also with new knowledge which can be exploited by DBPedia or any other dataset using the same URI for Ronaldo. These interlinked datasets containing semantic information are known as Linked Open Data[6].

Figure 1.1 [SBP14] shows some of the most well known Linked Open Data (RDF) datasets, crawled in April 2014. Each node in the graph corresponds to an RDF dataset,

3. `http://www.bbc.co.uk/blogs/bbcinternet/2010/07/bbc_world_cup_2010_dynamic_sem.html`
4. `http://wiki.dbpedia.org/`
5. `https://en.wikipedia.org/wiki/Main_Page`
6. `http://lod-cloud.net/`

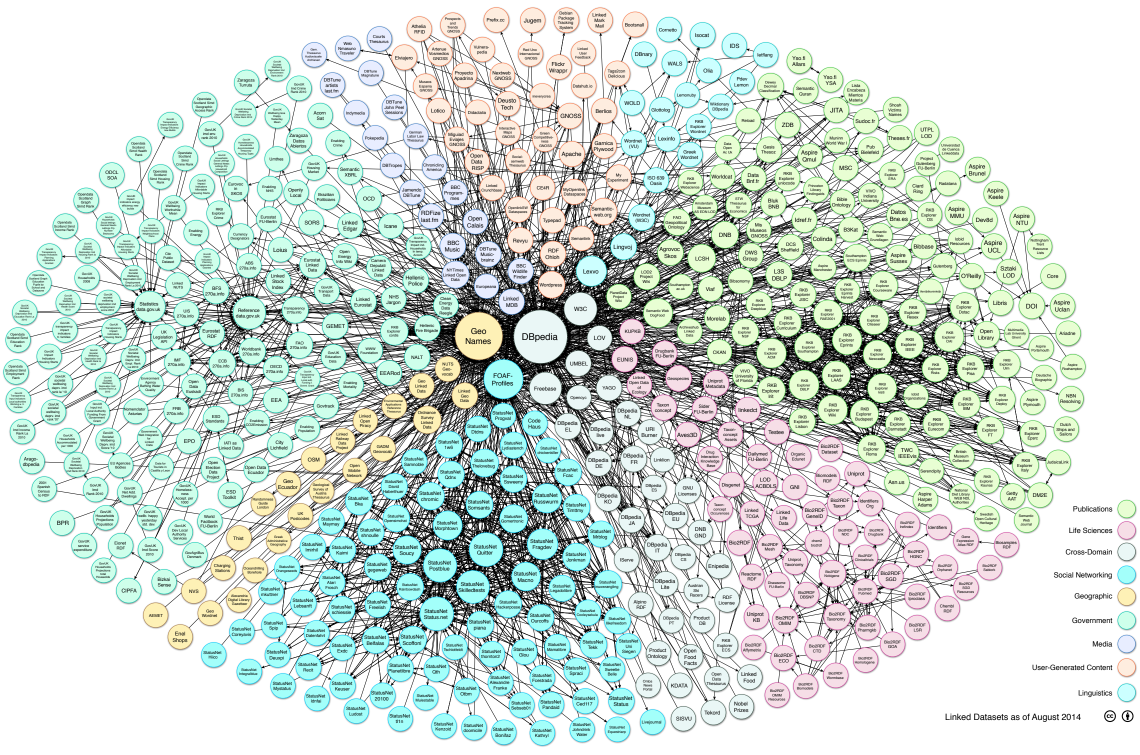


Figure 1.1: The Linked Open Data cloud as of April 2014.

while the node size reflects the size of the respective dataset. In addition, there is an edge between two nodes if the two datasets have some URIs in common. As reported in [SBP14], the size of the Semantic Web is growing, almost doubling its size every year.

This amount of RDF data would be worthless without the means to extract relevant information from them. There is an evident need for a query language and for query processing tools. Although various languages have been proposed for querying RDF data [KAC+02, MSR02, Sea04, MSP+04] the one standardized by W3C is SPARQL [W3C13]. SPARQL, following closely the RDF data model, is a graph-based language with many powerful features such as aggregation and optional clauses. The most frequently used dialect is that of Basic Graph Pattern queries (or BGP, in short).

1.2 Semantic Web data management

With respect to Semantic Web query processing, three main approaches have been identified. In the *warehousing* approach, the interesting data sources are gathered into a central repository, where they can be further processed and queried to return answers to the users. Among the best-known warehousing approaches are RDFSuite [ACK+01], Sesame [BKvH02], Jena [McB02], Virtuoso [Erl08], RDF-3X [NW10], and gStore [ZMC+11]. The second approach relies on the existence of SPARQL *endpoints*, where the data publishers provide a way to access their data through SPARQL queries. In this case, to evaluate a query over different endpoints (different data sources) the query is decomposed, sent to the proper sources, and then the intermediate results must be combined before returning the answer to the user. Illustrative systems for this category include DARQ [QL08], SPLENDID [GS11], FedX [SHH+11], ANAPSID [AV11]; the interested reader may also find in [RUK+13] a survey of query processing using federated endpoints. An alternative [PZÖ+14] is to send the complete query to all available data sources finding (local) partial matches, which are assembled together using centralized or distributed methods. The third approach, considers again individual data sources (similar to the endpoint approach) but the existence of SPARQL endpoints is optional thus some sources do not provide query processing capabilities. Additionally, the data sources may not be known in advance but are determined during query evaluation by following links. Representative works in this area are based on graph-traversal [LT11, Har13], indexes [UHK+11, TUY11], or a combination of both [LT10]. The aforementioned approaches each have their own advantages and disadvantages, and none of them is best in all circumstances. *In this thesis, we adopt the warehousing approach*, since, when data can be gathered in a warehouse (that is, when ownership, access rights, or resource limitations don't preclude it), the data warehouse setting is amenable to the best query evaluation performance.

Warehousing RDF data raises its own challenges. Large volumes of data have to be gathered and managed in a single repository. Efficient systems [SA09, FCB12, LPF+12] have been devised in order to handle large volumes of RDF data in a centralized setting, with RDF-3X [NW10] being among the best-known. However, as the amount of data continues to grow, it is no longer feasible to store the entire linked data sets on a single machine and still be able to scale to multiple and varied user requests. Thus, such huge data volumes have raised the need for distributed storage architectures and query processing frameworks.

1.3 Distributed systems & challenges

Building a distributed system requires first ensuring access to sufficient storage and computation resources. Making hardware equipment decision nowadays is mostly considered from a "buy or rent?" perspective, especially since the advent of Cloud computing [MG11] , advocating a pay-per-use model where computational resources are available in the cloud and rented in order to run applications. Although there are many disagreements[7] about the exact definition of Cloud computing, the pay-per-use model is widely agreed upon. Building a system based on the Cloud improves its flexibility and reduces the overall cost for its users. *The systems and techniques described in this thesis are designed and built for cloud-based infrastructures.* The design and implementation of a distributed RDF data management platform raises many challenges, that can be roughly grouped into the following four interrelated categories:

1. The first challenge is to provide an *architecture* that can be easily deployed in the cloud satisfying the requirements for scalability, elasticity, and performance.

7. `http://cloudcomputing.sys-con.com/node/612375/print`

2. The second challenge amounts to choosing a good *partitioning* scheme that distributes effectively the data to the available machines.
3. Third, it is important to build and maintain proper *indexes* to the data for improving access performance.
4. Last but not least, efficient *query optimization* algorithms are needed to build query plans that can be easily parallelized, in order to take advantage of the parallel processing infrastructure.

1.4 Contributions and outline

In the following, we provide an overview of the organization of the thesis and we outline our main contributions.

Chapter 2 provides the necessary background to follow the rest of the thesis. Moreover, it provides a systematic analysis and classification of existing works in the area of RDF data management.

Chapter 3 presents AMADA, an architecture for RDF data management using public cloud infrastructures. The contributions of this chapter are the following:
– We present an architecture for storing RDF data within the Amazon cloud that provides efficient query performance, both in terms of time and monetary costs.
– We consider hosting RDF data in the cloud, and its efficient storage and querying through a (distributed, parallel) platform also running in the cloud.
– We exploit RDF indexing strategies that allow to direct queries to a (hopefully tight) subset of the RDF dataset which provide answers to a given query, thus reducing the total work entailed by query execution.
– We provide extensive experiments on real RDF datasets validating the feasibility of our approach and giving insight about the monetary cost of storing RDF data in the cloud.

Chapter 4 presents CliqueSquare, an optimization approach for building massively parallel flat plans for RDF queries. The contributions of this chapter and *the main contributions of this thesis* are the following:
– We describe a search space of logical plans obtained by relying on *n-ary (star) equality joins*. The interest of such joins is that by aggressively joining many inputs in a single operator, they allow building flat plans.
– We provide a novel generic algorithm, called CliqueSquare, for exhaustively exploring this space, and a set of three algorithmic choices leading to eight variants of our algorithm. We present a thorough analysis of these variants, from the perspective of their ability to find one of (or all) the flattest possible plans for a given query.

- We show that the variant we call CliqueSquare-MSC is the most interesting one, because it develops a reasonable number of plans and is guaranteed to find some of the flattest ones.
- We have fully implemented our algorithms and validate through experiments their practical interest for evaluating queries on very large distributed RDF graphs. For this, we rely on a set of relatively simple parallel join operators and a generic RDF partitioning strategy, which makes no assumption on the kinds of input queries. We show that CliqueSquare-MSC makes the optimization process efficient and effective even for complex queries leading to robust query performance.

Chapter 5 concludes and outlines possible future directions as well as some ongoing work.

Chapter 2

Background and state-of-the-art

This chapter presents the background needed by the presentation of the research work performed in the thesis along with the related work in the area of RDF data management and massively parallel systems. Section 2.1 presents the Resource Description Framework (RDF) comprising the core of the Semantic Web. Section 2.2 discusses the recent developments in distributed storage infrastructures and provides a brief introduction to the MapReduce programming framework. Finally, Section 2.3 provides an extended overview of existing RDF data management systems in the centralized and distributed settings, emphasizing on the critical concepts of storage, indexing, query optimization, and infrastructure.

2.1 Resource Description Framework (RDF)

The Resource Description Framework (RDF) [W3C14a] is a family of W3C specifications originally intended to describe metadata [W3C04]. Since then it has evolved and matured, aiming to describe any kind of information available on the web.

The framework relies on an abstract data model that is naturally represented as a directed labeled graph, where nodes correspond to resources or literals and edges are used to describe the relations between the resources. The graph does not need to obey to a specific schema. Further, RDF graph may comprise implicit information, specified by means of a set of so-called *entailment rules*.

The standard language for querying RDF data is SPARQL [W3C08]. SPARQL is part of the W3C recommendations and it has an SQL-like syntax. The basic building block of the language is the graph pattern, that permits the retrieval of parts of the graph. The language also has many other features, such as optional query patterns, RDF graph constructions, property path querying by means of regular expressions etc.

We formalize the RDF data model in Section 2.1.1, while in Section 2.1.2 we present SPARQL, and the fragment that we consider in this work.

2.1.1 Data model

RDF data is organized in *triples* of the form $(s\ p\ o)$, stating that the subject s has the property (a.k.a. predicate) p whose value is the object o. *Unique Resource Identifiers* (URIs) are central in RDF: one can use URIs in any position of a triple to uniquely refer to some entity or concept. Notice that literals (constants) are also allowed in the o position.

RDF allows some form of incomplete information through *blank nodes*, standing for unknown constants or URIs. One may think of blank nodes as *labeled nulls* from the database literature [AHV95].

Definition 2.1.1 (RDF triple)**.** *Let U be a set of URIs, L be a set of literals, and B be a set of blank nodes. A well-formed* RDF triple *is a tuple $(s\ p\ o)$ from $(U \cup B) \times U \times (U \cup L \cup B)$.*

The syntactic conventions for representing valid URIs, literals, and blank nodes can be found in [W3C14a]. In this document, literals are shown as strings enclosed by quotation marks, while URIs are shown as simple strings (see also discussion on namespaces below).

RDF admits a natural graph representation, with each $(s\ p\ o)$ triple seen as an p-labeled directed edge from the node identified by s to the node identified by o.

Definition 2.1.2 (RDF graph)**.** *An* RDF graph *is a set of RDF triples.*

We use $val(G)$ to refer to the values (URIs, literals, and blank nodes) of an RDF graph G.

An example of an RDF graph from the domain of a university is shown in Figure 2.1. In the bottom part of the figure, the RDF graph is shown as a set of triples. The ovals represent URIs and the rectangles represent literals. The example RDF graph is a fictional instance from the LUBM benchmark [GPH05].

In some cases we need to work with many RDF graphs at the same time, while keeping their content separate. We achieve this by considering *named* RDF graphs where each graph is associated with a name that can be a URI or a blank node. The notion of an RDF triple is extended as follows to capture these needs.

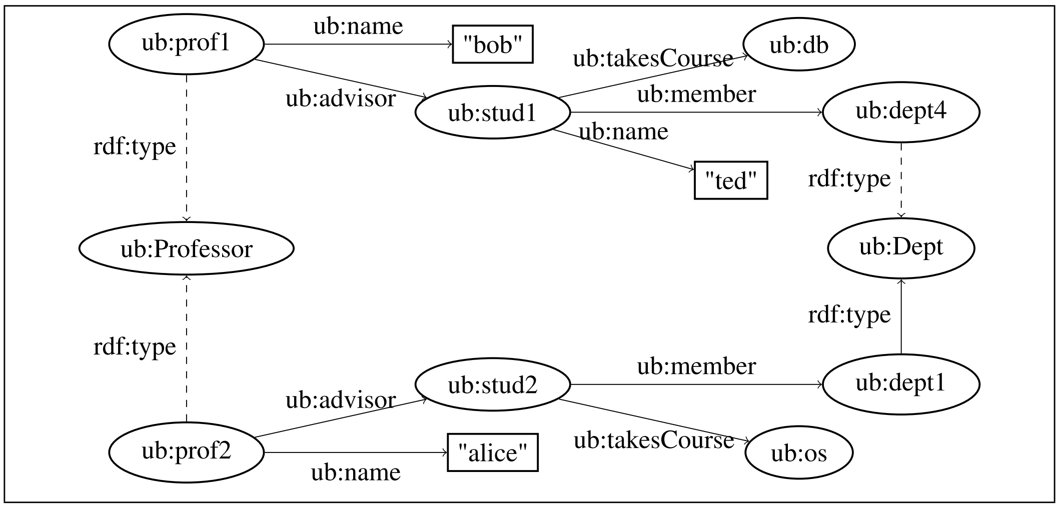
Definition 2.1.3 (RDF quad)**.** *Let U be a set of URIs, L be a set of literals, and B be a set of blank nodes. A well-formed* RDF quad *is a tuple $(s\ p\ o\ g)$ from $(U \cup B) \times U \times (U \cup L \cup B) \times (U \cup B)$.*

We are now able to capture multiple RDF graphs using the notion of an RDF dataset.

Definition 2.1.4 (RDF dataset)**.** *An* RDF dataset *is a set of RDF quads.*

An RDF dataset may contain only a single graph, in which case all the quads of the form $(s\ p\ o\ g)$ have the same value for g. In such cases, we may use the term RDF graph and RDF dataset interchangeably.

An example of an RDF dataset is shown in Figure 2.2. The dataset consists of two named graphs, ub:Professors graph, and ub:Students graph. In the bottom part of the figure, the dataset is shown as a set of quads.



```
(ub:prof1 ub:name "bob")                (ub:stud1 ub:member ub:dept4)
(ub:prof1 ub:advisor ub:stud1)          (ub:stud1 ub:takesCourse ub:db)
(ub:prof2 ub:advisor ub:stud2)          (ub:stud1 ub:name "ted")
(ub:prof2 ub:name "alice")              (ub:stud2 ub:member ub:dept1)
(ub:prof1 rdf:type ub:professor)        (ub:stud2 ub:takesCourse ub:os)
(ub:prof2 rdf:type ub:professor)        (ub:dept1 rdf:type ub:Dept)
                                        (ub:dept4 rdf:type ub:Dept)
```

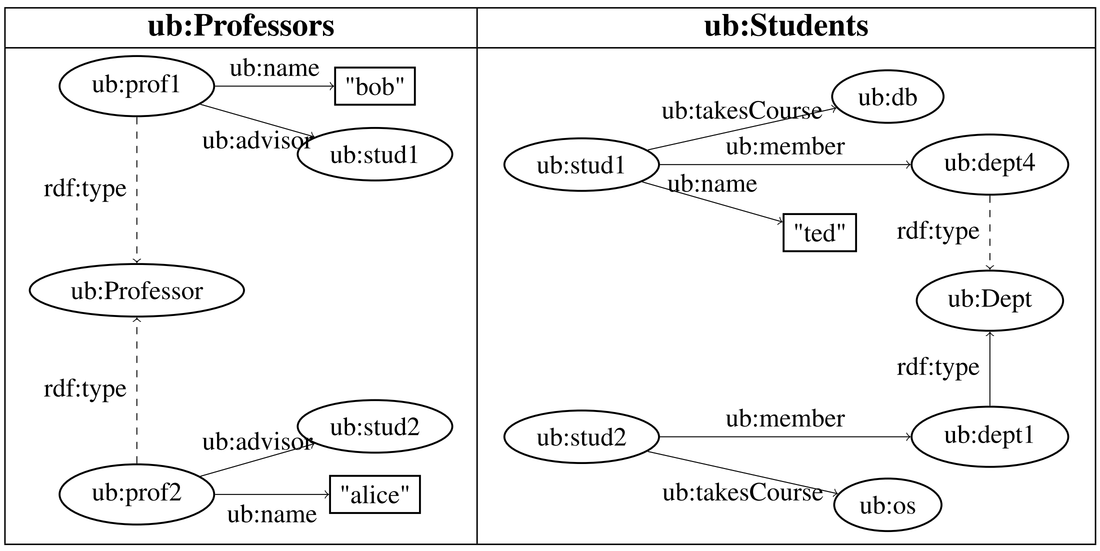
Figure 2.1: Example of an RDF graph.

Definition 2.1.5 (RDF merge)**.** *The* RDF merge *of two RDF graphs G_1, G_2, is the RDF graph G, defined as the set union of triples in G_1 and G_2 where blank nodes with the same labels in G_1 and G_2, are renamed to avoid such collisions.*

For example, merging the two graphs in Figure 2.2 is the graph shown in Figure 2.1.

Namespaces are supported in RDF as a means to support flexible choices of URIs as well as interoperability between different datasets. A namespace typically serves to identify a certain application domain. Concretely, a namespace is identified by a URI, which is used as a prefix of all URIs defined within the respective application domain. Thus, for instance, the URI `http://www.w3.org/1999/02/22-rdf-syntax-ns` is chosen by the W3C to represent the domain of a small set of predefined URIs which are part of the RDF specification itself; or, for instance, `http://swat.cse.lehigh.edu/onto/univ-bench.owl` is used by the fictious university of Lehigh to identify its domain of representation. To denote that the URI of a resource r is part of the application domain identified by a namespace URI u, the URI of u is a prefix of the URI of r. The suffix of r's URI is typically called *local name*; it identifies uniquely r among all the resources of the namespace u. This enables other application domains to use the same local name in conjunction with their respective namespace URIs, without causing confusions between the two. On the opposite, when one wishes to refer in a dataset to a specific resource from a specific namespace, the full URI (including the namespace URI prefix) must be used.

While the above mechanism is flexible, it leads to rather lengthy URIs, and bloats the



```
(ub:prof1 ub:name "bob" ub:Professors)
(ub:prof1 ub:advisor ub:stud1 ub:Professors)
(ub:prof2 ub:advisor ub:stud2 ub:Professors)
(ub:prof2 ub:name "alice" ub:Professors)
(ub:prof1 rdf:type ub:professor ub:Professors)
(ub:prof2 rdf:type ub:professor ub:Professors)
```

```
(ub:stud1 ub:member ub:dept4 ub:Students)
(ub:stud1 ub:takesCourse ub:db ub:Students)
(ub:stud1 ub:name "ted" ub:Students)
(ub:stud2 ub:member ub:dept1 ub:Students)
(ub:stud2 ub:takesCourse ub:os ub:Students)
(ub:dept1 rdf:type ub:Dept ub:Students)
(ub:dept4 rdf:type ub:Dept ub:Students)
```

Figure 2.2: Example of an RDF dataset comprising two named graphs.

space needed to represent a dataset. To reduce the space occupancy, within an RDF graph, a *local namespace prefix* is associated to a namespace URI, and serves as a short-hand to represent the latter. Thus, URIs are typically of the form *nsp*:*ln* where *nsp* stands for the local namespace prefix while *ln* represents the local name. For example, in Figure 2.1 `ub` is used as a local namespace prefix to replace `http://swat.cse.lehigh.edu/onto/univ-bench.owl`.

Resource descriptions can be enhanced by specifying to which *class(es)* a given resource belongs, by means of the pre-defined `rdf:type` property which is part of the RDF specification. For example, the triple `(:dept1 rdf:type :Dept)` in Figure 2.1 states that `:dept1` is of type `:Dept`.

Further, the RDF Schema [W3C14c] specification allows relating classes and properties used in a graph, through ontological (i.e., deductive) constraints expressed as triples using built-in properties: sub-class constraints `rdfs:subClassOf`, sub-property constraints `rdfs:subPropertyOf`, typing constraints of the first attribute `rdfs:domain` of a property and typing constraints of the second attribute `rdfs:range` of a property. RDFS constraints and the corresponding relational constraints are given in Table 2.1. For instance, if we know that `(ub:stud1 ub:member ub:dept4)` and that one can only be member of a department (expressed by the RDF Schema constraint `(ub:member rdfs:range ub:Dept)`), then due to the fourth constraint in Table 2.1, `ub:dept4` is necessarily a department (and the triple `(ub:dept4 rdf:type ub:Dept)` holds in the RDF graph). As another example, assuming that every advisor is

RDFS constraint	Relational modeling
`(s rdfs:subClassOf o)`	$\forall x[s(x) \rightarrow o(x)]$
`(s rdfs:subPropertyOf o)`	$\forall x,y[s(x,y) \rightarrow o(x,y)]$
`(p rdfs:domain c)`	$\forall x,y[p(x,y) \rightarrow c(x)]$
`(p rdfs:range c)`	$\forall x,y[p(x,y) \rightarrow c(y)]$

Table 2.1: Deductive constraints expressible in an RDF Schema

Premise (existing triples)	Conclusion (inferred triples)
(s rdf:type c_1) (c_1 rdfs:subClassOf c_2)	(s rdf:type c_2)
(s p_1 o) (p_1 rdfs:subPropertyOf p_2)	(s p_2 o)
(s p o) (p rdfs:domain c)	(s rdf:type c)
(s p o) (p rdfs:range c)	(o rdf:type c)

Table 2.2: Entailment rules combining schema and instance triples

a professor (described by the RDF schema constraint `(ub:advisor rdfs:domain ub:Professor)`) we can infer that `ub:prof1` and `ub:prof2` are professors (thus the triples `(ub:prof1 rdf:type ub:Professor)` and `(ub:prof2 rdf:type ub:Professor)` hold in the graph) due to the third constraint of Table 2.1. The previous examples reveal an important feature of RDF which is implicit information: triples which hold in the RDF graph, even though they may not be part of it explicitly (denoted with blue in Figure 2.1). The process of infering new triples based on existing ones (some of them may be RDF Schema constraints, while the others are simple triples, a.k.a. facts) is known as RDF entailment or *inference* and is guided by a set of entailment rules. Table 2.2 presents four of the most common entailment rules that are directly associated with the constrains of Table 2.1 and they are those that were used implicitly at the examples above. The full set of entailment rules is defined in the RDF Semantics [W3C14b].

Observe that unlike the traditional setting of relational databases, RDF Schema constraints are expressed with RDF triples themselves, and are part of the RDF graph (as opposed to relational schemas being separated from the relational database instances). Within an RDF dataset, the term *fact* is commonly used to denote a triple whose property is not one of the predefined RDF Schema properties.

Definition 2.1.6 (Saturation)**.** *The saturation of an RDF graph G, denoted G^{∞}, is obtained by adding to G all the implicit triples that derive from consecutive applications of the entailment rules on G, until a fixpoint is reached.*

It has been shown that under RDF Schema constraints, the saturation of an RDF graph is finite, and unique (up to blank node renaming).

Considering again the example of Figure 2.1, the graph without the solid edges is the unsaturated graph, while the complete graph (comprising both solid and dashed edges) is the saturated graph.

The problem of efficiently computing the saturation of an RDF dataset has been considered both in centralized settings [BK03, UMJ+13, MNP+14, GMR13, BGM15] and in distributed ones [UKOvH09, UvHSB11]. *In this thesis, we assume that all RDF datasets are saturated*, using such a technique.

2.1.2 Query language

The advent of the Semantic Web along with the arrival of the RDF data model entailed the need for a suitable declarative query language. The first steps have been done with RQL [KAC+02], a declarative query language closely inspired from the data model, providing ways of querying simultaneously the data and the schema. Other languages emerging at about the same time include SquishQL [MSR02], RDQL [Sea04], RSQL [MSP+04]. Inspired by the aforementioned efforts, the SPARQL [W3C08] has been proposed and has become a W3C standard in 2008. SPARQL has evolved subsequently; the current version (SPARQL 1.1 [W3C13]) resembles a complex relational query language such as SQL.

SPARQL has a variety of features. The simplest ones are: conjunctive graph pattern matching, selections, projections, and joins, while the more advanced allow arithmetic and alphanumeric comparisons, aggregations, nested sub-queries, and graph construction. In this work, we consider the most common fragment of SPARQL, named *basic graph pattern (BGP)* queries; from a database perspective, these correspond to conjunctive select-project-join (SPJ) queries.

A central role in composing BGPs is played by triple patterns.

Definition 2.1.7 (Triple Pattern)**.** *Let U be a set of URIs, L be a set of literals and V be a set of variables, a* triple pattern *is a tuple $(s\ p\ o)$ from $(U \cup V) \times (U \cup V) \times (U \cup L \cup V)$.*

Triple patterns are used to specify queries against a single RDF graph. To query datasets, possibly consisting of many datasets, we extend to the notion of a triple pattern to a quad pattern.

Definition 2.1.8 (Quad Pattern)**.** *Let U be a set of URIs, L be a set of literals, and V be a set of variable, a* quad pattern *is a tuple $(s\ p\ o\ g)$ from $(U \cup V) \times (U \cup V) \times (U \cup L \cup V) \times (U \cup V)$.*

Based on triples (or quad) patterns, one can express SPARQL BGP queries as below.

Definition 2.1.9 (BGP Query)**.** *A* BGP query *is an expression of the form*

$$\texttt{SELECT ?x}_1\texttt{...?x}_m\texttt{ WHERE \{ t}_1\texttt{...t}_n\texttt{ \}}$$

where $\texttt{t}_1\texttt{...t}_n$ triple (quad) patterns and $\texttt{?x}_1\texttt{...?x}_m$ distinguished variables appearing in $\texttt{t}_1\texttt{...t}_n$.

In the rest of the thesis we will use the terms *RDF query*, *SPARQL query*, and *BGP query*, interchangeably referring to the SPARQL fragment described by Definition 2.1.9. Furthermore, we use $var(q)$ to refer to the variables of a query q. An example BGP query, asking for advisors of students that are member of department four, is shown in Figure 2.3.

```
SELECT ?advisor ?student
WHERE {
      ?advisor ub:advisor ?student
      ?student ub:member ub:dept4
}
```

Figure 2.3: Example RDF query QA.

Definition 2.1.10 (Valid assignment). *Let q be a BGP query, and G be an RDF graph,* $\mu : var(q) \to val(G)$ is a valid assignment *iff* $\forall t_i \in q,\ t_i^\mu \in G$ *where we denote by* t_i^μ *the result of replacing every occurrence of a variable* $e \in var(q)$ *in the triple pattern* t_i *by the value* $\mu(e) \in val(G)$.

Definition 2.1.11 (Result tuple). *Let q be a BGP query, G be an RDF graph, μ be a valid assignment, and $\bar{x}$ be the head variables of q, the* result tuple of q based on μ, *denoted as* $res(q,\mu)$, *is the tuple:*

$$res(q,\mu) = \{\mu(x_1),\ldots,\mu(x_m) \mid x_1,\ldots,x_m \in \bar{x}\}$$

Definition 2.1.12 (Query evaluation). *Let q be a BGP query, and G be an RDF graph, the* evaluation of q against G *is:*

$$q(G) = \{res(q,\mu) \mid \mu : var(q) \to val(G) \text{ is a valid assignment}\}$$

where $res(q,\mu)$ is the result tuple of q based on μ.

The evaluation of query QA (shown in Figure 2.3) against the graph of Figure 2.1 is the tuple `(ub:prof1 ub:stud1)`.

Query evaluation only accounts for triples explicitly present in the graph. If the graph is not saturated, evaluating the query may miss some results which would have been obtained if the graph had been saturated. The following definition allows capturing results due both to the explicit and to the implicit triples in an RDF graph:

Definition 2.1.13 (Query answering). *The* answer of a BGP query q over an RDF graph G *is the evaluation of q over G^∞.*

It is worth noting that while the relational SPJ queries are most often used with set semantics, SPARQL, just like SQL, has bag (multiset) semantics. As a result BGP queries considered in this work have bag semantics.

2.2 Distributed storage and MapReduce

The vast amount of data available today, along with the needs for scalability, fault-tolerance and inexpensive performance, triggered the development of many distributed systems relying on distributed file systems and/or distributed key-value stores, and the

MapReduce framework. Section 2.2.1 briefly introduces the ideas behind a distributed file system, while Section 2.2.2 describes key-value stores along with some concrete systems. Section 2.2.3 outlines the MapReduce framework, and Section 2.2.5 discusses higher level languages and implementations for MapReduce.

2.2.1 Distributed file systems

Distributed file systems have their roots way back in the '80s [SGK+85]. However, with the emergence of the cloud computing, new distributed file systems are still being developed.

In an attempt to cover growing data processing needs, Google introduced the Google File System [GGL03] (GFS), a distributed file system which aims at providing performance, scalability, reliability and availability. However, it makes some radical design choices to support more effectively the following scenarios. First, node failures are not considered an exception but the rule. Second, the file size is big (in terms of GB) and the number of files moderate. Small files are supported but the performance tuning shifts towards the big ones. Third, updates are typically handled by appending to the file rather than overwriting. Finally, the API is flexible so as to facilitate the development process of applications.

In the same spirit, and closely following the GFS design principles, other distributed file systems were developed, like Apache's Hadoop Distributed File System [had11] (HDFS), and Amazon's S3 [s306]. HDFS became popular due to the opensource implementation that Apache provided.

2.2.2 Distributed key-value stores

A key-value store (key-value database) is a system for storing, retrieving, and managing associative arrays. An associative array is a data structure that can hold a set of `{key:value}` pairs such that each possible key appears just once. Other common names for associative arrays, include map, dictionary, and symbol table. Key-value stores have a history as long as relational databases. In contrast with relational databases, key-value stores have a very limited API and the vast-majority of them does not support join operations between different arrays. The most basic operations supported by all key-value stores are `Get(k,v)` and `Put(k,v)`.

Recently, key-value stores gained a lot of popularity due to their simplistic design, horizontal scaling, and finer control over availability. Google's Bigtable [CDG+06] inspired many of the key-value stores that are used nowadays. Bigtable takes the idea of associative arrays one step further, defining each array as a sparse, distributed, persistent multidimensional sorted map. The Bigtable's map indexes a value using a triple comprised from the row key, column key, and a timestamp. Each map implies a nested structure of the form `{rowkey:{columnkey:{time:value}}}`. By considering the map (array) as part of the nested structure the complete BigTable's architecture can be described as `{tablename:{rowkey:{columnkey:{time:value}}}}`. Using the ER diagram formalism, Key-Value stores similar to BigTable adopt the schema shown in Figure 2.4.

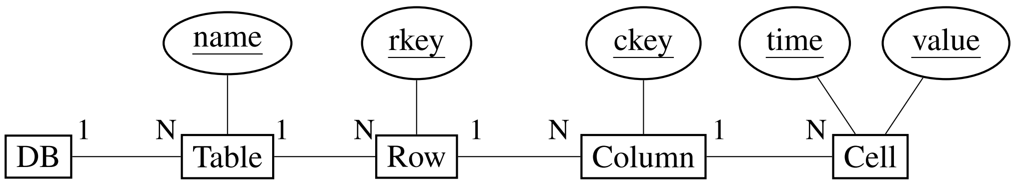


Figure 2.4: ER Diagram for common key-value stores.

Popular key-value stores that have been used by RDF systems include: Apache's Cassandra [cas08], Apache's Accumulo [acc08], Apache's HBase [hba08], Amazon's SimpleDB [sim07], and Amazon's DynamoDB [dyn12]. Although they share the basic elements of their interfaces, these systems differ with respect to their internal architecture, access control policies, authentication, consistency, etc. Below we briefly present Accumulo, Cassandra, and HBase, while we provide a slightly more elaborated overview for SimpleDB, and DynamoDB since they are used in AMADA (Chapter 3).

HBase is an open-source, distributed, versioned, non-relational database modeled after Google's Bigtable and implemented on top of HDFS. A data row in HBase is composed of a sortable row key and an arbitrary number of columns, which are further grouped into column families. A data cell can hold multiple versions of data which are distinguished by timestamps. Data stored in the same column family are stored together on the file system, while data in different column families might be distributed. HBase provides a B+ tree-like index on the row key by default. HBase supports ACID-level semantics on a per-row basis (row-level consistency). In addition, the notion of coprocessors is introduced, which allow the execution of user code in the context of the HBase processes. The result is roughly comparable to the relational database world's triggers and stored procedures.

Accumulo provides almost identical features to HBase, since it also follows Bigtable design pattern and is implemented on top of HDFS. In contrast with HBase and Bigtable, it provides the server Iterator model that helps increasing performance by performing large computing tasks directly on the servers and not on the client machine. By doing this, it avoids sending large amounts of data across the network. Furthermore, it extends the Bigtable data model, adding a new element to the key called "Column Visibility". This element stores a logical combination of security labels that must be satisfied at query time in order for the key and value to be returned as part of a user request. This allows data with different security requirements to be stored in the same table. As a consequence users can see only those keys and values for which they are authorized.

Cassandra is also inspired by Bigtable and implemented on top of HDFS, thus sharing a lot of similarities with Accumulo and HBase. Nevertheless, it has some distinctive features. It extends the Bigtable data model by introducing supercolumns. A storage model with supercolumns looks like: `{rowkey:{superkey:{columnkey:value}}}`. Supercolumns can be either stored based on the hash value of the supercolumn key or in sorted order. In addition, supercolumns can be further nested. Cassandra natively supports secondary indices, which can improve data access performance in columns whose values have a high level of repetition. Furthermore, It has configurable consistency. Both read and write consistency can be tuned, not only by level, but in extent. Finally, Cassandra provides an SQL-like language, CQL, for interacting with the store.

SimpleDB is a non relational data store provided by Amazon which focuses on high availability (ensured through replication), flexibility and scalability. SimpleDB supports a set of APIs to query and store items in the database. A SimpleDB data store is organized in domains. Each domain is a collection of items identified by their name. Each item contains one or more attributes; an attribute has a name and a set of associated values. There is a one to one mapping from SimpleDB's data model to the one proposed by Bigtable shown in Figure 2.4. Domains correspond to tables, items to rows, attributes to columns, and values to cells. The main operations of SimpleDB API are the following (the respective delete/update operations are also available):

- `ListDomains()` retrieves all the domains associated to one AWS account.
- `CreateDomain(D)` and `DeleteDomain(D)` respectively creates a new domain D and deletes an existing one.
- `PutAttributes(D, k, (a,v)+)` inserts or replaces attributes (a,v)+ into an item with name k of a domain D. If the item specified does not exist, SimpleDB will create a new item.
- `BatchPutAttributes` performs up to 25 `PutAttributes` operations in a single API call, which allows for obtaining a better throughput performance.
- `GetAttributes(D, k)` returns the set of attributes associated with item k in domain D.

It is not possible to execute an API operation across different domains as it is not possible to combine results from many tables in Bigtable. Therefore, if required, the aggregation of results from API operations executed over different domains has to be done in the application layer. AWS ensures that operations over different domains run in parallel. Hence, it is beneficial to split the data in several domains in order to obtain maximum performance. As most non-relational databases, SimpleDB does not follow a strict transactional model based on locks or timestamps. It only provides the simple model of conditional puts. It is possible to update fields on the basis of the values of other fields. It allows for the implementation of elementary transactional models such as some entry level versions of optimistic concurrency control.

AWS imposes some size and cardinality limitations on SimpleDB. These limitations include:

- Domains number: the default settings of an AWS account allow for at most 250 domains. While it is possible to negotiate more, this has some overhead (one must discuss with a sale representative etc. - it is not as easy as reserving more resources through an online form).
- Domain size: the maximum size of a domain cannot exceed 10 GB and the 109

attributes.
- Item name length: the name of an item should not occupy more than 1024 bytes.
- Number of (attribute, value) pairs in an item: this cannot exceed 256. As a consequence, if an item has only one attribute, that attribute cannot have more than 256 associated values.
- Length of an attribute name or value: this cannot exceed 1024 bytes.

DynamoDB is the sucessor of SimpleDB that resulted from combining the best parts of the original Dynamo [DHJ+07] design (incremental scalability, predictable high performance) with the best parts of SimpleDB (ease of administration of a cloud service, consistency, and a table-based data model that is richer than a pure key-value store).

The main operations of the DynamoDB API are the following (the respective delete/update operations are also available):
- `ListTables()` retrieves all the tables associated to one AWS account in a specific AWS Region.
- `createTable(T, Key(pk, rk?))` creates a new table `T` having a primary key `pk` and a range key `rk`.
- `PutItem(T, Key(hk, [rk]), (a,v)+)` creates a new item in the table `T` containing a set of attributes `(a,v)+` and having a key composed by a hash key `hk` and range key `rk`, or replaces it if it already existed. Specifying the range key is optional.
- `BatchWriteItem(item+)` puts and/or deletes up to 25 `Items` in a single request, thus obtaining better performance.
- `GetItem(T, Key(hk, [rk]), (a)*)` returns the item having the key `Key(hk, [rk])` in table `T`. Again, specifying the range key is optional. It is possible to retrieve only a subset of the attributes associated to an item by specifying their names `(a)*` in the request.

DynamoDB was designed to provide seamless scalability and fast, predictable performance. It runs on solid state disks (SSDs) for low-latency response times, and there is no limit on the request capacity or storage size for a given table. This is because Amazon DynamoDB automatically partitions the input data and workload over a sufficient number of servers, to meet the provided requirements. In contrast with its predecessor (SimpleDB), DynamoDB does not automatically build indexes on item attributes leading to more efficient insert, delete, and update operations, improving also the scalability of the system. Indexes can still be created if requested.

2.2.3 MapReduce

The MapReduce programming model have its roots way back in the '60s when the first functional programming languages like LISP [McC60] made their appearance. At the core of these languages, there are the functions *map* and *reduce* and most operations are expressed using these primitives. Inspired by the simplicity of functional languages, Google revisits the MapReduce model proposing a framework [DG04] for processing and generating large data sets. Nowadays, MapReduce is tightly connected with Google's proposal.

MapReduce emerged as an attempt to easily parallelize tasks in a large cluster of commodity computers without deep knowledge of parallel and distributed systems. The user writes a MapReduce program using the map and reduce primitive operations, and the framework is responsible for parallelizing the program, alleviating the user from tasks like resource allocation, synchronization, fault-tolerance, etc.

A MapReduce program is defined by jobs, each of which consists of three main phases:

- A *map* phase, where the input is divided into sub-inputs and each one is handled by a different map task (this procedure can also be customized if needed). The map task takes as an input key/value pairs, process them (by applying the operations defined by the user), and outputs again key/value pairs.
- A *shuffle* phase, where the key/value pairs emitted by the mappers are grouped and sorted by key, and then they are assigned to reducers.
- A *reduce* phase, where each reduce task receives key/value pairs (sharing the same key) and applies further user-defined operations writing the results into the file system.

In order to store the results from MapReduce operations usually a distributed file system (e.g., GFS, HDFS, S3) is used. The results can also be stored in the local file system but this is not very common.

Many recent massively parallel data management systems leverage MapReduce in order to build scalable query processors for both relational [LOÖW14] and RDF [KM15] data. The most popular open-source implementation of MapReduce is provided by the Apache's Hadoop [had11] framework. It has been used by many RDF data management platforms [RS10, HMM$^+$11, RKA11, KRA11, SPL11, HAR11, PKTK12, PKT$^+$13, LL13, PTK$^+$14, WZY$^+$15] and it is the framework on top of which we implemented CliqueSquare (see Chapter 4).

Following the success of the original MapReduce proposal, other systems and models have been proposed, which try to extend the MapReduce paradigm by eliminating some shortcomings of the original model and extending its expressive power. Among the most well known frameworks are Stratosphere [ABE$^+$14] and Apache Spark [ZCF$^+$10]. Both support the primitive map and reduce functions while extending the API with more operations. They are built to support HDFS and they are equiped with their own execution engines. The techniques and algorithms developed in this thesis can be easily adapted to use such richer platforms. On the contrary exploiting the additional primitives provided by these systems goes beyond the scope of this work and has been already explored in the context of XML and XQuery [CCMN15, CCM15].

2.2.4 MapReduce on relational data

The most important optimization algorithms that have been proposed for MapReduce and relational data are surveyed in [LOÖW14]. It is worth noticing that bushy plans are gaining popularity against left-deep in MapReduce frameworks, since applications can afford bigger optimization times. Additionally in [LOÖW14], there is a synopsis of the available join implementations that have been proposed for performing relational-style joins using the MapReduce framework. From them we can distinguish the replicated [AU10] join and its specialization replicated star-join that allow performing n-ary joins in one MapReduce job. The replicated join allows to join multiple relations in different columns while replicated star-join, as its name implies, joins multiple relations on the same column. A cost-based optimizer considering the general repartition join has been proposed in [WLMO11], but the size of the search-space leads to the exploitation of n-ary joins only at the first level of the plan while the subsequent levels use binary joins.

From a more theoretical perspective, [BKS13, AJR+14] study the problem of building n-ary join plans in MapReduce-like frameworks, with a focus on reducing the communication cost and the number of rounds (jobs); the authors provide concrete algorithms with formal cost guarantees expressed in terms of rounds and communication cost. In [AJR+14] they build upon the well-known query evaluation algorithm of Yannakakis [Yan81]. Yannakaki's algorithm receives as input a width-1 Generalized Hypertree Decomposition (GHD) of an acyclic query with n atoms and executes a sequence of $\Theta(n)$ *semi-joins* and *joins* within a time-bound polynomial in the combined size of the input and the output; *input* is the total size of all relations involved in the query and *output* is the size of the result. In [AJR+14] they propose different variations of Yannakaki's algorithm for evaluating conjunctive queries in the context of MapReduce starting from a given GHD. Additionaly, they provide algorithms for transforming the input GHD by reducing its depth and increasing its width considering the trade-offs between the number of MapReduce rounds (related with the depth of the GHD) and the communication cost (related with the width of the GHD). Furthermore, they describe an algorithm for building a minimum depth GHD of width-1 from an acyclic query. This has similarities with our optimization algorithm described in Chapter 4, however we build *minimum depth query plans* whereas a GHD does not correspond to a plan. Depending on the algorithm that receives as input the GHD, one or many plans may be produced based on it. The problem of generating a min-depth plan from a min-depth GHD is beyond the scope of [AJR+14], and of the present thesis.

Optimization algorithms for MapReduce and RDF data are discussed in Section 2.3.

2.2.5 Higher level languages for MapReduce

The arrival of the MapReduce framework improved greatly the development of massively parallel applications. Nevertheless, its API does not permit complex operations making the development of big programs a challenging task for intermediate programmers. To overcome the limitations of expressing everything using the primitive *map* and *reduce* functions, higher level languages were developed, providing a declarative way of writing programs that are then translated automatically to MapReduce jobs. Below, we briefly present two of the most popular languages that are used for writing MapReduce programs, namely Pig Latin and HiveQL.

Pig Latin [ORS+08] is a language for the analysis of very large datasets developed by Yahoo! Research. It is based on the well-known Apache Hadoop Framework, an open source implementation of Google's MapReduce. The implementation of Pig Latin for Hadoop, Pig, is an Apache top-level project that automatically translates a Pig Latin program into a series of MapReduce jobs. The data model of Pig Latin provides four different types:

- *atom* which contains a simple atomic value like a string or number;
- *tuple* which is a sequence of atoms of any type;
- *bag* which is a collection of tuples with possible duplicates;
- *map* which is a collection of data items where each item can be looked up by an associated key.

A Pig Latin program consists of a sequence of instructions where each instruction performs a single data transformation. PigLatin supports the following operators:

LOAD deserializes the input data and maps it to the data model of Pig Latin;

FOREACH can be used to apply some processing on every tuple of a bag;

FILTER allows removing unwanted tuples of a bag;

JOIN performs an equi- or outer- join between bags. It can also be applied to more than two bags at once (multi join);

UNION operator that can be used to combine two or more bags;

SPLIT partitions a bag into two or more bags that do not have to be distinct or complete.

Pig embodies a two-phase rule-based optimizer [GDN13]. In the first phase, traditional RDBMS techniques are used, where selections and projections are pushed down while cartesian products are dragged up to avoid creating large number of intermediate results early in the query plan. The optimizer does not take into account statistics (they are not available in general) and the choice of the join method is entirely left up to the user. In the second phase, it performs optimizations mostly associated to the MapReduce framework. It tries to identify and solve skew problems on the keys emitted from mappers to reducers, while another optimization focuses on reducing the number of jobs by grouping jobs performing aggregations on common inputs.

HiveQL is a part of the Hive [TSJ+09, TSJ+10] data warehousing solution built on top of Hadoop. It is an SQL-like declarative language that is compiled into MapReduce jobs, which are executed using Apache's Hadoop. Originally, it was developed by Facebook, while currently is being used by other companies such as Amazon and Netflix. HiveQL supports many of the most famous features of SQL such as subqueries, multiple types of equi-joins (inner, outer, and semi-joins), cartesian products, groupings and aggregations, as well as data definition statements for creating tables with specific serialization formats, partitioning, and bucketing columns. Hive's data model uses the classical relational tables, where the columns can be primitive (integers, floats, strings) or complex types (associative arrays, lists, and structs). It keeps metadata (schema and statistics) inside an RDBMS and not in HDFS, for fast access.

Hive is coupled with a simple transformation-based logical optimizer that applies selection and projection push-down, partition pruning, and simple join re-ordering based on the size of the relations. Furthermore, it provides a hint mechanism, where specific optimizations are explicitly given inside the query by the user. Most notably, it is possible to provide hints about performing map-joins. Recently, many hint-based optimizations can be applied automatically by the optimizer. Finally, the logical plan is translated into physical operators that are executed using Hive's engine and MapReduce.

2.3 RDF data management

The prevalence of the semantic web and the increasing number of RDF datasets attracted the interest of the data management community leading to a wide variety of approaches for the efficient management of RDF data. As was the case in the relational database setting, there are many challenges to be addressed in order to build an efficient RDF system. The main challenges include, but are not limited to: (i) choosing a good storage model and creating effective indexes; (ii) integrating smart partitioning techniques in the storage model, if the dataset is very large and a distributed architecture must be used; (iii) selecting the proper storage infrastructure; (iv) identifying an efficient query processing model, (v) developing smart optimization algorithms and (vi) handling the semantics of RDF datasets.

A system providing appropriate answers to the above challenges has yet to emerge. Current projects in this area have one primary research focus and possibly address some other secondary ones. Several surveys of RDF data management have appeared. The older ones [BG03, MKA$^+$02] mostly account the manners in which an RDBMS can be used for storing RDF data, providing also performance comparisons and insights of RDF systems [Lee04] at that time. The approach consisting of storing and manipulating RDF data using RDBMS has continued, as witnessed by subsequent surveys [SA09] and performance evaluations [SGD$^+$09]. As the volumes of semantic web data being considered kept increasing, novel indexing techniques were used, and different approaches, no longer relying on RDBMSs, have been studied [FCB12]. In particular, there has been extensive work on building distributed RDF data management systems using P2P infrastructures [FBHB11], while the past few years researchers turned into RDF data management in the cloud [LPF$^+$12, KM15].

2.3.1 Characterisation criteria

As shown in the multiple surveys mentioned above, RDF data management platforms can be examined from multiple perspectives, and many different classification criteria can be used. In the rest of this section, we present the criteria that were used to characterise the state-of-the-art in this thesis. In an attempt to provide a useful overview in which each individual approach can be placed, we purposely avoid discussing implementation details, and emphasize the general ideas instead.

Storage organization In the first place, we are interested in the way the data is organised inside the system. The latter means identifying the general *abstract schema* used for storing the data, whether the data is stored in tables or in graphs, in memory or in a particular disk-based platform etc. Ideally, we would like to describe such a schema relying only on the elements of a triple which are stored in each individual data structure. Integral part of the schema are, of course, the various indexes and the data partitioning techniques used by the platform.

In order to characterize the storage of each system concisely, we introduce the *storage description grammar* as follows.

Definition 2.3.1 (Storage Description Grammar)**.** *The storage description grammar is a tuple* (NS, TS, PR, S_0) *where*

- NS={S_0, EXP, P, ATR} *is the set of non-terminal symbols;*
- TS={"{", "}", "(", ")", "HP", "GP", "LP", "S", "P", "O", "U", "T", "C", "G", "$\overrightarrow{S}$", "$\overrightarrow{P}$", "$\overrightarrow{O}$", "$\overrightarrow{U}$", "$\overrightarrow{T}$", "$\overrightarrow{C}$", "$\overrightarrow{G}$", "*", STR} *is the set of terminal symbols;*
- PR *is the following set of production rules:*

S_0	$\rightarrow$	EXP ("{" EXP "}")?
EXP	$\rightarrow$	ATR+ | P "(" ATR+ ")" | STR | S_0
P	$\rightarrow$	"HP" | "GP" | "LP"
ATR	$\rightarrow$	"S" | "P" | "O" | "U" | "T" |"C" | "G"
	$\rightarrow$	"$\overrightarrow{S}$" | "$\overrightarrow{P}$" | "$\overrightarrow{O}$" | "$\overrightarrow{U}$" | "$\overrightarrow{T}$" | "$\overrightarrow{C}$" | "$\overrightarrow{G}$" | "*"

- STR *is a terminal symbol representing any string of finite length comprised from the characters "$\underline{A}$-$\underline{Z}$" and digits "$\underline{0}$-$\underline{9}$" or "ϵ" for the empty string;*
- S_0 *is the starting symbol.*

Definition 2.3.2 (Storage Description)**.** *A storage description is an expression accepted from the storage description grammar.*

The ATR non-terminal serves to denote a set of RDF data set elements stored in a specific storage structure. We use S to denote the subject values, P for the property values, O for the object values, U for resources (any URI value appearing in S, P or O), T for terms (any URIs or constants appearing in S, P or O), C for the classes of the RDF dataset, and G for the names of the graphs. A combination of several ATR symbols denotes the fact that the data structure holds the respective set of RDF data elements; for instance, either OS or SO denote a structure that stores (subject, object) tuples (in this case, pairs). Further, symbols annotated with an arrow denote an ordering of the respective data in the storage structure. Thus, $\overrightarrow{S}$ O and O $\overrightarrow{S}$ both denote (subject, object) pairs sorted by subject. If we have a combination of ATR symbols where there are multiple annotations with arrows then *the order of symbols in the combination denotes the order according to which data is sorted in the respective storage structure*. For example, $\overrightarrow{O}\,\overrightarrow{S}$ denotes (subject, object) pairs sorted first by the object then by the subject, whereas $\overrightarrow{S}\,\overrightarrow{O}$ denotes a set of such pairs sorted first by the subject and then by the object. Equivalently, we may also write $\overrightarrow{OS}$ and $\overrightarrow{SO}$ to describe the same storage structure. The asterisk (*) symbol is used as a shorthand for the complete graph (instead of writing SPO for subject, property, object, or SPOG for subject, property, object, graph), when the order of the attributes is not important.

The STR symbol represents string constants and it is used for denoting names of the storage structures (e.g., table, collection etc.) holding data. For example, $\underline{T}${SPO} represents a data structure named $\underline{T}$ and holding all triples of the dataset.

For a storage description to be complete, the partitioning of the data has to be specified. The existing works rely on three types of partitioning techniques:

- Hash Partitioning (**HP**), where the triples are classified in partitions using the value of a hash function on some of their attributes;

- List Partitioning (**LP**), where each partition is associated with a set of values, and all triples associated with these values are part of this partition (most of the times the property values are used to define the partitions);
- Graph Partitioning (**GP**), where the RDF graph is considered as a whole and graph partitioning techniques and tools (e.g., METIS [KK98]) are used to determined the triples belonging to each partition.

In order to model the partitioning, we use the P symbol followed by a sequence of RDF elements (ATR+). For example, HP(SO) denotes hash partitioning using the subject and object of a triple, while LP(P) denotes a list partitioning using the values of properties.

Furthermore, an expression of the form A{B} defines a map structure obtained from AB by grouping using the values of A. The brackets denote a nesting of information. For instance, the expression S{P{O}} means that triples are first grouped by subject; then, within each group, we split the triples again by the values of their properties, and for each such property we store the set of object values. The resulting data organization can be seen as a multiple-level map, or a two-level index.

The simplest meaningful storage description expression is SPO, which denotes a data structure storing all the (full) triples.

Storage infrastructure A second important aspect of an RDF store is the storage infrastructure being used. To store a given data collection such as those described by our grammar above, existing systems rely on RDBMS tables, flat files, main-memory structures, collections in a key-value store, other native RDF DBMS etc. These choices are compounded by the many different concrete systems implementing each of these paradigms, e.g., one can use either S3 or HDFS as a distributed file system, similarly either use Oracle or Postgres or DB2 etc. Since the concrete choice of a system does not change significantly the storage characterization we make here, below we organize the discussion around a set of classes of systems which may be used to hold an RDF data storage structure: DBMS, file system, memory, and key-value store.

Query optimization Third, we are interested in the kind of query optimization technique supported by the system. From the perspective of query optimization, one of the main concern is the shape of the query plan, which amounts to the space of possible query evaluation choices for a given query.

- Linear plans with binary joins (**LB**)
- Linear plans with n-ary joins, where each logical join operator may have 2 or more inputs (**LN**)
- Bushy plans with binary joins (**BB**)
- Bushy plans with n-ary joins only at the first level[1] (**BN1**)

1. The BN1 option is less standard than the other ones, yet it does occur in the recent RDF data management literature. It turns out that in a wide-scale distributed systems, the first-level joins, applied directly on the data extracted found in the distributed the store, enjoy some physical execution strategies capable of joining N inputs with a single operator. Thus, some systems consider (logical or physical) n-ary join operators only at this first level, and resort to more traditional binary joins for the subsequent operations in the logical plan. Our classification includes this option.

– Bushy plans with n-ary joins (**BN**).

Finally, the query processing paradigm can also vary significantly. We attempt to roughly group the existing approaches by considering the following processing models:

– Centralized execution using custom operators (**CECO**), where the query processing is done entirely on a single machine;
– Centralized execution using DBMS engine (**CEDB**) where the query processing is delegated completely on a single site database;
– Distributed execution using the MapReduce programming model (**DEMR**);
– Distributed execution using a parallel DBMS (**DEDB**), and
– Distributed execution using other (custom) operators (**DECO**).

In the sequel, we classify state-of-the-art systems according to their primary research focus, and then provide a detailed description of each, based on the criteria and using the grammar introduced in this section.

2.3.2 RDF storage & indexing

The earliest approaches for storing RDF using an RDBMS are described in [Lil00, Mel00]. The contributors advocate the use of the triple (SPO) and quad (GSPO) table for storing RDF data while simple dictionaries are proposed for reducing the space occupancy. Furthermore, they outline a schema where resources and literals are stored in separate tables and the triple table contains only the pointers to these tables. These ideas have been the core of RDF many data management systems. In the following, we will see how these ideas evolved over the years to create efficient systems able to handle billions of triples.

Redland [Bec01] is one of the earliest RDF data management systems. It was designed as a modular framework with a flexible API promoting the development of new applications relying on the RDF data model. Redland allows supports queries of exactly one triple pattern, thus no joins, and accordingly it does not have a query optimization algorithm. Redland uses three indexes to store the data, relying on hash table structures that reside either in main memory or in a persistent storage provided by the BerkeleyDB library [Ora94]. The first hash table uses as keys (subject, property) pairs, with the object value serving as value; the second hash table contains property-object keys and the subjects as values; finally, the third hash table has subject-object keys and the properties as values.

Storage description	Infrastructure	Plan shape	Processing
SP{O} + PO{S} + SO{P}	DBMS | memory	-	CECO

RDFSuite [ACK$^+$00, ACK$^+$01] is among the first works that advocated the use of database technology to support RDF data management. A formal data model for RDF data bases is proposed and the design of a persistent RDF store over PostgreSQL is presented. The proposed approach stores separately the RDF Schema information from the RDF instance information.

Four main tables are used for storing the RDF Schema information, namely Class, Property, SubClass, and SubProperty. The Class table holds all the different class resources that appear in the dataset while the Property table holds all the different properties with their domain and range. This work assumes a set of *restrictions* on the RDF graph: the domain and range of every property is exactly one RDF class, in other words there is exactly one RDFS constraint stating the domain, and one stating the range, of every property occurring in the RDF graph. As stated when presenting the RDF data model, these constraints do not hold in general. For this restricted setting, two tables named SubClass and SubProperty hold the rdfs:subClassOf and rdfs:subPropertyOf relationships.

At the instance level, there are two types of tables. For each class occurring in the dataset, a table named after that class holds all the resources that belong to this class (one-attribute table). For each property of the dataset, a table named after that property holds the source and the target connected if that property (two-attribute table). Appropriate indexes are built for every table and almost every attribute independently. The data is stored persistently in PostgreSQL; the query language supported is RQL [KAC+02]. The query evaluation engine relies on custom operators and on PostgreSQL. The optimizer of the system aims to push as much of the query evaluation as possible inside the DBMS so that it can take advantage of the DBMS optimizer.

Storage description	Infrastructure	Plan shape	Processing

$\underline{\text{CLS}}\{C\}$ + $\underline{\text{PTY}}\{P\{CC\}\}$
$\underline{\text{SCLS}}\{C\{C\}\}$ + $\underline{\text{SPTY}}\{P\{P\}\}$
LP(C){U}
LP(P){S{O}} + LP(P){O{S}}
	DBMS	-	CECO + CEDB

Sesame [BKvH02] is an RDF storage and querying platform advocating an extensible architecture. Sesame highlights the importance of having an intermediate *Repository Abstraction Layer* (RAL, in short) between the physical storage and the other modules (query module, data import/export module, etc.) of the system, in order for the query processor to remain independent of the particular storage. All RDF-specific methods are implemented inside this layer and form the API with which the other modules communicate. The code for translating the RDF methods to actual calls of the physical store is located inside this layer.

As in [ACK+01], Sesame's storage schema uses class and property tables. For each class from the RDF dataset, a single-attribute table stores the typed resources that correspond to it. Similarly, each property defines a two-attribute table that holds the source and the target node of the property (i.e., the table is named after the property value of a triple, and holds all subjects and objects corresponding to triples with the same property). The data are stored persistently in the underlying store, which may in principle be anything (due to RAL) from a RDBMS to flat files. In [BKvH02], the storage engine used was PostgreSQL. Queries are expressed in RQL [KAC+02] and are handled by the query module. The system has a query optimizer, however no details are provided about its functioning.

Storage description	Infrastructure	Plan shape	Processing
LP(C){U} + LP(P){SO}	DBMS | memory	-	CECO + CEDB

Jena2 [WSKR03] is the second generation of Jena [McB02], a semantic web toolkit specifically intended to support large datasets. The storage schema in Jena2 trades off space for time. It uses a triple table where resource URIs and simple literal values are stored directly in the table. Separate tables for URIs (resp., literals) are used only for resources (resp., literals) with long values (corresponding to very complex URIs or to long literals, e.g., paragraphs of text). Different RDF graphs are stored in different triple tables. In addition, Jena provides a general facility, called *property table*, to hold together triples commonly accessed together. Single-valued properties are part of a big table where subject values are stored in the first attribute and the rest of the attributes store property-object pairs (the attribute name is the property, while the attribute value holds the object value) associated with a given subject. Multi-valued properties are stored in separate tables having two attributes, storing the subject-object pairs related by a particular property. Finally, Jena2 proposes the use of a *property class* table. This holds all instances of a specified class, and also stores all the properties that have this class as domain as per the RDF Schema. The table has one attribute for the subject resources of the class, and zero or more attributes for the associated property values (the attribute name is the property, while the attribute value holds the object value). Jena2 provides a database driver interface and a generic implementation for the most common SQL databases like PostgreSQL, MySQL and Oracle. To access property class tables and for RDQL [Sea04] queries, the drivers dynamically generate SQL select statements.

Storage description	Infrastructure	Plan shape	Processing

LP(G){SPO} + S{PO}
L(P){SO} + L(C){S{PO}}
	DBMS | memory	-	CEDB

3store [HG03] stores triples from an RDF graph in a single table; the table has an attribute holding the id of the graph to which each triple belongs. 3store uses RDQL [Sea04] for expressing RDF queries. To evaluate them, the system relies on the underlying DBMS system (MySQL) by translating queries into SQL.

Storage description	Infrastructure	Plan shape	Processing
GSPO	DBMS	-	CEDB

Hexastore [WKB08] emerged as an attempt to address the limitations (creation of a large number of tables, poor performance for queries with unbound properties) of property-based partitioning approaches (see systems with LP(P) in their storage description). RDF triples are indexed in six possible ways, one for each permutation of the three RDF elements. Conceptually each index can be thought as a nested three-level structure one should access based on the first index attribute, then on the second, in order to get the value of the third; all the levels are sorted. Hexastore provides efficient single triple pattern lookups, and also allows fast merge-joins for any pair of triple patterns. The main disadvantage of Hexastore is the big storage requirement. To limit the amount of storage needed for the URIs, Hexastore uses *dictionary encoding* (whereas each resource is replaced with an integer ID), often applied to compress large URIs and literals. Update and insertion operations affect all six indices, hence can be slow. The prototype implementation relied in main-memory to store the indexes. The focus of this work was mostly to make the data storage efficient; the system is not complete, in particular no optimization algorithm is discussed.

Storage description	Infrastructure	Plan shape	Processing
S{P{O}} + S{O{P}} + P{S{O}}	memory	-	CECO
P{O{S}} + O{S{P}} + O{P{S}}			

Virtuoso Cluster Edition [Erl08] is a commercial system for RDF data management originally created for supporting relational data. Virtuoso stores RDF data into a single relational table with additional indexes; it also stores the graph from which the triple originates. Virtuoso delegates to the user the decision of creating the proper indexes. By default it builds an index using the concatenation of the graph id, subject, predicate, and object of a triple. This storage scheme is suggested for dealing with queries where the graph id is known. The recommended index when the graph id is not known is a concatenation of the subject, property, object and graph id. Virtuoso extensively uses bitmap indexes for improving performance and storage efficiency at the same time with other compression techniques. Its RDF query evaluation closely resembles the one of SQL, bar certain differences for what concerns the selectivity estimations. The cluster edition employs a hash partitioning scheme to partition the index to the available nodes. Intra- and inter-operator parallelism are inherited from its relational edition.

Storage description	Infrastructure	Plan shape	Processing
HP(GSPO){GSPO}	file system	LB | BB	CEDB

Clustered TDB [OSG08] is the clustered version of Jena TDB [2]. Clustered TDB builds three B+ tree indexes using the concatenation of subject, property, and object values. The system follows a shared-nothing approach: it applies a hash partitioning on the three indexes to split the data into the available nodes. Specifically, the SPO index is hash-partitioned by the subject, the POS by the property, and the OSP by the object. A relatively simple skew mechanism handles the values that appear too often in the dataset by combining the partition attribute with another attribute. The authors focus on building a clustered distributed RDF store rather than a complete data management system thus the current prototype does not include a query optimizer.

Storage description	Infrastructure	Plan shape	Processing
$HP(S)\{\overrightarrow{SPO}\} + HP(P)\{\overrightarrow{POS}\} + HP(O)\{\overrightarrow{OSP}\}$	file system	-	DECO

2. https://jena.apache.org/documentation/tdb/

Stratustore [SZ10] is the first work that exploits commercial cloud based services for RDF data management. In particular Stratustore uses Jena for parsing RDF, while it processes queries relying on Amazon's SimpleDB as the storage back-end. Stratustore builds two indexes using the key-value store, the first using the subject and property as a key and the object as a value, and the second using the subject and object and the property as a value. As is the case of many NoSQL databases, SimpleDB does not support complex query processing, thus SPARQL queries must be evaluated outside the SimpleDB system. Once the SPARQL query is parsed from Jena, the graph patterns are retrieved and grouped by Stratustore, based on the selected indexes. Then the grouped triple patterns are transformed into simple SELECT queries and are sent to SimpleDB. If the results need further processing (filters, joins, etc.) this is done in a single machine using Jena-provided operators.

Storage description	Infrastructure	Plan shape	Processing
S{P{O}} + S{O{P}}	key-value store	-	DEDB + CECO

CummulusRDF [LH11] is an RDF storage infrastructure relying on Apache's Cassandra key-value store. CummulusRDF uses three indexes: SPO, POS and OSP, based on which it is able to answer all single triple pattern queries. Two different implementation approaches were used, namely *Hierarchical* and *Flat*.

The hierarchical layout uses the row key, a super column key, and a column key to store the triples, while the Flat layout uses only the row key and column key. The Flat layout requires an extra index in order to support efficiently queries with triple patterns whose subject and object are unbound. CummulusRDF is not a complete system, since it only supports single triple pattern queries.

Storage description	Infrastructure	Plan shape	Processing
Hierarchical strategy			
S{P{O}} + P{O{S}} + O{S{P}}	key-value store	-	DEDB
Flat strategy			
S{PO} + PO{S} + O{SP} + P{PO}	key-value store	-	DEDB

2.3.3 RDF query optimization

Query optimization in relational databases is a well studied research area having its roots [SAC+79] back in the '70s. Early RDF data management systems relied entirely on relational optimizers, by storing and querying directly through a RDBMS). However, the graph structure of RDF and other specific features of the data model lead to optimization techniques being developed explicitly for the RDF model. Below, we present the main proposals in this area. To give context to each algorithm, we briefly introduce the general architecture of the system for which it was proposed, before explaining the optimization technique.

Atlas [KMM$^+$06] is a P2P system built on top of the Bamboo [RGRK04] Distributed Hash Table (DHT) implementation. Atlas incorporates various interesting strategies for query processing and optimization common in many P2P systems relying on DHTs. These strategies, namely QC, SBV, and QC*, were developed inside Atlas (to improve various aspects of the system) and are discussed below.

When using the QC strategy [LIK06], Atlas stores RDF triples three times, hashing them on the subject, property, and object value, exactly like RDFPeers (to be discussed shortly in Section 2.3.4). To evaluate the query it follows the same procedure with RDF-Peers where the triple patterns are ordered and executed sequentially each on one node, while the intermediate results are transfered from one to the next to evaluate the join. Conceptually, this algorithm amounts to left-deep plans with binary joins and cartesian products.

Following the SBV strategy [LIK06], Atlas stores RDF triples seven times using all the non-empty attribute subsets of a triple. To evaluate queries, the triple patterns are ordered and executed sequentially (as for QC) with the difference that after the evaluation of the first triple pattern, SBV pushes the bindings thus obtained into the next triple pattern. Different tuples of bindings lead to multiple instantiations of remaining sub-query to evaluate, that can be evaluated concurrently on multiple nodes exploiting the available indexes. The generated query plans are left-deep binary join trees; the binding-passing operator resembles binary index nested loop joins.

Following the QC* [KKK10] strategy, Atlas re-uses the QC storage and additionally provides a mapping dictionary from URIs and literals to unique integers. This strategy is explored in conjunction with three optimization algorithms: (i) the naive algorithm; (ii) the semi-naive algorithm; (iii) the dynamic algorithm. All of them rely on heuristics and build left-deep plans with binary joins. Query evaluation proceeds in the same way as QC. The naive algorithm orders the triple patterns by selectivity, but in contrast with QC, it does not allow cartesian products. The semi-naive algorithm builds a join graph where each edge represents a join between two triple patterns, then repeatedly picks the edge with the lowest cost (avoiding cartesian products, and focusing on identifying the cheapest join), and adds the nodes to the plan. The dynamic algorithm is an extension of the semi-naive, where after each join, the selectivity is re-estimated taking into account the actual results from the performed joins leading to more accurate estimations.

Storage description	Infrastructure	Plan shape	Processing
QC,QC* strategies			

H(S){SPO} + H(P){SPO}
H(O){SPO}
	DBMS	LB	CEDB + DECO

SBV strategy			

H(S){SPO} + H(P){SPO}
H(O){SPO} + H(SP){SPO}
H(SO){SPO} + H(PO){SPO}
H(SPO){SPO}
	DBMS	LB	CEDB + DECO

RDF-3X [NW10] is one of the best-known RDF data management systems. It employs exhaustive indexing (similar with Hexastore) over a table of triples materializing all possible permutations of subject-property-object values. This translates to a replication factor of six of the original dataset, that the authors overcome by applying smart compression techniques. Additionally, to avoid the problem of expensive self-joins they rely on compressed clustered B+ tree indexes (where the triples are sorted lexicographically) and efficient merge-join operations. The system relies on a Dynamic Programming algorithm that builds bushy plans with binary joins using state-of-the-art methods [NM11] for the cost estimation of joins.

Storage description	Infrastructure	Plan shape	Processing
$\overrightarrow{\text{SPO}}$ + $\overrightarrow{\text{SOP}}$ + $\overrightarrow{\text{PSO}}$ $\overrightarrow{\text{POS}}$ + $\overrightarrow{\text{OSP}}$ + $\overrightarrow{\text{OPS}}$	file system	BB	CECO

HadoopRDF [HKKT10, HMM$^+$11] relies on Hadoop for storing and querying the data, relying on property-based partitioning. The storage procedure is divided into two steps. First, the triples are grouped into files based on the value of the property; then, the file containing the triples with property value equal with rdf:type is further split based on the object value (that is, the exact type). A query is evaluated using a sequence of MapReduce jobs. HadoopRDF tries to minimize the number of jobs by aggressively performing as many joins as possible in each job. The optimization algorithm produces a single bushy plan, whose leaves are scan operators based on triple patterns. For a triple pattern whose property value is known, only a single file is read from HDFS. On the contrary, when the property is unbound, all the files residing in HDFS have to be read. In the sequel of the plan, n-ary joins may be involved.

Storage description	Infrastructure	Plan shape	Processing
LP(P){SO}	file system	BN	DEMR

Rapid+ [RKA11, KRA11] proposes an intermediate nested algebra to increase the degree of parallelism when evaluating joins, and thus reduce the number of MapReduce jobs. This is achieved by treating star joins (groups of triple patterns having as subject the same variable) as groups of triples and defining new operators on these triple groups. Specialized physical operators were developed to back-up the proposed algebra and they were integrated inside Pig, allowing the translation of logical plans (expressed using the proposed algebra) to MapReduce programs. Queries with k star-shaped subqueries are translated into a MapReduce program with k MapReduce jobs: 1 job for evaluating all star-join subqueries, and $k-1$ jobs for joining the subquery results thus obtained. Conceptually, Rapid+ builds bushy plans with n-ary joins at the first level since the intermediary results of the star-join subqueries are evaluated pair-wise and sequentially.

In [KRA12], the authors extend RAPID+ with a scan sharing technique applied in the reduce phase, to optimize queries where some property value appears more than once. Surprisingly, they do not pre-process the data. They assume that the triples are stored in flat files inside the HDFS using the typical subject, property, object representation.

However, when loading the data to process it through their physical operators, they use a special data structure, name RDFMap. RDFMap is an extension of Java's standard HashMap, providing (among other optimizations) an index-based access by the property value.

Storage description	Infrastructure	Plan shape	Processing
SPO	file system	BN1	DEMR

HSP [TSF+12] focuses on the problem of query optimization in the complete absence of statistics. The system is built on top of MonetDB [IGN+12] and applies the exhaustive indexing scheme used in RDF-3X. The optimization algorithm exploits the syntactic and structural variations of SPARQL triple patterns to produce a logical query plan without a cost model. The algorithm tries to detect the merge joins that can be performed by solving a maximum weight independent set problem. Finally, the logical plan is translated to a physical plan, executed by MonetDB. HSP produces bushy query plans with binary joins.

Storage description	Infrastructure	Plan shape	Processing
$\overrightarrow{SPO} + \overrightarrow{SOP} + \overrightarrow{PSO}$ $\overrightarrow{POS} + \overrightarrow{OSP} + \overrightarrow{OPS}$	DBMS	BB	CEDB

RDF-3X+CP [GN14] is an extension of the RDF-3X system with a new optimization procedure. In an attempt to overcome the cost of exploring the complete search space (that may be huge for RDF queries involving many self-joins) using a Dynamic Programming algorithm, new heuristic techniques are introduced. The original DP algorithm of RDF-3X is replaced by a new one that first decomposes the query into chain and star subqueries, and then it performs the DP algorithm separately on these subqueries. The system also uses a novel cost-estimation function that relies on the novel notion of *characteristic pairs* for providing more accurate cardinality estimations for the results of the subqueries.

Storage description	Infrastructure	Plan shape	Processing
$\overrightarrow{SPO} + \overrightarrow{SOP} + \overrightarrow{PSO}$ $\overrightarrow{POS} + \overrightarrow{OSP} + \overrightarrow{OPS}$	file system	BB	CECO

gStore [ZMC+11, ZÖC+14] utilizes disk-based adjacency lists in order to store the RDF graph. The adjacency list can be also seen as an index where for every subject we can retrieve the corresponding property-object values. gStore performs efficiently exact and wild-card SPARQL queries, as well as aggregate queries by exploiting materialized views. It compresses the data by encoding RDF entities and RDF classes using bit strings, creating a data signature graph. Further, it uses two auxiliary indexes: VS*-tree, and T-index. VS*-tree is used to prune out parts of the graph not relevant to the query, while T-index is used to speed up aggregate queries. The query is encoded using the same compression scheme as the data, and then it is asked against the VS*-tree to find candidate matches. Each match of the encoded query corresponding to a subgraph match of the original query is added to the results.

Storage description	Infrastructure	Plan shape	Processing
S{PO} + VS*-tree + T-index	file system	-	CECO

2.3.4 RDF query processing

This section presents RDF data management systems having invested significant effort in RDF query processing based on various infrastructures.

RDFPeers [CFYM04, CF04] is the first P2P system utilizing a Distributed Hash Table (DHT) to build an RDF repository. The system extends the MAAN [CFCS03] P2P implementation by incorporating RDF-specific storage and retrieval techniques. RDF triples are split over the available nodes hashing them by subject, property, and object, applying a more fine-grained partitioning strategy for properties that appear very often in the dataset. Inheriting the primitive operations that were designed for MAAN, it allows expressing single triple pattern queries, conjunctive queries with range filters, and disjunctive queries with range filters.

To evaluate single triple pattern queries, if there is some constant in the query then it is used to root the query to the proper node and return the results to the node that posed the query; if there are no constants in the query, the query is send to all nodes for evaluation. Disjunctive queries can be answered in a similar fashion, with an extra union operator performed in the end. To evaluate a larger conjunctive query, triple patterns are evaluated sequentially, each on the node having triples that match it; the intermediate results are forwarded from one node to another where where they are joined etc. Conceptually, this evaluation strategy resembles a left-deep plan with binary joins and possibly cartesian products. Although conjunctive and disjunctive queries can in principle be mixed, the authors do not provide a generic query planning algorithm for handling all of them.

Storage description	Infrastructure	Plan shape	Processing
H(S){SPO} + H(P){SPO} + H(O){SPO}	DBMS	LB	CEDB + DECO

Oracle [CDES05] proposes an approach for storing and querying RDF data within its RDBMS storage and processing engine.

RDF triples are stored in two tables: IdTriples ($\underline{\text{IT}}$) and UriMap. UriMap holds all resources and literals with their mappings to integer IDs, while IdTriples holds the triples (subjects, properties, objects using their integer mapping) plus the id of their originating RDF graph. In addition, all possible two-way joins between triple patterns are materialized, leading to six additional tables (which can be seen as materialized views denoted with $\underline{\text{V1}}$ to $\underline{\text{V6}}$). Finally, a property-based index ($\underline{\text{PI}}$) is defined for efficiently accessing the IdTriples table. To further improve performance, other data structures are materialized, namely subject-property matrices ($\underline{\text{SPM}}$), which are a slight variation of the property tables used in Jena.

At the core of Oracle's query processing capability is an SQL table function called RDF_MATCH [CDES05][3]. The main argument of the RDF_MATCH function is the BGP query, expressed using the SPARQL syntax. The interest of this approach is that RDF data can be queried jointly with relational data under a common language. To evaluate the query, they extend Oracle's interface by specifying rules for translating the content of RDF_MATCH function (the BGP query expressed in SPARQL) into an SQL query that is merged with the remaining SQL query (outside the RDF_MATCH function). Then, the SQL query resulting from the translation is sent to Oracle's query execution module.

Storage description	Infrastructure	Plan shape	Processing

IT{GSPO} + PI{P{GOS}}
V1{SS{POPO}} + V2{PP{SOSO}}
V3{OO{SPSP}} + V4{SP{POSO}}
V5{SO{POSP}} + V6{PO{SOSP}}
SPM{LP(P){SO}}
	DBMS	-	CEDB

4store [HLS09] is cluster-based RDF data management platform that operates on top of commodity machines adopting a shared nothing architecture. The cluster is divided into one or many *storage nodes* that hold the actual data and exactly one *processing (master) node* which is responsible for retrieving the data matching the triple patterns of the query from the storage nodes (using the so-called *bind function*) and evaluating locally the query using the fetched data. 4store partitions the data among the storage nodes according to the value of the subject. Then, at each node, two indexes (radix-tries) are built for every distinct property, the first using the subject, and the second using the object. The indexes store quads. In addition, at each node, there is a hash-based index whose key is the RDF graph ID, while the value is the set of triples of the graph. The optimization algorithm is based on heuristics; selections and projections are pushed down (inside the bind function) etc.J̇oin ordering is applied in a greedy fashion (using selectivity estimates) by executing the most selective join first, and then the remaining ones. The resulting plan is bushy; only binary joins are considered.

Storage description	Infrastructure	Plan shape	Processing

HP(S){LP(P){S{OG}}}
HP(S){LP(P){O{SG}}}
HP(S){G{SPO}}
	file system	BB	DECO

SHARD [RS10] is among the first systems that used Hadoop and HDFS to store and query RDF data. In SHARD, triples having the same subject are grouped in one line in a data file. Thus, for each subject, one can retrieve all the corresponding property-object pairs, which amounts to a simple form of file-resident subject index, although it is never exploited by the system. Query evaluation is done sequentially by processing one triple pattern at a time; one MapReduce job is used each to join the triples matching each triple

3. RDF_MATCH has been renamed to SEM_MATCH (`https://docs.oracle.com/database/121/RDFRM/sdo_rdf_concepts.htm#RDFRM592`)

pattern, with the previously created intermediate results. Conceptually, SHARD's query evaluation strategy leads to left deep query plans with binary joins, which correspond to a sequence of MapReduce jobs, and to potentially long query evaluation time.

Storage description	Infrastructure	Plan shape	Processing
S{PO}	file system	LB	DEMR

DB2RDF [BDK+13] insists on the RDF data management using RDBMS providing a new storage schema over relational tables and a syntactic optimizer for the efficient translation of SPARQL queries to SQL. It creates one table with as many columns as the underlying system tolerates storing in the first column triple subjects and in the rest of the columns properties and objects pairs associated with a given subject. Thus, one row of the table is of the form $[s_1|p_1|o_1|p_2|o_2|\ldots|p_n|o_n]$ where p_i and o_i are properties and objects having s_i as the subject. All the values of a same property (say, all the values of the p_1 property) in the dataset are always stored in the same column (say, the second). This resembles the property tables of Jena (Section 2.3.2) with the difference that in the same column of the table we may have multiple properties assigned (e.g., in the same table we may have a tuple of the form $[s_2|p_k|o_k|p_2|o_2'|\ldots|p_n|o_n']$) as long as the properties do not have the same resource (s_2 is not associated with the property p_1). To assign properties into columns, the authors propose two methods: (i) the first based on a graph coloring problem; (ii) and the second using some form of hashing. Similar to Jena if there are multi-valued properties then additional tables have to be built. The layout above almost completely eliminates subject-subject joins. A dual table is used for objects to eliminate also object-object joins.

Since a naive translation of SPARQL to SQL may lead to bad performance, the authors propose an algorithm for optimizing the translation of SPARQL to SQL. They consider SPARQL queries with joins, unions, and optional. The algorithm parses the query into a custom representation called dataflow graph where each operator (node) is assigned with a cost. The graph defines a space of flow trees, each of which uses some of the edges of the dataflow graph, and computes the query result. Since the problem of finding the minimum-cost flow tree is NP-hard they propose a heuristic algorithm. In the end the chosen flow tree is translated to SQL (this is one-to-one, i.e., there are no more choices at this point) and is executed by the RDBMS.

Storage description	Infrastructure	Plan shape	Processing
S{PO} + O{PS} + L(P){SO}	DBMS	-	CEDB

H$_2$RDF+ [PKTK12, PKT+13, PTK+14] is a recent RDF data management system using HBase and Hadoop's MapReduce processing framework. To store the data, it adopts the exhaustive indexing scheme of RDF-3X, building indexes on all permutations of subject, property, and object of a triple. To deal with the demanding size of the indexes, it employs compression techniques tailored around the Hadoop framework. The indexes are stored in HBase using only the key part of the index; HBase indexes are automatically sorted in key order.

For what concerns query processing, H_2RDF provides efficient MapReduce-based merge-join operators which take advantage of the existing indexes. The system is bundled with a greedy cost-based optimization algorithm that tries to perform as many joins as possible in each job. The joins are performed sequentially one after the other leading to left deep plans with n-ary joins (more than one triple pattern may join at the same job). H_2RDF+ has a dynamic optimization algorithm, which decides if the next join will be performed in a centralized or distributed fashion (using MapReduce) based on the estimated cost. If the join has to process large inputs, H_2RDF leverages the benefits of parallel execution through MapReduce, otherwise it performs the join in a single machine avoiding the overhead of starting a Hadoop job.

Storage description	Infrastructure	Plan shape	Processing
$\overrightarrow{\text{SPO}}$ + $\overrightarrow{\text{SOP}}$ + $\overrightarrow{\text{PSO}}$ $\overrightarrow{\text{POS}}$ + $\overrightarrow{\text{OSP}}$ + $\overrightarrow{\text{OPS}}$	key-value store	LN	CECO + DEMR

PigSPARQL [SPL11] is the first system to provide a full comprehensive translation from SPARQL 1.0 into the PigLatin (see Section 2.2.5) parallel data processing language. It allows any query expressed in SPARQL 1.0 to be executed using MapReduce jobs. To store the data, the system uses the property partitioning approach where triples are grouped in HDFS files based on the value of the property. The query optimization algorithm relies on heuristics. The join order is established based on the syntactic form of the triple patterns (triples patterns with more constants are considered more selective); selection and projections are pushed down. If the order features a sequence of joins on the same variable, an n-ary join is performed. Although not many details about the optimization algorithm are available, the optimization procedure described above reveals a left deep plan with n-ary joins.

Storage description	Infrastructure	Plan shape	Processing
LP(P){SO}	file system	LN	DEMR

Rya [PCR12] is an RDF data management platform built on top of the Accumulo key-value store. Rya is among the first systems that exploit entirely the row key of the key-value store to create clustered indexes (similar with those used in RDF-3X). Rya takes advantage of the key sorting provided by an Accumulo collection to store completely the subject, property, and object (concatenating the values) of the triple inside the key. Three indexes are built, using different permutations of subject, property, and object, so that all single-triple pattern queries can be answered efficiently. The query processing is done centralized in one machine using custom operators. Rya gathers and stores some simple statistics regarding the cardinality of the constants that appear in the dataset, to be able to order the triple patterns based on their estimated selectivity. From the available information, it appears left-deep plans with binary joins are used to evaluate the query. The optimizer uses index nested loop joins when there is an index on the join attribute. Rya also makes use of inter-operator parallelism to execute in parallel (threads) some joins.

Storage description	Infrastructure	Plan shape	Processing
$\overrightarrow{SPO}$ + $\overrightarrow{POS}$ + $\overrightarrow{OSP}$	key-value store	LB	CECO

Trinity.RDF [ZYW$^+$13] is the first system adopting a graph-oriented approach for RDF data management using cloud infrastructures. It stores the data as adjacency lists using Microsoft's Trinity [SWL13]. Trinity can be seen as distributed key-value store, supported by a memory storage module and a message passing framework. The basic storage scheme uses the node ID as a key and the matching edges (incoming and outgoing) as values. Conceptually, the scheme allows for each subject to find all property-object pairs (node to outgoing edges) and for each object to find all property-subject pairs (node to incoming edges). In that sense, it is similar with other approaches that use key-value stores and two indexes (S{PO}, O{PS}). Depending on the shape of the RDF graph, it may use another storage scheme that co-locates the adjacency lists of a graph node to the same machine.

To efficiently locate subject and objects, Trinity.RDF builds two additional indexes from properties to subjects and from properties to objects (in [ZYW$^+$13] this is the global index). Query processing in Trinity.RDF follows a graph exploration approach with custom operators. An optimizer derives the execution plan by ordering the triple patterns using a cost model that considers statistics, join estimations and communication cost. The plan is executed in parallel at each node (the nodes exchange messages to retrieve relevant triples) and then the results from all nodes are gathered to a master node, where a final join removes false positives. The optimization approach strongly resembles relational query optimization, considering left-deep binary joins that can choose between two access patterns for the leaf operators. The graph exploration approach comes very close to a relational plan that uses index nested loop joins and the suitable indexes for each join.

Storage description	Infrastructure	Plan shape	Processing
S{PO} + O{PS} + P{S} + P{O}	key-value store	LB	DECO

TriAD [GSMT14] is a cluster-based RDF data management system that combines and extends various techniques that lead to efficient RDF query processing. Similar with H-RDF-3X, it employs a graph partition strategy using the METIS graph partitioner [KK98] to divide the original RDF dataset into partitions. Based on the METIS result, they also build a summary graph used to restrict query processing to only its interesting part before evaluating the query. Every triple is replicated to two partitions using the subject and the object. Then, for every subject partition, it builds the indexes SPO, SOP and PSO, and for every object partition it builds the indexes OSP, OPS and POS. The indexes are sorted in lexicographic order and are stored in main memory. TriAD holds statistics for every possible combination of S, P and O from the data graph and the summary graph.

The query optimization algorithm is divided into two phases: (i) the first phase uses an exploratory algorithm the processes the query against the summary graph to identify the relevant partitions; (ii) the second phase uses a dynamic programming join ordering algorithm like the one of RDF-3X (bushy plans with binary joins) with an extended cost function to account for the distribution of the data, the gathered statistics, and the relevant partition information from the first phase. Finally, to perform joins, it leverages multi-threaded and distributed execution, based on an asynchronous MPI protocol.

Storage description	Infrastructure	Plan shape	Processing

GP(*){$\overrightarrow{\text{SPO}}$} + GP(*){$\overrightarrow{\text{SOP}}$} + GP(*){$\overrightarrow{\text{PSO}}$}
GP(*){$\overrightarrow{\text{POS}}$} + GP(*){$\overrightarrow{\text{OSP}}$} + GP(*){$\overrightarrow{\text{OPS}}$}
	memory	BB	DECO

2.3.5 RDF partitioning

Even in centralized architectures, partitioning the RDF graph in multiple tables or files can greatly improve performance for certain types of queries by avoiding overly large results (e.g., a join output too large for the available memory). The impact of partitioning is naturally even more visible in distributed systems, where the graph is distributed to multiple machines which can be exploited for parallelism; in exchange, one has to account also for data transfers across the network. In this section, we present approaches for distributed RDF graphs in a way that maximizes parallelism while attempting to reduce the network communications entailed by the evaluation of a query.

H-RDF-3X [HAR11] aims to scale up RDF data processing by partitioning the RDF dataset based on its graph structure, and storing the resulting partitions in a set of centralized RDF stores. Graph partitioning is again delegated to METIS [KK98]. Each partition is defined by a set of vertices, and triples are assigned to a partition set if their subject is among the nodes which METIS allocated to this set. This allows the execution of some type of queries locally at each processing node (each of which evaluates it on its locally stored data).

To enlarge the set of queries that can be thus parallelized processed in parallel, one can store next to the nodes in a partition set, all the nodes reachable by a path having one end node in the partition set, and whose path length is bounded by a certain constant k. Each partition set is then stored in an RDF-3X engine residing on the respective processing node.

The optimization algorithm decomposes the query into subqueries, each of which can be executed locally at each node without network communication. Finally, it joins the intermediary results from the subqueries using MapReduce jobs. The query plans produced for joining the results of the subqueries are left deep with binary joins.

Storage description	Infrastructure	Plan shape	Processing

GP(*){SPO} + GP(*){SOP}
GP(*){PSO} + GP(*){POS}
GP(*){OSP} + GP(*){OPS}
	DBMS	LB	CEDB+DEMR

SHAPE [LL13] partitions an RDF graph using a hash-based approach that considers also the semantics of the RDF graph. It tries to combine the effectiveness of the graph partitioning with the efficiency of the hashing to create partitions that allow many queries to run locally at each node. It starts by creating the *baseline partitions* by hashing the triples using their subject (resp. object). Then the baseline partitions are extended by: (i) hashing the objects (resp. subjects) of the triples that belong to this partition; (ii) hashing the subjects (resp. objects) of all the triples of the RDF graph; (iii) comparing the hash values, and associating to the baseline partition the new triples that hash to the same value. This procedure is known as *1-forward* (respectively, *1-reverse*) *partitioning*. The partitions can be extended further, following the k-hop notion introduced in H-RDF-3X, leading to the k*-forward*, k*-reverse*, and k*-bidirectional* partitioning approaches. The bidirectional partitioning expands the baseline partition using the hash of the subject and the hash of the object at the same time. The partitions are stored in centralized RDF stores.

Query evaluation and in particular query optimization are very similar to those of H-RDF-3X. The optimizer decomposes a queries into subqueries that can be evaluated locally, and combines their results through left deep plans with binary joins, executed using MapReduce.

Storage description	Infrastructure	Plan shape	Processing

HP(*){SPO} + HP(*){SOP}
HP(*){PSO} + HP(*){POS}
HP(*){OSP} + HP(*){OPS}
	DBMS	LB	CEBD+DEMR

PP-RDF-3X [WZY$^+$15] is the state-of-the-art on the partitioning of large RDF graphs. The authors suggest dividing the RDF graph into disjoint partitions using *end-to-end paths* as the smallest partition element (a single end-to-end path must reside completely inside a partition). Thus, every partition can be seen as a set of end-to-end paths. An end-to-end path can be roughly characterized as a *maximum-length path from a source node to a sink node*[4]. Setting up the store, then, requires finding all end-to-end paths and dividing them into disjoint sets. Since two or more end-to-end paths might contain common subpaths (common triples), the authors develop a cost model for finding the best trade-off when partitioning the graph, between replicating nodes (which improves performance but occupies more space) and disjointness. The problem is NP-Hard, thus they propose an approximate algorithm for solving it. Similar to SHAPE and H-RDF-3X, they store the derived partitions in local RDF-3X stores. Queries are then decomposed into queries which can be locally evaluated. The query plans combining subquery results are left-deep with binary joins; they make use of MapReduce jobs to produce the final result.

Storage description	Infrastructure	Plan shape	Processing

GP(*){SPO} + GP(*){SOP}
GP(*){PSO} + GP(*){POS}
GP(*){OSP} + GP(*){OPS}
	DBMS	LB	CEBD+DEMR

4. The exact definition is a bit more involved, allowing for directed cycles to occur in specific places with respect to the end-to-end path, in particular at the end; a sample end-to-end path is $u_1, u_2, u_3, u_4, u_5, u_3$, for some u_i URIs in the RDF graph, $1 \leq i \leq 5$.

2.3.6 Summary

We summarize here the main RDF data management ideas and concepts which we have analyzed and classified above.

Indexing Building and maintaining specialized indexes for providing fast access to RDF data started with the first RDF data management systems [Bec01, ACK$^+$01, BKvH02, McB02]. Among the most significant indexing proposals, the six-index approach where all the permutations of subject, property, and object, are concatenated sorted and stored in disk, memory, key-value store, or DBMS [NW10, TSF$^+$12, GN14, PTK$^+$14]. Of course there are approaches that considered fewer indexes but still on the idea of keeping the attributes sorted [PKTK12, PCR12]. The first system that followed this storage strategy is RDF-3X [NW10] although the idea of exhaustive indexing on the different permutations of SPO is bit older and known under the name Hexastore [WKB08]. Hexastore is based also on the idea of exhaustively indexing of the permutations and sorting them, but in addition it nests (groups) attributes within the index. This nested structure fits perfectly in key-value stores [SZ10, LH11], and thus some of the some of the six indexes have been implemented in such systems.

Partitioning Beyond indexing, partitioning is another way of allowing selective access to RDF data. In its simplest form it can be spotted early on in centralized systems [ACK$^+$01, BKvH02] where the subjects and objects were grouped into different structures (tables) based on the distinct property values that appear in the RDF dataset. This property partitioning approach became known as *vertical partitioning* [AMMH07]; it corresponds to the LP(P) prefix presented in the previous sections. As RDF datasets grow, partitioning becomes indispensable for large scale distributed systems. Vertical partitioning still remains among the most popular partitioning methods appearing in recent distributed systems [HLS09, HMM$^+$11, SPL11]. Another popular partitioning technique distributes the triples (tables, indexes, files, etc.) into disjoint partitions by applying a hash function on one (or several) of the attributes [CF04, LIK06, OSG08, Erl08, HLS09, KKK10]. Recently, more elaborated approaches were used, based on the structure of the RDF graph [HAR11, LL13, GSMT14, WZY$^+$15], aiming to maximize parallel processing and reduce the data transfers they require.

Adapting to the workload All the approaches discussed so far proposed a fixed solution that does not consider knowledge of the query workload. More recently, research has focused on exploiting information about the workload to structure the storage and indexes. In [CL10] a set of navigational paths appearing in the given workload is selected to be materialized, while in [DCDK11] focuses on recommending RDF indices. A methodology for organizing the storage of an RDF graph as a set of workload-inspired materialized views is presented in [GKLM10, GKLM11]. More recently, a distributed system [GHS12, HS13] exploited the query workload to decide how to partition the data in order to extract the best performance. While the above works recommend a storage model off-line (before actually running queries), a more recent vision [AÖD14] proposes an RDF storage model that can learn from the queries as they run, and adapt the storage online (in-between query evaluations); these ideas are developed in the Chameleondb prototype [AÖDH13, AÖD15, AÖDH15].

Infrastructure The size of the semantic web data volumes processed by early applications was sufficiently small to fit in memory [Bec01, McB02, BKvH02]. Subsequently, data storage was delegated to an RDBMS, which allowed manipulating bigger volumes of data also taking advantage of mature optimization techniques of RDBMS [ACK$^+$01, HG03, CDES05]. The limitations encountered by these RDBMS-based approaches have lead to the design of novel architectures specifically built from the grounds up, for RDF data [NW10, ZMC$^+$11, ZÖC$^+$14, AÖDH15]. In parallel, instead of relying only on centralized machines to store RDF data, systems were built around the concepts of P2P networks [NWQ$^+$02, CF04, MPK06, LIK06, KKK10], machine clusters [Erl08, OSG08, HLS09], distributed file systems [RS10, HMM$^+$11, RKA11, KRA11, SPL11], and key-value stores [SZ10, LH11, PCR12, ZYW$^+$13, PTK$^+$14]. With the price of main memory dropping as the technology advances, some recent distributed systems [ZYW$^+$13, GSMT14] opted for this direction. Last but not least, there are distributed systems relying on a federation of single-site triple stores [HAR11, LL13, WZY$^+$15].

Query optimization Early systems relied on RDBMS' optimizers [ACK$^+$01, BKvH02, WSKR03, HG03, CDES05]. These are typically based on dynamic programming (DP) using binary joins and various heuristics. DP algorithms without heuristics have been proposed also in native RDF systems like [NW10, GSMT14]. For large queries, DP has to be combined with (or replaced by) heuristics [TSF$^+$12, GN14] in order to scale. Surprisingly, a lot of distributed systems [OSG08, HLS09, KKK10, RS10, SPL11] use naive optimization algorithms combined with simple database techniques like selection-projection push-down producing plans; this cannot guarantee optimality. A line of works [LIK06, PCR12, ZYW$^+$13, ZÖC$^+$14] also use (index) nested loop joins for evaluation, thus the optimization algorithm, apart from join ordering, has to identify the proper access patterns when indexes are available. Systems focusing on partitioning [HAR11, LL13, WZY$^+$15] neglect the optimization part providing some very basic algorithms. All of the aforementioned approaches build left deep or bushy plans with binary joins. With the proliferation of MapReduce and distributed systems, providing more opportunities for parallel evaluations, bushy plans with n-ary joins can boost performance. In [RKA11, KRA11] the authors build bushy plans with n-ary joins at the first level while in [HMM$^+$11] a greedy optimization algorithm and heuristics are used in order to build a single bushy plans with n-ary joins all over. Some early systems do not address optimization at all [Bec01, WKB08, LH11, SZ10].

Query processing Centralized systems like [Bec01, WKB08, NW10, GN14, ZMC$^+$11, ZÖC$^+$14] rely entirely on custom operators to process the queries while others [WSKR03, HG03, CDES05, Erl08] simply translate the RDF query into a relational one and delegate optimization and execution to the relational server. Between the two categories there are approaches [ACK$^+$01, BKvH02, TSF$^+$12] based on custom operators while also pushing part of the query is inside the RDBMS. Other systems [SZ10, LH11, PCR12] use distribution stores while the processing is performed in a centralized machine. Among distributed systems, we can distinguish those [OSG08, HLS09, ZYW$^+$13, GSMT14] using their own operators and implementing the whole communication protocol from scratch, those [NWQ$^+$02, CF04, MPK06, LIK06, KKK10] that use and extend P2P implementations, and those exploiting the strengths of MapReduce to support the processing. The MapReduce based systems can be further divided into those [RS10, HMM$^+$11, RKA11, KRA11, SPL11] relying exclusively on the framework while others [HAR11, LL13, PTK$^+$14, WZY$^+$15] may perform some operations in a centralized environment.

2.4 Conclusion

In this chapter we presented RDF, the standard data model for representing Semantic Web resources, and its query language SPARQL. We recalled briefly the main features of distributed storage platforms, including distributed file systems and popular key-value stores. We described the general functionality of the MapReduce framework and outlined some higher level languages for exploiting its benefits using a declarative syntax. Finally, an extended overview of the existing RDF data management systems was presented and analyzed from the scope of storage, indexing, query optimization, and infrastructure.

Chapter 3

AMADA: RDF data management in the Cloud

In this chapter, we present AMADA, an architecture for RDF data management based on public cloud infrastructures. AMADA follows the Software as a Service (SaaS) model: the complete platform is running in the cloud and appropriate APIs are provided to the end users for storing and retrieving RDF data. Public cloud infrastructures are typically rented for a monetary cost. This chapter explores various RDF storage and querying strategies, revealing their strengths and weaknesses from a perspective seeking to strike a balance between performance and monetary costs.

The architecture of AMADA has been initially studied before my involvement in the project: in particular, an XML data management platform based on the same Amazon cloud services has been considered in [CCM12, CCM13], while RDF was the focus of [BGKM12, ABC$^+$12]. I joined the AMADA project subsequently. My work has revisited and extended the ideas in [BGKM12], to examine *more RDF storage strategies and processing techniques*. In addition, we *changed the underlying key-value store* (moving from SimpleDB to DynamoDB), and added to the system a novel *query optimizer*, which was lacking in [BGKM12, ABC$^+$12]. Specifically, for some storage and indexing strategies, I proposed a *cost-based optimizer*, while dictionary compression has been investigated for all strategies. Finally, we performed a larger set of experiments (350$\times$ more RDF triples) using real datasets that revealed interesting results, contradicting the conclusions drawn in [BGKM12] based on smaller datasets.

This chapter documents these extension. The material presented here follows closely the respective book chapter [BCG$^+$14]. The AMADA system has been open-sourced in March 2013 [1].

3.1 Introduction

The rapid growth of RDF data spawned the movement from centralized RDF data management systems to distributed ones. In addition, we have seen that relying on cloud

1. `http://cloak.saclay.inria.fr/research/amada/`

computing for building such systems is a very promising direction due to the scalability, fault-tolerance and elasticity features it offers. Nevertheless, building such a system requires addressing a set of challenges, outlined in Chapter 1. In this Chapter, our focus is on the design of a cloud-based architecture and how indexing techniques can improve performance.

Many recent works have focused on the performance versus monetary cost analysis of cloud platforms, and on the extension of the services that they provide. For instance, [BFG$^+$08] focuses on extending public cloud services with basic database primitives, while MapReduce extensions are proposed in [AEH$^+$11] for efficient parallel processing of queries in cloud infrastructures. In the Semantic Web community there has been also a movement to cloud-based RDF data management [KM15]. However, there are hardly any systems entirely designed around public cloud infrastructures, making it difficult to assess the suitability of commercial clouds for RDF data management.

RDF data management systems built using commercial clouds are Stratustore [SZ10] and Dydra [Dat11]. Stratustore explores only a single strategy for storing and querying RDF data in Amazon's SimpleDB while the cost factor of the cloud is not examined at all. Dydra is also developed on top of Amazon Web Services, but there are no available details regarding the system.

The focus of this chapter is on an architecture for storing RDF data within the Amazon cloud that provides efficient query performance, both in terms of time and monetary costs. We consider hosting RDF data in the cloud, and querying through a (distributed, parallel) platform also running in the cloud. Such an architecture belongs to the general Software as a Service (SaaS) setting where the whole stack from the hardware to the data management layer are hosted and rented from the cloud. At the core of our proposed architecture reside *RDF indexing strategies* that allow to direct queries to a (hopefully tight) subset of the RDF dataset which provide answers to a given query, thus reducing the total work entailed by query execution. This is crucial as, in a cloud environment, the total consumption of storage and computing resources translates into monetary costs.

This chapter is organized as follows. First, we briefly introduce the different parts of the Amazon Web Services (AWS) on which we build our platform, in Section 3.2. Then, we discuss the architecture of our system and the interactions between various AWS components in Section 3.3. In Section 3.4, we present our specific indexing and query answering strategies providing also some implementation details regarding the dictionary encoding technique that was used. Experiments validating the interest of our techniques are presented in Section 3.5. Finally, Section 3.6 summarizes this chapter.

3.2 Amazon Web Services

In AMADA, we store RDF graphs (RDF files) in *Amazon Simple Storage Service* (*S3*) and use Amazon *DynamoDB* for storing the indexes. SPARQL queries are evaluated against the RDF files retrieved from S3, within the *Amazon Elastic Compute Cloud* (*EC2*) machines and the communication among these components is done through the *Simple Queue Service* (*SQS*).

In the following we describe the services used by our architecture. We also introduce the parameters used by AWS for calculating the pricing of each of its services; the actual figures are shown in Table 3.1 (the notations in the table are explained in the following subsections). More details about AWS pricing can be found in [aws].

$ST^{\$}_{m,GB} = \$0.125$	$IDX^{\$}_{m,GB} = \$1.13$
$STput^{\$} = \$0.000011$	$IDXput^{\$} = \$0.00000032$
$STget^{\$} = \$0.0000011$	$IDXget^{\$} = \$0.000000032$
$VM^{\$}_{h,l} = \$0.38$	$QS^{\$} = \$0.000001$
$VM^{\$}_{h,xl} = \$0.76$	$egress^{\$}_{GB} = \$0.12$

Table 3.1: AWS Ireland costs as of February 2013.

3.2.1 Simple Storage Service

Amazon Simple Storage Service (S3) is a storage web service for raw data and hence, ideal for storing large objects or files. S3 stores the data in named buckets. Each object stored in a bucket has an associated unique name (key) within that bucket, some metadata, an access control policy for AWS users and a version ID. The number of objects that can be stored within a bucket is unlimited.

To retrieve an object from S3, the bucket containing it should be accessed, and within bucket the object can be retrieved by its name. S3 allows to access the metadata associated to an object without retrieving the complete entity. Storing objects in one or multiple S3 buckets has no impact on the storage performance.

Pricing. Each read file operation costs $STget^{\$}$, while each write operation costs $STput^{\$}$. Further, $ST^{\$}_{m,GB}$ is the cost charged for storing 1 GB of data in S3 for one month. AWS does not charge anything for data transferred to or within their cloud infrastructure. However, data transferred out of the cloud incurs a cost: $egress^{\$}_{GB}$ is the price charged for transferring 1 GB.

3.2.2 DynamoDB

Amazon DynamoDB is a key-value store that provides fast access to small objects, ensuring high availability and scalability for the data stored; we have presented it in Section 2.2.2. Here we report briefly some of the main feature that it provides. A DynamoDB database is organized in *tables*. Each table is a collection of *items* identified by a primary composite key. Each item contains one or more *attributes*; in turn, an attribute has a *name* and a set of associated *values*[2]. DynamoDB provides a very simple API to execute read and write operations. For the sake of readability in the rest of the chapter we will refer to those operations simply as `GetItem(T,K)` and `PutItem(T,(K,V))` where `T` is the table name, `K` is the key, and `V` is the value.

Pricing. Each item read and write API request has a fixed price, $IDXget^{\$}$ and $IDXput^{\$}$ respectively. One can adjust the number of API requests that a table can process per

2. An item can have any number of attributes, although there is a limit of 64 KB on the item size.

second. Further, DynamoDB charges $IDX^{\$}_{m,GB}$ for storing 1 GB of data in the index store during one month.

3.2.3 Elastic Compute Cloud

Amazon Elastic Compute Cloud (EC2) provides virtual machines, called *instances*, which users can rent to run their applications on. A developer can store in AWS the image or static data containing the software that an instance should run once it is started. Then, it can launch instances e.g. *large*, *extra-large*, etc that have different hardware characteristics, such as CPU speed, RAM size etc.

Pricing. The EC2 utilization cost depends on the kind of virtual machines used. In our system, we use large (l) and extra-large (xl) instances. Thus, $VM^{\$}_{h,l}$ is the price charged for using a large instance for one hour, while $VM^{\$}_{h,xl}$ is the price charged for using an extra-large instance for one hour.

3.2.4 Simple Queue Service

Amazon Simple Queue Service (SQS) provides reliable and scalable queues that enable asynchronous message-based communication between the distributed components of an application. This service prevents application message loss, even when some AWS components may be unavailable for some duration.

Pricing. $QS^{\$}$ is the price charged for any request to the queue service API, including send message, receive message, delete message, renew lease etc.

3.3 Architecture

In a setting where large amounts of RDF data reside in an elastic cloud-based store, and focus on the task of efficiently routing queries to only those graphs that are likely to have matches for the query. Selective query routing reduces the total work associated to processing a query, and in a cloud environment, total work also translates in financial costs. To achieve this, whenever data is uploaded in the cloud store, we index it and store the index in an efficient (cloud-resident) store for small key-value pairs. Thus, we take advantage of: *large-scale stores for the data itself; elastic computing capabilities to evaluate queries; and the fine-grained search capabilities of a fast key-value store, for efficient query routing*.

AMADA stores RDF graphs in S3, and each graph is treated as an uninterpreted BLOB[3] object. As explained in Section 3.2.1, one needs to associate a key to every resource stored in S3 in order to be able to retrieve it. Thus, we assign to each graph: (i) an URI consisting of the bucket name denoting the place where it is saved; (ii) the name of the graph. The combination of both (i) and (ii) uniquely describes he graph. Then the

3. A Binary Large OBject (BLOB) is a collection of binary data stored as a single entity in a database management system

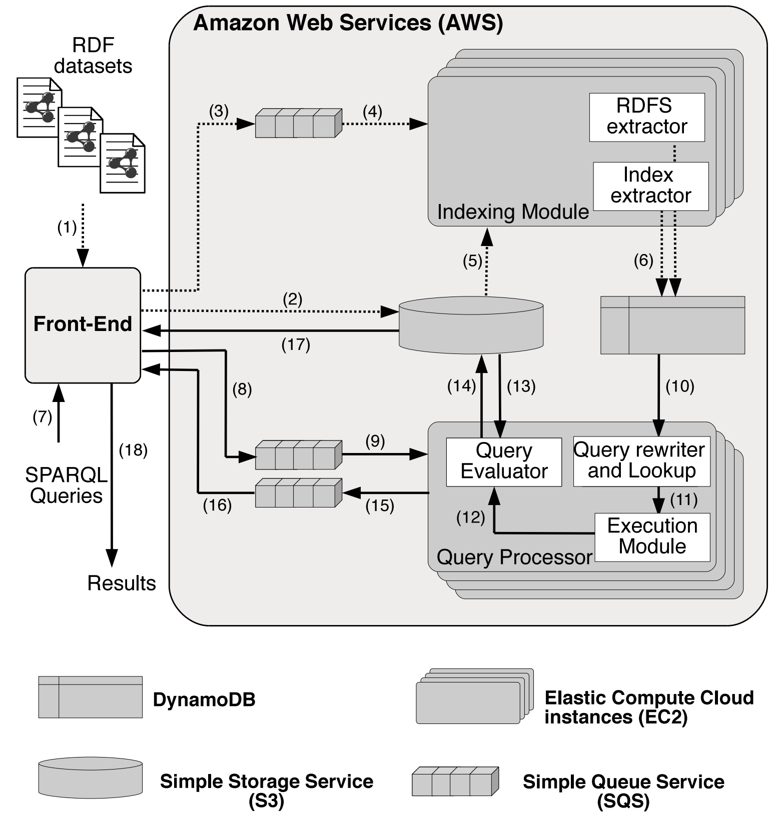


Figure 3.1: AMADA architecture based on AWS components.

triples comprising the RDF graphs are used to create indexes returning the URIs of relevant (with respect to a user query) graphs. Finally, we store our indexes in DynamoDB, as it provides fast retrieval for fine-granularity objects.

An overview of our system architecture is depicted in Figure 3.1. A user interaction with our system can be described as follows.

The user submits to the *front-end* component an RDF graph (**1**) and the front-end module stores the file in S3 (**2**). The front-end then creates a message containing the reference to the graph and inserts it to the *loader request queue* (**3**). Any EC2 instance running our *indexing module* receives such a message (**4**) and retrieves the graph from S3 (**5**). The indexing module, after transforming the graph into a set of RDF triples, creates the index entries and inserts them in DynamoDB (**6**).

When a user submits a SPARQL query to the front-end (**7**), the front-end inserts the corresponding message into the *query request queue* (**8**). Any EC2 instance running our *query processor* receives such a message and parses the query (**9**). Then, the query processor performs a lookup in the index stored in DynamoDB (**10**). Depending on the indexing strategy, the lookup will return data that can be used to answer the query directly (without scanning any data stored in S3), or data that can be used to find out which graphs contain information to answer the query. Any processing required on the data retrieved from DynamoDB is performed by the *execution module* (**11**). If a final results extraction step is required, the *local query evaluator* receives the final list of URIs pointing to the RDF graphs in S3 (**12**), retrieves them and evaluates the SPARQL query against these graphs (13). The results of the query are written to S3 (**14**) and a message is created and inserted into the *query response queue* (**15**). Finally, the front-end receives this message (**16**), retrieves the results from S3 (**17**) and the query results are returned to the user and deleted from S3 (**18**).

Although we use Amazon Web Services, our architecture could be easily adapted to run on top of other cloud platforms that provide similar services. Examples of such platforms include Windows Azure[4] and Google Cloud[5].

3.4 Indexing strategies

In the following, we introduce the different strategies we have devised to answer RDF queries efficiently within AMADA, both in terms of time and monetary costs.

To illustrate these strategies, we use the example RDF dataset shown in Figure 2.2 comprised from two RDF graphs, one holding information about the professors of the Lehigh university, and one holding information about the students of the same university. Each graph is identified by a name g and a set of triples of the form $(s\ p\ o)$. We will use $\mathcal{G}$ to denote all the RDF graphs that need to be indexed by our system, and for any graph $g \in \mathcal{G}$, we use $|g|$ to represent the total number of triples of this graph.

Furthermore, to illustrate query evaluation for each strategy, we use the example query QA shown in Figure 2.3. We use q to refer to an RDF query, and $|q|$ to denote the number

4. `http://www.windowsazure.com`
5. `https://cloud.google.com/`

of triple patterns existing in q.

As we have seen previously DynamoDB allows storing data using four levels of information `{tablename:{itemkey:{attributename:{attributevalue}}}}`. In order to describe the storage strategies we will use the storage description grammar from Definition 2.3.1.

For comparing the different strategies, we focus on the index size and the number of index look-ups entailed for each query. Thus, for each strategy, we will present analytical models for calculating data storage size and query processing costs in the worst case scenario, which is different for each strategy. In this chapter, we do not consider a complete cost model of AMADA that would include e.g. local processing, data transfer, etc. However, a full formalization of the monetary costs associated to the AMADA architecture can be found in [CCM13] (for XML data; the adaptation to RDF is quite direct).

In the sequel, Section 3.4.1 describes a data organization strategy that loads all the RDF data in DynamoDB, making it possible to answer queries only by looking up in the index. Section 3.4.2 presents indexing strategies that allow to find the RDF graphs that should be retrieved from S3 to answer a given query. Section 3.4.3 provides implementation details regarding the dictionary encoding techniques that were used in this work.

3.4.1 Answering queries from the index

The first strategy we describe relies exclusively on the index to answer queries. This is achieved by inserting the RDF data completely into the index, and answering queries based on the index without requiring accessing the dataset. We denote this strategy by QAS and we describe it in more details below. The same technique has been studied also in other works using key-value stores [SZ10, LH11] but not for DynamoDB, and without considering the associated monetary cost.

Indexing. A DynamoDB table is allocated for the subject, the property, and the object of an RDF triple. We use the subject, predicate, object values of each triple in the graph as the item keys in the respective DynamoDB table, and as attribute (name, value) pairs, the pairs: (predicate, object), (object, subject) and (subject, predicate) of the triple. Thus, each entry in the table completely encodes an RDF triple, and all database triples are encoded in three tables. In terms of our storage description language (Definition 2.3.1, Section 2.3.1) QAS storage is shown below.

Storage description
$\underline{\text{S}}$\{S\{P\{O\}\}\}+$\underline{\text{P}}$\{P\{O\{S\}\}\}+$\underline{\text{O}}$\{O\{S\{P\}\}\}

The organization of this strategy is illustrated in Table 3.2(a) while Table 3.2(b) illustrates it on the triples of our example.

Querying. When querying data indexed according to QAS, one needs to perform index look-ups in order to extract from the index sets of triples that match the triple patterns of the query, and then process these triples through relational operators (selections, projections and joins) which AMADA provides in its execution module of the query processor (Figure 3.1). In our implementation, we have used the relational operators of

(a) QAS indexing strategy

S table	
item key	(attr. name, attr. value)
subject	(predicate, object)
P table	
item key	(attr. name, attr. value)
predicate	(subject, object)
O table	
item key	(attr. name, attr. value)
object	(predicate, subject)

(b) Example of QAS index

S table	
item key	(attr. name, attr. value)
ub:prof1	(ub:name, "bob")
	(ub:advisor, ub:stud1)
	(rdf:type, ub:professor)
ub:prof2	(ub:advisor, ub:stud2)
	(ub:name, "alice")
...	...
P table	
item key	(attr. name, attr. value)
ub:name	(ub:prof1, "bob")
	(ub:prof2, "alice")
	(ub:stud1, "ted")
ub:advisor	(ub:prof1, ub:stud1)
	(ub:prof2, ub:stud2)
...	...
O table	
item key	(attr. name, attr. value)
"bob"	(ub:name, ub:prof1)
"alice"	(ub:name, ub:prof2)
ub:stud1	(ub:advisor, ub:prof1)
ub:stud2	(ub:advisor, ub:prof2)
...	...

Table 3.2: Sample entries using the QAS strategy.

ViP2P [KKMZ11] but any relational query processor supporting these operators could be used.

For each triple pattern appearing in a given RDF query, a `GetItem` DynamoDB call is executed. If the triple pattern has only one bound value, then depending on which element is bound on the triple pattern, the respective index is passed as a parameter to the call. Concretely, if the bound value is the subject, the first parameter of `GetItem` is the S index, if the bound value is the property, the first parameter of `GetItem` is the P index, otherwise (the bound value is the object) the first parameter of `GetItem` is the O index. In the case where two or three values of the triple pattern are bound, we choose the index to be accessed (S, P, or O) based on selectivity estimations. We have in our disposal statistics for the occurrences of a constant (URI or literal) in the dataset, thus we choose the appropriate index using the most selective value. For instance, if the subject and object of the triple pattern are bound to decide which index to use (S or P) we consult the available statistics; when the subject value is more selective we pick S, otherwise we pick P index.

For each triple pattern t_i, the resulting attribute name-value pairs retrieved from DynamoDB form a two-column relation R_i. If the triple pattern has only one bound value, the values of these columns hold bindings for the variables of t_i. Otherwise, if t_i has two bound values, a selection operation is used to filter out the values that do not match the triple pattern. These relations are then joined to compute the answer to the query.

For instance, consider the query of Figure 2.3. First, we define the following DynamoDB requests:

```
r1:  GetItem(P̲, ub:advisor)
r2:  GetItem(O̲, ub:dept4)
```

Request `r1` returns attribute name-value pairs (s_1, o_1) which form a relation R_1, while `r2` returns attribute name-value pairs (s_2, p_2) which form relation R_2. Then, a selection is applied to ensure that the values of the second column of R_2 are equal to the predicate `ub:member` (i.e., $\sigma_{2=\text{ub:member}}(R_2)$). The remaining values of the first column of R_2 are the bindings to the variable of the second triple pattern. Finally, a join is performed between the second column of R_1 and the first column of R_2 and the results of the join form the answer to the query QA, i.e., $R_1 \bowtie_{2=1} \pi_1(\sigma_{2=:member}(R_2))$.

When the query involves more than two triple patterns the order the joins has to be considered. A contribution of this thesis was to extend the AMADA prototype previously developed in the team [BGKM12, ABC$^+$12], where the order of join evaluation was fixed and did not consider data characteristics, with a *heuristic cost-based optimizer*, which exploits a set of simple RDF statistics we gather on the RDF dataset upon loading them into AWS. The principles of the optimizer are inspired from [SSB$^+$08, KKK10] (which we presented in Section 2.3.3). In a nutshell, it builds binary joins in a greedy fashion by repeatedly identifying the cheapest joins among those not already applied. Join costs are estimated using textbook formulas [RG03]. This optimization approach is an adaptation of the minimum selectivity heuristic [SMK97].

Analytical cost model. We now analyze the cost of the QAS indexing strategy as well as the number of required lookups while processing an RDF query.

We assume that the number of distinct subject, predicate and object values appearing in a dataset is equal to the size of the dataset itself, and thus equals to the number of triples (worst case scenario). In this indexing strategy we create three entries to DynamoDB for each triple in our dataset $g \in \mathcal{G}$. Therefore, the size of the index of this strategy is $\sum_{g \in \mathcal{G}} 3 \times |g|$.

To process queries, we perform one lookup for each triple pattern appearing in the query q. Thus, the number of lookups to DynamoDB is $|q|$.

3.4.2 Selective indexing strategies

In this section, we present three strategies for building RDF indexes within DynamoDB (i) the *term-based* strategy, (ii) the *attribute-based* strategy and, (iii) the *attribute-subset* strategy. It is important to notice that these strategies do not store the complete triples in the indexes; instead, they store "pointers" to the RDF graph(s) containing triples that match a specific condition. The system uses these indexes in order to identify among all the RDF graphs, those which may contribute to answer a given query. Then it loads these graphs from the S3 storage into an EC2 instance, merges them into a single graph (Definition 2.1.5), and processes the query there.

The techniques presented here are inspired from the domain of information retrieval. Specifically, the proposed indexes resemble closely the inverted files [Knu73] usually exploited in keyword search.

To the best of our knowledge, the strategies proposed here have not been exploited for RDF data management, prior to the AMADA system, despite being relatively simple.

1. The term-based strategy *had not been previously explored in AMADA either*, prior to my involvement in the work.
2. The attribute-based and attribute-subset strategies *were* exploited in [BGKM12], however here a new design is proposed, exploiting the full capabilities of a key-value store.

With respect to the second item above, the novel design we introduced in [BCG+14] and present below, makes indexing more efficient, in particular coupled with the move from SimpleDB to DynamoDB which was simultaneously performed. Recall from Section 2.2.2 that SimpleDB has hard limits on the size and number of items that can be stored. As a consequence (and due to the old design), in the SimpleDB-based implementation of AMADA, a special overflow mechanism, akin to building additional tables, was used in [BGKM12] to hold values that couldn't be added to the original data structure, for instance, due to the presence of an exceedingly popular (frequent) key. In turn, the presence of these additional tables lead to extra lookups required in order to retrieve the information needed to answer the query. Furthermore, the old design necessitates the existence of secondary indexes in order to retrieve efficiently the relevant RDF graphs for a given subject, property, or object value.

In contrast, using the new design, it is possible to retrieve all necessary information from the key-value store using just one level of `GetItem` operations, which also reduces the overall number of accesses to the key-value store. In addition, RDF graphs can be retrieved efficiently without the need for secondary indexes making the new design space-efficient. Last but not least, the move from SimpleDB to DynamoDB greatly improved look-up times. The differences between the old and the new design are detailed below in the respective section of each strategy.

Term-based strategy

This first indexing strategy, denoted RTS, relies on the RDF terms found within the datasets stored in S3. This strategy does not take into account whether a certain term is found as a subject, predicate or object inside a dataset.

Indexing. For each RDF term (URI or literal) appearing in an RDF graph, one DynamoDB item is created with the value of this term as key, the URI of the dataset that contains this RDF term as attribute name, and an empty string (denoted ϵ) as attribute value. The name of the graph is also used to retrieve the graph stored in S3. In terms of our storage description language (Definition 2.3.1, Section 2.3.1) RTS storage is shown below.

Storage description
$\underline{\mathrm{T}}\{\mathrm{T}\{\mathrm{G}\{\epsilon\}\}\}$

Table 3.3(a) depicts the general layout for this strategy, where v_i are the values of the RDF terms. Table 3.3(b) illustrates the index obtained for the running example.

(a) RTS indexing.

T table	
item key	(attr. name, attr. value)
v_1	$(g_1, \epsilon), (g_2, \epsilon), \ldots$
v_2	$(g_2, \epsilon), \ldots$
v_3	$(g_1, \epsilon), (g_2, \epsilon), \ldots$

(b) Sample RTS index entries.

T table	
item key	(attr. name, attr. value)
ub:prof1	(ub:Professors, ϵ)
ub:prof2	(ub:Professors, ϵ)
ub:Professor	(ub:Professors, ϵ)
ub:name	(ub:Professors, ϵ)
	(ub:Students, ϵ)
ub:takesCourse	(ub:Students, ϵ)
ub:advisor	(ub:Professors, ϵ)
rdf:type	(ub:Professors, ϵ)
	(ub:Students, ϵ)
ub:stud1	(ub:Professors, ϵ)
	(ub:Students, ϵ)
ub:stud2	(ub:Professors, ϵ)
	(ub:Students, ϵ)
ub:member	(ub:Students, ϵ)
ub:dept4	(ub:Students, ϵ)
ub:dept1	(ub:Students, ϵ)
ub:Dept	(ub:Students, ϵ)
ub:db	(ub:Students, ϵ)
ub:os	(ub:Students, ϵ)
"bob"	(ub:Professors, ϵ)
"alice"	(ub:Professors, ϵ)
"ted"	(ub:Students, ϵ)

Table 3.3: Sample index entries for the RTS strategy.

Querying. For each RDF term of an RDF query a `GetItem` look-up in the RTS index retrieves the URIs of the graphs containing a triple with the given term. For each triple pattern, the results of all the `GetItem` look-ups must be intersected, to ensure that all the constants of a triple pattern will be found in the same dataset. The union of all URI sets thus obtained from the triple patterns of a query provides the URIs of the graphs to retrieve from S3 and from the merge of which the query must be answered.

Using our running example, assume that we want to evaluate the query of Figure 2.3. The corresponding DynamoDB queries required in order to retrieve the corresponding graphs are the following:

```
r1:  GetItem(T, ub:advisor)
r2:  GetItem(T, ub:member)
r3:  GetItem(T, ub:dept4)
```

The datasets retrieved from the DynamoDB request `r1` is merged with those obtained by intersecting the results of `r2` and `r3`. The query is be then evaluated on the resulting (merged) graph to get the results.

Analytical cost model. We assume that each RDF term appears only once in a graph (worst case scenario) and thus, the number of RDF terms equals three times the number of triples. For each RDF term in a graph we create 1 entry in DynamoDB. Then, the number of items in the index for this strategy is $\sum_{g \in \mathcal{G}} 3 \times |g|$.

For query processing, the number of constants a query can have is at most $3 \times |q|$ (this upper bound is reached for boolean queries, where all variables are bound). Using this strategy, one lookup per constant in the query is performed to the index and thus, the number of lookups to DynamoDB is $3 \times |q|$.

Attribute-based strategy

The next indexing strategy, denoted ATT, uses each attribute present in an RDF graph and indexes it in a different table depending on whether it is subject, predicate or object.

Indexing. Let *element* denote any among the subject, predicate and object value of an RDF triple. For each triple of a graph and for each element of the triple, one DynamoDB item is created. The key of the item is named after the element value. As DynamoDB attribute name, we use the URI of the graph containing a triple with this value; as DynamoDB attribute value, we use ϵ. This index distinguishes between the appearances of an URI in the subject, predicate or object of a triple: one DynamoDB table is created for subject-based indexing, one for predicate- and one for object-based indexing. In terms of our storage description language (Definition 2.3.1, Section 2.3.1) ATT storage is shown below.

Storage description
$\underline{\text{S}}\{\text{S}\{\text{G}\{\epsilon\}\}\}, \underline{\text{P}}\{\text{P}\{\text{G}\{\epsilon\}\}\}, \underline{\text{O}}\{\text{O}\{\text{G}\{\epsilon\}\}\}$

In this strategy, false positives can be avoided (e.g., graphs that contain a certain URI but not in the position that this URI appears in the query will not be retrieved).

A general outline of this strategy is shown in Table 3.4. The old design of ATT [BGKM12] is shown in Table 3.4(a) while the new design is depicted in Table 3.4(b).

In the old design, graph URIs were used as primary keys for the three tables, instead of the values of subject, properties and objects. This design allows retrieving efficiently all the subject (resp. property and object) values s_i appearing in an RDF graph g_i by probing the $\underline{\text{S}}$-table (resp. $\underline{\text{P}}$-table, $\underline{\text{O}}$-table). However, using this indexing strategy, one needs to retrieve all the RDF graphs g_i where a subject value s_i appears; to efficiently support such look-ups, secondary indexes are needed. In [BGKM12] these secondary indexes are built automatically (through the use of SimpleDB). In most key-value stores, though, such secondary indexes are not built by default (users have to explicitly request them). In contrast, in the new design, there is no need for building secondary indexes since by default all key-value stores build an index on the primary key.

Another advantage of the new design occurs stems from the fact that an RDF graph usually has a large number of triples. Due to this, the indexes with the old design tend to have few items (few RDF graphs) with a lot of attributes (subjects, properties, and objects). Key-value stores rarely have limitations on the number of items, whereas quite often they impose limitations on the number of attributes per item. As a consequence, in [BGKM12] the limitations of SimpleDB (in particular the maximum number of attributes) were reached very fast (even for the small synthetic datasets used in the experimental evaluation of [BGKM12]) and required special overflow handling, which complicates the development and maintenance of the system but also harms performance.

Concretely, such overflow handling lead to look-ups in several tables needed for a single conceptual *get* operation.

The data from our running example using the new design proposed here leads to the index configuration outlined in Table 3.4(c).

(a) ATT indexing (old design).

$\underline{\text{S}}$ table	
item key	(attr. name, attr. value)
g_1	$(\underline{S}, s_1), (\underline{S}, s_2) \dots$
g_2	$(\underline{S}, s_1), (\underline{S}, s_3) \dots$
$\underline{\text{P}}$ table	
item key	(attr. name, attr. value)
g_1	$(\underline{P}, p_1), (\underline{P}, p_2) \dots$
g_2	$(\underline{P}, p_1), (\underline{P}, p_3) \dots$
$\underline{\text{O}}$ table	
item key	(attr. name, attr. value)
g_1	$(\underline{O}, o_1), (\underline{O}, o_2) \dots$
g_2	$(\underline{O}, o_1), (\underline{O}, o_3) \dots$

(b) ATT indexing (new design).

$\underline{\text{S}}$ table	
item key	(attr. name, attr. value)
s_1	$(g_1, \epsilon), (g_2, \epsilon), \dots$
s_2	$(g_2, \epsilon), \dots$
$\underline{\text{P}}$ table	
item key	(attr. name, attr. value)
p_1	$(g_1, \epsilon), (g_2, \epsilon), \dots$
p_2	$(g_2, \epsilon), \dots$
$\underline{\text{O}}$ table	
item key	(attr. name, attr. value)
o_1	$(g_1, \epsilon), (g_2, \epsilon), \dots$
o_2	$(g_2, \epsilon), \dots$

(c) Sample ATT index entries.

$\underline{\text{S}}$ table	
item key	(attr. name, attr. value)
ub:prof1	(ub:Professors, ϵ)
ub:prof2	(ub:Professors, ϵ)
ub:stud1	(ub:Students, ϵ)
ub:stud2	(ub:Students, ϵ)
ub:dept1	(ub:Students, ϵ)
ub:dept4	(ub:Students, ϵ)
$\underline{\text{P}}$ table	
item key	(attr. name, attr. value)
ub:advisor	(ub:Professors, ϵ)
ub:name	(ub:Professors, ϵ)
	(ub:Students, ϵ)
rdf:type	(ub:Professors, ϵ)
	(ub:Students, ϵ)
ub:member	(ub:Students, ϵ)
ub:takesCourse	(ub:Students, ϵ)
$\underline{\text{O}}$ table	
item key	(attr. name, attr. value)
ub:Professor	(ub:Professors, ϵ)
ub:stud1	(ub:Professors, ϵ)
ub:stud2	(ub:Professors, ϵ)
"bob"	(ub:Professors, ϵ)
"alice"	(ub:Professors, ϵ)
"ted"	(ub:Students, ϵ)
ub:dept1	(ub:Students, ϵ)
ub:dept4	(ub:Students, ϵ)
ub:db	(ub:Students, ϵ)
ub:os	(ub:Students, ϵ)
ub:Dept	(ub:Students, ϵ)

Table 3.4: Sample index entries for the ATT strategy.

Querying. For each RDF term (URI or literal) of an RDF query, a DynamoDB `GetItem` look-up is submitted to the $\underline{\text{S}}$, $\underline{\text{P}}$, or $\underline{\text{O}}$ table of the ATT index, depending on the position of the constant in the query. Each such look-up retrieves the URIs of the graphs which contain a triple with the given term in the respective position. For each triple pattern, the results of all the `GetItem` look-ups based on constants of that triple need to be intersected. This ensures that all the constants of a triple pattern appear in the same dataset. The union of all URI sets thus obtained from the triple patterns of a SPARQL query provides the URIs of the graphs to retrieve from S3, and from the merging of which the query must be answered.

Using our running example, assume that we want to evaluate the query of Figure 2.3. The corresponding DynamoDB queries required in order to retrieve the corresponding datasets are the following:

```
r1:  GetItem(P̲, ub:advisor)
r2:  GetItem(P̲, ub:member)
r3:  GetItem(O̲, ub:dept4)
```

The graph URIs retrieved from DynamoDB request `r1` will be merged with the datasets resulting from the intersection of those retrieved from the requests `r2` and `r3`. The query will be then evaluated on the resulting (merged) graphs to get the correct answers.

Analytical cost model. We assume that the number of distinct subjects, predicates and objects values appearing in a graph is equal to the size of the graph itself, and thus equal to the number of triples (worst case scenario). For each triple in a graph we create three entries in DynamoDB. Thus, the size of the index for this strategy will be $\sum_{g\in\mathcal{G}} 3 \times |g|$.

Given an RDF query q, one lookup per constant in a request is performed to the appropriate table. Thus, the number of lookups is $3 \times |q|$.

Attribute-subset strategy

The following strategy, denoted ATS, is also based on the RDF terms occurring in the datasets, but records more information on how terms are combined within these triples.

Indexing. This strategy encodes each triple $(s\ p\ o)$ by a set of seven patterns s, p, o, sp, po, so and spo, corresponding to all its non-empty attribute subsets. These seven patterns correspond to seven DynamoDB tables. For each triple, seven new items are created and inserted into the corresponding table. As attribute name, we use the URI of the graph containing this pattern; as attribute value we use ϵ. In terms of our storage description language (Definition 2.3.1, Section 2.3.1) ATS storage is shown below.

Storage description
$\underline{\text{S}}\{\text{S}\{\text{G}\{\epsilon\}\}\}+\underline{\text{P}}\{\text{P}\{\text{G}\{\epsilon\}\}\}+\underline{\text{O}}\{\text{O}\{\text{G}\{\epsilon\}\}\}$
$\underline{\text{SP}}\{\text{SP}\{\text{G}\{\epsilon\}\}\}+\underline{\text{PO}}\{\text{PO}\{\text{G}\{\epsilon\}\}\}+\underline{\text{SO}}\{\text{SO}\{\text{G}\{\epsilon\}\}\}$
$\underline{\text{SPO}}\{\text{SPO}\{\text{G}\{\epsilon\}\}\}$

A general outline of this strategy is shown in Table 3.5. The old design of ATS [BGKM12] is shown in Table 3.5(a) while the new design is depicted in Table 3.5(b). In the old design a single table is used to hold information about seven indexes. This is achieved by concatenating a string value ($\underline{\text{S}}$, $\underline{\text{P}}$, $\underline{\text{O}}$, and combinations of them) with the actual value that is indexed. Apart from the counterintuitive organization of the data, the design has various other disadvantages: (i) look-up operations requesting information from logically different indexes are executed against the same table, which limits parallelization opportunities (ii) the performance of the look-up operations degrades as the index grows in size (storing seven logically different indexes into a single table leads to a big, possibly slow, index); (iii) redundant information is stored for distinguishing among the indexed values; (iv) storage space limitations present in some key-value stores (e.g., SimpleDB) may be reached easier. The new design overcomes all of the aforementioned disadvantages. The data from our running example leads to the index configuration outlined in Table 3.5(c).

(a) ATS indexing (old design).

ATS table	
item key	(attr. name, attr. value)
S$|$$s_1$	$(g_1, \epsilon), (g_2, \epsilon), \ldots$
P$|$$p_1$	$(g_1, \epsilon), (g_2, \epsilon), \ldots$
O$|$$o_1$	$(g_1, \epsilon), (g_2, \epsilon), \ldots$
SP$|$$s_1 | p_1$	$(g_1, \epsilon), (g_2, \epsilon), \ldots$
PO$|$$p_1 | o_1$	$(g_1, \epsilon), (g_2, \epsilon), \ldots$
SO$|$$s_1 | o_1$	$(g_1, \epsilon), (g_2, \epsilon), \ldots$
SPO$|$$s_1 | p_1 | o_1$	$(g_1, \epsilon), (g_2, \epsilon), \ldots$

(b) ATS indexing (new design).

S table	
item key	(attr. name, attr. value)
s_1	$(g_1, \epsilon), (g_2, \epsilon), \ldots$
s_2	$(g_2, \epsilon), \ldots$
P table	
item key	(attr. name, attr. value)
p_1	$(g_1, \epsilon), (g_2, \epsilon), \ldots$
p_2	$(g_2, \epsilon), \ldots$
O table	
item key	(attr. name, attr. value)
o_1	$(g_1, \epsilon), (g_2, \epsilon), \ldots$
o_2	$(g_2, \epsilon), \ldots$
SP table	
item key	(attr. name, attr. value)
$s_1 \| p_1$	$(g_1, \epsilon), (g_2, \epsilon), \ldots$
$s_1 \| p_2$	$(g_1, \epsilon), (g_2, \epsilon), \ldots$
$s_2 \| p_1$	$(g_2, \epsilon), \ldots$
PO table	
item key	(attr. name, attr. value)
$p_1 \| o_1$	$(g_1, \epsilon), (g_2, \epsilon), \ldots$
$p_1 \| o_2$	$(g_1, \epsilon), (g_2, \epsilon), \ldots$
$p_2 \| o_1$	$(g_2, \epsilon), \ldots$
SO table	
item key	(attr. name, attr. value)
$s_1 \| o_1$	$(g_1, \epsilon), (g_2, \epsilon), \ldots$
$s_1 \| o_2$	$(g_1, \epsilon), (g_2, \epsilon), \ldots$
$s_2 \| o_3$	$(g_2, \epsilon), \ldots$
SPO table	
item key	(attr. name, attr. value)
$s_1 \| p_1 \| o_1$	$(g_1, \epsilon), (g_2, \epsilon), \ldots$
$s_1 \| p_2 \| o_2$	$(g_1, \epsilon), (g_2, \epsilon), \ldots$
$s_2 \| p_1 \| o_3$	$(g_2, \epsilon), \ldots$

(c) Sample ATS index entries.

S table	
item key	(attr. name, value)
ub:prof1	(ub:Professors, ϵ)
ub:prof2	(ub:Professors, ϵ)
ub:stud1	(ub:Students, ϵ)
…	…
P table	
item key	(attr. name, value)
ub:advisor	(ub:Professors, ϵ)
ub:member	(ub:Students, ϵ)
ub:name	(ub:Professors, ϵ)
	(ub:Students, ϵ)
…	…
O table	
item key	(attr. name, value)
ub:stud1	(ub:Professors, ϵ)
ub:stud2	(ub:Professors, ϵ)
ub:dept4	(ub:Students, ϵ)
…	…
SP table	
item key	(attr. name, value)
ub:prof1$\|$ub:advisor	(ub:Professors, ϵ)
ub:prof2$\|$ub:advisor	(ub:Professors, ϵ)
ub:stud2$\|$ub:member	(ub:Students, ϵ)
…	…
PO table	
item key	(attr. name, value)
ub:advisor$\|$ub:stud1	(ub:Professors, ϵ)
ub:advisor$\|$ub:stud2	(ub:Professors, ϵ)
ub:member$\|$ub:dept4	(ub:Students, ϵ)
…	…
SO table	
item key	(attr. name, value)
ub:prof1$\|$ub:stud1	(ub:Professors, ϵ)
ub:prof2$\|$ub:stud2	(ub:Professors, ϵ)
ub:stud2$\|$ub:dept4	(ub:Students, ϵ)
…	…
SPO table	
item key	(attr. name, value)
ub:prof1$\|$ub:advisor$\|$ub:stud1	(ub:Professors, ϵ)
…	…

Table 3.5: Sample index entries for the ATS strategy.

Querying. For each triple pattern of an RDF query the corresponding `GetItem` call is sent to the appropriate table depending on the position of the bound values of the triple pattern. The item key is a concatenation of the bound values of the triple pattern. The URIs obtained through all the `GetItem` calls identify the graphs on which the query must be evaluated.

For example, for the RDF query of Figure 2.3 we need to perform the following DynamoDB API calls:

```
r1:  GetItem(P, ub:advisor)
r2:  GetItem(PO, ub:member||ub:dept4)
```

We then evaluate the RDF query over the retrieved graphs.

Analytical cost model. For each triple in an RDF graph $g \in \mathscr{G}$, we create at most seven entries in DynamoDB. Thus, the size of the index for this strategy is $\sum_{g \in \mathscr{G}} 7 \times |g|$.

To answer a query q, we perform one lookup for each triple pattern appearing in the query (thus, $|q|$ lookups).

3.4.3 Dictionary encoding

The majority of RDF terms used are URIs, which consist of long strings of text. Since working with long strings is expensive in general, dictionary encoding has been used in many centralized RDF stores, e.g., [NW10]. The technique consists of assigning an integer code to each URI (or string) from the RDF dataset; then, data is stored, and queries are evaluated, based on these numerical values. The query results are then decoded again to the original RDF terms before returning them.

We also adopt dictionary encoding for the QAS strategy; URIs and literals are assigned integer codes with the help of the MD5 hash function. (Technically speaking, this function returns a 16-byte array, but for clarity of presentation, we will simply consider these codes as integers.) To decode results, a *dictionary table* which holds the reverse mappings, from the integer codes the original RDF terms, is required; we store it in Dynamo DB, with the integer codes as keys and the original URIs (or strings) as values. We use `GetItem` calls on this dictionary table to decode the encoded query results.

We also dictionary-encode the key items for the RTS, ATT and ATS indexes. This allows storing as index keys, 16-byte compact codes instead of arbitrarily long URIs or literals. For these strategies, only the encoding part is required, since the query results are computed by RDF-3X out of documents extracted from S3 and loaded on the fly. Additionally, RDF-3X employs dictionary encoding by itself.

Finally, note that dictionary encoding enables to encode RDF terms to binary values in parallel on different machines without any node coordination. This is because we rely on deterministic hash functions which always generate the same hash value for the same given input. On the other hand, hashing can lead to collisions, i.e., two different inputs can be mapped to the same hash value. In the RTS, ATT and ATS strategies, such a collision would only affect the number of datasets that need to be retrieved from S3 (false positives) and not the answers of the query. But even in the QAS strategy we can minimize the probability of a collision by choosing an appropriate hash function. For instance, for a 128-bit hash function, such as MD5, and assuming 2.6×10^{10} different inputs to the hash function, the probability of a collision is very low: 10^{-18} [BK04].

3.5 Experimental evaluation

The proposed architecture and algorithms have been fully implemented in our system AMADA [ABC$^+$12]. In this section we present an experimental evaluation of our strategies and techniques, complementing the experiments presented in [BGKM12] and leading to new conclusions.

3.5.1 Experimental setup

Our experiments were run in the AWS Ireland region in February 2013. For the local query evaluation needed by strategies RTS, ATT, and ATS we have used RDF-3Xv0.3.7 [6] [NW10], a widely known RDF research prototype, to process incoming queries on the graphs identified by our index look-ups. Thus, RDF-3X was available on the EC2 machine(s) used to process queries. For the QAS strategy, when the queries are processed directly on the data retrieved from DynamoDB, we relied on the physical relational algebraic select, project and join operators of our ViP2P project [KKMZ11].

We have used two types of EC2 instances to run the indexing module and query processor:

– **Large** (l), with 7.5 GB of RAM memory and 2 virtual cores with 2 EC2 Compute Units each.
– **Extra large** (xl), with 15 GB of RAM memory and 4 virtual cores with 2 EC2 Compute Units each.

An EC2 Compute Unit is equivalent to the CPU capacity of a 1.0-1.2 GHz 2007 Xeon processor.

To create the dataset we have used subsets of YAGO [7] and DBpedia [8] dumps consisting of approximately 35 million triples (5 GB in NTRIPLE syntax). It is worth noticing that the performance of the selective indexing strategies (i.e., RTS, ATT, ATS) can be greatly affected by the characteristics of the input graphs. AMADA stores and indexes the input graphs as they appear in the input. Pre-processing the input graphs (e.g., partitioning them into smaller graphs with specific properties) can improve the effectiveness of the indexes allowing the retrieval of RDF graphs to be done more accurately. As an extreme case consider the input dataset being a single very big graph; all incoming queries, even if they need a very small portion of the graph, will need to process the entire graph since all the entries in the index will point to the same file. In this work, a very simple pre-processing strategy has been exploited which divides the input dataset into equal-sized graphs. The original dataset is divided into 195 RDF graphs (files) each containing 174K triples on average. More elaborate pre-processing techniques can be used to further improve query performance.

For the query workload, we hand-picked nine queries with various *shapes* (simple, star, mixed), *sizes* (1 to 6 triple patterns), and *selectivities* (high, medium, and low). In addition, we varied the number of constants in the queries, and the number of RDF graphs that they need to access (for the strategies RTS, ATT, ATS). The complete query workload with further details regarding the queries can be found in Appendix A.1.

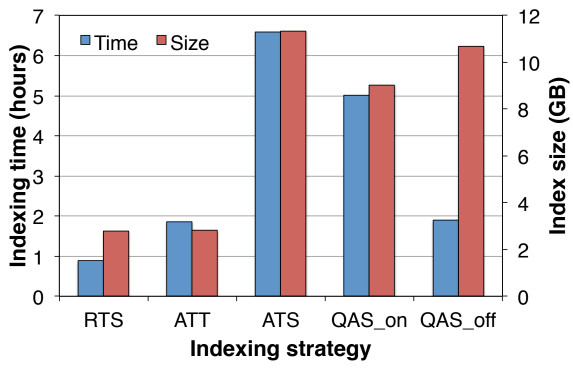
3.5.2 Indexing time and costs

In this section we study the performance of our four RDF indexing strategies. The RDF graphs (files) are initially stored in S3, from which they are gathered in batches by four l instances running the indexing module. We batched the datasets in order to

6. `http://code.google.com/p/rdf3x/`
7. `http://www.mpi-inf.mpg.de/yago-naga/yago/`
8. `http://dbpedia.org/`

minimize the number of calls needed to create the indexes into DynamoDB. Moreover, we used l instances (and not bigger instances) because we found out that DynamoDB is the bottleneck while indexing. We should also note that we used a *total* throughput capacity in our DynamoDB tables of $10,000$ write units. This means that if a strategy required more than one table (like ATS which needs seven tables), we divided the $10,000$ units among all tables (for ATS we used 1428 units per table).

We measure the indexing time and monetary costs of building the indexes in DynamoDB. For the strategies RTS, ATT and ATS we show results only with the dictionary on, as there is always a benefit from it. For the QAS strategy we show results both with (**QAS_on**) and without (**QAS_off**) the dictionary as the difference between the two leads to some interesting observations.



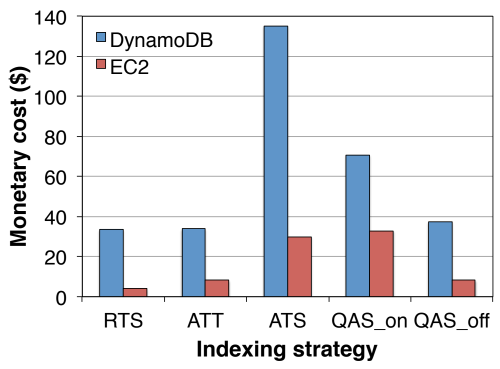
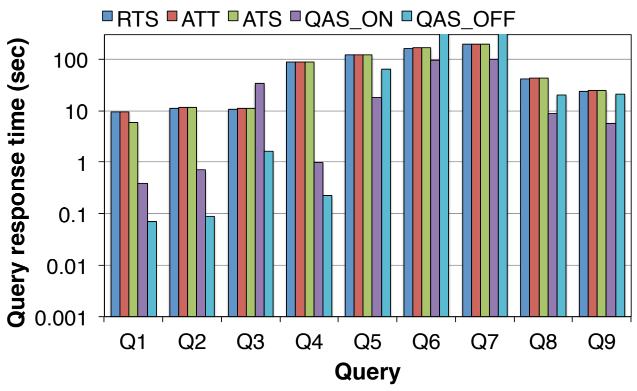


Figure 3.2: Indexing time and size (left) and cost (right).

In Figure 3.2 we demonstrate for each strategy the time required to create the indexes, their size and their indexing cost. Note that to add the items into DynamoDB we used the `BatchWriteItem` operation which can insert up to 25 items at a time in a table. We observe from the blue bars of the left graph of Figure 3.2 that the ATS index is the most time-consuming, since for each triple it inserts seven items into DynamoDB. The same holds for the size of the index, as the ATS occupies about 11 GB. In contrast, the RTS index which inserts only one item for each RDF term is more time-efficient. An interesting observation is that the QAS_off indexing requires significantly less time than when the dictionary is used. This is because inserting items in the dictionary table for each batch becomes a bottleneck. Also, the size of the QAS index with the dictionary is only slightly smaller than when the dictionary is not used, i.e., 9 GB in QAS_on vs. 10.6 GB in QAS_off. This is because of the dataset used in the experiments, where URIs do not occur many times across the triples and thus, the storage space gain is not very impressive.

In the graph at right in Figure 3.2, we show the monetary cost of DynamoDB and the EC2 usage when creating the index. Again, the ATS index is the most expensive one, both for DynamoDB and EC2. Moreover, we observe that the QAS_on is more expensive than QAS_off due to the increased number of items that we insert in the index when using the dictionary. The costs of S3 and SQS are constant for all strategies (0.0022$ and 0.0004$, respectively) and negligible compared to the costs of DynamoDB and EC2 usage. We thus



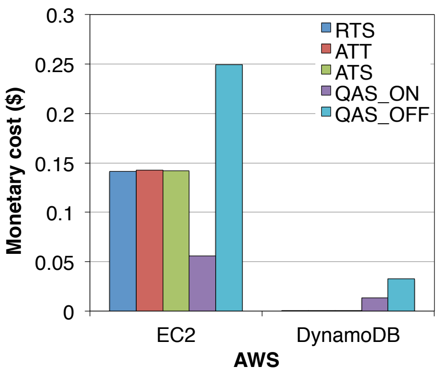


Figure 3.3: Querying response time (left) and cost (right).

omit them from the graph.

Although the information is not present in Figure 3.2, there is also a cost associated with DynamoDB (see Section 3.2.2) even when the system is idle, just for the storage space consumption. Due to this, the size of the index affects the money spent for keeping the index on a monthly basis. For example, the QAS_on index would cost about 10$ per month, while the QAS_off would cost an extra 2$ per month. On the other hand, the RTS, ATT strategies are more economical and would only cost about 3$ per month.

3.5.3 Querying time and costs

In this set of experiments, we measure the query response times and monetary costs for the strategies and the setup discussed previously. We ran one query after the other sequentially using one XL machine.

Figure 3.3 presents the response times of each query in each strategy and the total monetary cost for the whole query workload in each strategy regarding EC2 and DynamoDB usage. We observe that for the selective indexing strategies, i.e., RTS, ATS, and ATT, the queries accessing a small number of graphs (Q1, Q2, Q3, Q9, Q9) are very efficient and are executed in less than 50 seconds. As the number of graphs increases (Q4-Q7) so does the response time for these strategies. This is expected since the retrieved graphs have to be loaded in RDF-3X in order to answer the query; as this number increases, RDF-3X loading time also goes up. Out of these three strategies we cannot pick a winner since all strategies retrieve almost the same graphs from DynamoDB. The only cases where the retrieved graphs were different occurred for Q1 and Q5, where RTS and ATT lead to retrieving an additional RDF graph. Since the size of each RDF graph is relatively small there is no big difference in query response time from the one false positive that appeared. In practice when a triple pattern has two constants false positives may not appear often (a lot of them will be removed from the intersection that we perform between the graphs retrieved for the two constants).

Figure 3.4 shows the distribution of query response time for ATT strategy among: (i) the time to retrieve the relevant files from S3; (ii) the time to write these files to the local disk; (iii) the time to load these files into RDF-3X; (iv) the time to evaluate the query to

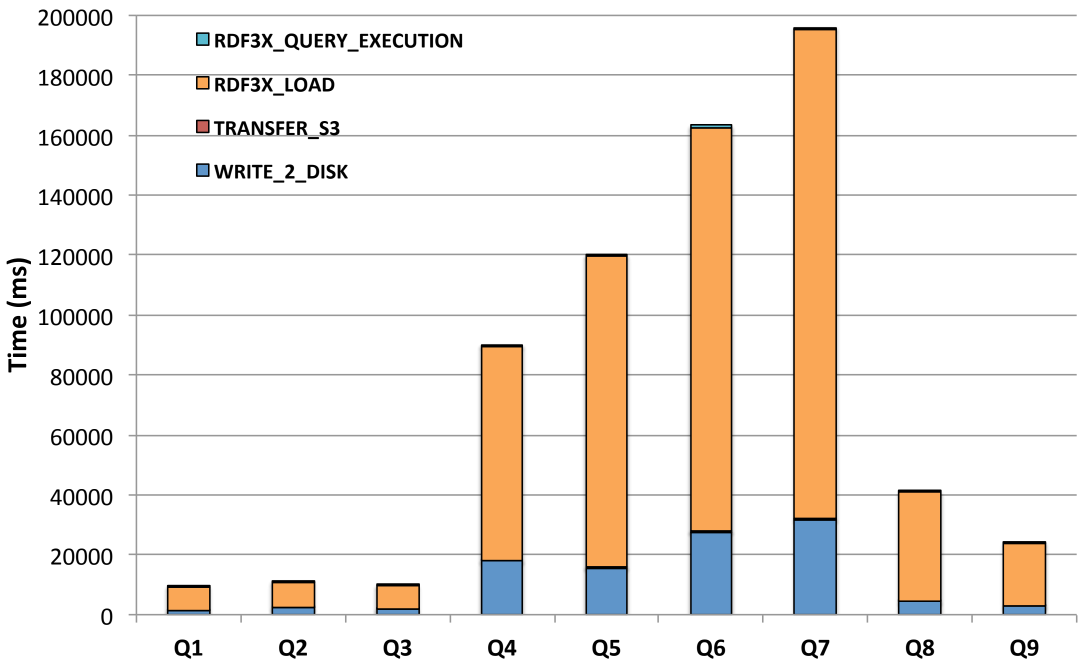


Figure 3.4: Querying response time distribution for ATT strategy.

RDF-3X. Notice that the trends discussed below generalize also to strategies RTS, and ATS thus there is no need for additional figures. As expected writting the files to the local disk and loading them in RDF-3X is the dominant factor affecting query performance, consuming in average 97% of the total time. It is thus evident that in order to bring down the query execution cost we have to invest in reducing the loading cost associated with RDF-3X (which constitutes in average 82% of the total time). There are various directions for achieving the latter result. A straight-forward change could be replacing RDF-3X with another processing engine with a more efficient loading process; this is not guaranteed to improve performance since a more efficient loading process might be associated with building less indexes and thus having an impact in query evaluation. Furthermore, one could eliminate entirely the processing engine and exploit in-house logical and physical operators. However, this requires building and optimizing query plans directly so it alters the original purpose of this strategy. Again, there is no guarantee that this approach will improve the overall performance. These are interesting directions that we would like to explore in our future work.

The strategies relying solely in DynamoDB to answer the queries (QAS_on and QAS_off) are better for highly selective queries (Q1, Q4, Q8, Q9) than those relying on RDF-3X. Especially the one using the dictionary encoding is good even for not very selective queries like Q6 and Q7. On the other hand, answering queries with low selectivity (Q6, Q7) without a dictionary through DynamoDB seems a bad idea due to the large number of items requested from DynamoDB and the large number of intermediate results that are loaded in memory. An interesting exception is Q3, for which the dictionary did not improve the performance. Note that the dictionary encoding invokes a big overhead for decoding the final results (transforming them from compact identifiers to their actual URI values), and especially if the number of returned results is large. If there are no joins in a query, as it is in the case of Q3, there is no profit from the dictionary encoding, and thus, decoding the large number of returned results is a big overhead.

In terms of monetary cost shown at right of Figure 3.3 we observe that the most expensive strategy regarding both EC2 and DynamoDB is QAS_off. For EC2, this can be easily explained by considering the query response times for this strategy and having in mind that queries Q6 and Q7 required more than 300 seconds to be evaluated, overwhelming the CPU for a large period of time. Regarding DynamoDB, the strategy is also expensive since the size of the items that need to be retrieved is significantly larger than for other strategies, which return only dataset names or compact encodings in the case of QAS_on. As anticipated, strategies RTS, ATS and ATT have almost the same EC2 costs, explained by their similar query response times.

3.5.4 Scalability

In this section we measure the total time for a workload of 27 queries (a mix of the queries presented in Appendix A.1) as the number of EC2 machines increases (scale-out) for strategies RTS and QAS_on. ATT and ATS present similar behavior with RTS and thus they are omitted from this experiment. Furthermore QAS_off performs most of the times worse than QAS_on so we chose to drop it from the graphs. The experiments were executed using XL machines varying their number from 1 to 8 and keeping the threads number (4) equal to the number of cores of each machine (allowing a concurrent execution of 4 queries per machine).

In Figure 3.5 we demonstrate how increasing the EC2 machines can affect the total response time for executing the whole query workload. The query response time follows a logarithmic equation where in the beginning and until reaching 4 EC2 instances the time is constantly dropping. Then by increasing the machines we cannot run faster due to the fact that all queries are distributed among machines and run in parallel. For example for our workload of 27 queries, using 8 machines will result into running 3 queries on each machine and due to the number of threads all queries will run in parallel and the total time will be equal with the less efficient query. Both strategies scale well with QAS_on being slightly worse due the large number of concurrent requests in DynamoDB.

Scaling-out the machines for DynamoDB is not feasible in the Amazon cloud. In general, similar services from AWS are usually offered as black boxes and the user does not have control over them other than specifying some performance characteristics, such as the throughput in DynamoDB. Finally, we have also experimented with scaling the size of the data in [BGKM12] and observed that the time for building the index scales linearly with the number of triples in the datasets, as it is also evident from our analytical cost model, so we omit it from this experimental evaluation.

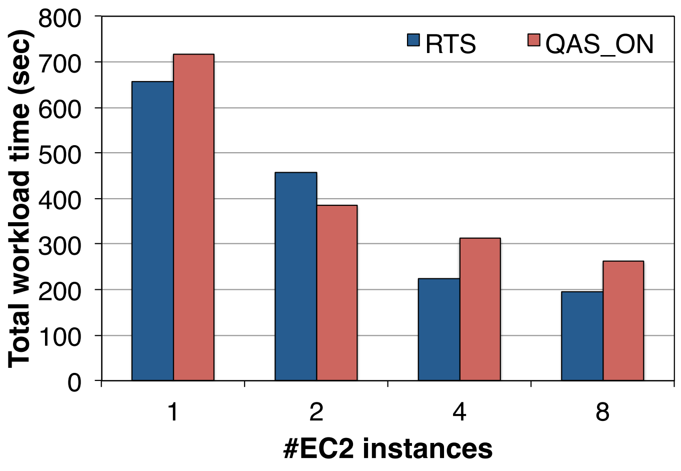


Figure 3.5: Total time for workload of #27 queries.

3.5.5 Experiments conclusion

Summing up, our baseline strategy RTS is the best, providing a good trade-off between indexing time, index size, query efficiency and overall monetary cost both for building the indexes and answering queries as well. Targeting query efficiency, QAS_on is the best strategy being 50% more expensive than RTS. In addition the size of the index for QAS_on is five times bigger in comparison with RTS, making the strategy highly expensive for long-term use. Among the discussed strategies, ATS can be considered as one of the worst, since it is the most expensive in terms of money and index size, whereas from the efficiency perspective the indexing time is high, and the query response time does not differ significantly from the other strategies (RTS and ATT) relying on RDF-3X to answer the queries.

In our previous work [BGKM12] ATS strategy performed significantly better than the other strategies. This behavior lead us to the preliminary conclusion that paying more for more indexes can significantly improve performance. However, revisiting the experiments after optimizing the design of ATT and ATS (Section 3.4.2) and on significantly larger data volumes, it turns out that the performance advantage of such more extensive indexing does not always hold. In addition, with the new strategies (RTS, QAS) we have shown that there are cheaper alternatives for achieving equal (RTS) or better performance (QAS). Finally, the experiments were performed on real datasets (in contrast with the synthetic benchmark used in [BGKM12]), providing robust results and validating the usefulness of our proposal in real-life scenarios.

3.6 Conclusion

This chapter described an architecture for storing and querying RDF data using off-the-shelf cloud services, in particular the AMADA platform we have developed and demonstrated recently [ABC+12, BGKM12]. The starting point of the present work is [BGKM12], however in this chapter we have presented a different set of strategies and accordingly new experiments, at a much larger scale than we had previously described in [BGKM12].

Within AMADA, we devised indexing techniques for identifying a tight subset of the RDF dataset which may contain results for a specific query, and we have examined a technique for answering a SPARQL query from the index itself. We presented analytical cost models for each strategy and evaluated their indexing and querying performance and monetary costs.

A direction we have not considered in this work is the parallelization of the task of evaluating a single query on a large RDF dataset. This is interesting, especially for non-selective queries, since the parallel processing capabilities of a cloud may lead to shorter response times. Chapter 4 is an important step towards this direction where a novel optimization algorithm is proposed for building massively parallel plans. The algorithm is combined with a MapReduce-based system that allows to process efficiently even bigger RDF graphs.

Chapter 4

CliqueSquare: Flat Plans for Massively Parallel RDF Queries

In this chapter, we present CliqueSquare, a distributed RDF data management system built on top of Hadoop. CliqueSquare incorporates a novel optimization algorithm that is able to produce massively parallel plans for RDF queries. The algorithm seeks to build *flat* plans, where the number of joins encountered on a root-to-leaf path in the plan is minimized. We present a family of optimization algorithms, relying on *n-ary (star) equality joins* to build flat plans, and compare their ability to find the flattest possibles. Inspired by existing partitioning and indexing techniques, we present a generic storage strategy suitable for storing RDF data in HDFS (Hadoop's Distributed File System) that binds well with the flat plans provided by the optimization algorithm. We provide algorithms for translating logical plans to physical plans and subsequently to MapReduce jobs based on the available physical operators. Finally, we present experimental results that validate the efficiency and effectiveness of the optimization algorithm demonstrating also the overall performance of the system.
An early version of this work was presented in a national conference [GKM+13], while this chapter closely follows the international conference publication [GKM+15] and the respective technical report [GKM+14]. The CliqueSquare system was demonstrated in a national [DGK+14] and an international conference [DGK+15], and finally open-sourced in January 2015 [1].

4.1 Introduction

In Chapter 3, we presented AMADA, an architecture for storing and querying RDF data using off-the-shelf cloud services; we explored how different indexing strategies can affect query performance. However, as we have seen in Chapter 1, building a distributed RDF data management system requires addressing two additional challenges: how to partition the RDF graph data across the nodes, and how to split the query evaluation across these nodes.

1. `https://sourceforge.net/projects/cliquesquare/`

Clearly, data distribution has an important impact on query performance. Accordingly, many previous works on distributed RDF query evaluation, such as [HAR11, LL13, WZY$^+$15, GHS12, HS13], have placed an important emphasis on the data partitioning process (workload-driven in the case of [GHS12, HS13]), with the goal of making the evaluation of certain shapes of queries *parallelizable without communications* (or *PWOC*, in short). In a nutshell, a PWOC query for a given data partitioning can be evaluated by taking the union of the query results obtained on each node.

In this work, we exploit a less elaborate partitioning technique that combines hash partitioning based on the three attributes of a triple and property partitioning (a.k.a. vertical partitioning). The combined partitioning technique is exploited for the first time in this work, and is implemented on top of a distributed file system. However, it is easy to see that no single partitioning can guarantee that *all* queries are PWOC; in fact, most queries do require processing across multiple nodes and thus, data re-distribution across nodes, a.k.a. shuffling.

The more complex the query is, the bigger will be the impact of evaluating the distributed part of the query plan. *Logical query optimization* – deciding how to decompose and evaluate an RDF query in a massively parallel context – has thus also a crucial impact on performance. As it is well-known in distributed data management [ÖV11], to efficiently evaluate queries one should maximize *parallelism* (both *inter-operator* and *intra-operator*) to take advantage of the distributed processing capacity and thus, reduce the response time.

In a parallel RDF query evaluation setting, intra-operator parallelism relies on join operators that process chunks of data in parallel. To increase inter-operator parallelism one should aim at building *massively-parallel (flat) plans*, having as few (join) operators as possible on any root-to-leaf path in the plan; this is because the processing performed by such joins directly adds up into the response time. Prior works (see Section 2.3) have binary joins organized in bushy or left-deep plans, bushy plans with n-ary joins (with $n > 2$) only in the first level of the plans and binary in the next levels, or n-ary joins at all levels but organized in left-deep plans. Such methods lead to high (non-flat) plans and hence high response times. HadoopRDF is the only one proposing some heuristics to produce flat plans [HMM$^+$11], but it has two major disadvantages: (i) it produces a single plan that can be inefficient; (ii) it does not guarantee that the plan will be as flat as possible.

In this chapter, we focus mostly on *the logical query optimization* of RDF queries, seeking to build *flat* query plans composed of *n-ary (star) equality joins*. Flat plans are most likely to lead to shorter response time in distributed/parallel settings. We describe a search space of logical plans obtained by relying on *n-ary (star) equality joins*. The interest of such joins is that by aggressively joining many inputs in a single operator, they allow building flat plans. In addition, we provide a novel generic algorithm, called CliqueSquare, for exhaustively exploring this space, and a set of three algorithmic choices leading to eight variants of our algorithm. We present a thorough analysis of these variants, from the perspective of their ability to find one of (or all) the flattest possible plans for a given query. We show that the variant we call CliqueSquare-MSC is the most interesting one, because it develops a reasonable number of plans and is guaranteed to find some of the flattest ones. To validate the usefulness of our optimization algorithm and to fulfill the requirement for a cloud-based data management system, we relied on Hadoop [had11] framework. We develop a storage strategy suitable for storing RDF data in Hadoop's Distributed File System that binds well with the flat plans provided by the optimization algorithm. Furthermore, we provide algorithms for translating logical plans to physical plans, and subsequently to MapReduce jobs based on the available physical operators and the chosen storage strategy.

We chose to build our system relying on MapReduce and Hadoop since by design it covers many of the properties and features that are requested [Aba09] from cloud-based systems. In addition, elastic MapReduce services are readily provided by important cloud providers such as Amazon [2] and Google [3].

It is worth noting that our optimization algorithms and the respective findings are not specific to RDF, but apply to any conjunctive query processing setting based on n-ary (star) equality joins. However, they are of particular interest for RDF, since (as noted e.g., in [NW10, GKLM11, TSF$^+$12]) RDF queries tend to involve more joins than a relational query computing the same result. This is because relations can have many attributes, whereas in RDF each query atom has only three, leading to syntactically more complex queries.

The Chapter is organized as follows. First, we discuss the architecture of the system and the interaction among the various components of CliqueSquare and Hadoop in Section 4.2. Section 4.3 introduces the logical model used in CliqueSquare for queries and query plans, and describes our generic logical optimization algorithm. In Section 4.4, we present our algorithm variants, their search spaces, and we analyze them from the viewpoint of their ability to produce flat query plans. Section 4.5 demonstrates how we store RDF data in HDFS, while Section 4.6 shows how to translate and execute our logical plans to MapReduce jobs, based on CliqueSquare storage. Section 4.7 experimentally demonstrates the effectiveness and efficiency of our logical optimization approach. Finally, Section 4.8 concludes our findings.

4.2 Architecture

We introduce CliqueSquare, a massively parallel cloud-based system that relies on efficient algorithms and Hadoop framework for storing and retrieving RDF data. RDF graphs are uploaded in the HDFS where they are partitioned and stored in flat HDFS files using MapReduce jobs. Then, RDF data can be retrieved efficiently from HDFS by composing RDF queries, that are parsed, optimized, and translated to MapReduce jobs relying heavily on a logical optimization module that generates massively parallel flat plans.

An overview of our system architecture is depicted in Figure 4.1. A user interaction with our system can be described as follows. The system is divided into the front-end component that runs in a centralized manner and the Hadoop component that operates on

2. `http://aws.amazon.com/elasticmapreduce/`
3. `http://www.qubole.com/google-compute-engine/`

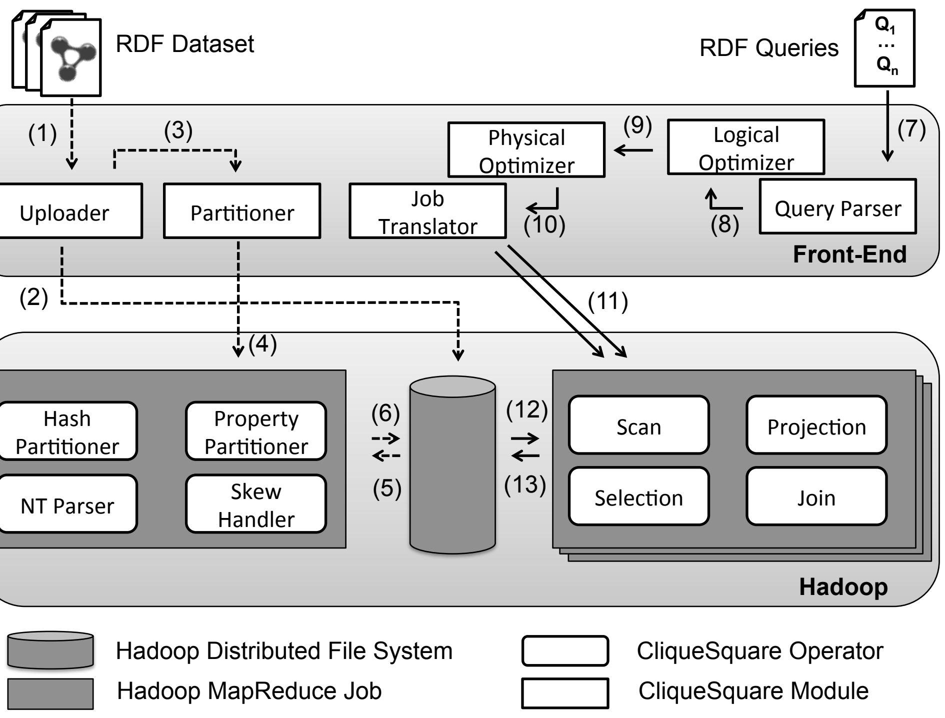


Figure 4.1: CliqueSquare architecture based on Hadoop Ecosystem.

a cluster of machines.

Users interact with the front-end component to store the RDF dataset to the HDFS (**1**). The RDF files comprising the dataset are uploaded to the HDFS using the *Uploader* module (**2**) and then a message informs the *Partitioner* (**3**) that the data are successfully uploaded to the HDFS and they are ready to be partitioned. The partitioner builds a MapReduce job customized with CliqueSquare operators (**4**) that reads the proper files from the HDFS (**5**), processes them exploiting the available machines in the cluster, and stores the partitioned data again in HDFS (**6**) (the original files can be deleted). The steps (**1**) to (**6**) are detailed in Section 4.5.

To retrieve RDF data from CliqueSquare users are able to submit their queries using the front-end component (**7**). First, the RDF query is parsed generating a graph representation (a.k.a. Variable graph) (**8**) that is given as input to the *Logical Optimizer*. The *Logical Optimizer* uses CliqueSquare optimization algorithm and provides a set of logical plans (**9**). Steps (**7**) to (**9**) are described in Section 4.3 and Section 4.4. The logical plans are translated to physical plans exploiting the partitioning of the data and the available physical operators; out of which one is chosen based on a cost model (**10**). The physical plan is processed from the *Job Translator*, which builds a sequence of MapReduce jobs whose execution is delegated to Hadoop (**11**). Every MapReduce job incorporates parts of the physical plan, reading the proper files from the HDFS (**12**) and writing the intermediate (final) results from the jobs back to the HDFS (**13**). Section 4.6 provides details for steps (**10**) to (**13**).

```
SELECT ?a ?b
WHERE {
      ?a p1 ?b
      ?a p2 ?c
      ?d p3 ?a
      ?d p4 ?e
      ?l p5 ?d
      ?f p6 ?d
      ?f p7 ?g
      ?g p8 ?h
      ?g p9 ?i
      ?i p10 ?j
      ?j p11 "C1"
}
```

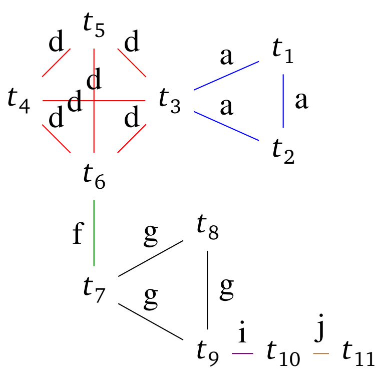


Figure 4.2: Query Q_1 and its variable graph $\mathbf{G_1}$.

4.3 Logical query model

This section describes the CliqueSquare approach for processing queries based on a notion of *query variable graphs*. We introduce these graphs in Section 4.3.1 and present the CliqueSquare optimization algorithm in Section 4.3.2.

4.3.1 Query model

We model an RDF query as a set of n-ary relations connected by joins. Specifically, we rely on a *variable (multi)graph representation*, inspired from the classical relational *Query Graph Model* (QGM) [HFLP89], and use it to represent incoming queries, as well as intermediary query representations that we build as we progress toward obtaining logical query plans. Formally:

Definition 4.3.1 (Variable graph)**.** *A* variable graph G_V *of an RDF query q is a labeled multigraph* (N, E, V)*, where* V *is the set of variables from q,* N *is the set of nodes, and* $E \subseteq N \times V \times N$ *is a set of labeled undirected edges such that: (i) each node* $n \in N$ *corresponds to a set of triple patterns in q; (ii) there is an edge* $(n_1, v, n_2) \in E$ *between two distinct nodes* $n_1, n_2 \in N$ *iff their corresponding sets of triple patterns join on the variable* $v \in V$*.*

Figure 4.2 shows a query and its variable graph, where every node represents a single triple pattern. More generally, one can also use variable graphs to represent *(partially) evaluated queries*, in which some or all the joins of the query have been enforced. A node in such a variable graph corresponds to a *set of triple patterns* that have been joined on their common variables, as the next section illustrates.

4.3.2 Query optimization algorithm

The CliqueSquare process of building logical query plans starts from *the query variable graph* (where every node corresponds to a single triple pattern), treated as an *initial state*, and repeatedly applies *transformations* that decrease the size of the graph, until it is reduced to only one node; a one-node graph corresponds to having applied all the query joins. On a given graph (state), several transformations may apply. Thus, there are many possible sequences of states going from the query (original variable graph) to a complete query plan (one-node graph). Out of each such sequence of graphs, CliqueSquare creates a logical plan. In the sequel of Section 4.3, we detail the graph transformation process, and delegate plan building to Section 4.4.

Variable cliques. At the core of query optimization in CliqueSquare lies the concept of *variable clique*, which we define as a set of variable graph nodes connected with edges having a certain label. Intuitively, *a clique corresponds to an n-ary (star) equi-join.* Formally:

Definition 4.3.2 (Maximal/partial variable clique). *Given a variable graph $G_V = (N, E, V)$, a* maximal (resp. partial) clique *of a variable $v \in V$, denoted $c\ell_v$, is the set (resp. a non-empty subset) of* all *nodes from N which are incident to an edge $e \in E$ with label v.*

For example, in the variable graph $\mathbf{G_1}$ of query $\mathrm{Q_1}$ (see Figure 4.2), the maximal variable clique of d, $c\ell_d$ is $\{t_3, t_4, t_5, t_6\}$. Any non-empty subset is a partial clique of d, e.g., $\{t_3, t_4, t_5\}$.

Clique Decomposition. The first step toward building a query plan is to decompose (split) a variable graph into several cliques. From a query optimization perspective, *clique decomposition corresponds to identifying partial results to be joined*, i.e., for each clique in the decomposition output, exactly one join will be built. Formally:

Definition 4.3.3 (Clique decomposition). *Given a variable graph $G_V = (N, E, V)$, a* clique decomposition $\mathscr{D}$ *of G_V is a set of variable cliques (maximal or partial) of G_V which covers all nodes of N, i.e., each node $n \in N$ appears in at least one clique, such that the size of the decomposition $|\mathscr{D}|$ is strictly smaller than the number of nodes $|N|$.*

Consider again our query $\mathrm{Q_1}$ example in Figure 4.2. One clique decomposition is $\mathrm{d_1} = \{\{t_1, t_2, t_3\}, \{t_3, t_4, t_5, t_6\}, \{t_6, t_7\}, \{t_7, t_8, t_9\}, \{t_9, t_{10}\}, \{t_{10}, t_{11}\}\}$; this decomposition follows the distribution of colors on the graph edges in Figure 4.2. A different decomposition is for instance $\mathrm{d_2} = \{\{t_1, t_2\}, \{t_3, t_4, t_5\}, \{t_6, t_7\}, \{t_8, t_9\}, \{t_{10}, t_{11}\}\}$; indeed, there are many more decompositions. We discuss the space of alternatives in the next section.

Observe that we do not allow a decomposition to have *more* cliques than there are nodes in the graph. This is because a decomposition corresponds to a step forward in processing the query (through its variable graph), and this advancement is materialized by the graph getting strictly smaller.

Based on a clique decomposition, the next important step is *clique reduction.* From a query optimization perspective, *clique reduction corresponds to applying the joins identified by the decomposition.* Formally:

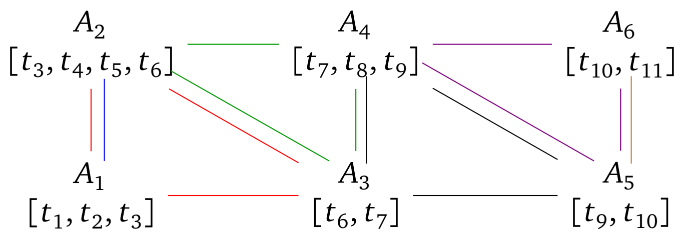


Figure 4.3: Clique reduction $\mathbf{G_2}$ of Q_1's variable graph (shown in Figure 4.2).

Algorithm 1: CliqueSquare algorithm

```
1  CliqueSquare ($G, states$)
       Input  : Variable graph $G$; queue of variable graphs $states$
       Output: Set of logical plans $QP$
2      $states = states \cup \{G\}$;
3      if $|G| = 1$ then
4          $QP \leftarrow$ createQueryPlan ($states$);
5      else
6          $QP \leftarrow \emptyset$;
7          $\mathcal{D} \leftarrow$ cliqueDecompositions($G$);
8          foreach $d \in \mathcal{D}$ do
9              $G' \leftarrow$ cliqueReduction($G, d$);
10             $QP \leftarrow QP \cup$ CliqueSquare ($G', states$);
11     return $QP$;
```

Definition 4.3.4 (Clique Reduction)**.** *Given a variable graph $G_V = (N, E, V)$ and one of its clique decompositions $\mathcal{D}$, the* reduction of G_V based on $\mathcal{D}$ *is the variable graph $G'_V = (N', E', V)$ such that: (i) every clique $c \in \mathcal{D}$ corresponds to a node $n' \in N'$, whose set of triple patterns is the union of the nodes involved in $c \subseteq N$; (ii) there is an edge $(n'_1, v, n'_2) \in E'$ between two distinct nodes $n'_1, n'_2 \in N'$ iff their corresponding sets of triple patterns join on the variable $v \in V$.*

For example, given the query Q_1 in Figure 4.2 and the above clique decomposition d_1, CliqueSquare reduces its variable graph $\mathbf{G_1}$ into the variable graph $\mathbf{G_2}$ shown in Figure 4.3. Observe that in $\mathbf{G_2}$, the nodes labeled A_1 to A_8 each correspond to several triples from the original query: A_1 corresponds to three triples, A_2 to four triples, etc.

CliqueSquare algorithm. Based on the previously introduced notions, the CliqueSquare query optimization algorithm is outlined in Algorithm 1. CliqueSquare takes as an input a variable graph G corresponding to the query with *some* of the predicates applied (while the others are still to be enforced), and a list of variable graphs $states$ tracing the sequence of transformations which have lead to G, starting from the original query variable graph. The algorithm outputs a set of logical query plans QP, each of which encodes an alternative way to evaluate the query.

The initial call to CliqueSquare is made with the variable graph G of the initial query, where each node consists of a single triple pattern, and the empty queue $states$. At each (recursive) call, cliqueDecompositions (line 7) returns a set of clique decompositions of G. Each decomposition is used by cliqueReduction (line 9) to reduce G into the variable graph G', where the n-ary joins identified by the decomposition have been applied. G' is in turn recursively processed, until it consists of a single node. When this is the case (line 3), CliqueSquare builds the corresponding logical query plan out of $states$ (line 4), as we explain in the next section. The plan is added to a global collection QP, which is returned when all the recursive calls have completed.

4.4 Query planning

We describe CliqueSquare's logical operators, plans, and plan spaces (Section 4.4.1) and how logical plans are generated by Algorithm 1 (Section 4.4.2). We then consider a set of alternative concrete clique decomposition methods to use within the CliqueSquare algorithm, and describe the resulting search spaces (Section 4.4.3). We introduce plan *height* to quantify its flatness, and provide a complete characterization of the CliqueSquare algorithm variants w.r.t. their ability to build the flattest possible plans (Section 4.4.4). Finally, we present a complexity analysis of our optimization algorithm (Section 4.4.5).

4.4.1 Logical CliqueSquare operators and plans

Let Val be an infinite set of data values, A be a finite set of attribute names, and $R(a_1, a_2, \ldots, a_n)$, $a_i \in A$, $1 \leq i \leq n$, denote a relation over n attributes, such that each tuple $t \in R$ is of the form $(a_1{:}v_1, a_2{:}v_2, \ldots, a_n{:}v_n)$ for some $v_i \in Val$, $1 \leq i \leq n$. In our context, we take Val to be a subset of $U \cup L$, and $A = var(tp)$ to be the set of variables occurring in a triple pattern $tp, A \subseteq V$. Every mapping $\mu(tp)$ from $A = var(tp)$ into $U \cup L$ leads to a tuple in a relation which we denote R_{tp}. To simplify presentation and without loss of generality, we assume $var(tp)$ has only those tp variables which participate in a join.

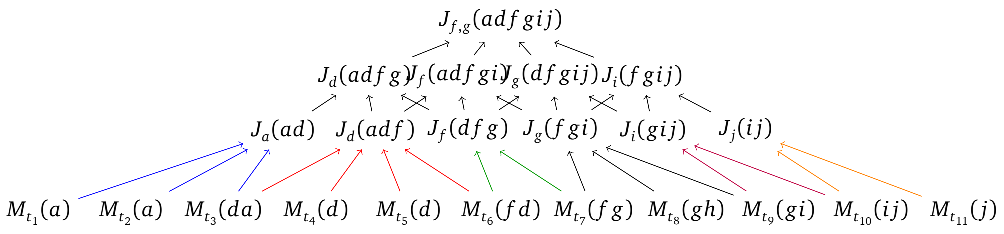


Figure 4.4: Sample logical plan built by CliqueSquare for Q1 (Figure 4.2).

We consider the following logical operators, where the output attributes are identified as $(a_1, \ldots, a_n)$:

- **Match**, $M_{tp}(a_1, \ldots, a_n)$, is parameterized by triple pattern tp and outputs a relation comprising the triples matching tp in the store.

- **Join**, $J_A(op_1, \dots, op_m)(a_1, \dots, a_n)$, takes as input a set of m logical operators such that A is the intersection of their attribute sets, and outputs their join on A.
- **Select**, $\sigma_c(op)(a_1, \dots, a_n)$, takes as input the operator op and outputs those tuples from op which satisfy the condition c (a conjunction of equalities).
- **Project**, $\pi_A(op)(a_1, \dots, a_n)$, takes as input op and outputs its tuples restricted to the attribute set A.

A *logical query plan* p is a rooted directed acyclic graph (DAG) whose nodes are logical operators. Node lo_i is a parent of lo_j in p *iff* the output of lo_i is an input of lo_j. Furthermore, a subplan of p is a sub-DAG of p.

The *plan space* of a query q, denoted as $\mathscr{P}(q)$, is the set of *all* the logical plans computing the answer to q.

4.4.2 Generating logical plans from graphs

We now outline the createQueryPlan function used by Algorithm 1 to generate plans. When invoked, the queue $states$ contains a list of variable graphs, the last of which (tail) has only one node and thus corresponds to a completely evaluated query.

First, createQueryPlan considers the first graph in $states$ (head), which is the initial query variable graph; let us call it $\mathbf{G_q}$. For each node in $\mathbf{G_q}$ (query triple pattern tp), a match (M) operator is created, whose input is the triple pattern tp and whose output is a relation whose attributes correspond to the variables of tp. We say this operator is *associated to* tp. For instance, consider node t_1 in the graph $\mathbf{G_1}$ of Figure 4.2: its associated operator is $M_{t_1}(a, b)$.

Next, createQueryPlan builds join operators out of the following graphs in the queue. Let $\mathbf{G_{crt}}$ be the current graph in $states$ (not the first). Each node in $\mathbf{G_{crt}}$ corresponds to a clique of node(s) from the previous graph in $states$, let's call it $\mathbf{G_{prev}}$.

For each $\mathbf{G_{crt}}$ node n corresponding to a clique made of a *single* node m from $\mathbf{G_{prev}}$, createQueryPlan *associates to* n the operator already associated to m.

For each $\mathbf{G_{crt}}$ node n corresponding to a clique of *several* nodes from $\mathbf{G_{prev}}$, create-QueryPlan creates a J_A join operator and *associates it to* n. The attributes A of J_A are the variables defining the respective clique. The parent operators of J_A are the operators associated to each $\mathbf{G_{prev}}$ node m from the clique corresponding to n; since $states$ is traversed from the oldest to the newest graph, when processing $\mathbf{G_{crt}}$, we are certain that an operator has already been associated to each node from $\mathbf{G_{prev}}$ and the previous graphs. For example, consider node A_1 in $\mathbf{G_2}$ (Figure 4.3), corresponding to a clique on the variable a in the previous graph $\mathbf{G_1}$ (Figure 4.2); the join associated to it is $J_a(abcd)$.

Further, if there are query predicate which can be checked on the join output and could not be checked on any of its inputs, a selection applying them is added on top of the join.

Finally, a projection operator π is created to return just the distinguished variables part of the query result, then projections are pushed down etc. A logical plan for the query Q_1 in Figure 4.2, starting with the clique decomposition/reduction shown in Figure 4.3, appears in Figure 4.4.

4.4.3 Clique decompositions and plan spaces

The plans produced by Algorithm 1 are determined by variable graphs sequences; in turn, these depend on the clique decompositions returned by cliqueDecompositions. Many clique decomposition methods exist.

First, they may use partial cliques or only maximal ones (Definition 4.3.2); maximal cliques correspond to systematically building joins with as many inputs (relations) as possible, while partial cliques leave more options, i.e., a join may combine only some of the relations sharing the join variables.

Second, the cliques may form an exact cover of the variable graph (ensuring each node belongs to exactly one clique), or a simple cover (where a node may be part of several cliques). Exact covers lead to tree-shaped query plans, while simple covers may lead to DAG plans. Tree plans may be seen as reducing total work, given that no intermediary result is used twice; on the other hand, DAG plans may enable for instance using a very selective intermediary result as an input to two joins in the same plan, to reduce their result size.

Third, since every clique in a decomposition corresponds to a join, decompositions having as few cliques as possible are desirable. We say a clique decomposition for a given graph is minimum among all the other possible decompositions if it contains the lowest possible number of cliques. Finding such decompositions amounts to finding minimum set covers [Kar72].

Decomposition and algorithm acronyms. We use the following short names for decomposition alternatives. **XC** decompositions are exact covers, while **SC** decompositions are simple covers. A $+$ superscript is added when only maximal cliques are considered; the absence of this superscript indicates covers made of partial cliques. Finally, **M** is used as a prefix when only minimum set covers are considered.

We refer to the CliqueSquare algorithm variant using a decomposition alternative $\mathscr{A}$ (one among the eight above) as CliqueSquare-$\mathscr{A}$.

CliqueSquare-MSC example. We illustrate below the working of the CliqueSquare-MSC variant (which, as we will show, is the most interesting from a practical perspective), on the query Q_1 of Figure 4.2. CliqueSquare-MSC builds out of the query variable graph $\mathbf{G_1}$ of Figure 4.2, successively, the graphs $\mathbf{G_3}$, then $\mathbf{G_4}$ and $\mathbf{G_5}$ shown in Figure 4.6. At the end of the process, $states$ comprises $[\mathbf{G_1}, \mathbf{G_3}, \mathbf{G_4}, \mathbf{G_5}]$. CliqueSquare plans are created as described in Section 4.4.2; the final plan is shown in Figure 4.5.

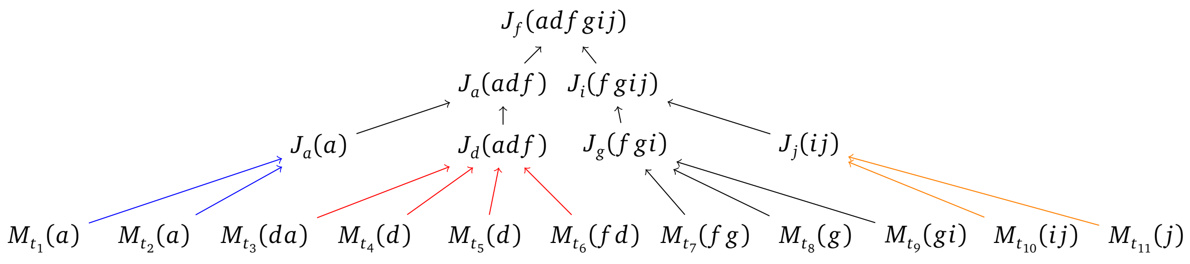


Figure 4.5: Logical plan built by CliqueSquare-MSC for Q1 (Figure 4.2).

$$\begin{tikzcd}\overset{A_1}{[t_1,t_2]} \arrow[r, no head, "a"] & \overset{A_2}{[t_3,t_4,t_5,t_6]} \arrow[r, no head, "f"] & \overset{A_3}{[t_7,t_8,t_9]} \arrow[r, no head, "i"] & \overset{A_4}{[t_{10},t_{11}]}\end{tikzcd}$$

(a) 1st call: graph $\mathbf{G_3}$

$$\begin{tikzcd}\overset{B_1}{[A_1,A_2]} \arrow[r, no head, "f"] & \overset{B_2}{[A_3,A_4]}\end{tikzcd} \qquad \overset{C_1}{[B_1,B_2]}$$

(b) 2nd call: graph $\mathbf{G_4}$ (c) 3rd call: graph $\mathbf{G_5}$

Figure 4.6: Variable graphs after each call of CliqueSquare-MSC.

The set of logical plans developed by CliqueSquare-$\mathscr{A}$ for a query q is termed *plan space of $\mathscr{A}$ for q* and we denote it $\mathscr{P}_{\mathscr{A}}(q)$; clearly, this must be a subset of $\mathscr{P}(q)$. We analyze the variants' plan spaces below.

Relationships between plan spaces. We have completely characterized the set inclusion relationships holding between the plan spaces of the eight CliqueSquare variants. Figure 4.7 summarizes them: an arrow from option $\mathscr{A}$ to option $\mathscr{A}'$ indicates that the plan space of option $\mathscr{A}$ *includes* the one of option $\mathscr{A}'$. For instance, CliqueSquare-SC (partial cliques, all set covers) has the biggest search space $\mathscr{P}_{\text{SC}}$ which includes all the others.

Theorem 4.4.1 (Plan spaces relationships)**.** *All the inclusion relationships shown in Figure 4.7 hold.*

$$\begin{tikzcd}
 & \mathscr{P}_{\text{SC}} \arrow[dl] \arrow[d] \arrow[dr] & \\
\mathscr{P}_{\text{SC}^+} \arrow[d] \arrow[dr] & \mathscr{P}_{\text{XC}} \arrow[dl] \arrow[dr] & \mathscr{P}_{\text{MSC}} \arrow[dl] \arrow[d] \\
\mathscr{P}_{\text{XC}^+} \arrow[dr] & \mathscr{P}_{\text{MSC}^+} \arrow[d] & \mathscr{P}_{\text{MXC}} \arrow[dl] \\
 & \mathscr{P}_{\text{MXC}^+} &
\end{tikzcd}$$

Figure 4.7: Inclusions between the plan spaces of CliqueSquare variants.

Proof. Recall that a decomposition is determined by three choices: (a) maximal or partial cliques; (b) exact or simple set cover; (c) minimum-size versus all covers. Therefore, we use a triple (o_1, o_2, o_3) where $o_i \in \{<, >, =\}$, $1 \leq i \leq 3$, to encode three relationships between two options, option i and option j:

- The symbol o_1 represents the relationship between the types of cliques used: since maximal cliques are a special case of partial cliques, o_1 is $<$ iff Option i uses maximal cliques while Option j uses partial cliques, $>$ if the opposite holds, and $=$ otherwise.

– The symbol o_2 represents the relationship between the types of cover used. Similarly, since exact covers are particular cases of set covers, o_2 is $<$ iff Option i relies on exact covers and Option j on general set covers; o_2 is $>$ in the opposite case, and $=$ otherwise.
– Finally, o_3 encodes the relationship between the size of the covers which are retained from the cover algorithms: minimum set covers being more restrictive, o_3 is $<$ iff Option i uses only the minimum covers while Option j uses them all, $>$ in the opposite case, and $=$ otherwise.

For example, comparing MXC$^+$ with XC$^+$ leads to the triple $(=,=,<)$: they both use maximal cliques and exact cover; MXC$^+$ considers only minimum covers while XC$^+$ considers them all. We say a symbol $s \in \{<,>\}$ *dominates* a triple (o_1, o_2, o_3) if the triple contains s and does not contain the opposite-direction symbol. For instance, $<$ dominates $(=,<,=)$ as well as $(<,<,=)$, but does not dominate $(<,>,=)$; $>$ does not dominate the latter, either. The following simple property holds:

Proposition 4.4.1 (Option domination)**.** *Let i, j be two options, $i, j \in$ {MXC$^+$, XC$^+$, MSC$^+$, SC$^+$, MXC, XC, MSC, SC} and (o_1, o_2, o_3) be the comparison triple of the options i and j. If $<$ (respectively, $>$) dominates (o_1, o_2, o_3), then the plan space of CliqueSquare using option i is included (respectively, includes) in the plan space of CliqueSquare using option j.*

	XC$^+$	MSC$^+$	SC$^+$	MXC	XC	MSC	SC
MXC$^+$	$(=,=,<)$	$(=,<,=)$	$(=,<,<)$	$(<,=,=)$	$(<,=,<)$	$(<,<,=)$	$(<,<,<)$
XC$^+$		$(=,<,>)$	$(=,<,=)$	$(<,=,>)$	$(<,=,=)$	$(<,<,>)$	$(<,<,=)$
MSC$^+$			$(=,=,<)$	$(<,>,=)$	$(<,>,<)$	$(<,=,=)$	$<,=,<)$
SC$^+$				$(<,>,>)$	$(<,>,=)$	$(<,=,>)$	$(<,=,=)$
MXC					$(=,=,<)$	$(=,<,=)$	$(=,<,<)$
XC						$(=,<,>)$	$(=,<,=)$
MSC							$(=,=,<)$
SC							

Table 4.1: Detailed relationships between decomposition options.

The reason for the above is that each comparison symbol encodes the relationship between the alternatives available to each algorithm. If neither $<$ nor $>$ dominates the comparison triple, it can be easily shown that the search spaces are incomparable (not included in one another). Table 4.1 shows the comparison triples for all pairs of decomposition options. The cell (row, col) corresponds to the comparison of option row and option col. The comparisons dominated by $<$ or $>$ (which entail a relationship between the respective search spaces) are highlighted.

$\square$

Optimization algorithm correctness. A legitimate question concerns the correctness of the CliqueSquare-SC, which has the largest search space: *for a given query q, does CliqueSquare-SC generate only plans from $\mathscr{P}(q)$, and all the plans from $\mathscr{P}(q)$?*

We first make the following remark. For a given query q and plan $p \in \mathscr{P}(q)$, it is easy to obtain a set of equivalent plans $p', p'', \ldots \in \mathscr{P}(q)$ by pushing projections and selections up and down. CliqueSquare optimization should not spend time enumerating p *and* such variants obtained out of p, since for best performance, σ and π should be pushed down as much as possible, just like in the traditional setting. We say two plans $p, p' \in \mathscr{P}(q)$ are *similar*, denoted $p \sim p'$, if p' can be obtained from p by moving σ and π up and down. We denote by $\mathscr{P}^{\sim}(q)$ the space of equivalence classes obtained from $\mathscr{P}(q)$ and the equivalence relation $\sim$. By a slight abuse of notations, we view $\mathscr{P}^{\sim}(q)$ to be a set of *representative* plans, one (arbitrarily chosen) from each equivalence class.

Based on this discussion, and to keep notations simple, *in the sequel we use $\mathscr{P}(q)$ to refer to $\mathscr{P}^{\sim}(q)$*, and we say an algorithm CliqueSquare-$\mathscr{A}$ is *correct* iff it is both *sound*, i.e., it produces only representatives of some equivalence classes from $\mathscr{P}^{\sim}(q)$, and *complete*, i.e., it produces a representative from every equivalence class of $\mathscr{P}^{\sim}(q)$.

Theorem 4.4.2 (CliqueSquare-SC correctness)**.** *For any query q, CliqueSquare-SC outputs the set of all the logical plans computing the answers to q: $\mathscr{P}_{\mathsf{SC}}(q) = \mathscr{P}(q)$.*

Proof. We first show that $\mathscr{P}_{\mathsf{SC}}(q) \subseteq \mathscr{P}(q)$ holds, before proving $\mathscr{P}(q) \subseteq \mathscr{P}_{\mathsf{SC}}(q)$.

Soundness. $\mathscr{P}_{\mathsf{SC}}(q) \subseteq \mathscr{P}(q)$ directly follows from our plan generation method starting from a sequence of variable graphs produced within CliqueSquare-SC (Section 4.4.2), by recursive **SC**-clique decompositions/reductions (Section 4.3.2).

Completeness. For proving $\mathscr{P}(q) \subseteq \mathscr{P}_{\mathsf{SC}}(q)$, consider any plan $p \in \mathscr{P}(q)$ and let us show that CliqueSquare-SC builds a plan $p' \in \mathscr{P}_{\mathsf{SC}}(q)$ similar to p ($p \sim p'$), i.e., disregarding the projection and selection operators. That is, we use p to refer to its *subplan* consisting of all its leaves (match operators) up to the last join operator, assuming all the σ and π operators have been pushed completely up.

The proof relies on three notions: (i) the height of a plan, (i) the subplans at a given level of a plan, and (iii) the equality of two plans up to a level.

For a given plan p, the *height* of p, denoted $h(p)$, is the largest number of successive join operators encountered in a root-to-leaf path of p; a *level l* of p is an integer between 0 and $h(p)$.

A subplan of p is a sub-DAG of p. For any subplan p' of p whose root is the node n, *p' is at level l*, for $0 \leq l \leq h(p)$ iff the longest n-to-leaf path is of size at most l, *and* the longest path from a direct parent of n (if any) to a leaf is of size at least $l+1$. In particular, the match operator leaves of a plan p are all the subplans of p at level 0, while p is its only subplan at level $h(p)$.

Finally, two plans are *equal up to a level* iff they have the same subplans at that level.

With the above notions in place, showing completeness amounts to proving the property:

$(*)$ *for any plan $p \in \mathscr{P}(q)$, CliqueSquare-SC produces a plan equal to p up to level $h(p)$.*

We prove this by induction on the level l of a sub-plan of p, as follows.

(Base) For $l = 0$, i.e., we consider p's leaves only, which are necessarily **match** operators, one for every triple pattern in q. Since CliqueSquare-SC is initially called with the variable graph G of q having a single triple per graph node, and with the empty queue *states*, any plan generated by CliqueSquare-SC using `createQueryPlan` has the same leaves as p. Therefore, any plan produced by CliqueSquare-SC is equal to p up to $l = 0$.

(Induction) Suppose that the above property $(*)$ holds up to level n, and let us show it also holds up to level $n+1$.

At level $n+1$, consider the new join operators that are not at level n. These operators correspond to the roots of subplans, i.e., of sub-DAGs of p, whose children are roots of sub-DAGs of p at level n. For any such new join operator $J_A(op_1, \ldots, op_m)(a_1, \ldots, a_n)$, consider a plan p' produced by CliqueSquare-SC that is equal to p up to level n (p' exists thanks to the induction hypothesis).

By construction, p' has been produced from the $states$ variables of CliqueSquare-SC in which the n^{th} variable graph G_q^n has one node per root of subplan of p at level n. Any new join operator $J_A(op_1, \ldots, op_m)(a_1, \ldots, a_n)$ introduced in p at level $n+1$ has as children $op_1, \ldots, op_m$ operators at level n. Since every operator $op_1, \ldots, op_m$ outputs the set of attributes A, the nodes corresponding to these operators form some cliques (as many as there are variables in A) in G_q^n. As, by definition, any such clique can be found by a SC clique decomposition, there exists a plan $p'' \in \mathscr{P}_{\text{SC}}(q)$ generated by CliqueSquare-SC from the $states$ variable whose first n variable graphs are equal to those from which p' has been generated, and whose $n+1^{th}$ graph has a node corresponding to $J_A(op_1, \ldots, op_m)(a_1, \ldots, a_n)$. Therefore, there exists a plan produced by CliqueSquare-SC that is equal to p up to $n+1$.

□

4.4.4 Height optimality and associated algorithm properties

To decrease response time in our parallel setting, we are interested in flat plans, i.e., having few join operators on top of each other. First, this is because flat plans enjoy the known parallelism advantages of bushy trees. Second, while the exact translation of logical joins into physical MapReduce-based ones (and thus, in MapReduce jobs) depends on the available physical operators, and also (for the first-level joins) on the RDF partitioning, it is easy to observe that overall, *the more joins need to be applied on top of each other, the more successive MapReduce jobs are likely to be needed by the query evaluation.* We define:

Definition 4.4.1 (Height optimal plan)**.** *Given a query q, a plan $p \in \mathscr{P}(q)$ is height-optimal (HO in short) iff for any plan $p' \in \mathscr{P}(q)$, $h(p) \leq h(p')$.*

We classify CliqueSquare algorithm variants according to their ability to build *height optimal plans*. Observe that the height of a CliqueSquare plan is exactly the number of graphs (states) successively considered by its function createQueryPlans, which, in turn, is the number of clique decompositions generated by the sequence of recursive CliqueSquare invocations which has lead to this plan. Notice that height optimal plans are not necessarily globally optimal. An example of this can be seen in Figure 4.4 depicting a height optimal plan generated by CliqueSquare-SC$^+$. The plan is height optimal but it is not globally optimal since it incurs a lot of redunandant processing (most notably joins J_f, J_g).

Definition 4.4.2 (HO-completeness)**.** *CliqueSquare-$\mathscr{A}$ is* height optimal complete *(HO-complete in short) iff for any query q, the plan space $\mathscr{P}_{\mathscr{A}}(q)$ contains all the HO plans of q.*

Definition 4.4.3 (HO-partial and HO-lossy)**.** *CliqueSquare-$\mathscr{A}$ is* height optimal partial *(HO-partial in short) iff for any query q, $\mathscr{P}_{\mathscr{A}}(q)$ contains* at least one *HO plan of q. An algorithm CliqueSquare-$\mathscr{A}$ which is not HO-partial is called HO-lossy.*

An HO-lossy optimization algorithm may find *no* HO plan for a query q_1, *some* HO plans for another query q_2 and *all* HO plans for query q_3. In practice, an optimizer should provide uniform guarantees for any input query. Thus, only HO-complete and HO-partial algorithms are of interest.

Table 4.2 classifies the eight CliqueSquare variants we mentioned, from the perspective of these properties.

Theorem 4.4.3 (CliqueSquare HO properties)**.** *The properties stated in Table 4.2 hold.*

HO-complete	SC
HO-partial	SC$^+$, MSC$^+$, MSC
HO-lossy	MXC$^+$, XC$^+$, MXC, XC

Table 4.2: HO properties of CliqueSquare algorithm variants.

Proof.

CliqueSquare-SC is HO-complete. This is a direct corollary of Theorem 4.4.2. Since for any query q, CliqueSquare-SC computes $\mathscr{P}(q)$, it therefore computes all the optimal plans for q.

CliqueSquare-SC$^+$ is HO-partial. First let us show that CliqueSquare-SC$^+$ is not HO-complete.

We show that SC$^+$ is not SO-complete based on the example of the query in Figure 4.8. SC$^+$ can produce only one plan for this query, joining $\{t_1, t_2\}$, and $\{t_2, t_3\}$ in the first level and then joining the resulting two intermediate relations in the next level. In contrast, SC is allowed to consider partial cliques, thus it may also build another HO plan as follows: choose as first cover $\{\{t_1, t_2\}, \{t_3\}\}$, and in the subsequent level join the result of $t_1 \bowtie t_2$ with t_3. SC$^+$ cannot build this plan.

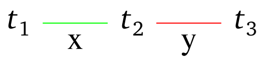


Figure 4.8: Query for which CliqueSquare-SC$^+$fails to find all HO plans.

Now, let us show that for any height optimal plan p for a query q, CliqueSquare-SC$^+$ computes a plan p' for q with the same height ($h(p) = h(p')$), therefore CliqueSquare-SC$^+$ is HO-partial. This follows from the HO-completeness of CliqueSquare-SC. Let p be any height optimal plan for a query q, built by CliqueSquare-SC. Consider the plan p' resulting from applying successfully to p the following changes:

1. pushing completely up the selection and projection operators;
2. starting from level 1 of p up to $h(p)$, replace *each* join operator resulting from a non-maximal clique for a given variable by the join operator obtained from the maximal version of this clique. Then add a projection operator on top of this maximal-clique join, to restrict its output to exactly the attributes that the original join used to return, and move this projection operator completely up.

Observe that p' also computes the answer to q (because no matter how much larger the newly introduced joins are, all the extra predicates that they bring were also going to be enforced in p) and that p and p' have the same height. Since p' is obtained from decompositions made of maximal-cliques only, p' is in the output of CliqueSquare-SC$^+$.

CliqueSquare-MSC is HO-partial. First let us show that CliqueSquare-MSC is not HO-complete.

Consider the query depicted in Figure 4.9. The only plan MSC produces for this query is shown in Figure 4.10. However, the plan shown in Figure 4.11 is also HO. This counterexample demonstrates that MSC is HO-partial.

Now, let us show that for any height optimal plan p for a query q, CliqueSquare-MSC computes a plan p' for q with the same height ($h(p) = h(p')$), therefore CliqueSquare-MSC is HO-partial.

We first introduce the following notions for a Join operator op at level l in a plan p:

– let $par(op)$ be the parents of op, for $1 < l \leq h(p)$, that is: the set of operators from level $l - 1$ that beget op, i.e., that are reachable from op within p.

– let $gp(op)$ be the grandparents of op, for $2 < l \leq h(p)$, that is: the set of operators from level $l - 2$ that beget op, i.e., that are reachable from op within p.

Let p be an HO plan produced by CliqueSquare-SC. (Recall that CliqueSquare-SC is HO-complete, thus it computes all the HO plans.) We next show that if *up to level* l, for $1 < l < h(p)$, the nodes of p result from an MSC decomposition based on the nodes one level below, then we can build from p a plan p' of the same height, computing the same output as p (thus computing q), and whose nodes are obtained from MSC clique decompositions *up to level* $l + 1$. Applying this process repeatedly on p, then on p' etc. eventually leads to an MSC plan computing q.

Observe that at the level $l = 1$, p has only Match operators for the input relations. Thus, at $l = 1$, any p produced by CliqueSquare-SC coincides with any HO plan, because the leaf operators are the same in all plans. Similarly, at $l = h(p)$, the plan p consists of a

$$\begin{tikzcd}
t_1 \arrow[r, dash, "\text{x}"'] & t_2 \arrow[r, dash, "\text{y}"'] & t_3 \arrow[r, dash, "\text{z}"'] & t_4
\end{tikzcd}$$

Figure 4.9: Query QX illustrating that minimum covers may lead to missing plans.

$$\begin{tikzcd}
 & & & J_y(xyzw) & & & \\
 & J_x(xy) \arrow[urr] & & & & J_z(yz) \arrow[ull] & \\
M_{t_1}(x) \arrow[ur] & & M_{t_2}(xy) \arrow[ul] & & M_{t_3}(yz) \arrow[ur] & & M_{t_4}(z) \arrow[ul]
\end{tikzcd}$$

Figure 4.10: Logical plan for the query QX using minimum-cover decompositions.

$$\begin{tikzcd}
 & & & J_y(xyzw) & & & \\
 & J_x(xy) \arrow[urr] & & J_y(xyz) \arrow[u] & & J_z(yz) \arrow[ull] & \\
M_{t_1}(x) \arrow[ur] & & M_{t_2}(xy) \arrow[ul] \arrow[ur] & & M_{t_3}(yz) \arrow[ul] \arrow[ur] & & M_{t_4}(z) \arrow[ul]
\end{tikzcd}$$

Figure 4.11: HO plan for the query QX obtained from non-minimum decompositions.

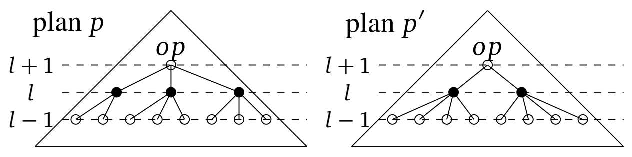


Figure 4.12: Modified operators (black nodes) related to op, between p and p'.

single Join operator, thus the level $h(p)$ *is* indeed obtained by an MSC clique decomposition.

Now, let us consider a level l, $1 < l < h(p)$, the first level of p from 2 up to $h(p) - 1$ *not* resulting from an MSC clique decomposition based on the operators at the previous level $l - 1$.

We build from p a plan p' computing q, with the same height, and resulting from MSC clique decompositions up to its level $l + 1$, as follows.

Let d be one of the MSC clique decompositions corresponding to level $l - 1$ of p. Let p' be a copy of the plan p up to $l-1$, and having at level l the Join operators corresponding to d. Observe that the connections between the operators from level $l - 1$ and l in p' are determined by d, as they are built by CliqueSquare's function `createQueryPlan`.

Now, let us show how to connect the level l of p' to (a copy of) the level $l+1$ of p, so as to make p' identical to p level-by-level between $l + 1$ and $h(p)$.

For every operator op in p' at level $l + 1$, which is identical to that of p, we connect op to a *minimal* subset of operators from level l in p', such that $gp(op)$ in p is a subset of $gp(op)$ in p'. Observe that the operators in $par(op)$ in p' are guaranteed to contain all the variables in op, because (i) any node has at most the variable present in all its parents (thus, grandparents etc.)[4] and (ii) op and $gp(op)$ were connected in p. Because all the input variables of op are provided by $par(op)$ in p', the join predicates encoded by op can be applied in p' exactly as in p (all operators at level $l+1$, are unchanged between p and p'). If the nodes in $par(op)$ in p' bring some variables not in p, we project them away prior to connecting the $par(op)$ operators to op in p'.

The plan p' satisfies the following: (i) it is syntactically correct, i.e., all operators have legal inputs and outputs, (ii) it computes q because, by construction, it is a CliqueSquare-SC HO-plan for q, and (iii) it is based on MSC decompositions up to level l (thus, one step higher than p). This concludes our proof.

CliqueSquare-MSC$^+$ is HO-partial. First let us show that CliqueSquare-MSC$^+$ is not HO-complete.

CliqueSquare-MSC$^+$ is not HO-complete because (i) CliqueSquare-MSC is not HO-complete and (ii) CliqueSquare-MSC$^+$ outputs a subset of the plans produced by CliqueSquare-MSC (Proposition 4.4.1).

Now, let us show that for any height optimal plan for a query q, CliqueSquare-MSC$^+$ computes a plan for q with same height, therefore it is HO-partial. This follows from the fact that CliqueSquare-MSC is HO-partial. Let p be any height optimal plan for a query q that is computed by CliqueSquare-MSC. Consider the plan p' resulting from applying successfully to p the following changes:

1. pushing completely up the selection and projection operators,
2. Starting from level 1 of p up to $h(p)$, replace each join operator resulting from a non-maximal clique for a given variable by the join operator resulting from the maximal version of this clique. Then add a projection operator on top of it to output the same relation as the previous join operator, and move this projection operator completely up.

Observe that p' computes the answer to the query, too, because no matter how much larger the newly introduced joins are, all predicates they bring are enforced at some point in p, too. Further, p and p' have the same height. Since p' is obtained from minimum decompositions (picked by CliqueSquare-MSC) now made of maximal-cliques only, p' is in the output of CliqueSquare-MSC$^+$.

MXC$^+$, XC$^+$, MXC and XC are HO-lossy. Consider the query shown in Figure 4.13. An exact cover algorithm cannot find an HO plan for this query. This is because the redundant processing introduced by considering simple (as opposed to exact) set covers may reduce the number of levels. For instance, using MSC$^+$, one can evaluate the query in Figure 4.13 in two levels: in the first level, the cliques $\{t_1, t_2\}$, $\{t_2, t_3\}$, $\{t_2, t_4\}$ are processed; in the second level, all the results are joined together using the common variables xyz. On the other hand, any plan built only from exact covers requires an extra level: t_2 is joined with the nodes of only one of its cliques, and thus, there is no common

4. We say "at most" because in p, some variables present in the parent may have been projected away at an upper level. However, for simplicity, we ignore projections throughout the proof; it is easy to see that one can first pull up all projections from p, then build p' out of p as we explain, and finally push back all necessary projections on p', with no impact on the number of levels.

variable among the rest of the triple patterns. This requires an extra level in order to finish processing the query.

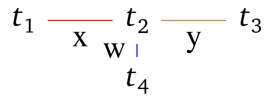


Figure 4.13: Query on which XC CliqueSquare variants are HO-lossy.

□

When MXC$^+$ and XC$^+$ fail. It turns out that the CliqueSquare algorithm based on the MXC$^+$ and XC$^+$ may fail to find *any* plan for some queries, such as the one shown in Figure 4.8. For this query, the maximal clique decomposition returns the cliques $\{t_1, t_2\}$, $\{t_2, t_3\}$, out of which no exact cover of the query nodes can be found. In turn, this translates into CliqueSquare-MXC$^+$ and CliqueSquare-XC$^+$ failing to find a query plan! Thus, we do not consider MXC$^+$ and XC$^+$ further.

4.4.5 Time complexity of the optimization algorithm

Through some simplification, we study the complexity of the CliqueSquare algorithm by focusing only on the total number of clique reductions performed. While there are many other operations involved, the decompositions applied by the algorithm are the main factor determining how many computations are overall performed.

Let n be the number of nodes in the variable graph, and let $T(n)$ denote the number of clique reductions. The size of the problem (number of nodes) reduces at every level (every recursive call); the reduction rate largely depends on the chosen decomposition method. Thus, we analyze the complexity of each of our eight option separately.

Recall that our decomposition methods can be classified in two major categories: (i) those based on minimum set covers only, and (ii) those using minimum or non-minimum covers.

Decompositions based on minimum covers. The decompositions using minimum set covers, namely MXC$^+$, MSC$^+$, MXC, and MSC, reduce the size of the graph by a factor of at least 2 at each call, as we explain below:

- Since we only consider connected query graphs, a graph of n nodes has at least $n-1$ edges.
- This graph admits at least one clique; if it has exactly one, the graph size is divided by n in one level of decomposition.
- At the other extreme, assuming each edge is labeled with a different variable; in this case, selecting $\lceil (n-1)/2 \rceil$ edges is guaranteed to lead to a minimum cover. Thus, in the next (recursive) call, the graph will have at most $\lceil (n-1)/2 \rceil$ nodes, and the reduction divides the size of the problem by a factor of 2.

We denote by $D(n)$ the number of possible decompositions. Given that at each step the algorithm performs as many reduction as there are possible decompositions, the following recurrence relation can be derived:

$$T(n) \leq D(n)T(\lceil (n-1)/2 \rceil) \tag{4.1}$$

where $T(1) = 1$.

Decompositions based on any covers. For the decompositions that are not using minimum covers, namely XC$^+$, SC$^+$, XC, SC, the size of the graph in the worst case is smaller by 1 (this follows from the Definition 4.3.3). In this case, the recurrence relation is:

$$T(n) \leq D(n)T(n-1) \tag{4.2}$$

where $T(1) = 1$.

The number of decompositions $D(n)$ depends on the graph and the chosen method. The first parameter affecting the number of decompositions $D(n)$ is the **set of cliques**.

Given a query q and its variable graph G_V, the join variables JV of q determine the number of maximal/partial cliques of the graph (we only consider non-trivial queries with at least one join variable, $|JV| \geq 1$).

Counting maximal cliques. The number of maximal cliques in the graph is equal to the number of join variables. When $|JV| = 1$, there is exactly one maximal clique.

Lemma 4.4.1. *A variable graph G_V has at most $2n+1$ maximal cliques.*

The proof is trivial since any conjunctive query q cannot have more than $2n+1$ distinct variables. Thus, the variable graph at any level does not have nodes that contain new variables so the number of maximal cliques is bound by the number of distinct variables existing in the query.

Counting partial cliques. Let $c\ell_u$ be the maximal clique corresponding to a variable $u \in JV$; from the definition of partial cliques it follows that the number of all non-empty partial cliques of $c\ell_u$ is equal to $2^{|cl_u|} - 1$. For two variables v_1, $v_2 \in JV$, the maximal cliques $c\ell_{v_1}$ and $c\ell_{v_2}$ may have some common nodes; in this case, some partial cliques for v_1 are also partial cliques for v_2.

To count the clique overlapping the following factor is introduced:

$$OF = \sum\nolimits_{v_1, v_2 \in JV : c\ell_{v_1} \cap c\ell_{v_2} \neq \emptyset} 2^{|c\ell_{v_1} \cap c\ell_{v_2}|}$$

Thus, the total *number of partial cliques* is given by:

$$\sum_{u \in JV} (2^{|cl_u|} - 1) - OF \tag{4.3}$$

Lemma 4.4.2. *A variable graph G_V has at most $2^n - 1$ partial cliques.*

The proof is trivial since even in the case where we can take all combinations of nodes as partial cliques this number cannot exceed the power set.

The second parameter that affects the number of decompositions $D(n)$ is the decomposition method. Below we establish the complexity of CliqueSquare algorithm for each decomposition method, considering the worst case scenario for $D(n)$.

Complexity of CliqueSquare-SC. Out of $2^n - 1$ partial cliques, we search for set covers of size at most $n-1$. The maximum number of decompositions (the maximum number of covers) satisfies:

$$D(n) \leq \sum_{k=1}^{n-1} \binom{2^n - 1}{k} \tag{4.4}$$

The equation above corresponds to the total number of sets that will be examined in CliqueDecomposition function. Only some of them are valid covers, thus the above is a very rough bound. This is easy to see for example considering the case where $k = 1$. The candidate covers according to the equation will be $2^n - 1$, but only one of them is really a cover.

Complexity of CliqueSquare-MSC. A variable graph of n nodes is sure to contain a cover of size $\lceil n/2 \rceil$. MSC decomposition searches for minimum covers in the graph, thus if there exists a cover with size $\lceil n/2 \rceil$ there is no need to consider bigger covers. Furthermore, a cover smaller than $\lceil n/2 \rceil$ may exist, but since we consider the worst case (where the number of decompositions is maximized), we rely on $\lceil n/2 \rceil$ as the size of the minimum set cover. Based on this, the number of decompositions satisfies:

$$D(n) \leq \binom{2^n - 1}{\lceil n/2 \rceil} \tag{4.5}$$

The worst case is constructed by taking the worst case for the number of cliques and the worst case for the size of the minimum set cover. These two worst cases, though, might never appear together in practice. For example, the $2^n - 1$ partial cliques appear when we have a single clique query, but in that case the minimum set cover is exactly one. On the other hand, a minimum set cover of size $n/2$ is typical for chain queries, for which the number of partial cliques is $2n - 1$.

Complexity of CliqueSquare-XC. When there are $2^n - 1$ partial cliques, there is exactly 1 maximal clique; in this case the exact cover problem is equivalent with the problem of finding the partitions of a set. The number of ways to partition a set of n objects into k non-empty subsets is described by *Stirling partition numbers of the second kind* and is denoted as $\left\{ {n \atop k} \right\}$. The total number of decompositions satisfies:

$$D(n) \leq \sum_{k=0}^{n-1} \left\{ {n \atop k} \right\} \tag{4.6}$$

Complexity of CliqueSquare-MXC. Given a variable graph G_V with n nodes, in the worst case the size of the minimum set cover is equal with $\lceil n/2 \rceil$ (the graph is connected). In addition we have seen that for exact cover decompositions the maximum number cannot exceed the number given by Equation 4.6. Using again the equivalence of the exact cover problem with set partitioning, we are interested in non-empty partitions of size $k = \lceil n/2 \rceil$. The total number of decompositions satisfies:

$$D(n) = \left\{ {n \atop \lceil n/2 \rceil} \right\} \tag{4.7}$$

Again, the above upper bound is based on two mutually exclusive worst cases: 1-clique queries for which we the number of exact covers is given by the Stirling numbers, and chain queries which maximize the size of the set cover ($n/2$).

Complexity of CliqueSquare-SC$^+$. A variable graph of n nodes has at most $2n+1$ maximal cliques (Lemma 4.4.1). Similarly to SC decompositions, we search for set covers of size at most $n-1$. The total number of decompositions satisfies:

$$D(n) \leq \sum_{k=1}^{n-1} \binom{2n+1}{k} \tag{4.8}$$

Complexity of CliqueSquare-MSC$^+$. As before the maximum number of maximal cliques is $2n+1$ (Lemma 4.4.1). Since the graph is connected, in the worst case, the size of the minimum set cover is $\lceil n/2 \rceil$. Thus, the total number of decompositions satisfies:

$$D(n) \leq \binom{2n+1}{\lceil n/2 \rceil} \tag{4.9}$$

Complexity of CliqueSquare-XC$^+$. Recall again Lemma 4.4.1. Due to the clique overlap, for every clique that is chosen, at least one other clique becomes ineligible. In practice the cliques may overlap even more but we are interested only in the worst case (the ones that generates the most decompositions). Assuming that there are $2n+1$ maximal cliques, we can only pick a maximum of $n+1$[5] cliques, since any extra selection will make some clique ineligible. From the above, we conclude that the total number of decompositions satisfies:

$$D(n) \leq \sum_{k=1}^{n-1} \binom{n+1}{k} \tag{4.10}$$

Complexity of CliqueSquare-MXC$^+$. Similar with XC$^+$ there are at most $n+1$ eligible cliques. Given that we are looking for a minimum cover, the size of the cover is constant (similar with the other options supporting minimum covers). Again the binomial coefficient is used to estimate the worst case and the total number of decompositions satisfies:

$$D(n) \leq \binom{n+1}{\lceil n/2 \rceil} \tag{4.11}$$

Table 4.3 summarizes the number of decompositions for each option when considering the worst case scenarios. Note that the worst cases are not reached by all algorithms on the same queries, therefore Figure 4.3 does not provide an easy way to compare the efficiency of the optimization variants. We have obtained interesting comparison results experimentally by testing all variants against a large set of synthetic queries (see Section 4.7.2).

5. Notice that $\lceil (2n+1)/2 \rceil = \lceil n + 1/2 \rceil = n+1$ since $n \in \mathbb{N}^+$

MXC$^+$	**MSC$^+$**	**MXC**	**MSC**	**XC$^+$**	**SC$^+$**	**XC**	**SC**
$\binom{n+1}{\lceil n/2 \rceil}$	$\binom{2n+1}{\lceil n/2 \rceil}$	$\left\{ {n \atop \lceil n/2 \rceil} \right\}$	$\binom{2^n-1}{\lceil n/2 \rceil}$	$\sum_{k=1}^{n-1} \binom{n+1}{k}$	$\binom{2n+1}{\lceil n/2 \rceil}$	$\sum_{k=0}^{n-1} \left\{ {n \atop k} \right\}$	$\sum_{k=1}^{n-1} \binom{2^n-1}{k}$

Table 4.3: Upper bounds on the complexity of CliqueSquare variants on a query of n nodes.

4.5 Storage

This section describes how CliqueSquare partitions (Section 4.5.1) and places RDF data in HDFS using MapReduce jobs (Section 4.5.2). In addition, it presents how to handle skew (Section 4.5.3) in the values of properties and briefly discusses some fault-tolerance issues (Section 4.5.4).

4.5.1 RDF partitioning

We start from the observation that the performance of MapReduce jobs suffers from shuffling large amounts of intermediate data between the map and reduce phases. Therefore, our goal is to partition and place RDF data so that the largest number of joins are evaluated at the map phase itself. This kind of joins are known as *co-located* or *partitioned* joins [RG03, ÖV11]. In the context of RDF, queries tend to involve many clique joins (e.g., subject-subject, subject-object, property-object, subject-subject-subject, subject-subject-property,etc.). Co-locating such joins as much as possible is therefore an important step towards efficient query processing.

By default HDFS replicates each dataset three times for fault-tolerance reasons. CliqueSquare exploits this data replication to partition and store RDF data in three different ways. In detail, it proceeds as follows.

1. We partition each triple and place it according to its subject, property and object values, as in [CFYM04]. Triples that share the same value in any position $(s\ p\ o)$ are located within the same compute node.
2. Then, unlike [CFYM04], we partition triples within each compute node based on their placement $(s\ p\ o)$ attribute. We call these partitions *subject*, *property*, and *object partition*. Notice that given a type of join, e.g., subject-subject join, this local partitioning allows for accessing fewer triples.
3. We further split each partition within a compute node by the value of the property in their triples. This property-based grouping has been first advocated in [HMM$^+$11] and also resembles the vertical RDF partitioning proposed in [AMMH07] for centralized RDF stores. Finally, we store each resulting partition into an HDFS file. By using the value of the property as the filename, we benefit from a finer-granularity data access during query evaluation.

CliqueSquare reserves a special treatment to triples where the property is rdf:type. In many RDF datasets, such statements are very frequent which, in our context, translates into an unwieldy large property partition corresponding to the value rdf:type. To avoid the performance problems this may entail, CliqueSquare splits the property partition of rdf:type into several smaller partitions, according to their object value. This enables working with finer-granularity partitions.

In contrast e.g., to Co-Hadoop [ETÖ$^+$11], which considers a single attribute for co-locating triple, our partitioner co-locates them on the three attributes (one for each data replica). This allows us to perform *all first-level joins* in a plan (s-s, s-p, s-o, s-s-s, s-s-o, etc.) *locally* in each compute node during query evaluation.

Below we describe CliqueSquare storage in terms of our storage description language (Definition 2.3.1, Section 2.3.1).

Storage description
HP(S){LP(P){SO}}+HP(P){LP(P){SO}}+HP(O){LP(P){SO}}

4.5.2 MapReduce partitioning process

CliqueSquare partitions RDF data in parallel for performance reasons. For this, it leverages the MapReduce framework and partitions input RDF data using a single MapReduce job. We describe the *map*, *shuffle*, and *reduce* phases of this job in the following.

Map phase. For each input triple (s_1 p_1 o_1), the map function outputs three key-$value$ pairs. CliqueSquare uses the triple itself as value of each output key-$value$ pair and creates composite keys based on the subject, property, and object values. The first part of the composite key is used for routing the triples to the reducers, while the second part is used for grouping them in the property-based files. In specific, CliqueSquare computes the three keys as follows: one key is composed of the subject and property values (i.e., $s_1|p_1$); one key is composed of the object and property values (i.e., $o_1|p_1$); and one key is composed of the property value itself (i.e., p_1), but, if p_1 is rdf:type, CliqueSquare then concatenates the object value to this key (i.e., rdf:type|o_1).

Shuffle phase. CliqueSquare uses a customized partitioning function to shuffle the key-$value$ pairs to reduce tasks based on the first part of the composite key. The reduce task (node) to which a key-value pair is routed is determined by hashing this part of the key. As a result, CliqueSquare sends any two triples having the same value x (as a subject, property, or object, irrespectively of where x appears in each of these two triples) to the same reduce task. Then, all triples belonging to the same reduce task are grouped by the second part of the composite key (the property value).

Reduce phase. The MapReduce framework then invokes the reduce function for each computed group of triples. The reduce function, in turn, stores each of these groups into a HDFS file (with a replication factor of one), whose file name is derived from the property value and a string token indicating if it is a subject, property, or object partition.

Algorithm 2 shows the pseudocode of the MapReduce job for partitioning data as explained above. Notice that since the property is included in the output key of the map function, we omit it from the value, in order to reduce the data transferred in the network and the data we store in HDFS.

Let us now illustrate our RDF partitioning approach (Algorithm 2) on the sample RDF graph of Figure 2.1 and a three-nodes cluster. Figure 4.14 shows the result after the rout-

Algorithm 2: RDFPartitioner job

```
1  Map(key, t)
       Input : File offset key; String value of a triple t
2      fileName ← ∅;
3      outputValue← ∅;
4      if t.property = "rdf:type" then
5          fileName ← t.property + "#" + t.object;
6          outputValue ← t.subject;
7      else
8          fileName ← t.property;
9          outputValue ← t.subject + t.object;
10     emit ((t.subject + "|" + fileName + "-S"), outputValue);
11     emit ((t.property + "|" + fileName + "-P"), outputValue) ;
12     emit ((v.object + "|" + fileName + "-O"), outputValue) ;
13 Reduce(key, triples)
       Input : Triple's attribute value|fileName; Collection of triples
14     file ← reducerID + fileName;
15     writeHDFS (triples,file);
```

ing of the shuffle phase. We underline the first part of the composite key used in the customized partitioning function. For example, the input triple (`:stud1 :takesCourse :db`) is sent: to node n_1 because of its subject value; to n_2 because of its object value; and to n_3 because of its property value. Next, each node groups the received triples based on the property part of the composite keys. Figure 4.15 shows the final result of the partitioning process assuming that the number of reduce tasks is equal to the number of nodes.

The advantage of our storage scheme is twofold. First, many joins can be performed locally during query evaluation. This is an important feature of our storage scheme as it reduces data shuffling during query processing and hence leads to improved query response times. Second, our approach strikes a good compromise between the generation of either too few or too many files. Indeed, one could have grouped all triples within a node (e.g., all triples on n_1 in Figure 4.15) into a single file. However, such files would have grown quite big and hence increase query response times. In contrast, the files stored by CliqueSquare have meaningful names, which can be efficiently exploited to load only the data relevant to any incoming query. Another alternative would be to omit the grouping by property values and create a separate file for each subject/property/object partition within a node. For instance, in our example, node n_2 has nine subject/property/object values (see underlined values in Figure 4.14) while only six files are located in this node (Figure 4.15). However, handling many small files would lead to a significant overhead within MapReduce jobs.

node n_1	**node** n_2	**node** n_3
(ub:stud1 ub:takesCourse ub:db)	(ub:stud1 ub:takesCourse ub:db)	(ub:prof1 ub:advisorOf ub:stud1)
(ub:stud1 ub:member ub:dept4)	(ub:stud1 ub:member ub:dept4)	(ub:prof1 ub:name "bob")
(ub:stud1 ub:name "ted")	(ub:dept1 rdf:type Dept)	(ub:prof2 ub:advisor ub:stud2)
(ub:prof1 ub:advisor ub:stud1)	(ub:stud2 ub:member ub:dept1)	(ub:prof2 ub:name "alice")
(ub:stud2 ub:takesCourse ub:os)	(ub:prof1 ub:name "bob")	(ub:stud1 ub:name "ted")
(ub:prof2 ub:advisor ub:stud2)	(ub:prof1 ub:advisor ub:stud1)	(ub:stud1 ub:name "ted")
(ub:stud2 ub:member ub:dept1)	(ub:prof2 ub:advisor ub:stud2)	(ub:prof1 ub:name "bob")
(ub:dept1 rdf:type ub:Dept)	(ub:stud2 ub:takesCourse ub:os)	(ub:prof2 ub:name "alice")
(ub:stud1 ub:member ub:dept4)	(ub:prof2 ub:name "alice")	(ub:stud1 ub:takesCourse ub:db)
(ub:stud2 ub:member ub:dept1)	(ub:dept1 type ub:Dept)	(ub:stud2 ub:takesCourse ub:os)
(ub:prof1 rdf:type ub:Professor)	(ub:prof1 rdf:type ub:Professor)	(ub:prof1 rdf:type ub:Professor)
(ub:prof2 rdf:type ub:Professor)	(ub:prof2 rdf:type ub:Professor)	(ub:prof2 rdf:type ub:Professor)
(ub:dept4 rdf:type ub:Dept)	(ub:dept4 rdf:type ub:Dept)	
	(ub:dept4 rdf:type ub:Dept)	

Figure 4.14: Data partitioning process: triples arriving at each node after the routing of the shuffle phase.

4.5.3 Handling skewness in property values

In practice, the frequency distribution of property values in RDF datasets is highly skewed, i.e., some property values are much more frequent than others [KOvH10, DKSU11]. Hence, some property-based files created by CliqueSquare may be much larger than others, degrading the global partitioning time due to unbalanced parallel efforts: processing them may last a long time after the processing of property files for non-frequent properties.

To tackle this, Map tasks in CliqueSquare keep track of the number of triples for each property file. When the number of triples reaches a predefined threshold, the Map task decides to split the file and starts sending triples into a new property file. For example, when the size of the property file `(takesCourse-P)` reaches the threshold, the Map task starts sending `(takesCourse)` triples into the new property file `(takesCourse-P_02)`, which may if necessary overflow into another partition `(takesCourse-P_03)` etc. The new property files end up to different Reduce tasks to ensure load balancing.

4.5.4 Fault-Tolerance

Fault-tolerance is one of the biggest strengths of HDFS as users do not have to take care of this issue for their applications. Fault-tolerance in HDFS is ensured through the replication of data blocks. If a data block is lost, e.g., because of a node failure, HDFS simply recovers the data from another replica of this data block. CliqueSquare also replicates RDF data three times. However, each replica is partitioned differently (based on the subject, property, and object). As a result, the copies of data blocks do not contain the same data. Consequently, some triples from the RDF data might be lost in case of a node failure. This is because such triples might belong to data blocks that were stored on the failing node.

Thus, fault-tolerance is a big challenge in this scenario. The trivial solution to the problem would be to set the replication factor of HDFS to a value greater than one. The problem with this approach would be that the data which are already triplicated in HDFS

node n_1		
1_takesCourse-S	1_member-S	1_advisor-O
(ub:stud1 ub:db)	(ub:stud1 ub:dept4)	(ub:prof1 ub:stud1)
(ub:stud2 ub:os)	(ub:stud2 ub:dept1)	(ub:prof2 ub:stud2)
1_name-S	1_type#Dep-O	1_member-P
(ub:stud1 "ted")	(ub:dept1)	(ub:stud1 ub:dept4)
	(ub:dept4)	(ub:stud2 ub:dept1)
1_type#Professor-O		
(ub:prof1)		
(ub:prof2)		

node n_2		
2_takesCourse-O	2_member-O	2_name-O
(ub:stud1 ub:db)	(ub:stud1 ub:dept4)	(ub:prof1 "bob")
(ub:stud2 ub:os)	(ub:stud2 ub:dept1)	(ub:prof2 "alice")
2_advisor-P	2_type#Dept-S	2_type#Dept-P
(ub:prof1 ub:stud1)	(ub:dept1)	(ub:dept1)
(ub:prof2 ub:stud2)	(ub:dept4)	(ub:dept4)
2_type#Professor-P		
(ub:prof1)		
(ub:prof2)		

node n_3		
3_advisor-S	3_name-S	3_name-P
(ub:prof1 ub:stud1)	(ub:prof1 "bob")	(ub:prof1 "bob")
(ub:prof2 ub:stud2)	(ub:prof2 "alice")	(ub:prof2 "alice")
		(ub:stud1 "ted")
3_name-O	3_takesCourse-P	3_type#Professor-S
(ub:stud1 "ted")	(ub:stud1 ub:db)	(ub:prof1)
	(ub:stud2 ub:os)	(ub:prof2)

Figure 4.15: Data partitioning process: triples in files at each node after the reduce phase.

would be multiplied even more depending on the specified value.

Another simple solution to this problem is to partition a computing cluster into three groups of computing nodes. Each group is responsible of storing a different replica. This would avoid losing triples in case of node failures. However, this does not avoid CliqueSquare to read a large number of data blocks to recover the failed data blocks (stored on the failing node). The database community recognizes this issue as a challenging and interesting problem. Hence, some research projects (e.g., Las Vegas Project[6]) already started to deal with this problem. This is an interesting research direction that we would like to investigate in the future.

6. `http://database.cs.brown.edu/projects/las-vegas/`

4.6 Plan evaluation on MapReduce

We now discuss the MapReduce-based evaluation of our logical plans. We first present the translation of logical plans into physical ones (Section 4.6.1) exploiting the Clique-Square storage (described in Section 4.5), then show how a physical plan is mapped to MapReduce jobs (Section 4.6.2) and finally introduce our cost model (Section 4.6.3).

4.6.1 From logical to physical plans

We define a *physical plan* as a rooted DAG such that (i) each node is a physical operator and (ii) there is a directed edge from op_1 to op_2 iff op_1 is a parent of op_2. To translate a logical plan, we rely on the following physical MapReduce operators:

– **Map Scan**, $MS[regex]$, parameterized by a regular expression[7] $regex$, outputs one tuple for each line of every file in HDFS that matches the regular expression $regex$.

– **Filter**, $\mathscr{F}_{con}(op)$, where op is a physical operator, outputs the tuples produced by op that satisfy logical condition con.

– **Map Join**, $MJ_A(op_1, \dots, op_n)$, is a directed join [BPE+10] that joins its n inputs on their common attribute set A.

– **Map Shuffler**, $MF_A(op)$, is the repartition phase of a repartition join [BPE+10] on the attribute set A; it shuffles each tuple from op on A's attributes.

– **Reduce Join**, $RJ_A(op_1, \dots, op_n)$, is the join phase of a repartition join [BPE+10]. It joins n inputs on their common attribute set A by (i) gathering the tuples from $op_1, \dots, op_n$ according to the values of their A attributes, (ii) building on each compute node the join results.

– **Project**, $\pi_A(op)$, is a simple projection (vertical filter) on the attribute set A.

We translate a logical plan p_l into a physical plan, operator by operator, from the bottom (leaf) nodes up, as follows.

match: Let M_{tp} be a match operator (a leaf in p_l), having $k \geq 1$ outgoing (parent-to-child) edges.

1. For each such outgoing edge e_j of M_{tp}, $1 \leq j \leq k$, we create a Map Scan operator $MS[regex]$ where $regex$ is computed based on the triple pattern tp and the target operator of edge e_j. The $regex$ is specified in Table 4.4. The table contains all the possible forms of a single triple pattern and the variables that may be requested from the child operator of $MS[regex]$ (target of edge e_j). If the query is composed only from a single triple pattern then any of the regular expressions appearing in the

7. A regular expression is a sequence of characters that define a search pattern, mainly for use in pattern matching with strings, or string matching.

Triple Pattern	$?s$	$?p$	$?o$
$(?s\ ?p\ ?o)$	$*-S$	$*-P$	$*-O$
$(?s\ ?p\ o)$	$*-S$	$*-P$	
$(?s\ p\ ?o)$	$*p-S$		$*p-O$
$(?s\ p\ o)$	$*p-S$		
$(s\ ?p\ ?o)$		$*-P$	$*-O$
$(s\ ?p\ o)$		$*-P$	
$(s\ p\ ?o)$			$*p-O$
$(s\ p\ o)$			

Table 4.4: Regular expressions identifying which files (from the HDFS) are scanned given a triple pattern and a variable requested by the child operator.

columns $?s$, $?p$, $?o$ of Table 4.4 can be used without compromising performance. For the existential query that is composed from the triple pattern $(s\ p\ o)$ we heuristically choose the file $H[s]\_p-S$ where $H[s]$ is the hash of the attribute s.

2. If the triple pattern tp has a constant in the subject and/or object, a *filter* operator $\mathcal{F}_{con}$ is added on top of $MS[regex]$, where con is a predicate constraining the subject and/or object as specified in tp. Observe the filter on the property, if any, has been applied through the computation of the $regex$ expression.

join: Let J_A be a logical join then two cases may occur.

1. J_A is a first level join (i.e., none of its ancestors is a join operator) and is transformed into a Map Join, MJ_A.
2. J_A is not a first level join and is transformed into a Reduce Join, RJ_A; a Reduce join cannot be performed directly on the output of another Reduce Join, thus a Map Shuffler operator is added, if needed.

select: σ_c operator is mapped directly to the $\mathcal{F}_c$ physical operator.
project: π_A operator is mapped directly to the respective physical operator.

For illustration, Figure 4.16 depicts the physical plan of Q1 built from its logical plan shown in Figure 4.5. Only the right half of the plan is detailed since the left side is symmetric.

4.6.2 From physical plans to MapReduce jobs

As a final step, we map a physical plan to MapReduce programs as follows:

– projections and filters are always part of the same MapReduce task as their parent operator;
– Map joins along with all their ancestors are executed in the same Map task, (iii) any other operator is executed in a MapReduce task of its own. The MapReduce tasks are grouped in MapReduce jobs in a bottom-up traversal of the task tree.

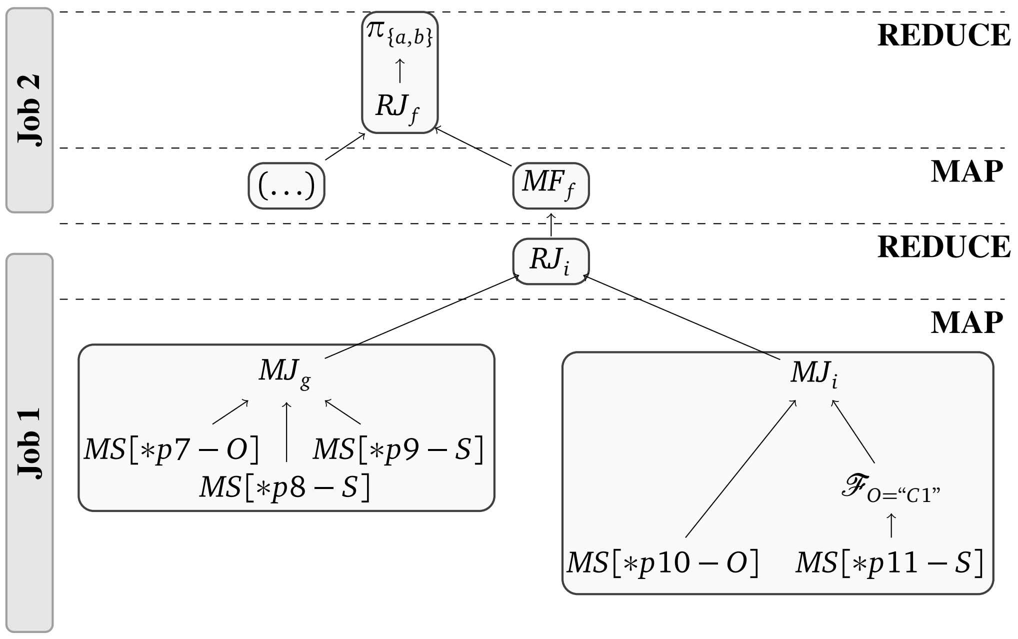


Figure 4.16: Part of Q1 physical plan and its mapping to MapReduce jobs.

Figure 4.16 shows how the physical plan of Q1 is transformed into a MapReduce program (i.e., a set of MapReduce jobs); rounded boxes show the grouping of physical operators into MapReduce tasks.

4.6.3 Cost model

We now define the cost $c(p)$ of a MapReduce query plan p, which allows us choosing a query plan among others, as an estimation of the total work $tw(p)$, required by the MapReduce framework, to execute p: $c(p) = tw(p)$. The total work accounts for (i) scan costs, (ii) join processing costs, (iii) I/O incurred by the MapReduce framework writing intermediary results to disk, and (iv) data transfer costs.

Observe that for full generality, our cost model takes into account many aspects (and not simply the plan height). Thus, while some of our algorithms are guaranteed to find plans as flat as possible, priority can be given to other plan metrics if they are considered important. In our experiments, the selected plans (based on this general cost model) were HO for all the queries but one (namely Q14).

To estimate $tw(p)$, we introduce the *own costs* of each operator op as follows: $c_{io}(op)$ is the I/O cost of operator op, $c_{cpu}(op)$ is its CPU incurred cost, while $c_{net}(op)$ is its data transfer cost.

The cost of a Map scan operator, MS, is mainly the I/O operations for reading the corresponding file from HDFS. The cost of a filter operator, $\mathscr{F}_c$, is mainly the CPU cost for checking whether condition c is satisfied. π_A operator involves only CPU processing for removing the appropriate attributes in A. Regarding the cost of a Map shuffler operator, MF_A, the I/O cost for reading intermediate results from the HDFS is measured, as well as the I/O cost for forwarding the results to the reducer (writing the results to disk). The cost of a Map join operator, MJ_A, occurs from CPU operations for joining locally the input relations on attributes A and from I/O writes to disk (the joining results are written to disk before shuffling). Finally, a Reduce join operator, RJ_A, entails network load for transferring the intermediate results to the reducers for attributes A, CPU cost for the computation of the join results and I/O cost for writing the results to disk. To sum up, the cost of each operator consists of the following individual costs:

- $c(MS) = c_{io}(MS)$
- $c(\mathscr{F}_{con}) = c_{cpu}(\mathscr{F}_{con})$
- $c(\pi_A) = c_{cpu}(\pi_A)$
- $c(MF_A) = c_{io}(MF_A)$
- $c(MJ_A) = c_{cpu}(MJ_A) + c_{io}(MJ_A)$
- $c(RJ_A) = c_{net}(RJ_A) + c_{cpu}(RJ_A) + c_{io}(RJ_A)$

The individual costs can be estimated as follows:

- $c_{io}(MS[regex]) = \sum_{f \in regex} |f| \times c_{read}$
- $c_{io}(MF_A(op)) = |op| \times c_{read} + |op| \times c_{write}$
- $c_{io}(MJ_A[op_1, \ldots, op_n]) = |op_1 \bowtie_A \ldots \bowtie_A op_n| \times c_{write}$
- $c_{io}(RJ_A[op_1, \ldots, op_n]) = |op_1 \bowtie_A \ldots \bowtie_A op_n| \times c_{write}$
- $c_{cpu}(\mathscr{F}_{con}(op)) = |op| \times c_{check}$
- $c_{cpu}(\pi_A(op)) = |op| \times c_{check}$
- $c_{cpu}(MJ_A(op_1, \ldots, op_n]) = c_{join}(op_1 \bowtie_A \ldots \bowtie_A op_n)$
- $c_{cpu}(RJ_A(op_1, \ldots, op_n)) = c_{join}(op_1 \bowtie_A \ldots \bowtie_A op_n)$
- $c_{net}(RJ_A(op_1, \ldots, op_n)) = (|op_1| + \ldots + |op_n|) \times c_{shuffle}$

where $|R|$ denotes the cardinality of relation R. c_{read} and c_{write} is the time to read and write one tuple from and to disk, respectively. $c_{shuffle}$ represents the time to transfer one tuple from one node to another through the network and $c_{join}(op_1 \bowtie_A ... \bowtie_A op_n)$ the cost of the join. Finally c_{check} is the time spend on performing a simple comparison on a part of the tuple.

While the performance of a MapReduce program can be modeled at much finer granularity [JOSW10, HDB11], the simple model above has been sufficient to guide our optimizer well, as our experiments demonstrate next.

4.7 Experimental evaluation

We have implemented the CliqueSquare optimization algorithms together with our partitioning scheme, and the physical MapReduce-based operators in a prototype we onward refer to as CSQ. We present our experimental setting in Section 4.7.1. Then, in Section 4.7.2, we perform an in-depth evaluation of the different optimization algorithms presented in Section 4.4.3 to identify the most interesting ones. In Section 4.7.3, we are interested in the performance (execution time) of the best plans recommended by our CliqueSquare optimization algorithms, and compare it with the runtime of plans as created by previous systems: linear or bushy, but based on binary joins. Finally, in Section 4.7.4, we compare CSQ query evaluation times with those of two state-of-the-art MapReduce-based RDF systems and show the query robustness of CSQ.

4.7.1 Experimental setup

Cluster. Our cluster consists of 7 nodes, where each node has: one 2.93GHz Quad Core Xeon processor with 8 threads; 4×4GB of memory; two 600GB SATA hard disks configured in RAID 1; one Gigabit network card. Each node runs CentOS 6.4. We use Oracle JDK v1.6.0_43 and Hadoop v1.2.1 for all experiments with the HDFS block size set to 256MB.

Dataset and queries. We rely on the LUBM [GPH05] benchmark, since it has been extensively used in similar works such as [HMM+11, HAR11, ZYW+13, PKT+13]. We use the LUBM10k dataset containing approximately 1 billion triples (216 GB). The LUBM benchmark features 14 queries, most of which return an empty answer if RDF reasoning (inference) is not used. Since reasoning was not considered in prior MapReduce-based RDF databases [HAR11, PKT+13, LL13], to evaluate these systems either the queries were modified, or empty answers were accepted; the latter contradicts the original benchmark query goal. We modified the queries as in [PKT+13] replacing generic types (e.g., <Student>, of which no explicit instance exists in the database) with more specific ones (e.g., <GraduateStudent> of which there are some instances). Further, the benchmark queries are relatively simple; the most complex one consists of only 6 triple patterns. To complement them, we devised other 11 LUBM-based queries with various selectivities and complexities, and present them next to a subset of the original ones to ensure variety across the query set. The complete workload can be found in Appendix A.

4.7.2 Plan spaces and CliqueSquare variant comparison

We compare the 8 variants of our CliqueSquare algorithms w.r.t. : (i) the total number of generated plans, (ii) the number of height-optimal (HO) plans, (iii) their running time, and (iv) the number of duplicate plans they produce.

Setup. We use the generator of [GKLM11] to build 120 synthetic queries whose shape is either *chain*, *star*, or *random*, with two variants *thin* or *dense* for the latter: dense ones have many variables in common across triples, while thin ones have significantly less, thus they are close to chains. The queries have between 1 and 10 (5.5 on average) triple patterns. Each algorithm was stopped after a time-out of 100 seconds.

Comparison. Figure 4.17 shows the *search space size* for each algorithm variant and query type. The total number of generated plans is measured for each query and algorithm; we report the average per query category. As illustrated in Section 4.4.3, MXC$^+$and XC$^+$ fail to find plans for some queries (thus the values smaller than 1). SC and XC return an extremely large number of plans, whose exploration is impractical. For these reasons, MXC$^+$, XC$^+$, XC, and SC are not viable alternatives. In contrast, MSC$^+$, SC$^+$, MXC, and MSC produce a reasonable number of plans to choose from.

Figure 4.18 shows the average *optimality ratio* defined as the number of HO-plans divided by the number of all produced plans. We consider this ratio to be 0 for queries for which no plan is found. While the ratio for MSC$^+$, MXC, and MSC is 100% for this workload (i.e., they return *only* HO plans), this is not guaranteed in general. SC$^+$ has

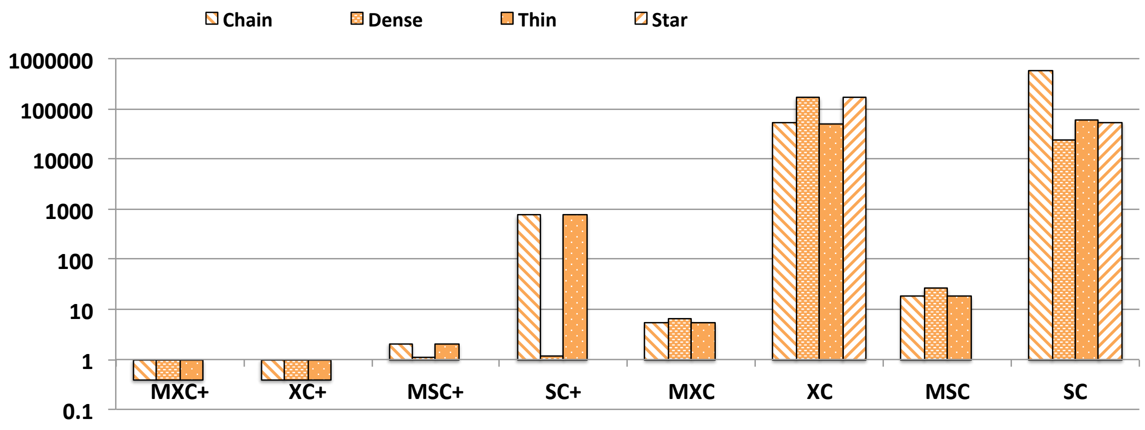


Figure 4.17: Average number of plans per algorithm and query shape.

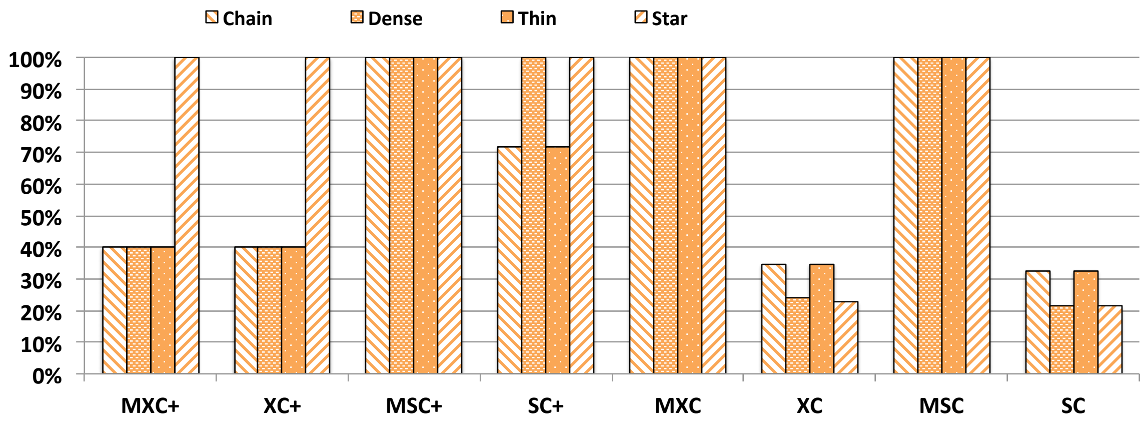


Figure 4.18: Average optimality ratio per algorithm and query shape.

a smaller optimality ratio but still acceptable. On the contrary, although XC finds some optimal plans, its ratio is relatively small.

Options MSC$^+$, MXC, and MSC lead to the shortest *optimization time* as shown in Figure 4.19. MSC is the slowest among these three algorithms, but it is still very fast especially compared to a MapReduce program execution, providing an answer in less than $1s$.

Given that our optimization algorithm is not based on dynamic programming, it may end up producing the same plan more than once. In Figure 4.20 we present the average *uniqueness ratio*, defined as the number of unique plans divided by the total number of produced plans. Dense queries are the most challenging for all algorithms, since they allow more sequences of decompositions which, after a few steps, can converge to the same (and thus, build the same plan more than once). However, in practice, as demonstrated in the figure, our dominant decomposition methods, MSC$^+$, MXC, and MSC produce very few duplicate plans.

Summary. Based on our analysis, the optimization algorithms based on MSC$^+$, MXC,

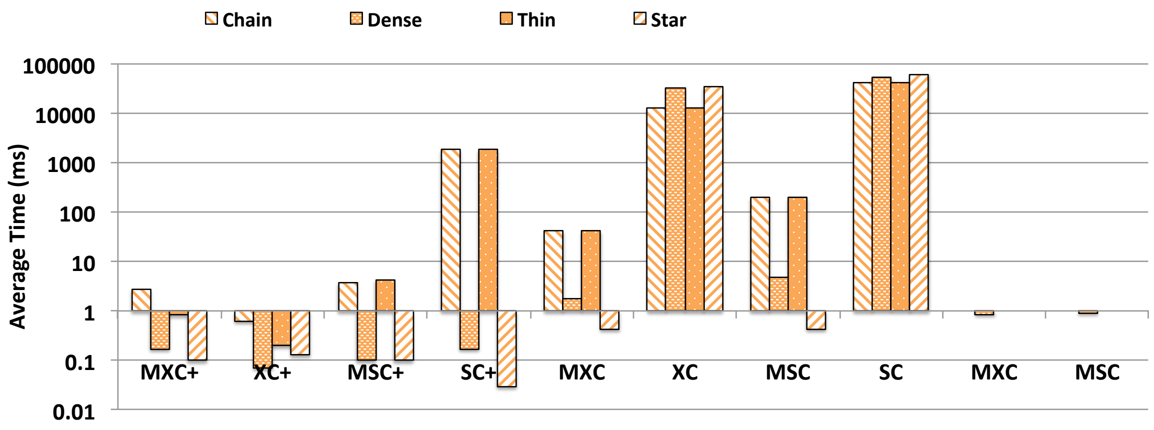


Figure 4.19: Average optimization time (ms) per algorithm and query shape.

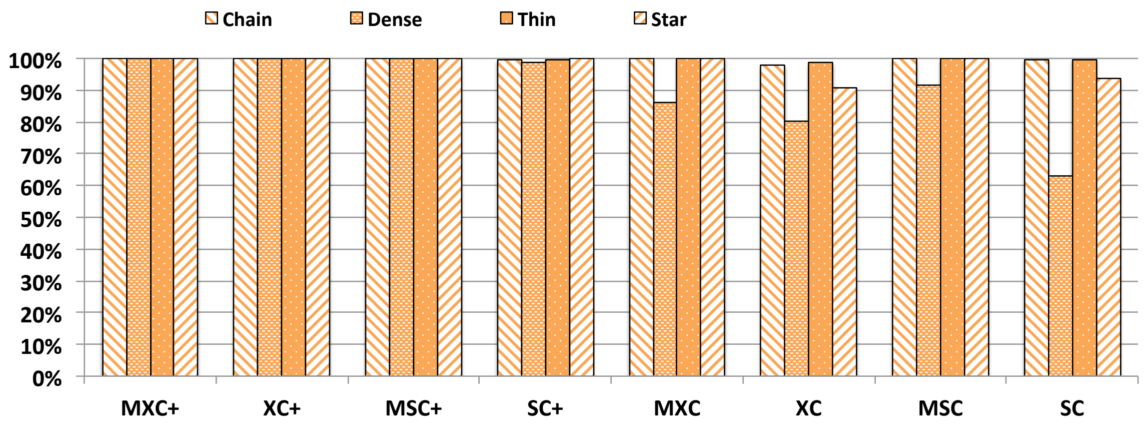


Figure 4.20: Average uniqueness ratio per algorithm and query shape.

and MSC return sufficiently many HO plans to chose from (with the help of a cost model), and produce these plans quite fast (in less than one second, negligible in an MapReduce environment). However, Theorem 4.4.3 stated that MXC is HO-lossy; therefore, we do not recommend relying on it in general. In addition, recalling (from Theorem 4.4.1) that the search space of MSC is a superset of those of MSC$^+$, and given that the space of CliqueSquare-MSC is still of reasonable size, we consider it the best CliqueSquare algorithm variant, and we rely on it exclusively for the rest of our evaluation.

4.7.3 CliqueSquare plans evaluation

We now measure the practical interest of the flat plans with n-ary joins built by our optimization algorithm.

Plans. We compare the plan chosen by our cost model among those built by CliqueSquare-MSC, against the *best binary bushy* plan and the *best binary linear* plan for each query. To find the best binary linear (or bushy) plan, we build them all, and then select the cheap-

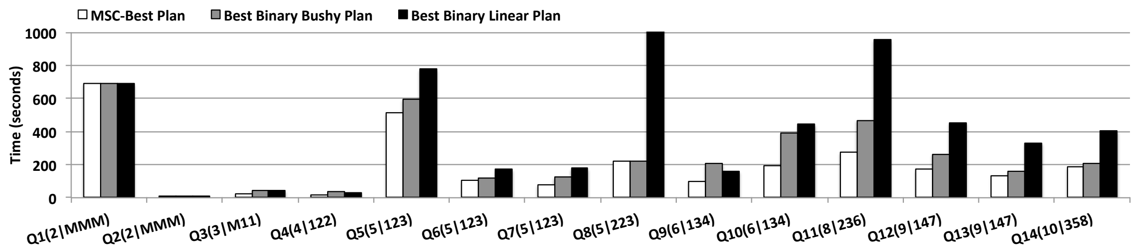


Figure 4.21: Plan execution time (in seconds) comparison for LUBM10k: MSC-plans, bushy-plans, and linear-plans.

est using the cost function described in Section 4.6.3. We translate all logical plans into MapReduce jobs as described in Section 4.6 and execute them on our CSQ prototype.

Comparison. Figure 4.21 reports the execution times (in seconds) for 14 queries (ordered from left to right with increasing number of triple patterns). In the x-axis, we report, next to the query name, the number of triples patterns followed (after the | character) by the number of jobs that are executed for each plan (where M denotes a map only job). For example, Q3(3|M11) describes query Q3, which is composed of 3 triple patterns, and for which MSC needs a map only job while the bushy and linear plans need 1 job each. The optimization time is not included in the execution times reported. This strongly favors the bushy and linear approaches, because the number of plans to produce and compare is bigger than that for MSC.

For all queries, the MSC plan is faster than the best bushy plan and the best linear plan, by up to a factor of 2 (for query Q9) compared to the binary bushy ones, and up to 16 (for query Q8) compared to the linear ones. The three plans for Q1 (resp. Q2) are identical since the queries have 2 triple patterns. For Q8, the plan produced with MSC is the same as the best binary bushy plan, thus the execution times are almost identical. As expected the best bushy plans run faster than the best linear ones, confirming the interest of parallel (bushy) plans in a distributed MapReduce environment.

Summary. CliqueSquare-MSC plans outperform the bushy and linear ones, demonstrating the advantages of the n-ary star equality joins it uses.

4.7.4 CSQ system evaluation

We now analyze the query performance of CSQ with the MSC algorithm and run it against comparable massively distributed RDF systems, based on MapReduce. While some memory-based massively distributed systems have been proposed recently [ZYW+13, GSMT14], we chose to focus on systems comparable with CSQ in order to isolate as much as possible the impact of the query optimization techniques that are the main focus of this work.

Systems. We pick SHAPE [LL13] and H_2RDF+ [PKT+13], since they are the most efficient RDF platforms based on MapReduce; the previous HadoopRDF [HMM+11] is largely outperformed by H_2RDF+ [PKT+13] and [HAR11] is outperformed by [LL13].

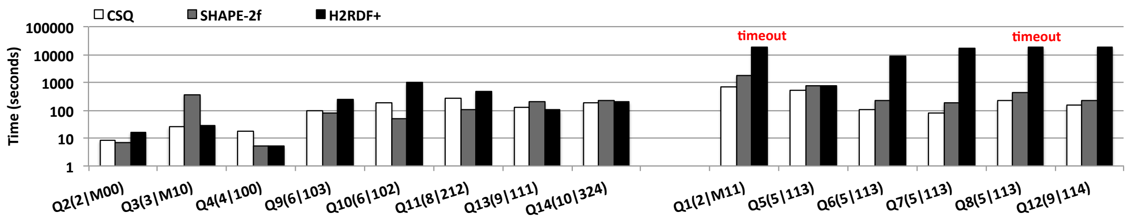


Figure 4.22: Query evaluation time (in seconds) comparison for LUBM10k: CSQ, SHAPE and H_2RDF+.

H_2RDF+ is open source, while we used our own implementation of SHAPE. SHAPE explores various partitioning methods, each with advantages and disadvantages. We used their 2-hop forward partitioning ($2f$) since it has been shown to perform the best for the LUBM benchmark. More details about the aforementioned systems can be found in Section 2.3.

Comparison. While CSQ stores RDF partitions in simple HDFS files, H_2RDF+ uses HBase, while SHAPE uses RDF-3X [NW10]. Thus, SHAPE and H_2RDF+ benefit from index access *locally* on each compute node, while our CSQ prototype can only scan HDFS partition files. We consider two classes of queries: *selective* queries (which on this 1 billion triple database, return less than 0.5×10^6 results) and *non-selective* ones (returning more than 7.5×10^6 results).

Figure 4.22 shows the running times: selective queries at the left, non-selective ones at the right. As before, next to the query name we report the number of triple patterns followed by the number of jobs that the query needs in order to be executed in each system (M denotes one map only job). H_2RDF+ sometimes uses map-only jobs to perform first-level joins, but it performs each join in a separate MapReduce job, unlike CSQ (Section 4.6).

Among the 14 queries of the workload, four (Q2, Q4, Q9, Q10) are PWOC for SHAPE (not for CSQ) and one (Q3) is PWOC for CSQ (not for SHAPE). These five queries are selective, and, as expected, perform better in the system which allows them to be PWOC. For the rest of the queries, where the optimizer plays a more important role, CSQ outperforms SHAPE for all but one query (Q11 has an advantage with $2f$ partitioning since a larger portion of the query can be pushed inside RDF-3X). The difference is greater for non-selective queries since a bad plan can lead to many MapReduce jobs and large intermediary results that affect performance. Remember that the optimization algorithm of SHAPE is based on heuristics without a cost function and produces *only one* plan. The latter explains why even for selective queries (like Q13 and Q14 which are more complex than the rest) CSQ performs better than SHAPE.

We observe that CSQ significantly outperforms H_2RDF+ for all the non-selective queries and for most of the selective ones, by 1 to more than 2 orders of magnitude. For instance, Q7 takes 4.8 hours on H_2RDF+ and only 1.3 minutes on CSQ. For queries Q1 and Q8 we had to stop the execution of H_2RDF+ after 5 hours, while CSQ required only 3.6 and 11 minutes, respectively. For selective queries the superiority of CSQ is less but it still outperforms H_2RDF+ by an improvement factor of up to 5 (for query Q9). This is because H_2RDF+ builds left-deep query plans and does not fully exploit parallelism; H_2RDF+ requires more jobs than CSQ for most of the queries. For example, for query Q12 H_2RDF+ initiates four jobs one after the other. Even if the first two jobs are map-only, H_2RDF+ still needs to read and write the intermediate results produced and pay the initialization overhead of these MapReduce jobs. In contrast, CSQ evaluates Q12 in a single job.

Summary. While SHAPE and H_2RDF+ focus mainly on data access paths techniques and thus perform well on selective queries, CSQ performs closely (or better in some cases), while it outperforms them significantly for non-selective queries. CSQ evaluates our complete workload in 44 minutes, while SHAPE and H_2RDF+ required 77 min and 23 hours, respectively. We expect that such systems can benefit from the logical query plans built by CliqueSquare to obtain fewer jobs and thus, lower query response times.

4.8 Conclusion

This chapter presented CliqueSquare, a distributed RDF data management system built on top of Hadoop. Our work focused on the *logical optimization of large conjunctive (BGP) RDF queries*, featuring many joins. We are interested in building *flat* logical plans to diminish query response time, and investigate the usage of *n-ary (star) equality joins* for this purpose.

We have presented a generic optimization algorithm and eight variants thereof, which build tree- or DAG-shaped plans using n-ary star joins and we have formally characterized their ability to find the flattest possible plans. We have explored a generic storage strategy suitable for storing RDF data in HDFS (Hadoop's Distributed File System) and provided algorithms for translating logical plans to physical plans and subsequently to MapReduce jobs.

The experimental results demonstrated the efficiency and effectiveness of our best variant CliqueSquare-MSC as well as the overall performance of the system against comparable systems.

The logical optimization approach proposed in this chapter is general enough and can be used in any massively parallel conjunctive query evaluation setting, contributing to shorten query response time.

Acknowledgments. The physical operators of the system as well as the physical plan parser were developed in collaboration with Benjamin Djahandideh, engineer in our group since October 2013. In addition, significant refactoring of the code has been performed by him before the open-source release.

Chapter 5

Conclusion and Future Work

The Semantic Web emerged as an attempt to improve the interaction of systems with data, so that they are able to reason efficiently about the information they encounter. The main building blocks of the Semantic Web is the RDF data model and the SPARQL query language. In order to make machines capable of reasoning about information, the latter has to be under a common data model, RDF in our case. Consequently, to process information expressed using the RDF data model the standard way is through the use of SPARQL.

The size of the Semantic Web grows rapidly, thus necessitates the design and implementation of massively parallel distributed systems for RDF data management. Towards this direction, cloud-computing provides the ideal infrastructures for building scalable, fault-tolerant, and elastic systems able to cover the unpredictable user requirements.

In this thesis, we have studied various architectures for storing and retrieving RDF data using cloud-based services and technologies. We have explored the performance and cost of warehousing RDF data into commercial cloud infrastructures, examining different storage and processing strategies. Furthermore, we focused on the optimization and parallelization of RDF queries, in order to exploit the opportunities offered by massively parallel frameworks.

5.1 Thesis summary

This thesis provides solutions for warehousing RDF data using cloud-based environments focusing on two different problems that we summarize below.

Warehousing Semantic Web data using commercial cloud services. We presented AMADA, an architecture for RDF data management using public cloud infrastructures.

- We proposed an architecture for storing RDF data within the Amazon cloud that shows a good behavior, both in terms of query processing time and monetary costs.
- We considered hosting RDF data in the cloud, and its efficient storage and querying through a (distributed, parallel) platform also running in the cloud.
- We exploited RDF indexing strategies that allow to direct queries to a (hopefully

tight) subset of the RDF dataset which provide answers to a given query, thus reducing the total work entailed by query execution.
- We provided extensive experiments on real RDF datasets validating the feasibility of our approach and giving insight about the monetary cost of storing and querying RDF data in commercial clouds.

Building massively-parallel (flat) plans for RDF queries. We presented CliqueSquare, an optimization approach for building massively parallel flat plans for RDF queries.
- We described a search space of logical plans obtained by relying on *n-ary (star) equality joins*. The interest of such joins is that by aggressively joining many inputs in a single operator, they allow building flat plans.
- We provided a novel generic algorithm, called CliqueSquare, for exhaustively exploring this space. We presented a thorough analysis of eight concrete variants of this algorithm, from the perspective of their ability to find one of (or all) the flattest possible plans for a given query.
- We showed that the variant we call CliqueSquare-MSC is the most interesting one, because it develops a reasonable number of plans and is guaranteed to find some of the flattest ones.
- We have fully implemented our algorithms and validate through experiments their practical interest for evaluating queries on very large distributed RDF graphs. For this, we relied on a set of relatively simple parallel join operators and a generic RDF partitioning strategy, which makes no assumption on the kinds of input queries. We show that CliqueSquare-MSC makes the optimization process efficient and effective even for complex queries.

5.2 Ongoing work

In Section 2.3, we have examined a wide variety of RDF data management systems. At the core of such platforms lies a specific strategy for partitioning, indexing and storing the data (inside one or many nodes). The optimization module of these systems is tightly connected with the underlying storage. Making the slightest changes to the storage (e.g., introducing a new index) requires also modifying the optimizer to consider these changes. If the changes are not propagated to the optimizer then in the best case we may experience performance degradation while in the worst case the correctness of the results may be compromised. In addition, lately there have been discussions [AÖD14] for systems [AÖDH15] where the internal storage changes on the fly. Such systems require the optimizer to be able to adapt automatically to build valid and efficient plans.

Furthermore, the optimization procedure in distributed systems heavily relies on the partitioning of the data. Finding which partitions should be accessed and detecting possible co-partition joins [LOÖW14] are usually choices hard-coded inside the optimizer. If the partitioning changes, these choices have to be revisited. We envision a generic optimization algorithm which can build efficient query plans for RDF queries over *any* storage as long as the storage model is formally described.

In order to be able to build efficient query plans over any possible storage, accurate information about how the data are stored (partitions, indexes, etc.) is needed. In Section 2.3, we have introduced the storage description grammar (Definition 2.3.1) as an attempt to describe the storage of various RDF data management platforms (distributed or not) under a common model. Although this grammar is useful for providing a meaningful overview of a system, the absence of formal semantics and the high level of abstraction prevent us from using it directly. As an alternative, we are working on defining a formal model relying on well-established concepts from relational databases. More precisely, we are looking into defining the storage model using parameterized conjunctive query views [PGGU95], able to capture the various access patterns as well as the partitioning of the data. We outline this proposal below.

An example of a storage description, modeling the CliqueSquare storage (Section 4.5), can be seen in Figure 5.1. The relation $T(s, p, o)$ models the RDF graph (the triple table) and the relation $H_{10}(x, y)$ models a hash function that given an input x outputs its hash value y, which is an integer number in the range of one to ten. At the root of the storage description we have a virtual view $V0$ representing the dataset before it is stored. Then, we have the views $V1$, $V2$ and $V3$, which model the partitioning of the data on ten machines using the subject, the property, and the object of a triple. On each machine, triples are then grouped by property and stored inside the file system. This is captured by the views $V4$, $V6$, and $V8$. Remember that CliqueSquare handles triples with the rdf:type property differently ($V5$, $V7$, $V9$) storing them in files according to the value of the object ($V10$, $V11$, $V12$). We can observe that only the views $V4$, $V6$, $V8$, $V10$, $V11$ and $V12$, correspond to actual files in the file system and thus we consider them to be materialized by adding an * exponent; all the other views are virtual. Finally, we use the symbol ξ to denote the input parameters for the views, i.e., the values that we have to provide in order to access the data of the view. For example, if one wants to find the triples that match the triple pattern `(ub:stud1 ub:takesCourse ?o)`, she needs to look into the file named after `ub:takesCourse` located in the node where `ub:stud1` hashes. Equivalently, we have to access the view $V4$ with parameter ξ_1 equals with the hash value of `ub:stud1` and parameter ξ_2 equals to `ub:takesCourse`.

The modeling described above relies on views with binding patterns and integrity constraints, thus the problem boils down to a rewriting problem where we are looking for equivalent rewritings of the input RDF query using the view-based storage description. To solve the problem our first thoughts lean towards a variation of the well-known Chase and Backchase algorithm [DPT99].

We are currently working on finalizing the storage description language and proving the correctness of the rewriting algorithm. Next, we plan to implement our proposal on a prototype system and perform experiments using (i) real [MLAN11] and synthetic RDF benchmarks [GPH05, SHLP09, AHÖD14] and (ii) storage descriptions corresponding to real and fictional RDF data management systems.

$$\begin{array}{l}
V0(s,p,o) \leftarrow T(s,p,o) \\
\hline
V1(s,p,o,\xi_1) \leftarrow V0(s,p,o), H_{10}(s,\xi_1) \\
V2(s,p,o,\xi_1) \leftarrow V0(s,p,o), H_{10}(p,\xi_1) \\
V3(s,p,o,\xi_1) \leftarrow V0(s,p,o), H_{10}(o,\xi_1) \\
\hline
V4^*(s,o,\xi_1,\xi_2) \leftarrow V1(s,\xi_2,o,\xi_1), \xi_2 \neq \text{"rdf:type"} \\
V5(s,o,\xi_1) \leftarrow V1(s,\text{"rdf:type"},o,\xi_1) \\
V6^*(s,o,\xi_1,\xi_2) \leftarrow V2(s,\xi_2,o,\xi_1), \xi_2 \neq \text{"rdf:type"} \\
V7(s,o,\xi_1) \leftarrow V2(s,\text{"rdf:type"},o,\xi_1) \\
V8^*(s,o,\xi_1,\xi_2) \leftarrow V3(s,\xi_2,o,\xi_1), \xi_2 \neq \text{"rdf:type"} \\
V9(s,o,\xi_1) \leftarrow V3(s,\text{"rdf:type"},o,\xi_1) \\
\hline
V10^*(s,\xi_1,\xi_3) \leftarrow V5(s,\xi_3,\xi_1) \\
V11^*(s,\xi_1,\xi_3) \leftarrow V7(s,\xi_3,\xi_1) \\
V12^*(s,\xi_1,\xi_3) \leftarrow V9(s,\xi_3,\xi_1)
\end{array}$$

Figure 5.1: View-based storage description for CliqueSquare.

5.3 Perspectives

Semantic Web data management in the cloud is still an area with a lot of potential for future research. Below we outline various perspectives to this thesis work.

Optimization for full SPARQL queries. Although (BGP) conjunctive queries are the most common fragment of SPARQL used in practice, extensions thereof are also important. For instance in the context of RDF analytics [CGMR14], queries with grouping, aggregation and optional clauses are central. Optimization beyond the fragment of conjunctive queries is thus necessary, and particularly challenging in distributed and cloud-computing architectures. In general, the optimization of RDF queries in cloud-based architectures is not sufficiently developed, with many systems relying on heuristics, or worse not even having an optimization algorithm. In order to support efficiently RDF analytics (and other application domains) in the cloud, we have to extend our algorithm to consider a bigger fragment of SPARQL.

Flat plans for centralized systems. In Chapter 4, we have shown the performance benefit of flat plans in a massively parallel setting. However, we have not examined the effects of flat plans in centralized systems. We believe that existing centralized systems could exploit flat plans to achieve better performance by utilizing multiple available processors and possibly benefiting from short optimization times. In Section 4.7.2, we have shown that flat plans can be produced efficiently by various CliqueSquare variants. In addition further pruning and/or memorization techniques could be used to bring the optimization time even lower. Flat plans combined with multi-core processors is an area worth exploring. Still, the algorithms would need some modifications since n-ary joins are not directly supported in centralized systems and additional proofs may be needed to retain the flatness guarantees after the modifications. Finally, extensive experiments should be performed to validate the claims.

Reformulation-based query answering for massively parallel RDF systems. We have seen in Section 2.1 that, in order to provide complete answers to user queries, we have to account for the implicit triples deriving from the semantics of RDF. In this thesis, we have assumed that the RDF dataset is saturated. Database saturation though may not be the best choice [GMR13], especially when there are large amounts of data that can introduce many implicit triples, which need to be stored in the system. Another option to obtain complete answers is through query reformulation. This technique though puts significant burden to the query optimizer that needs to generate plans involving hundreds of triple patterns. Reformulation-based optimizations have been studied in centralized RDF systems [BGM15]. In these works, a crucial part of the optimizer is the cost-model. Distributed systems have various additional factors that affect the cost (e.g., data partitioning, different types of joins, jobs overhead in case of MapReduce frameworks, etc.), thus adapting and extending these techniques is essential for enabling efficient reformulation-based query answering in massively parallel frameworks.

Statistics and estimations for distributed RDF systems. Since the advent of relational databases, statistics and accurate result estimations have been of great importance. Stepping on existing knowledge many RDF optimizers utilized relational-style statistics and estimations to improve query performance. Nevertheless, it has been shown [NM11, GN14] that gathering and using statistics by considering the peculiarities of RDF can greatly improve the performance. Cloud-based RDF data management systems though either do not use statistics at all or rely on simple statistics and estimations adopted from relational databases. It is evident that by extending works like [NM11, GN14] to take into account other factors like the distribution and the significantly bigger size of the data we could achieve further performance improvements. The domain of RDF statistics counts very few works, thus there are still many possibilities for deriving new models. Graph databases, semi-structured data, and distributed architectures, could be influential towards this direction.

Pricing model for RDF systems on commercial cloud infrastructures. Storing (and querying) data in public (commercial) clouds is an attractive option for companies and organizations, especially if their resource requirements change over time. Public clouds services usually come at a price. In order to provide a full solution for a cloud-based RDF store, a smart pricing model has to be established. In Chapter 3, we have outlined the monetary costs of the index, which are a first ingredient of a comprehensive pricing scheme. Working in this direction, the ultimate goal would be to formalize a cost model of different indexing and query answering strategies that expresses the trade-off between their efficiency/performance and associated monetary costs. In addition, a full-fledged cloud-based RDF data management system should provide a variety of options (for storage and querying) to a prospect customer being able to accommodate her performance needs and budget restrictions. In this thesis, we have witnessed the cost and performance of some storage and processing strategies but a lot more could be considered (see Section 2.3).

Result caching and view-based rewriting in Hadoop-based RDF systems. Among the proliferation of cloud-based systems, Apache's Hadoop is among the top choices for building RDF data management platforms. One particularity of Hadoop's MapReduce framework is that all intermediate results of a query (the results of Hadoop jobs) are written to disk and usually are discarded after the final result is returned to the user. Caching these results and considering them as views could definitely improve the performance of the system. Caching RDF query results [MUA10] and view-based rewritings of RDF queries [GKLM11] have both been examined in the context of centralized RDF systems. Building on existing knowledge we could extend our optimizer to be able to reuse intermediate query results for improving performance, exploiting views provided for free.

Bibliography

[Aba09] Daniel J. Abadi. Data management in the cloud: Limitations and opportunities. *IEEE Data Eng. Bull.*, 32(1):3–12, 2009.

[ABC$^+$12] Andrés Aranda-Andújar, Francesca Bugiotti, Jesús Camacho-Rodríguez, Dario Colazzo, François Goasdoué, Zoi Kaoudi, and Ioana Manolescu. AMADA: web data repositories in the Amazon cloud. In Xue-wen Chen, Guy Lebanon, Haixun Wang, and Mohammed J. Zaki, editors, *21st ACM International Conference on Information and Knowledge Management, CIKM'12, Maui, HI, USA, October 29 - November 02, 2012*, pages 2749–2751. ACM, 2012.

[ABE$^+$14] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, Felix Naumann, Mathias Peters, Astrid Rheinländer, Matthias J. Sax, Sebastian Schelter, Mareike Höger, Kostas Tzoumas, and Daniel Warneke. The stratosphere platform for big data analytics. *VLDB J.*, 23(6):939–964, 2014.

[acc08] Apache Accumulo. `http://accumulo.apache.org/`, 2008.

[ACK$^+$00] Sofia Alexaki, Vassilis Christophides, Gregory Karvounarakis, Dimitris Plexousakis, Karsten Tolle, Bernd Amann, Irini Fundulaki, Michel Scholl, and Anne-Marie Vercoustre. Managing RDF metadata for community webs. In Stephen W. Liddle, Heinrich C. Mayr, and Bernhard Thalheim, editors, *Conceptual Modeling for E-Business and the Web, ER 2000 Workshops on Conceptual Modeling Approaches for E-Business and The World Wide Web and Conceptual Modeling, Salt Lake City, Utah, USA, October 9-12, 2000, Proceedings*, volume 1921 of *Lecture Notes in Computer Science*, pages 140–151. Springer, 2000.

[ACK$^+$01] Sofia Alexaki, Vassilis Christophides, Gregory Karvounarakis, Dimitris Plexousakis, and Karsten Tolle. The ICS-FORTH RDFSuite: Managing voluminous RDF description bases. In *SemWeb*, 2001.

[AEH$^+$11] Alexander Alexandrov, Stephan Ewen, Max Heimel, Fabian Hueske, Odej Kao, Volker Markl, Erik Nijkamp, and Daniel Warneke. MapReduce and PACT - Comparing Data Parallel Programming Models. In *BTW*, 2011.

[AHÖD14] Günes Aluç, Olaf Hartig, M. Tamer Özsu, and Khuzaima Daudjee. Diversified stress testing of RDF data management systems. In Peter Mika, Tania Tudorache, Abraham Bernstein, Chris Welty, Craig A. Knoblock, Denny Vrandecic, Paul T. Groth, Natasha F. Noy, Krzysztof Janowicz, and Carole A. Goble, editors, *The Semantic Web - ISWC 2014 - 13th International Semantic Web Conference, Riva del Garda, Italy, October 19-23, 2014. Proceedings, Part I*, volume 8796 of *Lecture Notes in Computer Science*, pages 197–212. Springer, 2014.

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[AJR^{+}14] Foto N. Afrati, Manas Joglekar, Christopher Ré, Semih Salihoglu, and Jeffrey D. Ullman. GYM: A multiround join algorithm in mapreduce. *CoRR*, abs/1410.4156, 2014.

[AMMH07] Daniel J. Abadi, Adam Marcus, Samuel Madden, and Katherine J. Hollenbach. Scalable semantic web data management using vertical partitioning. In Christoph Koch, Johannes Gehrke, Minos N. Garofalakis, Divesh Srivastava, Karl Aberer, Anand Deshpande, Daniela Florescu, Chee Yong Chan, Venkatesh Ganti, Carl-Christian Kanne, Wolfgang Klas, and Erich J. Neuhold, editors, *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, pages 411–422. ACM, 2007.

[AÖD14] Günes Aluç, M. Tamer Özsu, and Khuzaima Daudjee. Workload matters: Why RDF databases need a new design. *PVLDB*, 7(10):837–840, 2014.

[AÖD15] Günes Aluç, M. Tamer Özsu, and Khuzaima Daudjee. Clustering RDF databases using tunable-LSH. *CoRR*, abs/1504.02523, 2015.

[AÖDH13] Günes Aluç, M. Tamer Özsu, Khuzaima Daudjee, and Olaf Hartig. chameleon-db: a workload-aware robust RDF data management system. *University of Waterloo, Tech. Rep. CS-2013-10*, 2013.

[AÖDH15] Günes Aluç, M. Tamer Özsu, Khuzaima Daudjee, and Olaf Hartig. Executing queries over schemaless RDF databases. In Johannes Gehrke, Wolfgang Lehner, Kyuseok Shim, Sang Kyun Cha, and Guy M. Lohman, editors, *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*, pages 807–818. IEEE, 2015.

[AU10] Foto N. Afrati and Jeffrey D. Ullman. Optimizing joins in a MapReduce environment. In Ioana Manolescu, Stefano Spaccapietra, Jens Teubner, Masaru Kitsuregawa, Alain Léger, Felix Naumann, Anastasia Ailamaki, and Fatma Özcan, editors, *EDBT 2010, 13th International Conference on Extending Database Technology, Lausanne, Switzerland, March 22-26, 2010, Proceedings*, volume 426 of *ACM International Conference Proceeding Series*, pages 99–110. ACM, 2010.

[AV11] Maribel Acosta and Maria-Esther Vidal. Evaluating adaptive query processing techniques for federations of SPARQL endpoints. In *10th International Semantic Web Conference (ISWC) Demo Session*, 2011.

[aws] Amazon Web Services. `http://aws.amazon.com/`.

[BCG$^+$14] Francesca Bugiotti, Jesús Camacho-Rodríguez, François Goasdoué, Zoi Kaoudi, Ioana Manolescu, and Stamatis Zampetakis. SPARQL query processing in the cloud. In Andreas Harth, Katja Hose, and Ralf Schenkel, editors, *Linked Data Management.*, pages 165–192. Chapman and Hall/CRC, 2014.

[BDK$^+$13] Mihaela A. Bornea, Julian Dolby, Anastasios Kementsietsidis, Kavitha Srinivas, Patrick Dantressangle, Octavian Udrea, and Bishwaranjan Bhattacharjee. Building an efficient RDF store over a relational database. In Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 121–132. ACM, 2013.

[Bec01] Dave J. Beckett. The design and implementation of the Redland RDF application framework. In Vincent Y. Shen, Nobuo Saito, Michael R. Lyu, and Mary Ellen Zurko, editors, *Proceedings of the Tenth International World Wide Web Conference, WWW 10, Hong Kong, China, May 1-5, 2001*, pages 449–456. ACM, 2001.

[BF00] Tim Berners-Lee and Mark Fischetti. *Weaving the web - the original design and ultimate destiny of the World Wide Web by its inventor*. HarperBusiness, 2000.

[BFG$^+$08] Matthias Brantner, Daniela Florescu, David A. Graf, Donald Kossmann, and Tim Kraska. Building a Database on S3. In *SIGMOD*, 2008.

[BG03] Dave Beckett and Jan Grant. SWAD-europe deliverable 10.2: Mapping semantic web data with RDBMSes, 2003.

[BGKM12] Francesca Bugiotti, François Goasdoué, Zoi Kaoudi, and Ioana Manolescu. RDF data management in the amazon cloud. In Divesh Srivastava and Ismail Ari, editors, *Proceedings of the 2012 Joint EDBT/ICDT Workshops, Berlin, Germany, March 30, 2012*, pages 61–72. ACM, 2012.

[BGM15] Damian Bursztyn, François Goasdoué, and Ioana Manolescu. Optimizing reformulation-based query answering in RDF. In Gustavo Alonso, Floris Geerts, Lucian Popa, Pablo Barceló, Jens Teubner, Martín Ugarte, Jan Van den Bussche, and Jan Paredaens, editors, *Proceedings of the 18th International Conference on Extending Database Technology, EDBT 2015, Brussels, Belgium, March 23-27, 2015.*, pages 265–276. OpenProceedings.org, 2015.

[BK03] Jeen Broekstra and Arjohn Kampman. Inferencing and truth maintenance in RDF schema. In Raphael Volz, Stefan Decker, and Isabel F. Cruz, editors, *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*, volume 89 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2003.

[BK04] Mihir Bellare and Tadayoshi Kohno. Hash function balance and its impact on birthday attacks. In Christian Cachin and Jan Camenisch, editors, *Advances in Cryptology - EUROCRYPT 2004, International Conference on the Theory* *and Applications of Cryptographic Techniques, Interlaken, Switzerland, May 2-6, 2004, Proceedings*, volume 3027 of *Lecture Notes in Computer Science*, pages 401–418. Springer, 2004.

[BKS13] Paul Beame, Paraschos Koutris, and Dan Suciu. Communication steps for parallel query processing. In Richard Hull and Wenfei Fan, editors, *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2013, New York, NY, USA - June 22 - 27, 2013*, pages 273–284. ACM, 2013.

[BKvH02] Jeen Broekstra, Arjohn Kampman, and Frank van Harmelen. Sesame: A generic architecture for storing and querying RDF and RDF schema. In Ian Horrocks and James A. Hendler, editors, *The Semantic Web - ISWC 2002, First International Semantic Web Conference, Sardinia, Italy, June 9-12, 2002, Proceedings*, volume 2342 of *Lecture Notes in Computer Science*, pages 54–68. Springer, 2002.

[BPE$^+$10] Spyros Blanas, Jignesh M. Patel, Vuk Ercegovac, Jun Rao, Eugene J. Shekita, and Yuanyuan Tian. A comparison of join algorithms for log processing in MapReduce. In *SIGMOD*, 2010.

[cas08] Apache Cassandra. `http://cassandra.apache.org/`, 2008.

[CCM12] Jesús Camacho-Rodríguez, Dario Colazzo, and Ioana Manolescu. Building large XML stores in the Amazon cloud. In Anastasios Kementsietsidis and Marcos Antonio Vaz Salles, editors, *Workshops Proceedings of the IEEE 28th International Conference on Data Engineering, ICDE 2012, Arlington, VA, USA, April 1-5, 2012*, pages 151–158. IEEE Computer Society, 2012.

[CCM13] Jesús Camacho-Rodríguez, Dario Colazzo, and Ioana Manolescu. Web data indexing in the cloud: efficiency and cost reductions. In Giovanna Guerrini and Norman W. Paton, editors, *Joint 2013 EDBT/ICDT Conferences, EDBT '13 Proceedings, Genoa, Italy, March 18-22, 2013*, pages 41–52. ACM, 2013.

[CCM15] Jesús Camacho-Rodríguez, Dario Colazzo, and Ioana Manolescu. Paxquery: Efficient parallel processing of complex xquery. *IEEE Trans. Knowl. Data Eng.*, 27(7):1977–1991, 2015.

[CCMN15] Jesús Camacho-Rodríguez, Dario Colazzo, Ioana Manolescu, and Juan A. M. Naranjo. Paxquery: Parallel analytical XML processing. In Timos Sellis, Susan B. Davidson, and Zachary G. Ives, editors, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 1117–1122. ACM, 2015.

[CDES05] Eugene Inseok Chong, Souripriya Das, George Eadon, and Jagannathan Srinivasan. An efficient SQL-based RDF querying scheme. In Klemens Böhm, Christian S. Jensen, Laura M. Haas, Martin L. Kersten, Per-Åke Larson, and Beng Chin Ooi, editors, *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*, pages 1216–1227. ACM, 2005.

[CDG$^+$06] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber. Bigtable: A distributed storage system for structured data. In Brian N. Bershad and Jeffrey C. Mogul, editors, *7th Symposium on Operating Systems Design and Implementation (OSDI'06), November 6-8, Seattle, WA, USA*, pages 205–218. USENIX Association, 2006.

[CF04] Min Cai and Martin R. Frank. RDFPeers: a scalable distributed RDF repository based on a structured peer-to-peer network. In Stuart I. Feldman, Mike Uretsky, Marc Najork, and Craig E. Wills, editors, *Proceedings of the 13th international conference on World Wide Web, WWW 2004, New York, NY, USA, May 17-20, 2004*, pages 650–657. ACM, 2004.

[CFCS03] Min Cai, Martin R. Frank, Jinbo Chen, and Pedro A. Szekely. MAAN: A multi-attribute addressable network for grid information services. In Heinz Stockinger, editor, *4th International Workshop on Grid Computing (GRID 2003), 17 November 2003, Phoenix, AZ, USA, Proceedings*, pages 184–191. IEEE Computer Society, 2003.

[CFYM04] Min Cai, Martin R. Frank, Baoshi Yan, and Robert M. MacGregor. A subscribable peer-to-peer RDF repository for distributed metadata management. *J. Web Sem.*, 2(2):109–130, 2004.

[CGMR14] Dario Colazzo, François Goasdoué, Ioana Manolescu, and Alexandra Roatis. RDF analytics: lenses over semantic graphs. In Chin-Wan Chung, Andrei Z. Broder, Kyuseok Shim, and Torsten Suel, editors, *23rd International World Wide Web Conference, WWW '14, Seoul, Republic of Korea, April 7-11, 2014*, pages 467–478. ACM, 2014.

[CL10] Roger Castillo and Ulf Leser. Selecting materialized views for RDF data. In Florian Daniel and Federico Michele Facca, editors, *Current Trends in Web Engineering - 10th International Conference on Web Engineering, ICWE 2010 Workshops, Vienna, Austria, July 2010, Revised Selected Papers*, volume 6385 of *Lecture Notes in Computer Science*, pages 126–137. Springer, 2010.

[Dat11] Datagraph. Dydra. `http://dydra.com/`, 2011.

[DCDK11] Vicky Dritsou, Panos Constantopoulos, Antonios Deligiannakis, and Yannis Kotidis. Optimizing query shortcuts in RDF databases. In Grigoris Antoniou, Marko Grobelnik, Elena Paslaru Bontas Simperl, Bijan Parsia, Dimitris Plexousakis, Pieter De Leenheer, and Jeff Z. Pan, editors, *The Semanic Web: Research and Applications - 8th Extended Semantic Web Conference, ESWC 2011, Heraklion, Crete, Greece, May 29 - June 2, 2011, Proceedings, Part II*, volume 6644 of *Lecture Notes in Computer Science*, pages 77–92. Springer, 2011.

[DG04] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In Eric A. Brewer and Peter Chen, editors, *6th Symposium on Operating System Design and Implementation (OSDI 2004), San*

Francisco, California, USA, December 6-8, 2004, pages 137–150. USENIX Association, 2004.

[DGK$^+$14] Benjamin Djahandideh, François Goasdoué, Zoi Kaoudi, Ioana Manolescu, Jorge Quiané-Ruiz, and Stamatis Zampetakis. How to Deal with Cliques at Work. In *BDA'2014: 30e journées Bases de Données Avancées*, Grenoble-Autrans, France, October 2014.

[DGK$^+$15] Benjamin Djahandideh, François Goasdoué, Zoi Kaoudi, Ioana Manolescu, Jorge-Arnulfo Quiané-Ruiz, and Stamatis Zampetakis. CliqueSquare in action: Flat plans for massively parallel RDF queries. In Johannes Gehrke, Wolfgang Lehner, Kyuseok Shim, Sang Kyun Cha, and Guy M. Lohman, editors, *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*, pages 1432–1435. IEEE, 2015.

[DHJ$^+$07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In Thomas C. Bressoud and M. Frans Kaashoek, editors, *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*, pages 205–220. ACM, 2007.

[DKSU11] Songyun Duan, Anastasios Kementsietsidis, Kavitha Srinivas, and Octavian Udrea. Apples and oranges: a comparison of RDF benchmarks and real RDF datasets. In Timos K. Sellis, Renée J. Miller, Anastasios Kementsietsidis, and Yannis Velegrakis, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, pages 145–156. ACM, 2011.

[DPT99] Alin Deutsch, Lucian Popa, and Val Tannen. Physical data independence, constraints, and optimization with universal plans. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 459–470, 1999.

[dyn12] Amazon DynamoDB. `http://aws.amazon.com/dynamodb/`, 2012.

[Erl08] Orri Erling. Towards web scale RDF. Available at http://www.researchgate.net/publication/228809902_Towards_web_scale_RDF, 2008.

[ETÖ$^+$11] Mohamed Y. Eltabakh, Yuanyuan Tian, Fatma Özcan, Rainer Gemulla, Aljoscha Krettek, and John McPherson. CoHadoop: Flexible data placement and its exploitation in Hadoop. *PVLDB*, 4(9):575–585, 2011.

[FBHB11] Imen Filali, Francesco Bongiovanni, Fabrice Huet, and Françoise Baude. A survey of structured P2P systems for RDF data storage and retrieval. *T. Large-Scale Data- and Knowledge-Centered Systems*, 6790:20–55, 2011.

[FCB12] David C. Faye, Olivier Cure, and Guillaume Blin. A survey of RDF storage approaches. *ARIMA Journal*, 15:11–35, 2012.

[GDN13] Alan Gates, Jianyong Dai, and Thejas Nair. Apache Pig's optimizer. *IEEE Data Eng. Bull.*, 36(1):34–45, 2013.

[GGL03] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003*, pages 29–43, 2003.

[GHS12] Luis Galarraga, Katja Hose, and Ralf Schenkel. Partout: A distributed engine for efficient RDF processing. *CoRR*, abs/1212.5636, 2012.

[GKLM10] François Goasdoué, Konstantinos Karanasos, Julien Leblay, and Ioana Manolescu. RDFViewS: a storage tuning wizard for RDF applications. In Jimmy Huang, Nick Koudas, Gareth J. F. Jones, Xindong Wu, Kevyn Collins-Thompson, and Aijun An, editors, *Proceedings of the 19th ACM Conference on Information and Knowledge Management, CIKM 2010, Toronto, Ontario, Canada, October 26-30, 2010*, pages 1947–1948. ACM, 2010.

[GKLM11] François Goasdoué, Konstantinos Karanasos, Julien Leblay, and Ioana Manolescu. View selection in semantic web databases. *PVLDB*, 5(2):97–108, 2011.

[GKM$^+$13] François Goasdoué, Zoi Kaoudi, Ioana Manolescu, Jorge Quiané-Ruiz, and Stamatis Zampetakis. CliqueSquare: efficient Hadoop-based RDF query processing. In *BDA'13 - Journées de Bases de Données Avancées*, Nantes, France, October 2013.

[GKM$^+$14] François Goasdoué, Zoi Kaoudi, Ioana Manolescu, Jorge Quiané-Ruiz, and Stamatis Zampetakis. CliqueSquare: Flat Plans for Massively Parallel RDF Queries. Research Report RR-8612, INRIA Saclay, October 2014.

[GKM$^+$15] François Goasdoué, Zoi Kaoudi, Ioana Manolescu, Jorge-Arnulfo Quiané-Ruiz, and Stamatis Zampetakis. CliqueSquare: Flat plans for massively parallel RDF queries. In Johannes Gehrke, Wolfgang Lehner, Kyuseok Shim, Sang Kyun Cha, and Guy M. Lohman, editors, *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*, pages 771–782. IEEE, 2015.

[GMR13] François Goasdoué, Ioana Manolescu, and Alexandra Roatis. Efficient query answering against dynamic RDF databases. In Giovanna Guerrini and Norman W. Paton, editors, *Joint 2013 EDBT/ICDT Conferences, EDBT '13 Proceedings, Genoa, Italy, March 18-22, 2013*, pages 299–310. ACM, 2013.

[GN14] Andrey Gubichev and Thomas Neumann. Exploiting the query structure for efficient join ordering in SPARQL queries. In Sihem Amer-Yahia, Vassilis Christophides, Anastasios Kementsietsidis, Minos N. Garofalakis, Stratos Idreos, and Vincent Leroy, editors, *Proceedings of the 17th International Conference on Extending Database Technology, EDBT 2014, Athens, Greece, March 24-28, 2014.*, pages 439–450. OpenProceedings.org, 2014.

[GPH05] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. LUBM: A benchmark for OWL knowledge base systems. *J. Web Sem.*, 3(2-3):158–182, 2005.

[GS11] Olaf Görlitz and Steffen Staab. SPLENDID: SPARQL endpoint federation exploiting VOID descriptions. In Olaf Hartig, Andreas Harth, and Juan Sequeda, editors, *Proceedings of the Second International Workshop on Consuming Linked Data (COLD2011), Bonn, Germany, October 23, 2011*, volume 782 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2011.

[GSMT14] Sairam Gurajada, Stephan Seufert, Iris Miliaraki, and Martin Theobald. TriAD: a distributed shared-nothing RDF engine based on asynchronous message passing. In Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu, editors, *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 289–300. ACM, 2014.

[had11] Apache Hadoop. `http://hadoop.apache.org/`, 2011.

[HAR11] Jiewen Huang, Daniel J. Abadi, and Kun Ren. Scalable SPARQL querying of large RDF graphs. *PVLDB*, 4(11):1123–1134, 2011.

[Har13] Olaf Hartig. SQUIN: a traversal based query execution system for the web of linked data. In Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 1081–1084. ACM, 2013.

[hba08] Apache HBase. `http://hbase.apache.org/`, 2008.

[HDB11] Herodotos Herodotou, Fei Dong, and Shivnath Babu. MapReduce programming and cost-based optimization? crossing this chasm with Starfish. *PVLDB*, 4(12):1446–1449, 2011.

[HFLP89] Laura M. Haas, Johann Christoph Freytag, Guy M. Lohman, and Hamid Pirahesh. Extensible query processing in Starburst. In James Clifford, Bruce G. Lindsay, and David Maier, editors, *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data, Portland, Oregon, May 31 - June 2, 1989.*, pages 377–388. ACM Press, 1989.

[HG03] Stephen Harris and Nicholas Gibbins. 3store: Efficient bulk RDF storage. In Raphael Volz, Stefan Decker, and Isabel F. Cruz, editors, *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*, volume 89 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2003.

[HKKT10] Mohammad Farhan Husain, Latifur Khan, Murat Kantarcioglu, and Bhavani M. Thuraisingham. Data intensive query processing for large RDF graphs using cloud computing tools. In *IEEE International Conference on Cloud Computing, CLOUD 2010, Miami, FL, USA, 5-10 July, 2010*, pages 1–10. IEEE, 2010.

[HLS09] Steve Harris, Nick Lamb, and Nigel Shadbolt. 4store: The design and implementation of a clustered RDF store. In *In 5th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2009)*, pages 94–109, 2009.

[HMM$^+$11] Mohammad Farhan Husain, James P. McGlothlin, Mohammad M. Masud, Latifur R. Khan, and Bhavani M. Thuraisingham. Heuristics-based query processing for large RDF graphs using cloud computing. *IEEE Trans. Knowl. Data Eng.*, 23(9):1312–1327, 2011.

[HS13] Katja Hose and Ralf Schenkel. WARP: workload-aware replication and partitioning for RDF. In Chee Yong Chan, Jiaheng Lu, Kjetil Nørvåg, and Egemen Tanin, editors, *Workshops Proceedings of the 29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pages 1–6. IEEE Computer Society, 2013.

[IGN$^+$12] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, K. Sjoerd Mullender, and Martin L. Kersten. MonetDB: Two decades of research in column-oriented database architectures. *IEEE Data Eng. Bull.*, 35(1):40–45, 2012.

[JOSW10] Dawei Jiang, Beng Chin Ooi, Lei Shi, and Sai Wu. The performance of MapReduce: An in-depth study. *PVLDB*, 3(1):472–483, 2010.

[KAC$^+$02] Gregory Karvounarakis, Sofia Alexaki, Vassilis Christophides, Dimitris Plexousakis, and Michel Scholl. RQL: a declarative query language for RDF. In David Lassner, Dave De Roure, and Arun Iyengar, editors, *Proceedings of the Eleventh International World Wide Web Conference, WWW 2002, May 7-11, 2002, Honolulu, Hawaii*, pages 592–603. ACM, 2002.

[Kar72] Richard M. Karp. Reducibility among combinatorial problems. In *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York.*, pages 85–103, 1972.

[KK98] 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.

[KKK10] Zoi Kaoudi, Kostis Kyzirakos, and Manolis Koubarakis. SPARQL query optimization on top of DHTs. In Peter F. Patel-Schneider, Yue Pan, Pascal Hitzler, Peter Mika, Lei Zhang, Jeff Z. Pan, Ian Horrocks, and Birte Glimm, editors, *The Semantic Web - ISWC 2010 - 9th International Semantic Web Conference, ISWC 2010, Shanghai, China, November 7-11, 2010, Revised Selected Papers, Part I*, volume 6496 of *Lecture Notes in Computer Science*, pages 418–435. Springer, 2010.

[KKMZ11] Konstantinos Karanasos, Asterios Katsifodimos, Ioana Manolescu, and Spyros Zoupanos. The ViP2P platform: XML views in P2P. *CoRR*, abs/1112.2610, 2011.

[KM15] Zoi Kaoudi and Ioana Manolescu. RDF in the clouds: a survey. *VLDB J.*, 24(1):67–91, 2015.

[KMM$^+$06] Zoi Kaoudi, Iris Miliaraki, Matoula Magiridou, Antonios Papadakis-Pesaresi, and Manolis Koubarakis. Storing and querying RDF data in Atlas. In *ESWC (Demonstration)*, 2006.

[Knu73] Donald E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms, 2nd Edition*. Addison-Wesley, 1973.

[KOvH10] Spyros Kotoulas, Eyal Oren, and Frank van Harmelen. Mind the data skew: distributed inferencing by speeddating in elastic regions. In Michael Rappa, Paul Jones, Juliana Freire, and Soumen Chakrabarti, editors, *Proceedings of the 19th International Conference on World Wide Web, WWW 2010, Raleigh, North Carolina, USA, April 26-30, 2010*, pages 531–540. ACM, 2010.

[KRA11] HyeongSik Kim, Padmashree Ravindra, and Kemafor Anyanwu. From SPARQL to MapReduce: The journey using a nested triplegroup algebra. *PVLDB*, 4(12):1426–1429, 2011.

[KRA12] HyeongSik Kim, Padmashree Ravindra, and Kemafor Anyanwu. Scan-sharing for optimizing RDF graph pattern matching on mapreduce. In Rong Chang, editor, *2012 IEEE Fifth International Conference on Cloud Computing, Honolulu, HI, USA, June 24-29, 2012*, pages 139–146. IEEE, 2012.

[Lee04] Ryan Lee. Scalability report on triple store applications. Available at http://simile.mit.edu/reports/stores/, 2004.

[LH11] 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.

[LIK06] Erietta Liarou, Stratos Idreos, and Manolis Koubarakis. Evaluating conjunctive triple pattern queries over large structured overlay networks. In Isabel F. Cruz, Stefan Decker, Dean Allemang, Chris Preist, Daniel Schwabe, Peter Mika, Michael Uschold, and Lora Aroyo, editors, *The Semantic Web - ISWC 2006, 5th International Semantic Web Conference, ISWC 2006, Athens, GA, USA, November 5-9, 2006, Proceedings*, volume 4273 of *Lecture Notes in Computer Science*, pages 399–413. Springer, 2006.

[Lil00] Jonas Liljegren. Description of an RDF database implementation. `http://infolab.stanford.edu/~melnik/rdf/db-jonas.html`, 2000.

[LL13] Kisung Lee and Ling Liu. Scaling queries over big RDF graphs with semantic hash partitioning. *PVLDB*, 6(14):1894–1905, 2013.

[LOÖW14] Feng Li, Beng Chin Ooi, M. Tamer Özsu, and Sai Wu. Distributed data management using MapReduce. *ACM Comput. Surv.*, 46(3):31, 2014.

[LPF$^+$12] Yongming Luo, François Picalausa, George HL Fletcher, Jan Hidders, and Stijn Vansummeren. Storing and indexing massive RDF datasets. In *Semantic Search over the Web*, pages 31–60. Springer, 2012.

[LT10] Günter Ladwig and Thanh Tran. Linked data query processing strategies. In Peter F. Patel-Schneider, Yue Pan, Pascal Hitzler, Peter Mika, Lei Zhang, Jeff Z. Pan, Ian Horrocks, and Birte Glimm, editors, *The Semantic Web - ISWC 2010 - 9th International Semantic Web Conference, ISWC 2010, Shanghai, China, November 7-11, 2010, Revised Selected Papers, Part I*, volume 6496 of *Lecture Notes in Computer Science*, pages 453–469. Springer, 2010.

[LT11] Günter Ladwig and Thanh Tran. SIHJoin: Querying remote and local linked data. In Grigoris Antoniou, Marko Grobelnik, Elena Paslaru Bontas Simperl, Bijan Parsia, Dimitris Plexousakis, Pieter De Leenheer, and Jeff Z. Pan, editors, *The Semantic Web: Research and Applications - 8th Extended Semantic Web Conference, ESWC 2011, Heraklion, Crete, Greece, May 29-June 2, 2011, Proceedings, Part I*, volume 6643 of *Lecture Notes in Computer Science*, pages 139–153. Springer, 2011.

[McB02] Brian McBride. Jena: A semantic web toolkit. *IEEE Internet Computing*, 6(6):55–59, 2002.

[McC60] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Commun. ACM*, 3(4):184–195, 1960.

[Mel00] Sergey Melnik. Storing RDF in a relational database. `http://infolab.stanford.edu/~melnik/rdf/db.html`, 2000.

[MG11] Peter Mell and Tim Grance. The NIST definition of cloud computing. Available at http://simile.mit.edu/reports/stores/, 2011.

[MKA+02] Aimilia Magkanaraki, Grigoris Karvounarakis, Ta Tuan Anh, Vassilis Christophides, and Dimitris Plexousakis. Ontology storage and querying. *ICS-FORTH Technical Report*, 308, 2002.

[MLAN11] Mohamed Morsey, Jens Lehmann, Sören Auer, and Axel-Cyrille Ngonga Ngomo. DBpedia SPARQL benchmark - performance assessment with real queries on real data. In Lora Aroyo, Chris Welty, Harith Alani, Jamie Taylor, Abraham Bernstein, Lalana Kagal, Natasha Fridman Noy, and Eva Blomqvist, editors, *The Semantic Web - ISWC 2011 - 10th International Semantic Web Conference, Bonn, Germany, October 23-27, 2011, Proceedings, Part I*, volume 7031 of *Lecture Notes in Computer Science*, pages 454–469. Springer, 2011.

[MNP+14] Boris Motik, Yavor Nenov, Robert Piro, Ian Horrocks, and Dan Olteanu. Parallel materialisation of datalog programs in centralised, main-memory RDF systems. In Carla E. Brodley and Peter Stone, editors, *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada.*, pages 129–137. AAAI Press, 2014.

[MPK06] Akiyoshi Matono, Said Mirza Pahlevi, and Isao Kojima. RDFCube: A P2P-based three-dimensional index for structural joins on distributed triple stores. In Gianluca Moro, Sonia Bergamaschi, Sam Joseph, Jean-Henry Morin, and Aris M. Ouksel, editors, *Databases, Information Systems, and Peer-to-Peer Computing, International Workshops, DBISP2P 2005/2006, Trondheim, Norway, August 28-29, 2005, Seoul, Korea, September 11, 2006, Revised Selected Papers*, volume 4125 of *Lecture Notes in Computer Science*, pages 323–330. Springer, 2006.

[MSP+04] Li Ma, Zhong Su, Yue Pan, Li Zhang, and Tao Liu. RStar: an RDF storage and query system for enterprise resource management. In David A. Grossman, Luis Gravano, ChengXiang Zhai, Otthein Herzog, and David A. Evans, editors, *Proceedings of the 2004 ACM CIKM International Conference on Information and Knowledge Management, Washington, DC, USA, November 8-13, 2004*, pages 484–491. ACM, 2004.

[MSR02] Libby Miller, Andy Seaborne, and Alberto Reggiori. Three implementations of SquishQL, a simple RDF query language. In Ian Horrocks and James A. Hendler, editors, *The Semantic Web - ISWC 2002, First International Semantic Web Conference, Sardinia, Italy, June 9-12, 2002, Proceedings*, volume 2342 of *Lecture Notes in Computer Science*, pages 423–435. Springer, 2002.

[MUA10] Michael Martin, Jörg Unbehauen, and Sören Auer. Improving the performance of semantic web applications with SPARQL query caching. In Lora Aroyo, Grigoris Antoniou, Eero Hyvönen, Annette ten Teije, Heiner Stuckenschmidt, Liliana Cabral, and Tania Tudorache, editors, *The Semantic Web: Research and Applications, 7th Extended Semantic Web Conference, ESWC 2010, Heraklion, Crete, Greece, May 30 - June 3, 2010, Proceedings, Part II*, volume 6089 of *Lecture Notes in Computer Science*, pages 304–318. Springer, 2010.

[NM11] Thomas Neumann and Guido Moerkotte. Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In Serge Abiteboul, Klemens Böhm, Christoph Koch, and Kian-Lee Tan, editors, *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, pages 984–994. IEEE Computer Society, 2011.

[NW10] Thomas Neumann and Gerhard Weikum. The RDF-3X Engine for Scalable Management of RDF Data. *VLDBJ*, 19(1), 2010.

[NWQ$^+$02] Wolfgang Nejdl, Boris Wolf, Changtao Qu, Stefan Decker, Michael Sintek, Ambjörn Naeve, Mikael Nilsson, Matthias Palmér, and Tore Risch. EDUTELLA: a P2P networking infrastructure based on RDF. In David Lassner, Dave De Roure, and Arun Iyengar, editors, *Proceedings of the Eleventh International World Wide Web Conference, WWW 2002, May 7-11, 2002, Honolulu, Hawaii*, pages 604–615. ACM, 2002.

[Ora94] Oracle. Berkeley db. `http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/overview/index.html`, 1994.

[ORS$^+$08] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 1099–1110, 2008.

[OSG08] Alisdair Owens, Andy Seaborne, and Nick Gibbins. Clustered TDB: a clustered triple store for Jena, 2008.

[ÖV11] M. Tamer Özsu and Patrick Valduriez. *Distributed and Parallel Database Systems (3rd. ed.)*. Springer, 2011.

[PCR12] Roshan Punnoose, Adina Crainiceanu, and David Rapp. Rya: a scalable RDF triple store for the clouds. In Jérôme Darmont and Torben Bach Pedersen, editors, *1st International Workshop on Cloud Intelligence (colocated with VLDB 2012), Cloud-I '12, Istanbul, Turkey, August 31, 2012*, page 4. ACM, 2012.

[PGGU95] Yannis Papakonstantinou, Ashish Gupta, Hector Garcia-Molina, and Jeffrey D. Ullman. A query translation scheme for rapid implementation of wrappers. In *Deductive and Object-Oriented Databases, Fourth International Conference, DOOD'95, Singapore, December 4-7, 1995, Proceedings*, pages 161–186, 1995.

[PKT$^+$13] Nikolaos Papailiou, Ioannis Konstantinou, Dimitrios Tsoumakos, Panagiotis Karras, and Nectarios Koziris. H2RDF+: high-performance distributed joins over large-scale RDF graphs. In Xiaohua Hu, Tsau Young Lin, Vijay Raghavan, Benjamin W. Wah, Ricardo A. Baeza-Yates, Geoffrey Fox, Cyrus Shahabi, Matthew Smith, Qiang Yang, Rayid Ghani, Wei Fan, Ronny Lempel, and Raghunath Nambiar, editors, *Proceedings of the 2013 IEEE International Conference on Big Data, 6-9 October 2013, Santa Clara, CA, USA*, pages 255–263. IEEE, 2013.

[PKTK12] Nikolaos Papailiou, Ioannis Konstantinou, Dimitrios Tsoumakos, and Nectarios Koziris. H2RDF: adaptive query processing on RDF data in the cloud. In Alain Mille, Fabien L. Gandon, Jacques Misselis, Michael Rabinovich, and Steffen Staab, editors, *Proceedings of the 21st World Wide Web Conference, WWW 2012, Lyon, France, April 16-20, 2012 (Companion Volume)*, pages 397–400. ACM, 2012.

[PTK$^+$14] Nikolaos Papailiou, Dimitrios Tsoumakos, Ioannis Konstantinou, Panagiotis Karras, and Nectarios Koziris. H2RDF+: an efficient data management system for big RDF graphs. In Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu, editors, *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 909–912. ACM, 2014.

[PZÖ$^+$14] Peng Peng, Lei Zou, M. Tamer Özsu, Lei Chen, and Dongyan Zhao. Processing SPARQL queries over linked data-a distributed graph-based approach. *CoRR*, abs/1411.6763, 2014.

[QL08] Bastian Quilitz and Ulf Leser. Querying distributed RDF data sources with SPARQL. In Sean Bechhofer, Manfred Hauswirth, Jörg Hoffmann, and Manolis Koubarakis, editors, *The Semantic Web: Research and Applications, 5th European Semantic Web Conference, ESWC 2008, Tenerife, Canary Islands, Spain, June 1-5, 2008, Proceedings*, volume 5021 of *Lecture Notes in Computer Science*, pages 524–538. Springer, 2008.

[RG03] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems (3rd. ed.)*. McGraw-Hill, 2003.

[RGRK04] Sean C. Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling churn in a DHT (awarded best paper!). In *Proceedings of the General Track: 2004 USENIX Annual Technical Conference, June 27 - July 2, 2004,* *Boston Marriott Copley Place, Boston, MA, USA*, pages 127–140. USENIX, 2004.

[RKA11] Padmashree Ravindra, HyeongSik Kim, and Kemafor Anyanwu. An intermediate algebra for optimizing RDF graph pattern matching on mapreduce. In Grigoris Antoniou, Marko Grobelnik, Elena Paslaru Bontas Simperl, Bijan Parsia, Dimitris Plexousakis, Pieter De Leenheer, and Jeff Z. Pan, editors, *The Semanic Web: Research and Applications - 8th Extended Semantic Web Conference, ESWC 2011, Heraklion, Crete, Greece, May 29 - June 2, 2011, Proceedings, Part II*, volume 6644 of *Lecture Notes in Computer Science*, pages 46–61. Springer, 2011.

[RS10] Kurt Rohloff and Richard E. Schantz. High-performance, massively scalable distributed systems using the MapReduce software framework: the SHARD triple-store. In Eli Tilevich and Patrick Eugster, editors, *SPLASH Workshop on Programming Support Innovations for Emerging Distributed Applications PSI EtA - 2010), October 17, 2010, Reno/Tahoe, Nevada, USA*, page 4. ACM, 2010.

[RUK$^+$13] Nur Aini Rakhmawati, Jürgen Umbrich, Marcel Karnstedt, Ali Hasnain, and Michael Hausenblas. Querying over federated SPARQL endpoints - A state of the art survey. *CoRR*, abs/1306.1723, 2013.

[s306] Amazon S3. `http://aws.amazon.com/s3/`, 2006.

[SA09] Sherif Sakr and Ghazi Al-Naymat. Relational processing of RDF queries: a survey. *SIGMOD Record*, 38(4):23–28, 2009.

[SAC$^+$79] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. Access path selection in a relational database management system. In Philip A. Bernstein, editor, *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, May 30 - June 1.*, pages 23–34. ACM, 1979.

[SBP14] Max Schmachtenberg, Christian Bizer, and Heiko Paulheim. Adoption of the linked data best practices in different topical domains. In Peter Mika, Tania Tudorache, Abraham Bernstein, Chris Welty, Craig A. Knoblock, Denny Vrandecic, Paul T. Groth, Natasha F. Noy, Krzysztof Janowicz, and Carole A. Goble, editors, *The Semantic Web - ISWC 2014 - 13th International Semantic Web Conference, Riva del Garda, Italy, October 19-23, 2014. Proceedings, Part I*, volume 8796 of *Lecture Notes in Computer Science*, pages 245–260. Springer, 2014.

[Sea04] Andy Seaborne. RDQL - a query language for RDF. `http://www.w3.org/Submission/2004/SUBM-RDQL-20040109/`, 2004.

[SGD$^+$09] Florian Stegmaier, Udo Gröbner, Mario Döller, Harald Kosch, and Gero Baese. Evaluation of current RDF database solutions. In *Proceedings of the 10th International Workshop on Semantic Multimedia Database Technologies (SeMuDaTe), 4th International Conference on Semantics And Digital Media Technologies (SAMT), 2009*, pages 39–55. Citeseer, 2009.

[SGK$^+$85] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation of the sun network filesystem. In *Proceedings of the Summer USENIX conference*, pages 119–130, 1985.

[SHH$^+$11] Andreas Schwarte, Peter Haase, Katja Hose, Ralf Schenkel, and Michael Schmidt. FedX: A federation layer for distributed query processing on linked open data. In Grigoris Antoniou, Marko Grobelnik, Elena Paslaru Bontas Simperl, Bijan Parsia, Dimitris Plexousakis, Pieter De Leenheer, and Jeff Z. Pan, editors, *The Semanic Web: Research and Applications - 8th Extended Semantic Web Conference, ESWC 2011, Heraklion, Crete, Greece, May 29 - June 2, 2011, Proceedings, Part II*, volume 6644 of *Lecture Notes in Computer Science*, pages 481–486. Springer, 2011.

[SHLP09] Michael Schmidt, Thomas Hornung, Georg Lausen, and Christoph Pinkel. SP2Bench: A SPARQL performance benchmark. In Yannis E. Ioannidis, Dik Lun Lee, and Raymond T. Ng, editors, *Proceedings of the 25th International Conference on Data Engineering, ICDE 2009, March 29 2009 - April 2 2009, Shanghai, China*, pages 222–233. IEEE, 2009.

[sim07] Amazon SimpleDB. `http://aws.amazon.com/simpledb/`, 2007.

[SMK97] Michael Steinbrunn, Guido Moerkotte, and Alfons Kemper. Heuristic and randomized optimization for the join ordering problem. *VLDB J.*, 6(3):191–208, 1997.

[SPL11] Alexander Schätzle, Martin Przyjaciel-Zablocki, and Georg Lausen. PigSPARQL: mapping SPARQL to pig latin. In Roberto De Virgilio, Fausto Giunchiglia, and Letizia Tanca, editors, *Proceedings of the International Workshop on Semantic Web Information Management, SWIM 2011, Athens, Greece, June 12, 2011*, page 4. ACM, 2011.

[SSB$^+$08] Markus Stocker, Andy Seaborne, Abraham Bernstein, Christoph Kiefer, and Dave Reynolds. SPARQL basic graph pattern optimization using selectivity estimation. In Jinpeng Huai, Robin Chen, Hsiao-Wuen Hon, Yunhao Liu, Wei-Ying Ma, Andrew Tomkins, and Xiaodong Zhang, editors, *Proceedings of the 17th International Conference on World Wide Web, WWW 2008, Beijing, China, April 21-25, 2008*, pages 595–604. ACM, 2008.

[SWL13] Bin Shao, Haixun Wang, and Yatao Li. Trinity: a distributed graph engine on a memory cloud. In Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 505–516. ACM, 2013.

[SZ10] Raffael Stein and Valentin Zacharias. RDF on cloud number nine. In *4th Workshop on New Forms of Reasoning for the Semantic Web: Scalable and Dynamic*, pages 11–23. Citeseer, 2010.

[TSF$^+$12] Petros Tsialiamanis, Lefteris Sidirourgos, Irini Fundulaki, Vassilis Christophides, and Peter A. Boncz. Heuristics-based query optimisation for

SPARQL. In Elke A. Rundensteiner, Volker Markl, Ioana Manolescu, Sihem Amer-Yahia, Felix Naumann, and Ismail Ari, editors, *15th International Conference on Extending Database Technology, EDBT '12, Berlin, Germany, March 27-30, 2012, Proceedings*, pages 324–335. ACM, 2012.

[TSJ$^+$09] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive - A warehousing solution over a map-reduce framework. *PVLDB*, 2(2):1626–1629, 2009.

[TSJ$^+$10] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Anthony, Hao Liu, and Raghotham Murthy. Hive - a petabyte scale data warehouse using Hadoop. In Feifei Li, Mirella M. Moro, Shahram Ghandeharizadeh, Jayant R. Haritsa, Gerhard Weikum, Michael J. Carey, Fabio Casati, Edward Y. Chang, Ioana Manolescu, Sharad Mehrotra, Umeshwar Dayal, and Vassilis J. Tsotras, editors, *Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA*, pages 996–1005. IEEE, 2010.

[TUY11] Yuan Tian, Jürgen Umbrich, and Yong Yu. Enhancing source selection for live queries over linked data via query log mining. In Jeff Z. Pan, Huajun Chen, Hong-Gee Kim, Juanzi Li, Zhe Wu, Ian Horrocks, Riichiro Mizoguchi, and Zhaohui Wu, editors, *The Semantic Web - Joint International Semantic Technology Conference, JIST 2011, Hangzhou, China, December 4-7, 2011. Proceedings*, volume 7185 of *Lecture Notes in Computer Science*, pages 176–191. Springer, 2011.

[UHK$^+$11] Jürgen Umbrich, Katja Hose, Marcel Karnstedt, Andreas Harth, and Axel Polleres. Comparing data summaries for processing live queries over linked data. *World Wide Web*, 14(5-6):495–544, 2011.

[UKOvH09] Jacopo Urbani, Spyros Kotoulas, Eyal Oren, and Frank van Harmelen. Scalable distributed reasoning using mapreduce. In Abraham Bernstein, David R. Karger, Tom Heath, Lee Feigenbaum, Diana Maynard, Enrico Motta, and Krishnaprasad Thirunarayan, editors, *The Semantic Web - ISWC 2009, 8th International Semantic Web Conference, ISWC 2009, Chantilly, VA, USA, October 25-29, 2009. Proceedings*, volume 5823 of *Lecture Notes in Computer Science*, pages 634–649. Springer, 2009.

[UMJ$^+$13] Jacopo Urbani, Alessandro Margara, Ceriel J. H. Jacobs, Frank van Harmelen, and Henri E. Bal. Dynamite: Parallel materialization of dynamic RDF data. In Harith Alani, Lalana Kagal, Achille Fokoue, Paul T. Groth, Chris Biemann, Josiane Xavier Parreira, Lora Aroyo, Natasha F. Noy, Chris Welty, and Krzysztof Janowicz, editors, *The Semantic Web - ISWC 2013 - 12th International Semantic Web Conference, Sydney, NSW, Australia, October 21-25, 2013, Proceedings, Part I*, volume 8218 of *Lecture Notes in Computer Science*, pages 657–672. Springer, 2013.

[UvHSB11] Jacopo Urbani, Frank van Harmelen, Stefan Schlobach, and Henri E. Bal. QueryPIE: Backward reasoning for OWL horst over very large knowledge bases. In Lora Aroyo, Chris Welty, Harith Alani, Jamie Taylor, Abraham Bernstein, Lalana Kagal, Natasha Fridman Noy, and Eva Blomqvist, editors, *The Semantic Web - ISWC 2011 - 10th International Semantic Web Conference, Bonn, Germany, October 23-27, 2011, Proceedings, Part I*, volume 7031 of *Lecture Notes in Computer Science*, pages 730–745. Springer, 2011.

[W3C04] W3C. Resource description framework (RDF): Concepts and abstract syntax. `http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/`, 2004.

[W3C08] W3C. SPARQL query language for RDF. `http://www.w3.org/TR/rdf-sparql-query/`, 2008.

[W3C13] W3C. SPARQL 1.1 query language for RDF. `http://www.w3.org/TR/sparql11-query/`, 2013.

[W3C14a] W3C. RDF 1.1 concepts and abstract syntax. `http://www.w3.org/TR/rdf11-concepts/`, 2014.

[W3C14b] W3C. RDF 1.1 semantics. `http://www.w3.org/TR/rdf11-mt/`, 2014.

[W3C14c] W3C. RDF schema 1.1. `http://www.w3.org/TR/rdf-schema/`, 2014.

[WKB08] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. Hexastore: sextuple indexing for semantic web data management. *PVLDB*, 1(1):1008–1019, 2008.

[WLMO11] Sai Wu, Feng Li, Sharad Mehrotra, and Beng Chin Ooi. Query optimization for massively parallel data processing. In Jeffrey S. Chase and Amr El Abbadi, editors, *ACM Symposium on Cloud Computing in conjunction with SOSP 2011, SOCC '11, Cascais, Portugal, October 26-28, 2011*, page 12. ACM, 2011.

[WSKR03] Kevin Wilkinson, Craig Sayers, Harumi A. Kuno, and Dave Reynolds. Efficient RDF storage and retrieval in Jena2. In Isabel F. Cruz, Vipul Kashyap, Stefan Decker, and Rainer Eckstein, editors, *Proceedings of SWDB'03, The first International Workshop on Semantic Web and Databases, Co-located with VLDB 2003, Humboldt-Universität, Berlin, Germany, September 7-8, 2003*, pages 131–150, 2003.

[WZY$^+$15] Buwen Wu, Yongluan Zhou, Pingpeng Yuan, Ling Liu, and Hai Jin. Scalable SPARQL querying using path partitioning. In *ICDE*, 2015.

[Yan81] Mihalis Yannakakis. Algorithms for acyclic database schemes. In *Very Large Data Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings*, pages 82–94. IEEE Computer Society, 1981.

[ZCF$^+$10] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In Erich M. Nahum and Dongyan Xu, editors, *2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'10, Boston, MA, USA, June 22, 2010*. USENIX Association, 2010.

[ZMC$^+$11] Lei Zou, Jinghui Mo, Lei Chen, M. Tamer Özsu, and Dongyan Zhao. gStore: Answering SPARQL queries via subgraph matching. *PVLDB*, 4(8):482–493, 2011.

[ZÖC$^+$14] Lei Zou, M. Tamer Özsu, Lei Chen, Xuchuan Shen, Ruizhe Huang, and Dongyan Zhao. gStore: a graph-based SPARQL query engine. *VLDB J.*, 23(4):565–590, 2014.

[ZYW$^+$13] Kai Zeng, Jiacheng Yang, Haixun Wang, Bin Shao, and Zhongyuan Wang. A distributed graph engine for web scale RDF data. *PVLDB*, 6(4):265–276, 2013.

Appendix A

Detailed queries

A.1 Queries used in the AMADA experiments

This section lists the SPARQL queries used in the experimental section of Chapter 3. The namespaces that appear in the queries are given in Table A.1. The characteristics of the queries are summarized in Table A.2: $struct$ indicates the structure of each query ($simple$ for single triple pattern queries, $star$ for star-shaped join queries and mix for complex queries combining both star and path joins); $\#tp$ is the number of triple patterns; $\#c$ is the number of constant values each query contains; $\#results$ is the number of triples each query returns; $\#g$ is the number of distinct graphs $\#g$ which will be used from each strategy to answer the query.

Namespace	URI
dbpedia	`http://dbpedia.org/ontology/`
yago	`http://yago-knowledge.org/resource/`
rdf	`http://www.w3.org/1999/02/22-rdf-syntax-ns#`

Table A.1: URIs for namespaces used in the experimental queries of Chapter 3

Query	**struct**	**#tp**	**#c**	**#results**	**#g** by RTS	**#g** by ATT	**#g** by ATS
Q1	*simple*	1	2	1	2	2	1
Q2	*simple*	1	2	433	3	3	3
Q3	*simple*	1	1	72829	2	2	2
Q4	*star*	2	4	1	19	19	19
Q5	*star*	3	4	2895	26	25	25
Q6	*star*	3	3	50686	34	34	34
Q7	*star*	4	4	42785	39	39	39
Q8	*mix*	5	6	2	9	9	9
Q9	*mix*	5	5	12	5	5	5

Table A.2: Query characteristics.

Q1: SELECT ?birthplace WHERE { dbpedia:David_Beckham dbpedia:birthPlace ?birthplace . }

Q2: SELECT ?x WHERE { ?x yago:playsFor yago:FC_Barcelona . }

Q3: SELECT ?x WHERE { ?x yago:hasWonPrize ?y . } **Q4:** SELECT ?x WHERE { ?x rdf:type yago:wordnet_scientist_110560637 . ?x yago:diedOnDate "1842-07-19" . }

Q5: SELECT ?x ?z ?w WHERE { ?x rdf:type yago:wordnet_scientist_110560637 . ?x yago:diedOnDate ?w . ?x yago:wasBornOnDate ?z . }

Q6: SELECT ?gName ?fName ?type WHERE { ?p yago:hasGivenName ?gName . ?p yago:hasFamilyName ?fName . ?p rdf:type ?type . }

Q7: SELECT ?gName ?fName ?type WHERE { ?p yago:hasGivenName ?gName . ?p yago:hasFamilyName ?fName . ?p rdf:type ?type . ?p yago:wasBornOnDate ?date. }

Q8: SELECT ?gp ?loc ?name WHERE { ?gp rdf:type dbpedia:GrandPrix . ?gp dbpedia:location ?loc . ?gp dbpedia:firstDriver ?driver . ?driver dbpedia:birthPlace ?loc . ?driver foaf:name ?name . }

Q9: SELECT ?name1 ?name2 where { ?p1 yago:isMarriedTo ?p2 . ?p2 yago:hasGivenName ?name2 . ?p1 yago:hasGivenName ?name1. ?p2 yago:wasBornIn ?city . ?p1 yago:wasBornIn ?city . }

A.2 Queries used in the CliqueSquare experiments

This section lists the SPARQL queries used in the experimental section of Chapter 4. For the sake of simplicity some constants appear abbreviated. The characteristics of the queries are summarized in Figure A.1: number of triple patterns ($\#tps$), number of join variables ($\#jv$) and result cardinality for LUBM10k ($|Q|_{10k}$. The indicator (*original*) appears next to the query name when the query belongs to the default LUBM benchmark.

Queries	**Q1**	**Q2**	**Q3**	**Q4**	**Q5**	**Q6**	**Q7**
$\#tps$	2	2	3	4	5	5	5
$\#jv$	1	1	1	2	3	3	3
$\|Q\|_{10k}$	3.7B	1900	282.2K	93	56.1M	7.9M	25.1M
Queries	**Q8**	**Q9**	**Q10**	**Q11**	**Q12**	**Q13**	**Q14**
$\#tps$	5	6	6	8	9	9	10
$\#jv$	3	3	3	4	4	4	5
$\|Q\|_{10k}$	504.3M	2528	439.9K	1647	12.5M	871	1413

Figure A.1: Characteristics of the LUBM queries used in the experiments.

Q1: SELECT ?P ?S WHERE { ?P ub:worksFor ?D . ?S ub:memberOf ?D . }

Q2(original): SELECT ?X WHERE { ?X rdf:type ub:AssistantProfessor . ?X ub:doctoralDegreeFrom <http://www.University0.edu> }

Q3: SELECT ?P ?S WHERE { ?P ub:worksFor ?D . ?S ub:memberOf ?D . ?D ub:subOrganizationOf <University0> }

Q4(original): SELECT ?X ?Y WHERE { ?X rdf:type ub:Lecturer . ?Y rdf:type ub:Department . ?X ub:worksFor ?Y . ?Y ub:subOrganizationOf <University0> }

Q5: SELECT ?X ?Y ?Z WHERE { ?X rdf:type ub:UndergraduateStudent . ?Y rdf:type ub:FullProfessor . ?Z rdf:type ub:Course . ?X ub:takesCourse ?Z . ?Y ub:teacherOf ?Z }

Q6: SELECT ?X ?Y ?Z WHERE { ?X rdf:type ub:UndergraduateStudent . ?Y rdf:type ub:FullProfessor . ?Z rdf:type ub:Course . ?X ub:advisor ?Y . ?Y ub:teacherOf ?Z }

Q7: SELECT ?X ?Y ?Z WHERE { ?X a ub:GraduateStudent . ?Z ub:subOrganizationOf ?Y . ?X ub:memberOf ?Z . ?Z a ub:Department . ?Y a ub:University . }

Q8: SELECT ?X ?Y ?Z WHERE { ?X a ub:GraduateStudent . ?X ub:undergraduateDegreeFrom ?Y. ?Z ub:subOrganizationOf ?Y . ?Z a ub:Department . ?Y a ub:University . }

Q9(original): SELECT ?X ?Y ?Z WHERE { ?X a ub:GraduateStudent . ?X ub:undergraduateDegreeFrom ?Y. ?Z ub:subOrganizationOf ?Y . ?X ub:memberOf ?Z . ?Z a ub:Department . ?Y a ub:University . }

Q10(original): SELECT ?X ?Y ?Z WHERE { ?X rdf:type ub:Undergraduate Student . ?Y rdf:type ub:FullProfessor . ?Z rdf:type ub:Course . ?X ub:advisor ?Y . ?X ub:takesCourse ?Z . ?Y ub:teacherOf ?Z }

Q11: SELECT ?X ?Y ?E WHERE { ?X rdf:type ub:Undergraduate Student . ?X ub:takesCourse ?Y . ?X ub:memberOf ?Z . ?X ub:advisor ?W . ?W rdf:type ub:FullProfessor . ?W ub:emailAddress ?E . ?Z ub:subOrganizationOf ?U . ?U ub:name "University3" }

Q12: SELECT ?X ?Y ?Z WHERE { ?X rdf:type ub:FullProfessor . ?X ub:teacherOf ?Y . ?Y rdf:type ub:GraduateCourse . ?X ub:worksFor ?Z . ?W ub:advisor ?X . ?W rdf:type ub:GraduateStudent . ?W ub:emailAddress ?E . ?Z rdf:type ub:Department . ?Z ub:subOrganizationOf ?U }

Q13: SELECT ?X ?Y ?Z WHERE ?X rdf:type ub:FullProfessor . ?X ub:teacherOf ?Y . ?Y rdf:type ub:GraduateCourse . ?X ub:worksFor ?Z . ?W ub:advisor ?X . ?W rdf:type ub:GraduateStudent . ?W ub:emailAddress ?E . ?Z rdf:type ub:Department . ?Z ub:subOrganizationOf <University0>

Q14: SELECT ?X ?Y ?Z WHERE { ?X rdf:type ub:FullProfessor . ?X ub:teacherOf ?Y . ?Y rdf:type ub:GraduateCourse . ?X ub:worksFor ?Z . ?W ub:advisor ?X . ?W rdf:type ub:GraduateStudent . ?W ub:emailAddress ?E . ?Z rdf:type ub:Department . ?Z ub:subOrganizationOf ?U . ?U ub:name "University3" }