UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO DA UFSC
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Luiz Henrique Zambom Santana

A MIDDLEWARE FOR WORKLOAD-AWARE MANIPULATION OF RDF DATA STORED INTO NOSQL DATABASES

Florianópolis
2019

Luiz Henrique Zambom Santana

A MIDDLEWARE FOR WORKLOAD-AWARE MANIPULATION OF RDF DATA STORED INTO NOSQL DATABASES

Tese submetida ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Santa Catarina para a obtenção do título de doutor em Ciência da Computação.
Orientador: Prof. Ronaldo dos Santos Mello, Dr.

Florianópolis
2019

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Santana, Luiz Henrique Zambom Santana
A MIDDLEWARE FOR WORKLOAD-AWARE MANIPULATION OF RDF DATA STORED INTO NOSQL DATABASES / Luiz Henrique Zambom Santana Santana ; orientador, Ronaldo dos Santos Mello Mello, 2019.
96 p.

Tese (doutorado) - Universidade Federal de Santa Catarina, Centro Tecnológico, Programa de Pós-Graduação em Ciência da Computação, Florianópolis, 2019.

Inclui referências.

1. Ciência da Computação. 2. Semantic Web. 3. RDF/SPARQL. 4. Databases. 5. NoSQL. I. Mello, Ronaldo dos Santos Mello. II. Universidade Federal de Santa Catarina. Programa de Pós-Graduação em Ciência da Computação. III. Título.

Luiz Henrique Zambom Santana

A MIDDLEWARE FOR WORKLOAD-AWARE MANIPULATION OF RDF DATA STORED INTO NOSQL DATABASES

O presente trabalho em nível de doutorado foi avaliado e aprovado por banca examinadora composta pelos seguintes membros:

Prof. Dra. Vanessa Braganholo Murta
Universidade Federal Fluminense

Prof. Dr. João Eduardo Ferreira
Universidade de São Paulo

Profa. Dra. Carina Friedrich Dorneles
Universidade Federal de Santa Catarina

Certificamos que esta é a **versão original e final** do trabalho de conclusão que foi julgado adequado para obtenção do título de doutor em Ciência da Computação.

Documento assinado digitalmente
Jose Luis Almada Guntzel
Data: 17/10/2019 11:12:47-0300
CPF: 533.780.670-04

Prof. José Luís Almada Güntzel, Dr.
Coordenador do Programa

Documento assinado digitalmente
Ronaldo dos Santos Mello
Data: 17/10/2019 10:34:49-0300
CPF: 509.693.660-00

Prof. Ronaldo dos Santos Mello, Dr.
Orientador

Florianópolis, 18 de Agosto de 2019.

Este trabalho é dedicado aos meus pais Sonia e Luiz, aos meus irmãos Bruna e Dr. Eduardo e a meu filhinho lindo Vicente.

AGRADECIMENTOS

Agradeço aos meus pais, meus irmãos e meu filho pelo incentivo e paciência. Ao meu orientador por muita paciência e aos sábios conselhos durante esses quatro anos e meio. Aos colegas Geomar e Ângelo, aos professores do INE e à comunidade da UFSC.

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior- Brasil (CAPES) - Código de Financiamento 001

RESUMO

A Web Semântica tem quase 20 anos e muitos avanços foram propostos para atender a essa visão desde o artigo seminal de Berners-Lee et al., publicado em 2001. Entre esses avanços, os padrões propostos pelo W3C, como RDF e SPARQL, alcançaram uma versão madura e atualmente são empregados em muitos projetos acadêmicos e da indústria. Repositórios de grandes grafos RDF, chamados triplestores, são um tópico importante na área de gerenciamento de informações e conhecimento. Os triplestores evoluíram lado a lado com a pesquisa de banco de dados e, desde o advento dos bancos de dados NoSQL, vários triplestores incluem essa família de bancos de dados em suas arquiteturas. Esta tese apresenta um triplestore chamado WA-RDF, proposto como um middleware para manipular dados no formato RDF, mantidos em bancos de dados NoSQL, através de SPARQL. O WA-RDF propõe uma camada de armazenamento poliglota que fornece contribuições principalmente nas áreas de pesquisa de fragmentação de dados, particionamento de dados e mapeamento de RDF para múltiplos bancos de dados NoSQL. Um componente de monitoramento da carga de operações (WAc) é o pilar central do WA-RDF. Ele permite que, de acordo com o formato da consulta da carga de trabalho típica em um grafo RDF, o WA-RDF encontre a melhor estratégia de mapeamento para reduzir o tempo de resposta da consulta. Três rodadas diferentes de avaliações experimentais são descritas na tese. A primeira compara nossa proposta com um banco de dados multimodelo, demonstrando como uma solução de persistência poliglota baseada em um único modelo de acesso pode facilitar o desenvolvimento de aplicações. Em seguida, o middleware é comparado com trabalhos relacionados recentes usando o WatDiv, um benchmark RDF/SPARQL moderno e popular. Por fim, analisa-se a aplicação do WA-RDF no domínio de trajetórias semânticas de objetos em movimento.

Palavras-chave: NoSQL, RDF, middleware, workload, data fragmentation, data partitioning.

RESUMO EXPANDIDO

Introdução

Este trabalho apresenta o WA-RDF, um triplestore que propõe um middleware para armazenamento poliglota de dados no formato RDF em múltiplos bancos de dados NoSQL. O WA-RDF propõe avanços nas áreas de fragmentação, particionamento e mapeamento de dados RDF no contexto de bancos de dados NoSQL. Um componente de monitoramento da carga de operações (WAc) é o pilar central do WA-RDF. Ele permite que, de acordo com o formato de uma consulta típica executada sobre um grafo RDF, o WA-RDF encontre a melhor estratégia de mapeamento para reduzir o tempo de resposta desta consulta.

Objetivos

O objetivo geral desta tese é desenvolver um triplestore mais rápido e escalável que as abordagens atuais. Para atingir esse objetivo, os seguintes objetivos específicos foram considerados: projetar e desenvolver um componente dinâmico sensível à carga de trabalho capaz de identificar e registrar os tipos de consultas SPARQL; especificar uma arquitetura adequada para o middleware em termos de operações de manipulação de dados suportadas sobre os bancos de dados NoSQL, de acordo com as decisões baseadas no reconhecimento de carga de trabalho; projetar e desenvolver um esquema escalonável de particionamento e replicação para dados RDF; projetar e desenvolver uma estratégia de processamento de consultas sobre múltiplos bancos de dados NoSQLs; projetar um conjunto de experimentos para avaliar o middleware; executar os experimentos comparando o middleware com outros triplestores de úlltima geração.

Metodologia

A metodologia utilizada permitiu o desenvolvimento iterativo de protótipos do WA-RDF com posterior avaliação usando os benchmarks LUBM e WatDiv. Os protótipos reusaram ferramentas como o Apache Jena e os bancos de dados NoSQL MongoDB, Neo4j, Redis e Cassandra. Além disso, o WA-RDF foi comparado com o OrientDB para validar sua escalabilidade em relação a uma solução proposta pela indústria. Finalmente, o WA-RDF foi utilizado no contexto de uma aplicação que realiza manipulação de trajetórias semânticas de objetos móveis.

Resultados e Discussão

Os experimentos de avaliação desta proposta de tese demonstraram que o desempenho das consultas executadas sobre o WA-RDF apresenta uma escalabilidade linear em relação ao aumento de tamanho do volume de dados. O WA-RDF apresenta um melhor tempo médio de resposta quando comparado ao S2RDF, que é o estado da arte em termos de triplestore. Uma comparação em termos de facilidade de uso do WA-RDF em relação à outra solução que realiza persistência poliglota (OrientDB) mostra que o

uso de padrões da Web Semântica reduz a complexidade das aplicações em relação ao número de linhas de código. Além disso, o WA-RDF melhora consideravelmente o tempo de resposta em relação a uma solução concorrente, denominada SECONDO, no domínio de trajetórias semânticas.

Considerações Finais

A presente tese oferece evidências de que o uso de múltiplos bancos de dados NoSQL pode aumentar a escalabilidade de triplestores. Ademais, ela também demonstra que diferentes tipos de operações da carga de trabalho de uma aplicação podem se beneficiar de procedimentos específicos em relação à fragmentação, particionamento, mapeamento e armazenamento de dados. Em termos de trabalhos futuros, considera-se o uso e avaliação de outros bancos de dados NoSQL, o desenvolvimento de um triplestore que gerencia dados puramente em memória, a consideração de outras construções da linguagem SPARQL, a utilização de técnicas de aprendizado de máquina e a inclusão de bancos de dados NewSQL para a persistência de dados RDF.

Palavras-chave: NoSQL, RDF, middleware, workload, data fragmentation, data partitioning.

ABSTRACT

The Semantic Web has almost 20 years and many advances were proposed in order to meet this vision since the seminal paper of Berners-Lee et al., published in 2001. Among these advances, the standards guided by W3C, like RDF and SPARQL, achieved a mature version and are currently employed in many academic and industry projects. Repositories of large RDF graphs, called triplestores, are an important topic on information and knowledge management area. The triplestores evolved hand-to-hand with the database research, and since the advent of the NoSQL database, several triplestores include this family of databases in their architectures. This thesis presents a triplestore called WA-RDF (Workload-aware RDF), which is proposed as a middleware to manipulate data maintained by NoSQL databases using RDF and SPARQL. WA-RDF proposes a polyglot NoSQL storage layer that provides contributions mainly on data fragmentation, data partitioning and RDF-to-NoSQL mapping research areas. A Workload Awareness component (WAc) is the central pillar of WA-RDF. It allows that, accordingly to the query shape of typical workload over a RDF graph, WA-RDF finds the better mapping strategy in order to reduce the query response time. Three different rounds of experimental evaluations are described in the thesis. The first one compares our proposal to a multimodel database, demonstrating how a polyglot persistence solution based on a single access model can facilitate the development of applications. Then, the middleware is compared to recent baselines by using WatDiv, a modern and popular RDF/SPARQL benchmark. Finally, it is analyzed the application of WA-RDF in the domain of semantic trajectories of moving objects.

Keywords: NoSQL, RDF, middleware, workload, data fragmentation, triplestore.

LIST OF FIGURES

Figure 1 – Semantic Web Layers (Figure from https://www.w3.org/2001/12/semweb-fin/w3csw) . . . 15
Figure 2 – An example of a e-commerce application with polyglot persistence . 16
Figure 3 – Architectures for Polyglot Access . . . 17
Figure 4 – NoSQL Data Models . . . 24
Figure 5 – Architecture of the Estocada Approach . . . 26
Figure 6 – Pieces of a SPARQL Query . . . 27
Figure 7 – Main Types of SPARQL Query Shapes . . . 27
Figure 8 – WA-RDF Architecture . . . 40
Figure 9 – Fragment Creation . . . 47
Figure 10 – Fragment Partitioning . . . 48
Figure 11 – WAc: Workload Monitoring . . . 50
Figure 12 – Fragment creation . . . 51
Figure 13 – WA-RDF with Apache Spark for fragmentation and triple cleaning processing . . . 52
Figure 14 – Fragment distribution . . . 53
Figure 15 – Fragments redistribution . . . 53
Figure 16 – Internal Resource Mapping (IRM) . . . 56
Figure 17 – Overview of QProc . . . 60
Figure 18 – Merging of Fragments . . . 62
Figure 19 – Distributed Cache Management . . . 63
Figure 20 – Data Partitioning Scenario . . . 65
Figure 21 – Dictionary Design . . . 67
Figure 22 – Triple Update . . . 68
Figure 23 – WA-RDF with Apache Kafka for asynchronous processing . . . 70
Figure 24 – Example of RDF Data for a Moving Object Trajectories Application . 70
Figure 25 – Triples insertion at time t1 . . . 71
Figure 26 – Query at time t2 . . . 71
Figure 27 – Triples insertion at time t3 . . . 72
Figure 28 – Chain shape query at time t4 and triples insertion at time t5 . . . 72
Figure 29 – Experiments with WA-RDF using WatDiv benchmark . . . 76
Figure 30 – Rendezvous versus WA-RDF . . . 78
Figure 31 – WA-RDF and OrientDB comparison . . . 78
Figure 32 – Comparison of Query Performance over BerlinMOD between Secondo and Rendezvous . . . 80
Figure 33 – Comparison of Query Performance between Secondo and Rendezvous over a Larger Number of Nodes . . . 81

LIST OF TABLES

Table 1 – Comparative of related work in the RDF/NoSQL Converter category . 33
Table 2 – Comparative of related work in the Polystore category 34
Table 3 – Comparative of related work in the In-memory category 36
Table 4 – Related Work Comparison 37
Table 5 – Middleware Versions 40
Table 6 – A Taxonomy for RDF-to-NoSQL Mapping Strategies 44
Table 7 – Processing Time of the Mapping Categories (milliseconds) 45
Table 8 – Mapping GeoSPARQL into MongoDB 64
Table 9 – Qualitative comparison of WA-RDF and OrientDB 74

CONTENTS

1	**INTRODUCTION**	**14**
1.1	MOTIVATION	14
1.1.1	**Polyglot Persistence**	**16**
1.1.2	**NoSQL-based Triplestores**	**17**
1.2	HYPOTHESIS	18
1.3	OBJECTIVE	19
1.4	SCOPE	19
1.5	CONTRIBUTIONS	20
1.6	THESIS ORGANIZATION	21
2	**BACKGROUND**	**23**
2.1	NOSQL DATABASES	23
2.2	RDF AND SPARQL	26
3	**RELATED WORK**	**29**
3.1	RDF/NOSQL CONVERTER	29
3.2	POLYSTORE	32
3.3	IN-MEMORY	34
3.4	WORKLOAD-AWARE TRIPLESTORES	36
3.5	FINAL REMARKS	38
4	**WA-RDF**	**39**
4.1	WA-RDF ARCHITECTURE	40
4.1.1	**Architectural Decisions**	**42**
4.1.1.1	NoSQL databases	42
4.1.1.2	Mapping decisions	43
4.2	STORAGE	46
4.2.1	**Workload-awareness monitoring**	**49**
4.2.2	**Fragmentation**	**50**
4.2.3	**Mapping**	**54**
4.2.3.1	Internal Resource Mapping	55
4.3	QUERYING	56
4.3.1	**Querying details**	**59**
4.3.2	**Cache Management**	**62**
4.3.3	**GeoSPARQL**	**63**
4.4	PARTITIONING	64
4.4.1	**Dictionary Design**	**66**
4.5	UPDATE AND DELETE	67
4.5.1	**Pending updates and delete**	**68**
4.5.2	**Asynchronous Processing**	**69**

4.6 RUNNING EXAMPLE 69
4.7 FINAL REMARKS 73
5 EXPERIMENTAL EVALUATIONS 74
5.1 QUALITATIVE ANALYSIS 74
5.2 PERFORMANCE EVALUATION 74
5.2.1 WatDiv Benchmark 75
5.2.2 Comparison with a Industry Multimodel Database 77
5.2.3 Middleware Application in a Semantic Trajectory Domain 79
5.3 FINAL REMARKS 81
6 CONCLUSION 82
6.1 LIMITATIONS OF THE THESIS 83
6.2 FUTURE WORKS 83
6.3 PUBLICATIONS 84
REFERENCES 86
APPENDIX A – RDF TRIPLES FOR THE MAPPING EXPERIMENTS 95
A.1 LEHIGH UNIVERSITY BENCHMARK(LUBM) 95
APPENDIX B – CONFIGURATION APPENDIX 96

1 INTRODUCTION

In the last decade, the RDF data model, along with other *Semantic Web* technologies (BERNERS-LEE; HENDLER; LASSILA, et al., 2001), like OWL, RDFS, and SPARQL, was affected by a range of data management challenges, like data integration, search optimization, and information extraction. The main reason for that is the current scale of *Big Data* applications (*e.g.*, smart cities, sensor networks and eHealth), which generates very large datasets and need to efficiently store massive RDF graphs that goes beyond the processing capacities of existing RDF storage systems operating on a single node (HOSE; SCHENKEL, 2013).

Moreover, recent data-centric applications often deal with diverse and heterogeneous datasets, some of them highly structured and some of them with a little structure. For many scenarios, the biggest challenge is to transform a pile of information into knowledge available to a massive number of users. For instance, a *Smart City* application usually collects data from many sources, such as highly structured information from official datasets, unstructured datasets from sensor networks, and geographical datasets from GPS records (SANTANA; CHAVES, et al., 2016), which are processed into actionable knowledge to governments and citizens. Other scenarios such as *semantic trajectory management* (MELLO et al., 2019) can also benefit from the RDF model usage, both by the facilities on describing the models and the possibility of using scalable tools.

1.1 MOTIVATION

RDF is been around for almost 20 years already. However, it never got to the mainstream of the software development industry. There are multiple reasons for that: the verbosity, the low community engagement, but mainly, the lack of good tools to support it. In the recent years, many tools are trying to bridge the gap between the industry and the RDF standard. The Amazon cloud solution for RDF storage, called Neptune[1], for example, facilitates the access to a cloud-based triplestore. Products like Ontobroker[2], IQser[3], and the consideration of RDF data by Volvo to represent its autonomous car[4] shows that the Semantic Web technologies are gaining more space in the industry. On the other hand, RDF data management is still a very hot research topic. Its abstraction power, reason possibilities and easiness on creating knowledge representation, yet been simple to humans understand, is still very powerful. Recent advances on the use of RDF, like the Florence project (BELLINI; NESI, 2018) and the

[1] https://docs.aws.amazon.com/neptune
[2] http://www.semafora-systems.com/en/products/ontobroker/
[3] https://instantli.com/
[4] https://www.youtube.com/watch?v=mpLcTSG7VPg

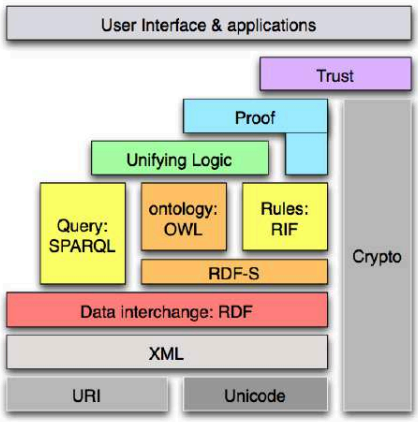

Figure 1 – Semantic Web Layers (Figure from https://www.w3.org/2001/12/semweb-fin/w3csw)

recently proposed benchmark for RDF as the spatial data (HUANG; RAZA, et al., 2019), show that the RDF usage is not only possible but desirable.

Traditionally, the Semantic Web is presented as a stack of layers where each layer supports the lower ones, as presented in Figure 1. The results of this thesis can benefit RDF and SPARQL layers and leverage other upper layers. In fact, this thesis aims to unlock or leverage use cases that deal with RDF data at large scale, like:

- Storage layer of smart city applications;
- Unified RDF views generation from multiple databases;
- Software development using multiple databases;
- Scalable reasoning by permitting plug-and-play integration with Apache Spark and other machine learning tools.

The research focus of this thesis is efficient management of RDF data persisted into multiple NoSQL databases, given the increasing need for RDF-based solutions and the benefits of polyglot persistence introduced by the several NoSQL data models (SADALAGE; FOWLER, 2012). The crossing of these two research topics (RDF and NoSQL-based data management) is still an open issue that can raise contributions, like cross-database consistency control, data mapping between the RDF data model and the NoSQL data models, and scalability improvement of NoSQL databases when dealing with RDF data. All of these points are considered in this thesis. We discuss more about polyglot persistence and NoSQL-based triplestores in the following.

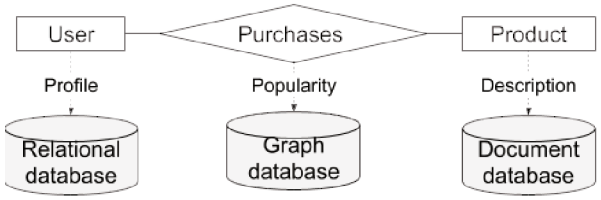

Figure 2 – An example of a e-commerce application with polyglot persistence

1.1.1 Polyglot Persistence

Since the advent of NoSQL databases (CATTELL, 2011; SADALAGE; FOWLER, 2012), many modern applications, instead of choosing only one single data management system, are considering the so-called *polyglot persistence* to satisfy their requirements. Services like StackShare[5] holds several applications that follow this approach. Uber[6], for instance, stores data into Postgres, Redis and Cassandra through its data layer.

Designing and developing a data layer have several contradictory requirements (WIESE, 2015). Many of the decisions necessarily have to accommodate different access patterns (write-heavy versus read-heavy workloads), data models (coexistence of social network features fitted to graph structures and billing components fitted to tables) and access methods (REST interface and query languages)(SHARP et al., 2013). Several domains benefit from the polyglot persistence, like e-commerce (SRIVASTAVA; SHEKOKAR, 2016) and e-healthcare (KAUR; RANI, 2015). Figure 2, for example, shows part of the storage strategy followed by an e-commerce application. It considers a relational database for maintaining user profiles, a graph database to store the popularity of products, and a document database to save product descriptions.

These problems have been faced by industry and academic communities in different ways. There are *polyglot frameworks* such as the *Spring Data*, which resides on the software client and allows access to different databases using the same abstractions[7]. On the other extreme there are the *multimodel databases*, such as *OrientDB*[8] and *CosmosDB*[9], which encapsulate different data models into a unique storage solution. Finally, approaches like *Estocada* (BUGIOTTI et al., 2015) propose a *middleware* to manage the communication between an application and different databases. Thus, we argue, based on a literature review, that there are three main architectures for a polyglot system, as illustrated by Figure 3.

In a client-server system, the polyglot component can exist into the client data

[5] https://stackshare.io/
[6] https://stackshare.io/uber-technologies/uber
[7] https://spring.io/projects/spring-data
[8] https://orientdb.com/
[9] https://azure.microsoft.com/en-us/services/cosmos-db/

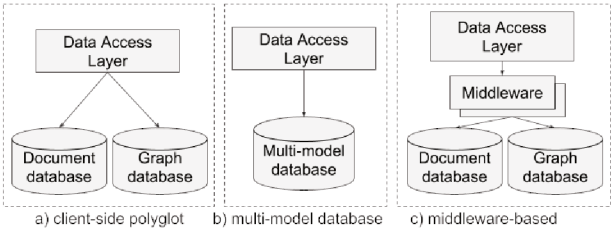

Figure 3 – Architectures for Polyglot Access

layer, such as *Spring Data* and *Doctrine*[10], which are an evolution of the *Object-Relational Mapping (ORM)* for accessing different databases using the same abstractions. These solutions are limited because they delegate several tasks to the application, like consistency management among the data sources. We also have the multimodel databases, such as *OrientDB* and *CosmosDB*. Their architecture is also restricted because most of the current solutions do not have a unique abstraction to access all the supported data models. Finally, some architectures propose a middleware to manage the communication between an application and databases with heterogeneous data models. Middlewares are in the core of recent proposals for NoSQL polyglot access. In this thesis, a middleware solution is proposed.

1.1.2 NoSQL-based Triplestores

Since the rise of the RDF format, the RDF storage has been evolving to meet the latest requirements of the Computer Science trends. This subject has been the focus of a great variety of surveys and benchmark papers, based, for instance, on relational databases and peer-to-peer systems (SAHOO et al., 2009; SAKR; AL-NAYMAT, 2010; FILALI et al., 2011; LUO et al., 2012). Out of this variety, since 2012 the NoSQL movement plays an important role in the related work.

There are many works related to polyglot NoSQL systems and scalable RDF data management (MA; CAPRETZ; YAN, 2016). Recently, the workload-awareness was recognized as an important feature for the development of scalable triplestores, as described by Aluç, Ozsu, and Daudjee (ALUÇ; ÖZSU; DAUDJEE, 2014). The lack of awareness can easily lead to important performance barriers: *(i)* naive data fragmentation; *(ii)* poor data localization; and *(iii)* unnecessarily large intermediate results. Thus, although many works propose NoSQL-based triplestores, the current solutions fall short because of RDF structural diversity and dynamism of SPARQL workloads. Thus, the cutting-edge of triplestores are normally the workload-aware and NoSQL-based solutions.

Even following different approaches, there are not proposals in the literature re-

[10] https://www.doctrine-project.org

garding RDF data management with workload-awareness that consider polyglot persistence into NoSQL databases with more than one data model. Besides, the close-related proposals present several limitations, as follows:

- None of the proposals is able to rearrange the RDF dataset already persisted into the physical storages, when the workload changes, in order to improve query processing;
- Lack of support to an efficient data fragmentation strategy, i.e., queries access RDF data in a triple-based level, which leads to several join processings that limit query performance;
- Naive support to partitioning and data replication, which leads to cross-partition joins that also slow down the query response;
- Current indexing solutions are complex and, in most of the cases, generates index structures for all the permutations of the RDF triple components;
- Related solutions, like S2RDF (SCHÄTZLE; PRZYJACIEL-ZABLOCKI; SKILEVIC, et al., 2016) and Rainbow (GU; HU; HUANG, 2014), are optimized to only one type of workload, usually star-shaped queries;
- No support for data reorganization in order to improve data localization for further query processing;
- Most of related work do not support update and deletion processes. This is even more evident among the workload-aware triplestores because data rearrangement is required.

These limitations guided the design and implementation of a new triplestore. This triplestore, materialized as a middleware that accesses multiple NoSQL databases, is called *WA-RDF*. WA-RDF is based on a workload-awareness approach that rules RDF data partitioning, fragmentation and caching strategies. Moreover, it considers well-known *Big Data* frameworks that facilitate the processing and background operations, as well as enable update, delete and data reorganization tasks.

1.2 HYPOTHESIS

As stated before, the motivation of this thesis is to enable the usage of multiple NoSQL databases to manage RDF data in a more efficient way than the current state-of-the-art proposals. We claim that such efficiency can be reached by considering that:

1. The performance of RDF-based applications can benefit of a dynamic workload-aware triplestore. Such an approach permits to preprocessing query joins through an efficient data fragmentation the partitioning of an RDF graph;

2. Multiple NoSQL databases with different data models can coexist in a triplestore;
3. A middleware connecting an RDF-data-centric application to multiple and heterogeneous NoSQL databases is a promising architectural solution for developing a new triplestore.

1.3 OBJECTIVE

This thesis is related to the joint subjects of RDF data management and NoSQL databases. The state-of-the-art reveals that this is an important and open topic. Thus, the main objective of this thesis is to propose a workload-aware middleware for RDF data manipulation based on multiple and heterogeneous NoSQL databases that be faster and more scalable then the state-of-the-art approaches when measured by the current Semantic Web benchmarks. In order to achieve this main objective, the following specific objectives must be addressed:

- To design and develop a dynamic workload-aware component able to identify and register the types of SPARQL queries;
- To specify a suitable architecture for the middleware in terms of the supported data manipulation operations and the NoSQL databases mix, accordingly to the workload-aware decisions;
- To design and develop a scalable partitioning and replication schema for RDF data;
- To design and develop a query processing strategy that uses multiple and heterogeneous NoSQL databases;
- To design a set of experiments to evaluate the middleware with the *WatDiv* benchmark, which is the more frequent used benchmark for triplestores;
- To execute the experiments by comparing the middleware against the state-of-the-art baselines.

1.4 SCOPE

The development of the proposed middleware can be achieved by many different ways. However, this thesis have a limited scope due to several constraints, including time and infrastructure. So, the following aspects constitute its scope:

- **Mapping**: effective ways to translate RDF data into the NoSQL data models;

- **Workload-awareness**: monitoring and registering the workload is crucial for effectively answer incoming queries;
- **Fragmentation, partitioning and replication**: these managements allow our middleware be scalable as the dataset increases.

Based on this scope, this thesis presents results that expand the state-of-the-art. However, other important aspects are out of the scope or are not deeply considered:

- Relational databases: traditional relational databases are not taken into consideration in this thesis. There are multiple motivations for this decision, including its limited scalability, its rigid schema, and the fact that the tabular model is satisfied by a columnar NoSQL database;
- Indexing: indexing is an important topic for triplestores. Many options were studied in the last years and this thesis option was to create a lightwheight index that is used only as a support for activities such as fragmentation, and for responding the simplier queries. This thesis do not evaluate the efficacy of the indexes proposed for the middleware;
- In-memory: in-memory capabilities is a new trend for triplestores. The proposed middleware explores it partially by using *Apache Spark* framework[11];
- Caching: cache support for RDF data is another important topic that was not completely explored in the middleware. An architectural contribution in that sense could be to develop a cache manager that considers the middleware fragmentation strategy to divide the cache into a local and remote cache. Some details about this issue are given in Chapter 4;
- NewSQL movement: recent relational databases (e.g., Google Spanner and VoltDB) comprise the so-called *NewSQL movement* (PAVLO; ASLETT, 2016). The NewSQL database family aims at providing scalable and ACID-compliant databases based on the SQL access interface. This thesis do not consider NewSQL databases for two reasons: *(i)* their development was in an initial stage during the beginning of this thesis; *(ii)* some NoSQL data models are more suitable to represent RDF graphs than the relational data model.

1.5 CONTRIBUTIONS

We consider the main contribution of this thesis a set of strategies that are joint-supported by a new middleware proposal responsible to process manipulation

[11] https://spark.apache.org/

operations over RDF data stored into NoSQL databases. The middleware advocates for an architecture where the data management is workload-awareness. These strategies are the following:

- **RDF-to-NoSQL mapping**

  RDF triples are converted into NoSQL document and graph data. Moreover, the middleware considers the NoSQL key/value database for caching and the NoSQL columnar database for the partitioning dictionary. Thus, the middleware defines processes to convert RDF data into the NoSQL data models and vice-versa.

- **Workload-awareness**

  The cornerstone of the middleware architecture is a *workload-awareness component (WAc)* that monitors and registers information about each incoming SPARQL query. WAc classifies a query into simple, star, chain and complex. The workload information is considered during triple insertion.

- **Multiple databases in the main storage**

  WA-RDF is the first workload-aware triplestore to employ two NoSQL databases in the storage level. Systems like ScalaRDF and Rainbow consider multiple databases (in both cases, they use a key-value database as cache) but only one storage is responsible for maintaining the RDF data. Additionally, WA-RDF is able to, during a triple storage, decides on translating a RDF triple into a NoSQL document or graph database (or both) according to the usual query workload.

- **Full data manipulation support**

  WA-RDF is the unique workload-aware triplestore that support all the data manipulation operations over RDF data (insert, update, delete and query operations). As presented in Section 4.5.1, this is possible due to its queuing strategy.

- **Query processing**

  A query processing strategy aims at mainly avoiding joins between data partitions to reduce the volume of intermediate query results and dynamically choose the best NoSQL node to query.

All of these strategies are proven to be effective through a set of experiments that compared performance and scalability of the middleware against close related work.

1.6 THESIS ORGANIZATION

The rest of this thesis is organized as follows. Chapter 2 provides a short background and chapter 3 discusses related work. Chapter 4 details the middleware, its

novel strategies, as well as some relevant design and implementation issues that contributes to its good performance. Chapter 5 presents the set of experiments considered to validate the proposal, and chapter 6 is dedicated to the conclusion.

2 BACKGROUND

This chapter introduces the necessary concepts to understand the rest of this thesis. It begins with an overview of Big Data as a motivating scenario for NoSQL databases and their related technologies. In the following, we focus on RDF data.

2.1 NOSQL DATABASES

The vast pile of data generated by persons and application systems today, available in several data sources, is a gold mine (CALDAROLA; PICARIELLO; CASTELLUCCIA, 2015). Modern enterprises, for example, are using them as an unlimited source of knowledge and insights about the habits and preferences of their current and potential costumers. However, storing data and extracting information from these sources is not a trivial task. By definition, *Big Data* is a moving target since today's Big Data is not the Big Data of tomorrow. Thus, the term Big Data itself is going to disappear in the next years, or it will return to be called just data (CALDAROLA; PICARIELLO; CASTELLUCCIA, 2015). In fact, when we talk about Big Data we mean the capability of computational systems to generate, store and share large volumes of data with different natures.

In the Big Data context, one of the first categories of database systems proposed to deal with this tremendous amount of data with potential heterogeneous representations is called NoSQL. The standard features of these database systems are: no relational data model; no join support; distribution; massive horizontal scaling; no fixed and flexible schema; replication support; procedural query processing instead of a standard declarative query language; and consistency guarantee within a node and eventually consistent assurance across the cluster.

The term NoSQL was coined by Carlo Strozzi in 1998 to describe his database, which offered an access interface not based on SQL. Eric Evans revisited the concept in 2009 to describe non-relational and distributed databases that do not adhere to atomicity, consistency, isolation and durability, essential properties of relational databases. During the subsequent years, many discussions set some formalization to this subject. A consensus attempt, as defined by Sadalage and Fowler (MARZ; WARREN, 2015; SADALAGE; FOWLER, 2012), is that NoSQL comprises database systems that use more than one storage mechanism, including new types of which are not compatible with the relational database model (RAMAKRISHNAN; GEHRKE, 2000). Moreover, NoSQL databases are organized into four categories (see Figure 4) regarding their data models:

- *Key/Value*: this type of database focuses on reducing the data search latency. Its data structure is similar to a hash map, which limits the storage to simple and

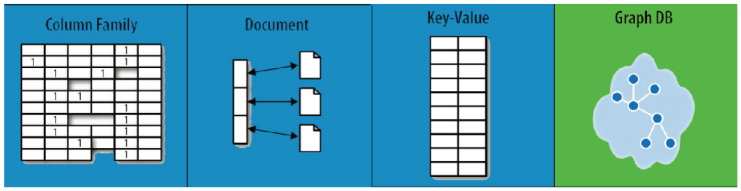

Figure 4 – NoSQL Data Models

flat information. It is especially attractive to the game industry, mobile and Web applications. Examples of this type of database are Voldemort (SUMBALY et al., 2012), Amazon DynamoDB (BRUNOZZI, 2012) and Redis (PHAM, 2013);

- *Document*: this type of database stores semistructured data represented in one of the many formats used by current computational systems (e.g., XML and mainly JSON[1]). Examples of this type of database are CouchDB (LERNER, 2010) and MongoDB (ABRAMOVA; BERNARDINO, 2013);

- *Columnar*: this type of database can store data that does not respect a rigid schema. In other words, data instances usually have a standard set of properties (columns), but may have a different number of columns. Examples of this type of database are Apache HBase (KUHLENKAMP; KLEMS; RÖSS, 2014), Apache Cassandra (ABRAMOVA; BERNARDINO, 2013), and Apache Parquet[2]. The last one is a novel columnar storage format created for Apache Hadoop and Spark, which has been used by many research initiatives (SCHÄTZLE; PRZYJACIEL-ZABLOCKI; NEU, et al., 2014; SCHÄTZLE; PRZYJACIEL-ZABLOCKI; SKILEVIC, et al., 2016);

- *Graph*: this type of database is suitable to model real-world relationships (e.g., between people and organizations) that can be represented by the graph theory. So, its data model allows the definition of nodes and edges, where nodes usually represent real-world entities, and edges represent associations between them. Both of them (nodes and edges) may hold attributes. Examples of this kind of database are TitanDB, Trinity and Neo4J (KOLOMIČENKO; SVOBODA; MLÝNKOVÁ, 2013; SHAO; WANG; LI, 2012).

From the software engineering perspective, the main factor that allowed the development of Big Data applications was the advent of the *MapReduce* programming model (DEAN; GHEMAWAT, 2008). This model was developed by Google for its search engine and aims to process significant amounts of data through batch jobs execution, being able to handle a large volume of data with high throughput. The main open source implementation of MapReduce is *Hadoop*, which provides a complete set of tools that

[1] https://www.w3.org/TR/json-ld/
[2] https://parquet.apache.org/

range from Big Data processing to Big Data persistence in a distributed file system. Thus, MapReduce and Hadoop are on the edge between a development tool and a data management system. Although the focus of this thesis is not on software engineering, MapReduce is essential not only because it is considered in the development of most NoSQL databases, but also because many data management solutions employ it in distinct ways (SCHÄTZLE; PRZYJACIEL-ZABLOCKI; NEU, et al., 2014; PAPAILIOU; DIMITRIOS TSOUMAKOS, et al., 2015; PAPAILIOU; KONSTANTINOU; TSOUMAKOS; KOZIRIS, 2012).

Nevertheless, the MapReduce paradigm and Hadoop cannot work for all the use cases of the ever involving IT market needs. In this sense, there are applications that regardless the definition of massive data repositories are also heavily accessed, and perform in-memory most of the time. Some examples of such systems are social networks, stock market analysis and telephony domains, network monitoring, fraud exposure, and military environments. Due to it, several new architectures and frameworks are been proposed to support the size and *urgency* of the Big Data (GRAY et al., 2014; TOSHNIWAL et al., 2014). These solutions, primarily lead by Apache Software Foundation, includes *Apache Storm, Apache Spark* and *Apache Flume*. The capabilities of the applications based on these solutions are in-memory processing to avoid reading to/from disk whenever possible, integration with distributed computing technologies, and real-time execution plans, being an alternative to the MapReduce ideas (LIU; IFTIKHAR; XIE, 2014).

Additionally, it is important to point out the relation of NoSQL databases with the concept of Cloud Computing. NoSQL databases are usually offered as part of a cloud solution, which permits on-demand computational power, where processing, storage, memory and network capabilities can grow (or reduce) with high elasticity, optimizing costs and facilitating the application operation (KAOUDI; MANOLESCU, 2015). In such a context, the *Lambda Architecture* (PERERA; SUHOTHAYAN, 2015) is the most recent advance in the integration of Big Data, NoSQL databases and Cloud Computing. This architecture contributes to develop a suitable approach for NoSQL databases application by introducing the idea of views over Big Data repositories, so it is possible to slice and dice the data according to the necessary information retrieval.

Finally, the *Polyglot Persistence* is an important background concept of this thesis. This concept, firstly coined by Marting Fowler[3], argues that complex applications can accomodate several data storage techniques, models and technologies for different natures of data. For instance, as presented in Figure 2, an e-commerce application can benefit of this issue to improve scalability and simplify the data management by using more than one storage type.

Approaches like Estocada (BUGIOTTI et al., 2015), BigDAWG (DUGGAN et al.,

[3] https://martinfowler.com/bliki/PolyglotPersistence.html

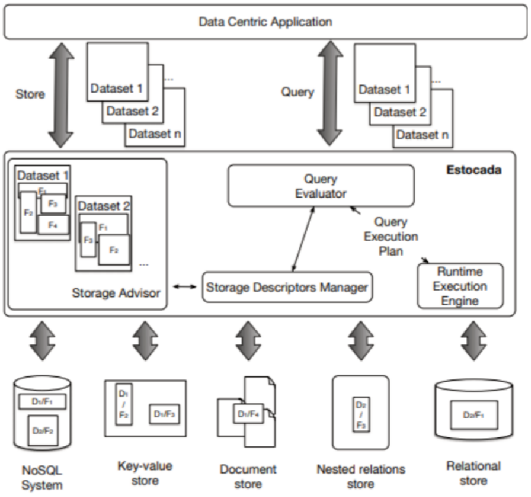

Figure 5 – Architecture of the Estocada Approach

2015) and Presto (MAMMO; BANSAL, 2015) are examples of middlwares that are expanding the possibilities of the NoSQL databases to decide which storage and data model can better handle parts of a dataset. The architecture of this thesis is specially inspired by the *Estocada* architecture (see Figure 5), where a middleware connects diverse and heterogeneous data stores, and by defining fragments, can route parts of the dataset to the specific database model where each fragment is more suited.

2.2 RDF AND SPARQL

One of the most important pillars of this thesis is the *Semantic Web*, as envisioned by Tim Berners-Lee in 2001 (BERNERS-LEE; HENDLER; LASSILA, et al., 2001). The Semantic Web offers, as practical value, the development of applications that can handle complicated human queries based not only on simple matches of raw data but also on its meaning. When the Semantic Web was introduced, the current amount of information generated and available could not be foreseen by most of the specialists, but the need for data integration was already argued as one of its fundamental purposes. Thus, the effort of developing the Semantic Web was harvested mainly in the form of well-established standards for expressing shared meaning, defined by the WWW Consortium (W3C)[4], like *Resource Description Framework (RDF)* and *the Simple Protocol and RDF Query Language (SPARQL)*.

RDF is expressed by triples that define a relationship between two resources. RDF triples can be modeled as graphs, where the resources, called *subject* (S) and *object* (O), are vertexes, and the relationship, called *predicate* (P), is a directed edge from the subject to the object. For instance, we can define a predicate *:owns* between

[4] https://www.w3.org/

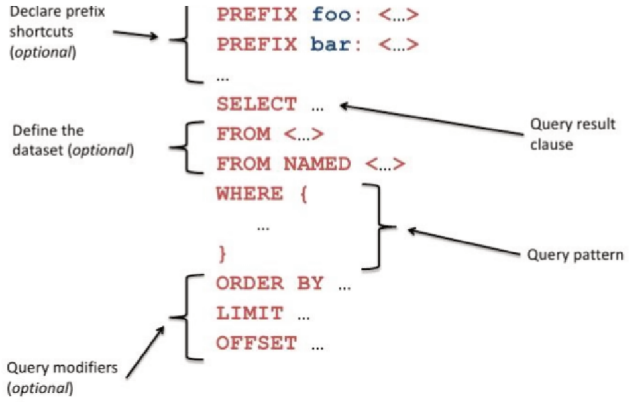

Figure 6 – Pieces of a SPARQL Query

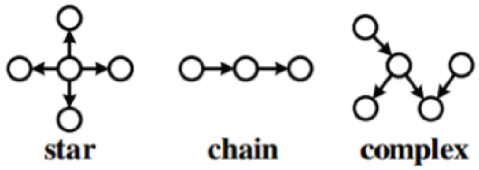

Figure 7 – Main Types of SPARQL Query Shapes

two resources: *:person* (subject) and *:car* (object).

SPARQL is a query language for searching and retrieving RDF data. As presented in Figure 6, a query statement in SPARQL consists of triple patterns, conjunctions, disjunctions and optional patterns. The *triple pattern* defines the RDF subject, predicate and object to be searched, *conjunctions and disjunctions* express the intended relations between the searched resources, and the *optional patterns* combine two graph models. Moreover, sets of triple patterns define *Basic Graph Patterns (BGP)*, being each BGP a function that transforms the RDF dataset into result sets. These result sets are the answer to a SPARQL query in the form of RDF triples.

Traditionally, SPARQL queries can be categorized into *star*, *chain* and *complex* queries (GALLEGO et al., 2011), as presented on Figure 7. These shapes depend on the location of variables in triple patterns, which can influence the query performance disregarding the edge direction (SCHÄTZLE; PRZYJACIEL-ZABLOCKI; SKILEVIC, et al., 2016). The *star* shape is characterized by subject-subject (S-S) or object-object (O-O) joins within triple patterns as the join variable is located on the subject or the object. The *chain* shape is composed of object-subject (O-S) joins, *i.e.*, the join variable is in the subject location in one pattern and on object location in the other one. *Complex* shapes are combinations of the two previous ones, which can be connected by simple query filters like an object equals to a constant value.

Given, for instance, the SPARQL query:

```
SELECT DISTINCT ?X ?Y ?Z
WHERE {
?Y rdf:type ub:Faculty.
?Y ub:teacherOf ?Z.
?X ub:advisor ?Y.
?X ub:takesCourse ?Z.
?X rdf:type ub:Student.
?Z rdf:type ub:Course.}
```

we have a star shape around the subject *?X* and a chain shape composed of *?X ub:advisor ?Y, ?Y ub:teacherOf ?Z,* and *?Z rdf:type ub:Course*.

The development of systems capable of storing RDF and retrieving data via SPARQL (usually called *triplestores*) has a long tradition. This kind of system figures at multiple surveys, as presented in the next chapter. During this time, many advances were presented like the pioneers AllegroGraph (CHANG; MILLER, 2009), Stardog (CERANS et al., 2012), YARS (HARTH; DECKER, 2005), Hexastore (WEISS; KARRAS; BERNSTEIN, 2008), 4store (HARRIS; LAMB; SHADBOLT, 2009), SPIDER (CHOI; SON, et al., 2009), RDF-3X (NEUMANN; WEIKUM, 2010), SHARD (HUANG; ABADI; REN, 2011), swStore (ABADI et al., 2009), SOLID (CUESTA; MARTINEZ-PRIETO; FERNÁNDEZ, 2013), S2X (SCHÄTZLE; PRZYJACIEL-ZABLOCKI; BERBERICH, et al., 2015), BigOWLIM (KIRYAKOV et al., 2010) and Sesame (BROEKSTRA et al., 2002).

As detailed in the next chapter, the works in the literature related to this thesis comprise triplestores that uses NoSQL databases as the main storage.

3 RELATED WORK

Triplestores, also called RDF stores, is a database for the storage and retrieval of triples. Many triplestore proposals use NoSQL databases for RDF storage, and it is possible to group them in many different classifications. As the main focus of this thesis is on NoSQL persistence and its implications, we organize the related work into three categories that denote, in our opinion, the latest advances in NoSQL databases:

- *RDF/NoSQL converter*: this category is related to middlewares that convert RDF data into NoSQL data models and vice-versa. Most of the seminal proposals (e.g., CumulusRDF (LADWIG; HARTH, 2011) and Jena-HBase (KHADILKAR et al., 2012)) are included in this category.
- *Polystore*: polystore solutions are gaining momentum in the last few years. As cited in Chapter 2, proposals like Estocada (BUGIOTTI et al., 2015), BigDAWG (DUGGAN et al., 2015) and Presto (MAMMO; BANSAL, 2015) are expanding the possibilities of the NoSQL databases to smartly decide which kind of storage, with an specific data model, can better handle parts of the considered dataset. In recent years, a handful of works were published to explain how polystore databases can better manage the RDF data complexity;
- *In-memory*: in-memory databases is a current trend for Big Data management (STOREY; SONG, 2017). This category comprises works that use structured in-memory solutions (e.g., Parquet, Memcached and Redis) for storing data in one of the NoSQL data models.

The related approaches on each category are detailed in the next sections, and a comparison of them per category is shown in the Tables 1, 2 and 3, respectively. For each category, we analyze, at least, the work *focus*, *physical storage* and *NoSQL data model*. Also, for each category we added specific features. It is worth mentioning that an approach may fit into more than one category. In this case, we decided to put it into the category that it offers more contribution.

An extended version of this chapter was submitted as a survey to the *Transactions on Knowledge and Data Engineering (TKDE)* journal and is currently under review.

3.1 RDF/NOSQL CONVERTER

There are many proposals that fit into this category. RDFJoin (MCGLOTHLIN, J.; KHAN, L., 2009), for example, uses vertical partitioning and sextuple indexing for storing data on MonetDB and LucidDB databases. Their data model focus on performance optimization of join queries. They also employ BitMat (ATRE; SRINIVASAN; HENDLER,

2008) for speedup lookups and joins since each subject and object URI is converted to an integer that represents the corresponding bit index in a bit matrix. Their tests using LUBM benchmark show that RDFJoin outperforms Hexastore on join time. RDFKB (Resource Description Framework Knowledge Base) (MCGLOTHLIN, J. P.; KHAN, L. R., 2009) also uses MonetDB for RDF datasets, having focus on inference and knowledge management. It supports reasoning at storage time instead of during query processing. In comparison to RDFJoin, the queries are simplified and the performance is enhanced. RDFKB and RDFJoin were proposed by the same research group.

Stratustore (STEIN; ZACHARIAS, 2010) is an RDF store implemented over Amazon SimpleDB using Jena framework. It accomplishes a triple-oriented mapping, i.e., a triple is mapped to a data item with the attributes subject, predicate and object. AMADA (ARANDA-ANDÚJAR et al., 2012) is a platform for RDF persistence based on the Amazon Web Services (AWS) that also considers SimpleDB. It operates in a Software as a Service (SaaS) approach, enabling upload, index, store and query over large volumes of Web data. The NoSQL usage is limited to data indexing since the RDF triples are stored on Amazon S3 distributed filesystem.

The HBase NoSQL database is considered by many works. H2RDF (PAPAILIOU; KONSTANTINOU; TSOUMAKOS; KOZIRIS, 2012) is a fully distributed RDF store that combines the MapReduce processing framework with HBase. The main focus is on simple and multi-joins. Vaibhav et al. (KHADILKAR et al., 2012) proposed Jena-HBase, which stores RDF indices in HBase and directly carries out queries through HBase APIs. Map-Side Index Nested Loop Join (MAPSIN join) (SCHÄTZLE; PRZYJACIEL-ZABLOCKI; DORNER, et al., 2012) combines the scalable indexing capabilities of HBase with MapReduce paradigm provided by Hadoop. The central goal of this work is the large-scale join processing while maintaining the flexibility of commonly used reduce-side joins, leveraging the effectiveness of map-side joins. The authors demonstrate that, for most queries, MAPSIN join-based query execution outperforms reduce-side join-based execution by an order of magnitude. RDFChain (CHOI; JUNG; LEE, 2013) also combines MapReduce and HBase. The storage scheme of RDFChain aims to decrease the number of storage accesses. On confronted with the other proposals, their cost-based map-side join reduces the number of map jobs since it uses statistics to process several join strategies in a single map job. At last, Rainbow (GU; HU; HUANG, 2014) manages RDF data with two layers aiming to deal with RDF in a scalable way. The bottom layer is represented by HBase, which is responsible for storing the RDF data. The top layer is represented by an in-memory cache to speed up query processing.

The Cassandra NoSQL columnar database is used by CumulusRDF (LADWIG; HARTH, 2011). This work introduces hierarchical and flat layout types for mapping RDF data to Cassandra data model. Another approach called SPOVC (MULAY; KUMAR,

2012) organizes RDF data for efficient evaluation of SPARQL queries through three types of indexes (subject, predicate, object) on top of any column oriented database. The main techniques used by them are the horizontal partitioning of the logical indexes and specific indexes for values and classes. SPOVC extends SW-Store (ABADI et al., 2009), which uses vertical partitioning of PostgreSQL to respond SPARQL queries with filter patterns having range conditions and regular expressions, obtaining a two/three-fold performance improvement.

The work of Pham (PHAM, 2013; PHAM; BONCZ, 2016) proposes a self-organizing RDF storage in MonetDB where on RDF ingestion time the system automatically detects structure in the data. During the ingestion, MonetDB can be used as a key/value or a document storage. This knowledge is then used to store the data in a structured form to accelerate RDF joins for queries with star patterns.

Graph databases are considered by the work of Bouhali and Laurent(BOUHALI; LAURENT, 2015), which defines transformation rules from RDF data to the graph database model. They propose a direct RDF graph-database graph mapping, along with other four mapping strategies (Literal Datatype Mapping, Literal Mapping, Resource Type Mapping and Structural Mapping) to transform a RDF graph into a database graph. During the evaluation, tests with real data over Neo4j graph database obtained promising performance results.

There is a handful of works that is independent of storage but focuses only on the translation between RDF and JavaScript Object Notation (JSON). As JSON is the common storage format adopted by the NoSQL document databases, we consider these works as following the document data model. The traditional Semantic Web framework *Apache Jena*[1] includes a component that has the capability of converting RDF to JSON for persisting RDF data in NoSQL document databases. This work is independent of database solution, and do not include partitioning or indexing solutions. W3C RDF/JSON[2] is a concrete syntax for RDF into JSON, as defined in the RDF Concepts and Abstract Syntax W3C Recommendation. On considering also NoSQL document databases, the work of Tomaszuk (TOMASZUK, 2010) adopts MongoDB as a RDF triplestore by defining an alternative mean for serializing RDF triples using JSON.

Rya (PUNNOOSE; CRAINICEANU; RAPP, 2012) uses a columnar NoSQL database called Accumulo to develop a scalable system for storing and retrieving RDF data in a cluster of nodes. It introduces a serialization format for storing RDF data, an indexing method to provide fast access to data and query processing techniques for speeding up the evaluation of SPARQL queries. Experimental evaluations show that the system scales RDF data storage to billions of triples and provides millisecond query

[1] https://jena.apache.org/
[2] https://www.w3.org/TR/json-ld/

times.

The Hive+HBase described in the survey of Cudre-Maurox et al. (CUDRÉ-MAUROUX et al., 2013) uses Apache Hive, an SQL-like data warehousing tool that allows querying using MapReduce. This proposal relies on the translation from SPARQL to SQL and vice-versa. It is independent of database system and do not also cover partitioning or indexing solutions. The code implementation is available at Github author's profiles[3].

TriAl (LIBKIN; REUTTER; VRGOČ, 2013) combines the idea of RDF triplestores with that of graphs with data. This acronym for *Triple Algebra* introduces a language that works directly over triples and it is closed, i.e., they produce sets of triples rather than graphs. The closure is assured by replacing the cartesian product with a family of join operations. The authors also provide examples of the usefulness of TriAl in querying graph, RDF and social network data.

Table 1 presents the works classified as RDF/NoSQL Converters. The works are compared according to its focus, storage and data model. Some works do not propose a database system, but only a data model, so they are independent of a specific storage. Its important to notice here the volume of the works in this category: fifteen published since 2009, and none, to the best of our knowledge, since 2015. Moreover, most of them focuses on join processing or RDF-to-NoSQL mapping, are based on Columnar databases and some of them employ the MapReduce paradigm to process joins. This is probably influenced by the sound success of SW-Store (ABADI et al., 2009), the first approach to propose vertical storage for RDF triples.

As commented before, works that are mere RDF/NoSQL converters are getting rare during the last years. Probably, the reason for that is the difficulty to achieve the necessary scale using approaches that negligence partition, indexing, fragmentation and caching. The following sections show that polystores and multimodel solutions, alongside with an in-memory RDF manipulation, are the future for massive RDF data management.

3.2 POLYSTORE

The Polystore category means works that persist RDF data into more than one storage solution that usually hold different data models (multimodel solutions). In general, the goal of this kind of approach is to exploit the structural differences between parts of the RDF dataset in order to optimize the query response time. Two works fit into this category.

The project Presto defines a storage layer for RDF called Presto-RDF (MAMMO; BANSAL, 2015). Presto allows querying data in different NoSQL databases (e.g., Cas-

[3] https://github.com/ahaque/hive-hbase-rdf

Work	**Focus**	**Proc.**	**Storage**	**Model**
RDFJoin (2009)	Join processing	No	MonetDB/ LucidDB	Columnar
RDFKB (2009)	Join processing with reasoning	No	MonetDB	Columnar
Stratustore (2010)	RDF-to-columnar database mapping	No	SimpleDB	Columnar
Tomaszuk (2010)	RDF-to-JSON mapping	No	Independent	Document
Cumulus RDF (2011)	RDF-to-columnar database mapping	No	Cassandra	Columnar
Jena (2011)	RDF-to-JSON mapping	No	Independent	Document
AMADA (2012)	RDF storage in the cloud	No	SimpleDB	Columnar
H2RDF (2012)	Join processing	M/R	HBase	Columnar
Jena-HBase (2012)	Jena-based processing	No	HBase	Columnar
MAPSIN (2012)	Join processing	M/R	HBase	Columnar
SPOVC (2012)	Horizontal data partitioning	No	Independent	Columnar
Rya (2012)	Scalable RDF storage and querying	No	Accumulo	Columnar
Pham (2013)	RDF structure analysis	No	MonetDB	Columnar
Hive+HBase (2013)	SPARQL-to-SQL mapping	M/R	HBase	Columnar
RDFChain (2013)	Join processing	M/R	HBase	Columnar
TriAL (2013)	RDF-to-graph database mapping	No	Independent	Graph
RDF/JSON (2014)	RDF-to-JSON mapping	No	Independent	Document
Rainbow (2014)	Scalable querying	No	HBase	Columnar
Bouhali (2015)	RDF storage into graph databases	No	Graph	Graph

Table 1 – Comparative of related work in the RDF/NoSQL Converter category

Work	**Focus**	**Physical Storage**	**Model**	**Inter.**	**In-mem.**
Presto-RDF (2015)	Polystore queries	Redis and Cassandra	Key/Value and Columnar	No	Yes
xR2RML (2016)	Queries over views of RDF data	Independent	Independent	Pivot	No

Table 2 – Comparative of related work in the Polystore category

sandra, Redis), relational databases or even proprietary data stores. A single Presto query can combine data from multiple sources, allowing multimodel queries. Presto query execution is fast because of in-memory processing, so this work also fits as an *in-memory* solution.

xR2RML (MICHEL; FARON-ZUCKER; MONTAGNAT, 2016a,b) describes how native database entities can be mapped to RDF, and whereby an SPARQL query is translated into a pivot abstract query language independent of the database. Then, the pivot query is translated into the target database query language, considering the particular database capabilities and query optimization that can be implemented at this level. The pivot language is composed of clauses equivalent to the SPARQL language and a set of rules to transform this language into the target database query language. The focus of the work is the query mapping to document NoSQL databases, despite the authors argue that any NoSQL database system could be considered.

Table 2 compares the works classified as polystore into physical storage, model, intermediate structures and in-memory capabilities. Their purpose is either to define a middleware that enables several NoSQL database access or to delegate the storage task to a tool, like Presto. The only work that uses an intermediate structure for representing data is xR2RML. This is an important gap in the state-of-art, since other architectures like Estocada (BUGIOTTI et al., 2015) and other native triplestores, like WARP (HOSE; SCHENKEL, 2013), already consider such a structure to reduce the number of joins and therefore accelerate the query results.

3.3 IN-MEMORY

This category holds approaches whose main focus is on reducing query and ingestion latency by using the memory as the main storage.

Trinity.RDF (ZENG et al., 2013) introduces a distributed, memory-based graph engine for web scale RDF data on top of the Trinity NoSQL graph database. Instead of managing RDF data in triplestores, the authors developed an RDF store in its native graph form. The results show that it achieves better performance for SPARQL queries than the state-of-the-art approaches. Furthermore, since the data is stored in its native

graph form, the system can support other operations (e.g., random walks, reachability) on RDF graphs as well.

Papailiou et al. (PAPAILIOU; DIMITRIOS TSOUMAKOS, et al., 2015) presents a novel system that addresses graph-based, workload-adaptive indexing of large RDF graphs by caching SPARQL query results. At the heart of the system lies an SPARQL query canonical labeling algorithm that is used to uniquely index and reference SPARQL query graphs as well as their isomorphic forms, and a dynamic programming planner to generate the optimal join execution plan, examining the utilization of both indexed primitive triples and cached query results. By monitoring SPARQL queries, the system can identify and cache query results, substantially reducing the average response time of a workload. This approach persists RDF data into HBase columnar database.

H2RDF+ (PAPAILIOU; KONSTANTINOU; TSOUMAKOS; KARRAS, et al., 2013) is also based on HBase and uses six tables to store all possible triple permutations (SPO, SOP, PSO, POS, OPS, OSP). It maintains index statistics to estimate triple pattern selectivity as well as join output size and cost. From these estimations, H2RDF+ adaptively decides if queries are executed in a centralized way over a single node, or in a distributed way via MapReduce processing. It offers merge and sort-merge joins for both MapReduce and local execution. This work is an evolution of H2RDF.

A work that considers the in-memory capabilities of the NoSQL Key/Value Redis is ScalaRDF (HU et al., 2016). It presents a novel protocol that extends the consistent hashing protocol to achieve efficient data placement and elastic resource scale out/in. It allows for low-cost data store in the event of RDF data update and powerful cluster resource joining or departing. In this context, merely local data redistribution is necessary, avoiding the holistic data redistribution.

The columnar format *Apache Parquet*[4] seems to be a trend on in-memory triple-store proposals. Sempala (SCHÄTZLE; PRZYJACIEL-ZABLOCKI; NEU, et al., 2014) is a SPARQL-over-SQL approach that provides interactive time for SPARQL query processing on Hadoop. It stores RDF data in Parquet and uses Impala, a massively parallel processing SQL query engine for Hadoop, as the execution layer on top of it. SPARQL queries are translated into Impala SQL for improving performance. The work of Schätzle et al. (SCHÄTZLE; PRZYJACIEL-ZABLOCKI; SKILEVIC, et al., 2016) uses Parquet as storage solution and S2RDF (SPARQL on Spark for RDF), an SPARQL processor based on the in-memory cluster computing framework Spark. It comes with a novel partitioning schema for RDF called *ExtVP (Extended Vertical Partitioning)* that accomplishes semi-join reductions. The optimizations of ExtVP are applicable to all SPARQL query shapes regardless of its diameter.

A comprehensive experimental evaluation compares S2RDF with other state-of-art SPARQL processors for Hadoop to demonstrate its superior performance on diverse

[4] https://parquet.apache.org

Work	Focus	Proc.	Storage	Inter.	Data Model
Trinity. RDF (2013)	RDF data manager	No	Trinity	No	Graph
Sempala (2014)	Parallel processing	M/R	Parquet	No	Columnar
H2RDF+ (2014)	Triple indexing and distributed query processing	M/R	HBase	No	Columnar
Papailiou et al. (2015)	Workload-based query caching	No	HBase	Yes	Columnar
ScalaRDF (2016)	Parallel processing	No	Redis	No	Key/ Value
S2RDF (2016)	Data partitioning	Spark	Parquet	No	Columnar
PRoST (2018)	Data partitioning	Spark	Parquet	No	Columnar

Table 3 – Comparative of related work in the In-memory category

query workloads using the recent synthetic WatDiv benchmark and the real-world YAGO dataset. PRoST (COSSU et al., 2018) is an evolution of S2RDF that considers the same tools but defines a multi strategy that replicates RDF data by using a vertical partitioning approach and property tables. During the query processing, PRoST selects the partition type to be accessed using an statistic-based optimization where the total number of triples and the number of distinct subjects for each predicate are the main drivers.

Table 3 summarizes the works presented in this section. These works have different manners to organize the RDF information into memory before flushing it to disk. Apache Parquet is the rule for the papers using Spark and HDFS storage. HBase is considered by two works mainly because it fits when using MapReduce (Hadoop) jobs. Except the work of Papailiou et al., no other work considers an intermediate data model to represent RDF data (column *Inter.*). The lack of an intermediate representation forces the triplestore to manipulate each triple individually, which reduces the scalability because this design ignore the graph nature of the RDF data, specially the connections between the triples. So, data manipulation could consider a subgraph granularity instead of a triple granularity.

3.4 WORKLOAD-AWARE TRIPLESTORES

Table 4 shows works that exploit workload-awareness to speed up SPARQL queries or support polyglot storage of RDF data, including our proposal. The adopted abbreviations are **WA** for workload-awareness support, **S** for storage type, **F** for data fragmentation capabilities, **L** and **IR** for data localization and intermediate result processing optimizations, respectively.

The *AdaptRDF* approach (MAHMOUDINASAB; SAKR, 2012) consists firstly of a vertical partitioning phase that uses the workload information to generate an efficient

Work	**WA**	**S**	**F**	**L**	**IR**
AdaptRDF (2012)	Query stream	RDBMSs	No	Yes	No
WARP (2013)	Logs	Independent	Yes	No	No
Cerise (2014)	History	Native	No	Yes	Yes
Rainbow (2014)	Pre-calculated	Columnar	No	No	No
Papailiou et al. (2015)	Query results	Columnar	No	Yes	No
Presto-RDF (2015)	No	K/V, Columnar	No	No	No
xR2RML (2016)	No	Independent	Yes	No	Yes
ScalaRDF (2016)	No	Graph	No	No	No
S2RDF (2016)	Star queries	Columnar	No	No	Yes
WA-RDF (2019)	Monitoring	NoSQL	Yes	Yes	Yes

Table 4 – Related Work Comparison

relational schema that reduces the number of joins. Secondly, in the adjustment phase, any change in the workload is considered to create a sequence of pivoting and unpivoting operations to adapt the underlying schema and maintain the efficiency of the query processing. *WARP* (HOSE; SCHENKEL, 2013) presents a replication method on top of a graph-based partitioning that takes the workload into account to create a cost-aware query optimization and provide efficient execution. *Cerise* (KOBASHI et al., 2014) is a distributed RDF data store that adapts the underlying storage and query execution according to the history of queries. It colocates (on the same data segment) data that are accessed frequently together to reduce overall disk and network latency. In turn, the work of Papailiou et al. (PAPAILIOU; DIMITRIOS TSOUMAKOS, et al., 2015) focuses on monitoring SPARQL queries in order to cache the more requested query results.

As presented in Table 4, the fragmentation problem is discussed only by *WARP* and *xR2RML* (MICHEL; FARON-ZUCKER; MONTAGNAT, 2016a), which define expanded fragments to avoid unnecessary joins. On the other hand, the localization is the main contribution of *AdaptRDF*, *Cerise* and the work of Papailiou et al., which monitor query changes to better rearrange the localization of the underlying schema.

The more referenced approaches in the literature that propose triplestores based on NoSQL as data storage are *Rainbow* (GU; HU; HUANG, 2014) (a query processor), *ScalaRDF* (HU et al., 2016) (an in-memory solution) and *S2RDF* (SCHÄTZLE; PRZYJACIEL-ZABLOCKI; SKILEVIC, et al., 2016) (a query processor). *Rainbow* is a distributed triplestore that uses the *HBase* columnar database and an in-memory cache to speed up query processing. Based on a previous analysis of the dataset and the expected workload, it decides where the RDF data will be maintained. *ScalaRDF* introduces a distributed in-memory triple store that uses *Redis* as a fault-tolerant and distributed RDF store. Additionally, *S2RDF* proposes a *Spark*-based SPARQL query processor that offers very fast response time for star queries by extending the vertical

partitioning. Their partition scheme uses the *Apache Parquet*[5] columnar format to store the triples excluding unnecessary data from query processing. In order to reduce the intermediate results, S2RDF maintains statistics about the size of the dataset tables and places the subqueries corresponding to the smallest tables at the beginning of joining in order to reduce the intermediate result size.

According to Table 4, WA-RDF is the only work to address fragmentation, localization and intermediate results. Also, WA-RDF monitors and classifies the workload and store the triples into NoSQL databases.

3.5 FINAL REMARKS

The current scenario of the state-of-art NoSQL triplestores can be summarized as an *almost complete delegation of the storage mechanism to the NoSQL databases*. This limitation facilitates the development of the triplestore but considerably reduces the opportunities inherent of the difference between the RDF data model and the NoSQL data models.

Therefore, WA-RDF was designed and implemented as a workload-aware middleware. This middleware is able to rearrange the dataset storage in the NoSQL databases when the workload is modified, and also avoids data manipulation at a triple level. Instead, it defines fragments that reduces the number of joins at query processing time. The partition and replication mechanisms avoid cross-partition access, and the usage of cutting-edge processing frameworks facilitates the support for update and deletion operations. Also, WA-RDF is a polyglot solution that deals with different NoSQL databases and data models while considering a single abstraction for manipulating RDF data. Next chapter details the proposed middleware.

[5] https://parquet.apache.org/

4 WA-RDF

This chapter presents the solution proposed by this thesis, which is called *WA-RDF (Workload-Aware-based RDF data management)*. WA-RDF is a workload-aware middleware for storing and querying RDF data in multiple NoSQL database nodes.

We published four papers with the ideas described in this chapter. In the SBBD conference, we focus on the middleware architecture (SANTANA; SANTOS MELLO, 2017b). In the IRI conference, we focus on the polyglot access and RDF-to-NoSQL mapping (SANTANA; SANTOS MELLO, 2019a). In the DEXA conference, we focus on query processing (SANTANA; SANTOS MELLO, 2019b). In the ADBIS conference, the focus is on the workload management (SANTANA; SANTOS MELLO, 2019c).

The implementation of WA-RDF is based on Java. The programming language used for the development was the simplest decision of this thesis because the core Semantic Web implementations are inherited from Jena. Apache Jena[1] is implemented in Java and it is the most popular framework for the Semantic Web. Jena is important because it contains many components for parsing RDF data and SPARQL queries.

The current version of WA-RDF was developed using Apache Jena version 3.2.0 with Java 1.8, and we use MongoDB 3.4.3, Neo4J 3.2.5, Cassandra 3.11 and Redis 5.0.3 as the document, graph, columnar and key/value NoSQL databases, respectively. The choice for these solutions was based on their current high popularity in *DB-engines ranking*[2]. WA-RDF also employs Apache Spark 2.4.0 as the processing framework for map/reduce algorithms and Apache Kafka 2.1.0 as the queue manager.

Four versions of the proposed middleware was implemented during the development of this thesis. Table 5 presents the versions and the features contained on each one. The first version, developed during 2015 and 2016, focused only on NoSQL database integration, i.e., it accessed multiple NoSQL databases through a unique interface and a unified data model, and used JSON for storing and querying data. The second version, called *Rendezvous* (SANTANA; SANTOS MELLO, 2017b) included Web Semantic standards and the workload-aware notion of query shapes. This version considered document and columnar databases as the main storages, and a key/value database as the cache. In the third version, renamed to *WA-RDF*, the storage database for chain shaped queries was changed to the graph data model, and the columnar database was used as the distributed dictionary. Finally, in the last version, Apache Spark and Apache Kafka were added in order to increase the scalability of complex and iterative algorithms (e.g., fragmentation and query result generation) and background activities (e.g., deletion, update and replication).

[1] https://jena.apache.org/
[2] https://db-engines.com/en/ranking

Name	Year	Features
N/A	2015-2016	NoSQL integration with a JSON-based data storage and querying
Rendezvous	2017	Polyglot persistence (NoSQL document and colunar), consideration of Semantic Web standards as well as partitioning and replication support
WA-RDF	2018	Consideration of NoSQL graph database as a storage
WA-RDF	2019	A complete triplestore (support to update and deletion operations)

Table 5 – Middleware Versions

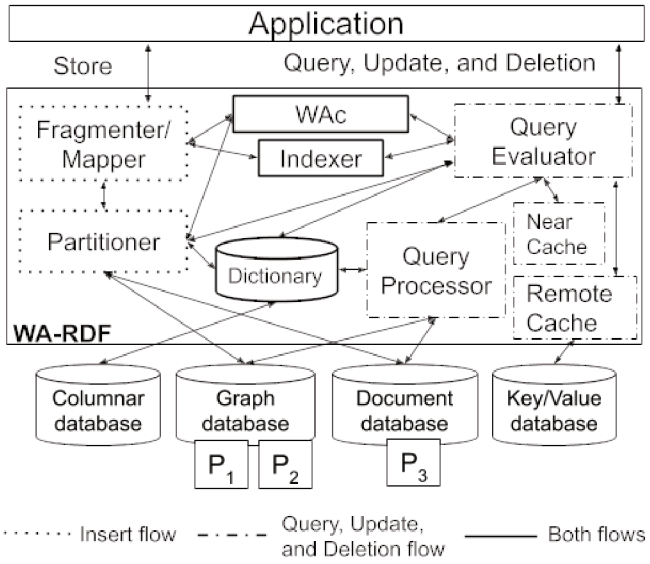

Figure 8 – WA-RDF Architecture

4.1 WA-RDF ARCHITECTURE

WA-RDF inspiration comes from *Estocada* (BUGIOTTI et al., 2015), which argues (as discussed in Chapter 2) that a mixed-model layer, relying on a set of diverse and heterogeneous data stores, can provide performance advantages for the applications using this layer. However, *Estocada* is neither a workload-aware approach nor a storage solution for RDF data. Another idea we borrowed from *Estocada* is a *fragment-based* storage that is entirely transparent to the client applications. It means that the data unit for manipulation purposes in *WA-RDF* follows a fragment granularity most of the time. Figure 8 gives an overview of the *WA-RDF* architecture.

An *RDF-based Application* issues *store* or *query* requests to *WA-RDF*, which is normally deployed into multiple dedicated physical nodes. When an application submits a *store* request for a triple to the *Fragmenter/Mapper* component, *WA-RDF* expands

this triple to a fragment F_{RDFi} and maps F_{RDFi} to the target NoSQL database(s). This process is controlled by the *WAc*, which is the main component of our middleware. During a triple storage, it decides on translating F_{RDFi} to a NoSQL document or graph database (or both) according to the usual query workload, and indexes it with the aid of the *Indexer* component. Once F_{RDFi} is created, the *Partitioner* registers this fragment into the *Dictionary* repository (supported by a NoSQL columnar database) and stores it in the NoSQL databases.

When an *RDF-based Application* submits a SPARQL *query* request, the *Query Evaluator* component decomposes this query into subqueries and reports to the *WAc* about them. In the following, the *Query Evaluator* verifies, with the aid of the *Dictionary*, the partitions on which the triples for the query are potentially located. Based on this information, it checks which triples are available in the *Near Cache* (a data structure in the main memory of the server) and the *Remote Cache* (a remote NoSQL key/value database), and sends the SPARQL subqueries for the missing triples to the *Query Processor* component that, in turn, translates them to graph and/or document NoSQL database queries. Finally, the *Query Processor* sends back the query results to the *Query Evaluator* that translates them back to RDF triples with the aid of the *Dictionary*, and returns the result to the *RDF-based Application*.

WA-RDF also supports *update* and *deletion* operations. In both cases, the operation execution starts at the *Query Evaluator*, which verifies if the update or deletion is valid. If so, it redirects the modification to the *Partitioner*. This component, in turn, accesses the *Dictionary* to find out the fragments that will be affected by the operation and finally performs the changes in the NoSQL databases.

In order to use WA-RDF, a programmer needs to deploy the servers accordingly to the Appendix B. The *RDF-based Application* will call the servers using a RESTful interface[3]. A store operation is mapped to a POST call with the triples in the body specified by JSON-LD (JSON-based Serialization for Linked Data)[4]. A update opreation is mapped to a PUT method, a query operation to a GET method, and a deletion operation to a DELETE method.

The main purpose of *WA-RDF* is to store large RDF graphs. In such a scenario, the number of RDF triples can easily surpass the performance capacity (*e.g.*, disk, memory, CPU) of a single server. When it occurs, *WA-RDF* distributes the RDF fragments among potentially many NoSQL nodes. A fragment is our smallest grain of distribution, *i.e.*, during the partitioning process we deal with fragments instead of triples. Nevertheless, a query can eventually access data in multiple partitions, forcing *WA-RDF* to join data from different partitions. Since a join operation is very costly, we try to avoid join processes by replicating fragments that are potentially part of a join.

[3] https://restfulapi.net/

[4] https://www.w3.org/2001/sw/wiki/JSON-LD

In short, whenever the typical workload for a fragment spans more than one partition, our *partitioning* scheme replicates the boundary fragments of the partition. Boundary fragments hold triples that are connected to triples that are stored in other partitions.

WA-RDF also provides an RDF indexing strategy. In this context, a traditional approach is to build indexes for the full set of permutations of each triple component (*subject (S), predicate (P)* and *object (O)*). Although this method has been designed to accelerate joins by some orders of magnitude, the overhead with large index space limits its scalability and makes it heavyweight. Hence, we developed a *hashmap index* with subject and object keys following the patterns *S-PO* and *O-PS*. In *WA-RDF*, the *Indexer* component is responsible to manage these indexes. It is accessed in two situations: *(i)* during the fragment creation and storage; and *(ii)* to process queries with one triple pattern. We detail the storage and querying strategies in the following.

4.1.1 Architectural Decisions

During the development of the current version of WA-RDF, many decisions about the organization of the components and tools that could help on each part of the development and models were accomplished. This section presents two architectural decisions regarding WA-RDF that were the result of several researches that culminate with the following mix: the considered NoSQL databases and how to map RDF data to the choosen NoSQL databases.

4.1.1.1 NoSQL databases

The best mix of NoSQL databases in the middleware was achieved only after a few rounds of design and implementation trials. In the first version, the intended contribution was to map and access RDF data through all the NoSQL database categories, but after the initial thesis development, we figure out that the most important challenge was on matching the dataset and workload characteristics to the best NoSQL database features.

In the Rendezvous version, the columnar database was considered in order to store the triples on which the typical workload is represented by chain shaped queries. The vertical partitioning schema influenced this solution, as discussed in our SBBD paper (SANTANA; SANTOS MELLO, 2017b). The objective of this mapping was quickly discovering the triples through the predicates of the querying triple patterns. However, this version had generated multiple database accesses by queries on which the triple patterns have query variables in the join, instead of values. Due to it, the next version, as presented in our ADBIS paper (SANTANA; SANTOS MELLO, 2019c), considered a graph database.

This new solution (WA-RDF) solved the columnar problem of multiple accesses and introduced two benefits: it was possible to map the fragments into a node-edge-

node format that summarizes the chain fragments, and it was possible to join multiple chain fragments with only one access because RDF is also a graph. The columnar database was considered in the design of the middleware dictionary as the usage of column families allows a fast access to the content of each data partition (more details are given in Section 4.4.1). Additionally, the document database fits as the best option to star-shaped fragments. Its hierarchical querying capabilities facilitates queries that have triple partners that are transversals. The best performance for the middleware was extracted of these choices. Finally, the key/value database application was more obvious because its common usage is for caching. Redis database was also used as the indexer, as a common use case considered by other triplestores.

The design of WA-RDF allows to easily exchange databases, since the only dependency is the database connectors. However, MongoDB, Neo4j, Cassandra and Redis plays an important role in the middleware. As stated before, they were chosen based on the DB ranking popularity. However, some important features of them facilitated the WA-RDF implementation, specially Neo4j and Redis. Neo4j more important feature is the possibility of adding and querying properties in the edges. Redis, in turn, is very useful for two reasons: *(i)* the *expire* solution[5], which removes unused fragments; *(ii)* to store complex serialized Java objects.

The usage of a graph database instead of the document database is a manner to avoid deep hierarchical structures that would limit the querying power of the middleware. In this case, we transform each chain fragment into a structure formed by two nodes connected by an edge. The first node represents the first subject of the chain, and the last node represents the last object of the chain. In the edge properties, we add the triples of the chain in a list. This structure is only possible because the modern graph databases (*e.g.*, Neo4j, Titan) allows for queries over edge properties.

4.1.1.2 Mapping decisions

One of the most important and frequent processes accomplished by WA-RDF is the mapping of RDF data to the data structures of the NoSQL databases. In order to decide for the best mapping processes, we analyze related work that provides similar solutions. We propose a taxonomy to organize the considered approaches according to their mapping strategies, including the Rendezvous version. Table 6 shows the categories of this taxonomy and the works that fit on each of them.

The first category is called *Flat triple* since the idea is to map each triple component into a correspondent concept of a NoSQL data model, like a column or a JSON key. In the work of Tomaszuk (TOMASZUK, 2010), for example, each triple is a document with S, P and O keys, and the respective triple part value. This strategy can be used only by document and columnar NoSQL databases because it needs a group-

[5] https://redis.io/commands/expire

Mapping Strategy	Approach
Flat triple (FT)	Tomaszuk, Cumulus (flat strategy), Jena-HBase (simple strategy)
Triple permutation (TP)	Rya, ScalaRDF, Jena-HBase (index strategy), Rainbow (key/value)
Graph (G)	Trial, Trinity.RDF, Bouhali, Papailiou
Vertical Partitioning (VP)	Rendezvous (columnar), Pham, S2RDF, PRoST
Pre-processed joins (PPJ)	Rendezvous (document), RDFJoin, RDFKB, Stratustore, AMADA, RDFChain, W3C RDF/JSON, Rainbow (columnar)
Hierarchical (H)	Cumulus (hierarchical strategy), MAPSIN, Hive+HBase

Table 6 – A Taxonomy for RDF-to-NoSQL Mapping Strategies

ing structure (*e.g.*, document, table) that is not available at the key/value and graph NoSQL databases. In this mapping strategy, each triple is mapped to one structure of the NoSQL database.

The *Triple permutation* category includes works that consider some permutations of an RDF triple (SP-O, SO-P, OP-S, S-PO, P-SO, and O-SP). In the surveyed approaches, this mapping is normally used by the key/value NoSQL stores, but it is also present in JenaHBase and in the SQL-based system Hexastore (WEISS; KARRAS; BERNSTEIN, 2008), commonly cited as one of the first to employ this strategy. Different from it, the *Graph* category comprises approaches that accomplish a direct map from S, P and O to nodes and edges of a graph, as used by Trinity.RDF, Bouhali, and Papailiou. It reduces the complexity of the mapping process.

The *Pre-processed joins* category holds works that store S, P, and O in a way that improves join processing. This strategy is adopted for all the NoSQL data models, and usually brings a great improvement in the SPARQL query processing. The mapping solutions in this category define structures (*e.g.*, tables, documents) that store together RDF data usually requested in the same queries. It avoids, for example, table scans and joins of multiple documents.

The *Vertical partitioning (VP)* category can be understood as the opposite of the *Pre-processed joins* category as it breaks the triple in smaller parts instead of generating new structures. The most common mapping solution in this case is to define a two-column table for every RDF predicate. The SQL-based solution SW-Store (ABADI et al., 2009) is one of the first works that employs this strategy, which is specially considered by the surveyed works based on columnar NoSQL databases. This category includes different strategies. One example is S2RDF. Its ExtVP approach adds pre-processed joins to the vertically partitioned tables in order to improve the query response time.

Cat.	Work	t1	t2	t3	t4	t5	AVG
FT	Tomaszuk	0.105	0.096	0.108	0.18	0.375	**0.173**
TP	Rya	0.147	0.128	0.141	0.175	0.208	**0.160**
G	Trinity.RDF	8.335	8.708	8.541	8.357	8.714	**8.531**
PPJ	RDFJoin	3.728	5.287	4.845	4.374	5.287	**4.608**
VP	S2RDF	0.69	0.466	1.971	2.09	2.268	**1.497**
H	MAPSIN	4.148	0.698	3.381	0.699	4.013	**2.588**

Table 7 – Processing Time of the Mapping Categories (milliseconds)

Finally, the *Hierarchical* category considers nested structures to improve the query response time. Different from *flat triple* strategy, each triple part is stored inside a structure defined by another triple part (*e.g.*, the P defines the key of a document, where the values are the Os). In this strategy, each triple is mapped into many structures of the NoSQL database in different permutations of the triple parts. In MAPSIN, for example, a triple is mapped to the SPO table, which has S as row key, P as column name, O as the column value, as well as the OPS table with O as row key, P as column name, and S as the column value.

In order to evaluate the performance of these mapping categories in a fair way, one representative of each category was implemented as a plain Java (*i.e.*, without frameworks) application based only on Apache Jena 3.10.0. Table 7 shows the category and an implemented work for a given category. The processing time was measured considering the creation of a triple, the execution of the mapping strategy over it and the storage into the database. The triples sample used in this experiment are described in Appendix A. They come from the LUMB (GUO; PAN; HEFLIN, 2005) benchmark, the most cited one in the surveyed papers.

The experiments were run 1000 times for each triple and the average processing time was collected. Our infrastructure is a Ubuntu 18, processor Intel Core i7 with 16 Gb of memory. All the databases are local to the Java client, and the source code for the tests can be found in the authors' GitHub[6].

The *graph* strategy is more than twice slower than the other options. Although the mapping is straightforward, the graph database (Neo4J) is slower than the other NoSQL databases. It may be the reason why so few works fit into this category. The second slower is the *pre-processed joins* strategy. This is due to the multiple calls to Cassandra. Besides, the results for *t2* and *t5* shows that the more connected a triple is the slower the pre-computed processing time will be. The *hierarchical* strategy is dependent on the existence of the predicate column. In the experiments, the predicate *http://www.w3.org/2001/vcard-rdf/3.0FN* is created for *t1*, so the times for *t2* and *t4* are reduced. The `VP` strategy is three times faster than the *hierarchical* one mainly because

[6] https://github.com/lhzsantana/er-mapping

it does not have to add new columns when a triple has a predicate that is not present in Cassandra. The *flat triple* strategy is around ten times faster than *VP* since the last one has to check if each table exists and creates new structures when they do not exist. Finally, the *triple permutation* strategy is slightly faster than *flat triple* even with multiple calls. The low latency of Redis makes it a very promising solution.

Based on this analysis, the design of the WA-RDF mapping strategy followed these goals:

- Mappings must be performed in background;
- Redis is the right choice for caching due to its low latency even in a multiple triple permutation;
- MongoDB can be used for storing the most common shapes, which are the star-shaped queries in the considered benchmarks, and in general as presented in Gallego et al. (GALLEGO et al., 2011);
- A direct mapping from RDF to Neo4j is too costly, so WA-RDF uses a simplification approach to reduce the query time. This simplification allows returning multiple chains in only one Neo4j access;
- Cassandra can be a bottleneck in multiple calls, so it can be used only for a limited number of calls, which is the case of the middleware Dictionary.

4.2 STORAGE

When a new RDF triple t_{new} = (s, p, o) is inserted through *WA-RDF*, the hashmap indexes are checked to decide if t_{new} is more frequent on star or chain-shaped queries. Algorithm 1 shows the workload-based triple storage strategy. The input parameter is t_{new}, and it generates an RDF fragment *f* that is stored in one or more partitions. Suppose, for example, that we have two new triples `C p10 M` and `C p2 D` (Figure 9 (ii) and Figure 9 (vi), respectively) to be inserted into the RDF graph of Figure 9 (i). An RDF fragment represents an expansion of t_{new} (called *core triple*) with all of its neighbors according to a *n-hop replication horizon* managed by *WA-RDF*. The n-hop replication horizon is a value equivalent to the number of connected triples that WA-RDF considers to expand new triple into a fragment. It is formally defined in the following.

Definition 4.2.1 *(N-hop replication horizon). The n-hop replication horizon for a new triple* $t_{new} = (s_{new}, p_{new}, o_{new})$ *is a value* n *equals to the length of the longest chain of triple patterns in the current workload that is related to the predicate of* t_{new}*, i.e.,* $n_{t_{new}} = max(length(WAc.get(t_{new}.p_{new})))$*, where WAc.get() is a function that get the triple patterns related to the predicate of a triple.*

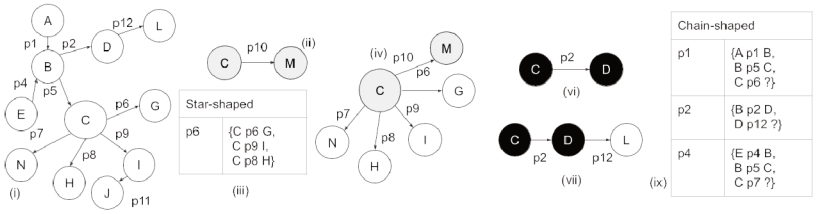

Figure 9 – Fragment Creation

For example, the n-hop for the triple `C p10 M` (Figure 9 (ii)) is 1 according to the frequent star-shaped query in the hash of Figure 9 (iii). For triple `C p2 D` (Figure 9 (vi)), WA-RDF checks the hash of Figure 9 (ix), and returns a n-hop of size 2. The n-hop is used to avoid frequent joins by wisely expanding the core triple to include its neighbors until a maximal distance n. These auxiliary hash structures (Figure 9 (iii) and (ix)) are maintained by the WAc component.

Input: Triple t_{new}
if *!exists(t_{new})* **then**
    f = *new* Fragment;
    f.core = t_{new};
    indexSPO.put(t_{new}.s, t_{new});
    indexOPS.put(t_{new}.o, t_{new});
    f.shapes = getShapes(t_{new});
    hop = 1;
    if *f.shapes.contains('chain')* **then**
        hop = chainHop(t_{new});
    if *f.shapes.contains('star')* **then**
        if(starHop(t) $>$ hop) hop = starHop(t_{new});
    f.triples = expand(t_{new}, hop, f.shapes);
    writeToPartitions(f);
end

Algorithm 1: Workload-based triple storage

Back to Algorithm 1, if t_{new} does not exists (line 1), a new RDF fragment *f* is generated (line 2) and it initially holds the core triple (line 3). Next, the core triple is indexed in an SPO and OPS fashion (lines 4 and 5) in order to reduce response time of queries without joins and facilitate the query expansion.

From line 6 to line 11, Algorithm 1 obtains the shapes and the n-hop size for the core triple. The n-hop size is defined as the size in terms of triple patterns of the biggest query in the typical workload for the core triple. It initially finds the shapes and registers them in the fragment *f* (line 6). If neither the predicate nor the subject exist in the chain and star hashmaps, respectively (no shape is found), it defaults to a star-shaped query with one triple n-hop size (hop = 1) (line 7). Otherwise, it determines the hop based on the found shapes (lines 8 to 11). In line 12, t_{new} is expanded to the n-hop size. *f.triples*

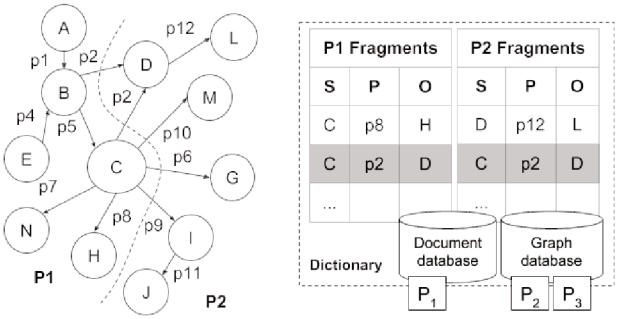

Figure 10 – Fragment Partitioning

is an array with up to 2 positions: one for the chain fragment and another one for the star fragment. In the example of Figure 9, the new triple `C p10 M` is expanded to the (RDF) *star fragment* shown in Figure 9 (iv), and the new triple `C p2 D` to the (RDF) *chain fragment* shown in Figure 9 (vii). An RDF fragment is formally defined as follows.

Definition 4.2.2 *(RDF Fragment). An RDF Fragment is a set* $F_{RDFi} = \{t_{RDF}\}$ *of RDF triples* $t_{RDF} = (s, p, o)$ *whose content may overlap with other fragment* F_{RDFj}*. After the star or chain fragment is created, WA-RDF distributes it among potentially NoSQL nodes (line 13). A NoSQL node can store one or more partitions. We discuss RDF data partitioning further on in this section.*

It is important to observe here that a core triple can generate two RDF fragments (star and chain). It happens when the subject of this core triple is in the star hashmap and its predicate is in the chain hashmap at the same time. If a star fragment is generated, it is mapped to a JSON document and stored into a NoSQL document database. This JSON document is called a *document fragment*. If a chain fragment is generated, we have a mapping to a NoSQL graph database. In this case, we have a *graph fragment*. More details about the mapping processes accomplished by WA-RDF are given in Section 4.2.3.

We now explain the partitioning strategy of WA-RDF. Given the RDF graph of Figure 10 (the resulting graph after the storage of the triples `C p10 M` and `C p2 D` into the graph of Figure 9), the fragments are stored in document partitions (for instance, P_1) and/or in graph partitions (for instance, P_2 and P_3). A fragment is the finest unit for a partition. As defined in the following, an RDF partition is a set of fragments stored into the same physical NoSQL node, and a fragment can be replicated in multiple partitions.

Definition 4.2.3 *(RDF Partition). An RDF Partition* P_m *of an RDF graph* G*, such that* $G \subseteq P_1 \cup P_2 \cup ...P_n$*, is a set of RDF fragments* $P_m = \{F_{RDFi}\}$*, being not required that* $P_m \cap P_t = \emptyset$*, for* $m \neq t$*.*

We also introduce the concept of *partition boundary* as follows.

Definition 4.2.4 *(Partition Boundary). Given* $SP = \{P_1, P_2, ..., P_n\}$ *the set of RDF partitions, the partition boundary* B_{P_i} *of a partition* $P_i \subset SP$ *is the a set of RDF fragments* $B_{P_i} = Fb_{P_1} \cup Fb_{P_2} ... \cup Fb_{P_r}$*, where each* $Fb_{P_k} \in B_{P_i}$ *has one or more RDF triples* $t_i F_{P_k} = (s_i, p_i, o_i)$ *with* $o_i = s_j$*, being* s_j *the subject of other triple* $t_j F_{P_j}$ *of a partition* P_j *so that* $t_j F_{P_j} = (s_j, p_j, o_j)$*.*

The *Dictionary* shown in Figure 10 registers each fragment location. It registers in a columnar database information for each partition to keep track of the RDF elements stored in each partition (represented by the tables *P1 Fragments and P2 Fragments*), so during a query request we can avoid accessing unnecessary partitions that cannot answer this query. If a WA-RDF node manages more than one partition of a NoSQL database type, in face of a new core triple we have to decide which is the best partition to store its fragments. For doing so, WA-RDF finds out the typical workload for the triples that belong to the fragment generated by the core triple. With this information, we can query the partition sets in the *Dictionary* to verify in which partition this fragment can be more useful in sense that joins outside the fragment can be answered within a single partition.

WA-RDF uses a boundary replication parameter to avoid unnecessary joins between partitions. Thus, in Figure 10, the boundary replication (with size n = 1) is presented between partitions *P1* and *P2*. WA-RDF boundary replication will repeat the fragment with core triple `C p2 D` in partitions *P1* and *P2*. The fragment replication permits that a query with triple patterns `x? p8 H. x? p2 D.` be responded only by querying partition P1, and a query with triple patterns `x? p2 D. D p12 L.` be responded only by partition P2, avoiding, in both cases, cross-partition joins.

It may happen that no workload information is available for a new triple. In this case (we call it *cold start*) the storage is defaulted to the document database.

4.2.1 Workload-awareness monitoring

A workload-aware approach is the cornerstone of WA-RDF. Based on it, WA-RDF decides where and how to place data, which influences fragmentation, mapping, partitioning and querying strategies. First of all, it is essential to explain how this component monitors the workload.

WA-RDF registers information about the triple patterns of each incoming SPARQL query. For instance, if we have a SPARQL query `SELECT ?x WHERE { ?x p1 y?. y? p2 z?. z? p3 D }` (q_1) at time t_1, and we have a query `SELECT ?x WHERE { ?x p1 y?. x? p4 a?. x? p5 b?}` (q_2) at time t_2 ($t_1 < t_2$), WA-RDF stores, with the aid of WAc, information about the triple patterns, the BGPs and their shapes, the time that the query was most recently received, and how many times WA-RDF received this query, as illustrated in Figure 11. WAc is composed of the following in-memory

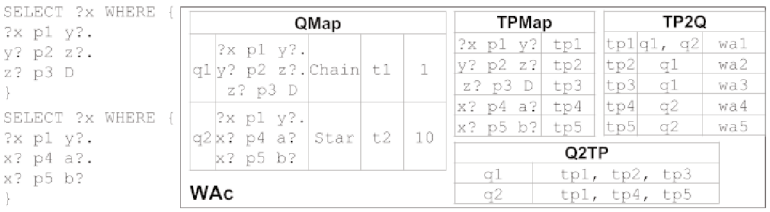

Figure 11 – WAc: Workload Monitoring

data structures: *QMap* (registers the queries); *TPMap* (registers the triple patterns); *TP2Q* (maps triple patterns to queries and identifies the workload version for each triple pattern); and *Q2TP* (maps queries to triple patterns).

Next sections detail how WAc manages data fragmentation, improves data localization and avoids unnecessary intermediate query results during WA-RDF running. Such enrichments represent an advance w.r.t. the state-of-the-art regarding NoSQL-based triplestores, as discussed in Section 1.1.2.

4.2.2 Fragmentation

An efficient data fragmentation management in a distributed database is fundamental to speed up query processing. The goal of the WA-RDF fragmentation process is to minimize the number of joins to be executed by preprocessing these joins during a triple insertion.

When WA-RDF receives a new RDF triple $t_{new} = \{s, p, o\}$, it persists t_{new} into a document NoSQL database with a temporary JSON format {*f=s, p-i=o*}. This mapping format is discussed later in this section, but it allows t_{new} to be queried while WA-RDF creates a fragment. In a background process, WA-RDF checks if t_{new} matches any triple pattern present in *TPMap*. For instance, `A p3 D` matches `z? p3 D` (tp3) in Figure 11, but `A p3 C` does not match any triple pattern. If a match occurs, based on *TP2Q* information, WA-RDF verifies in *QMap* if the typical workload is star or chain (or both) according to the match. In any case, WA-RDF expands t_{new} to a fragment representing the union of all the queries that hold the match. If t_{new} does not match any workload information, it is solely registered in WAc and the temporary JSON format still remains.

In the example of Figure 12 (1), a new triple `A p1 B` matches the triple pattern `?x p1 y?`, which, in turn, is related to queries with both shapes in the typical workload of Figure 11. Hence, if Figure 12 (2) represents the current graph managed by WA-RDF, the fragments formed by the new triple `A p1 B` are the star {`A p1 B, F p6 A, A p4 D, A p5 E`} and the chain {`A p1 B, B p2 C, C p3 D`} (Figure 12 (3)). As presented in Figure 12 (4), after the fragments creation, they are mapped and stored into a document or graph (or both) databases.

WA-RDF transforms each document fragment into a JSON document. The most

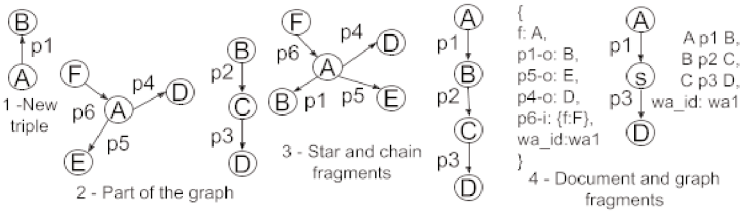

Figure 12 – Fragment creation

connected node, *i.e.*, the node that has more incoming or outcoming predicates, is the value for the document key *f*. Moreover, for each predicate, it is created a key with the predicate value concatenated with *-i* for incoming and *-o* for outcoming. The value is the resource connected to this predicate keys, or a subdocument with the same structure. One example is shown in Figure 12 (4). The benefit of this document mapping is to solve the typical queries with only one filter access.

For the graph NoSQL database, we could do a straightahead mapping from the fragment. However, the following mapping leads to a better performance. Firstly, WA-RDF finds the longest chain, creates graph nodes for the first subject S_1, the last object O_n of this chain, and a summary node S. Further, it maps the predicates linked to S_1 and O_n to edges, and S contains, as properties, all the triples between S_1 and O_n. Finally, WA-RDF connects S to all the nodes that are not in the longest chain. This procedure is repeated to all the sizes of chains until S would have only one triple. As shown in Figure 12 (4), the graph fragment `A p1 B, B p2 C, C p3 D` is mapped to the nodes `A`, S and `D`, and S contains `A p1 B`, `B p2 C` and `C p3 D` as properties. This mapping strategy is efficient when there are more than one chain query connected. For instance, if a query with the triple patterns `?x p1 y?. ?y p2 z?. ?z p3 D. ?z p4 ?k. ?k p5 ?l. ?l p6 ?m` is issued, WA-RDF can solve it with only one database access.

As illustrated in Figure 12 (4), every new fragment receives an identifier from WAc. For document fragments, it is added to the key *wa_id* in the body of the document. For graph fragments, it is a property called *wa_id* of the node S. This identifier is used by WA-RDF to register where each fragment is located, as further detailed.

Algorithm 2 describes the fragmentation process accomplished by WA-RDF. Its input is a new triple t_{new}, and its return is the list of triples $\{t_1, t_2, ..., t_n\}$ for a created fragment f. Firstly, the algorithm creates a list of the triple patterns (line 1). If the list is empty then the temporary fragment is returned (lines 2 and 3). Otherwise, by using a *map-reduce* programming approach, all the queries that match the triple patterns are analyzed (lines 5 and 6). The *map* function removes the triples that are not connected to t_{new} or does not match any triple pattern. During the map phase, f is created from the cleaned fragments (lines 7 and 8) and, in the reduce phase, the triples are connected to t_{new} (line 9). Finally, in line 10 the fragment is stred into the NoSQL databases and

the temporary fragment is deleted in line 11.

Input: t_{new}
Output: f = $\{t_1, t_2, ..., t_n\}$
triplePatterns=TPMap.matches(t_{new});
if *triplePatterns.size()*$= 0$ **then**
    f=getTemporaryFragment(t_{new});
else
    queries=QMAP.getQueries(triplePatterns);
    fragment=analyze(queries);
    f = fragment
    .map(cleaned: (fragment, triplePatterns))
    .reduce((cleaned, t_{new}), f: connect(cleaned, t_{new}));
    persistToNoSQL(f);
    deleteTemporaryFragment(t_{new});
return f;

Algorithm 2: Map-reduce-based fragmentation

In terms of implementation, it was decided to use the *Apache Spark*[7] framework for two reasons: the in-memory nature of the processing and its map/reduce implementation. The Apache Spark is connected to the components Fragment/Mapper and Query Processor, as shown in Figure 13.

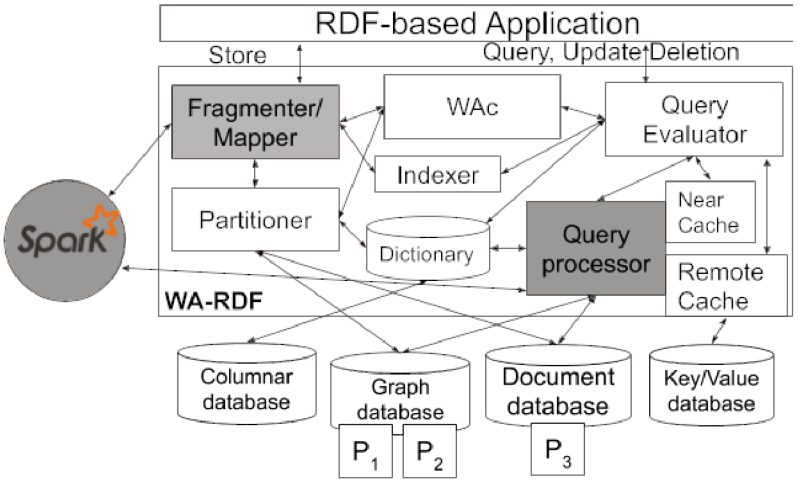

Figure 13 – WA-RDF with Apache Spark for fragmentation and triple cleaning processing

Other options could be *Apache Hadoop* and *Apache Storm*, but Apache Spark is by far the fastest and more scalable current technology for Big Data processing (ZAHARIA, 2016). Also, this framework is gaining more attention than the others and enable for future works on machine learning (ZAMBOM SANTANA; SANTOS MELLO; ROISENBERG, 2015).

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

WA-RDF also manages multiple nodes of each NoSQL database, where each node maintains a partition of the data. As presented in Figure 14, a *Dictionary* component maintains the workload identifiers of the persisted fragments and the number of triples for each workload identifier (wa_{id}) maintained in data partitions into the NoSQL databases. The goal of the Dictionary is to help on the querying process, avoiding unnecessary searches and reducing the calls and round trips to the NoSQL databases. For instance, as illustrated in Figure 14, when a WA-RDF installation with three partitions receives a query, it matches the triple patterns in *TPMap*, get the wa_{id} from *TP2Q*, and finds out which partitions to query from the Dictionary.

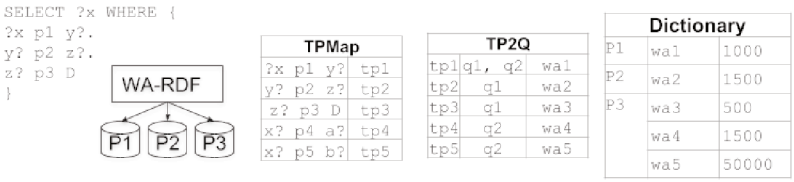

Figure 14 – Fragment distribution

The choice for using wa_{id} to distribute the fragments into partitions can lead to skewed data placement. We try to balance it with a round-robin process on the partitions when new fragments are inserted or updated. Moreover, periodically WA-RDF checks the Dictionary to find skewed partitions. In the example of Figure 15 (left), partition *P3* has too many triples for the workload *wa5*. In this case, WA-RDF checks *TP2Q* to see the queries causing the skewness, get the fragments, and replicate or move data. WA-RDF moves data (replication and deletion) to evenly distribute the storage size and the query load among the servers. This is not the case of the example, where deleting the fragemnts from *P3* would only move the skewness to the other partitions. In this case, WA-RDF only replicates data and balances the query load (Figure 15 (right)). As discussed in Chapter 5, the additional storage need generated by this replication is usually irrelevant in comparison to the whole dataset.

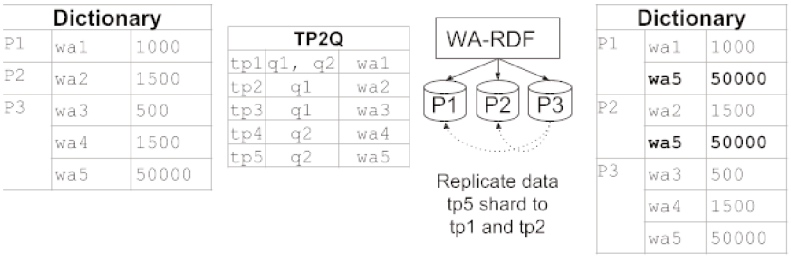

Figure 15 – Fragments redistribution

In a dynamic architecture, the workload of SPARQL queries changes over time.

To deal with it, WA-RDF maintains only the most relevant queries in WAc. The relevance *R* for each query is calculated by the sum of how frequent (f) this query is, the number of placeholders (p) in the query (the more placeholders a query has, the more triples it potentially matches), the size (s) of the query (the number of triple patterns), divided by the novelty (n) of the query (how long since it was lastely received). During WA-RDF development, other measures of relevance were considered. However, preliminary experiments revealed that the following formula obtained the best results.

$$R = (f + p) * s/n \tag{1}$$

Instead of adding immediately the new queries in the WAc, WA-RDF keeps them in a queue and waits until the size of this queue passes a soft dynamic threshold *T*. This threshold is calculated by multiplying the number of querying threads (q) to the number of queries in the last 10 seconds (t), divided by the average of queries in the last minute (m). The relevance R is calculated if T is bigger than the number of querying threads. This threshold is used to avoid the process to be fired for every new query.

$$T = q * t/m \tag{2}$$

Also, when the threshold is reached, WAc defines a new wa_{id} as well as deletes and recreates all the fragments stamped with the old workload identifier. This change leads to modifications on how the data is placed in the architecture by changing the fragments, the partitions and the data storage in the NoSQL databases. During fragment recreation, WA-RDF performs in a sub-optimum placement, i.e., the old and the new fragments coexists in the architecture, and if a new query arrives, it can be responded with an old version of the fragmentation schema. However, this task does not affect the consistency of the architecture. It is important to notice that the workload version only changes if queries with new structures gain relevance, which is usually seldom.

4.2.3 Mapping

RDF data is mapped to each one of the NoSQL databases by WA-RDF. A columnar fragment is created when a new triple is registered to the Dictionary. Document and graph fragments are used as main storages, as explained in Section 4.2.2. Finally, a key-value fragment is registered in the cache after a query answer is returned.

We formally describe all of these mappings in the following, remembering that, given an RDF triple a $t_{RDF} = (s, p, o)$ where $t_{RDF}.s$ is the subject, $t_{RDF}.p$ is the predicate and $t_{RDF}.o$ is the subject, an RDF *fragment* is a set $F_{RDFi} = \{t_{RDF}\}$ of connected RDF triples.

Definition 4.2.5 *(Columnar fragment). A columnar fragment is a tuple $f_{cf} = (k_cf, C)$ where $f_{cf}.k_cf$ is the name (key) of the column family and $f_{cf}.C$ is a set of columns*

(key-value pairs) $f_{cf}.C = \{(n_c : v)\}$*, being* n_c *the column name (or column key) and* v *an atomic value. The mapping of an RDF fragment* F_{RDFi} *to columnar fragments proceeds as follows: for each predicate* $t_k.p$ *of a triple* $t_k \in F_{RDFi}$*: generate a columnar fragment* $f_{cfi} = (k_cfi, \{c1, c2\})$ *such that* $f_{cf}.k_cfi \leftarrow t_k.p$*,* $f_{cf}.C.c1 \leftarrow t_k.s$ *and* $f_{cf}.C.c2 \leftarrow t_k.o$*.*

Definition 4.2.6 *(Document fragment). A document fragment is a tuple* $f_{df} = (k_d, A)$ *where* $f_{df}.k_d$ *is the JSON document key and* $f_{df}.A = \{(k_\alpha : v)\}$ *is a set of attributes, being* k_α *the attribute key and* v *a value whose domain can be atomic, a list, a set or a tuple. In short, the core triple* t_{core} *in the RDF fragment* F_{RDFi} *is mapped to a document whose key is* $t_{core}.s$*, and each outgoing predicate from the subject becomes a document attribute with a key* $t_{core}.p$*. If* F_{RDFi} *is 1-hop, the attribute value of each outgoing predicate is the object* $t_{core}.o$ *reached from it. Otherwise, the predicate value is an inner document that maintains the target object as the inner document key, and its outgoing predicates as attributes. If any of these outgoing predicates is, in turn, an n-hop, n > 1, the generation of other inner documents proceeds recursively.*

Definition 4.2.7 *(Graph fragment). A graph fragment is a triple* $f_{gf} = (s_gf, T, o_gf)$ *where* s_gf *is a vertex representing the first subject of a chain,* o_gf *is a vertex representing the last object of a chain, and* T *= {*t_n*} denotes an edge that holds a set of triples as property, i.e., the intermediary triples between* s_gf *and* o_gf*, including the object of the first triple and the subject of the last triple. A graph fragment summarizes a chain of triples by transforming this chain into a triple where the subject of the first triple and the object of the last triple are mapped to two vertexes, and the edge between these two vertexes is created with a property that maintains all the triples of the chain.*

Definition 4.2.8 *(Key-value fragment). A key-value fragment is maintained in the WA-RDF cache with the form of triples. The fragments are stored when there is a response of a SPARQL query. So, a key-value fragment formed by each part of the triple, where the key is the part (subject, predicate, object) and the value is the triple itself.*

Examples of mappings for a columnar, document, graph and key-value fragment are given, respectively, in Section 4.4.1, in the Figure 9 (v) for the star fragment of Figure 9 (iv), in Figure 9 (viii) for the RDF triple of Figure 9 (vi) that is expanded to the chain fragment of Figure 9 (vii), and in Section 4.3.2.

4.2.3.1 Internal Resource Mapping

RDF triples and triple patterns managed by WA-RDF on its different data structures (cache, dictionary, indexes and NoSQL databases) cannot be stored in raw format so data replication would be prohibitive. Thus, in order to avoid this problem, WA-RDF uses an internal binary format called *Internal Resource Mapping (IRM)* that maps a triple resource (S, P, O) to a binary code stored in Redis.

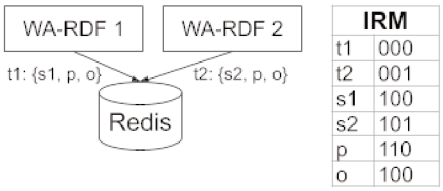

Figure 16 – Internal Resource Mapping (IRM)

Figure 16 exemplifies the mapping. Each new triple stored in WA-RDF receives a binary representation, as well as each resource of the triple. All WA-RDF servers access the mapping repository for translating from and to the binary code. The binary code is sequential, with a prefix for triple (0) and for resources (1). In terms of implementation, each part of the triple is an object of the *Resource*[8] class of *Apache Jena*. Thus, to identify each part of the triple, WA-RDF uses the *getId()* method of the Resource class. The triple id is the concatenation of the three triple parts.

4.3 QUERYING

From a performance point of view, the most important task accomplished by WA-RDF is the query processing. We start with an overview of this task, which is summarized by the Algorithm 3.

The input of the Algorithm 3 is the set of *triple patterns* from the query and the output is the result set *R*. If the query has only simple triple patterns - i.e., there are no joins -, the result is retrieved from SPO and OPS indexes (lines 1 and 2). Otherwise, Algorithm 3 looks for the shapes of the query to define its execution plan. Firstly (lines 4 to 6), WA-RDF loads the triple patterns into two multilevel hash tables *mhtSPO* and *mhtOPS* in order to speedup the further steps. Then, it looks for S-S star shapes (lines 11 to 14), O-O star shapes (lines 15 to 18) and chains (lines 20 to 26).

Star shapes are identified when a subject has more than 2 entries in the *mhtSPO* (line 11), or an object have more than 2 entries on the *mhtOPS* (line 15). In this case, it expands the star shape with all the entries from the multilevel hash tables, registers the results in the star hashmap and add it to the query execution plan stored into the set *stars* that will be later translated to the document database query language (line 28). The triple patterns that do not define star shapes are expanded to chains (line 20). If the expanded chain has size 1 (*i.e.*, the triple pattern itself), the indexes are accessed to get the result triples (line 22 and 23). Otherwise, the expanded chain is registered in the chain hashmap and added to the query execution plan stored into the set *chains*, which is later translated to the graph database query language (line 29). Finally, with

[8] https://jena.apache.org/documentation/javadoc/jena/org/apache/jena/rdf/model/Resource.html

the aid of the *Dictionary*, after the *stars* and *chains* sets are processed by the document and graph databases, the algorithm returns the result set R (line 30).

Input: SPARQL query triple patterns = $\{tp_1, tp_2, ..., tp_n\}$, where
tp$_i = (s_i, p_i, o_i)$
Output: Result set R = $\{t_1, t_2, ..., t_m\}$
if n *== 1* **then**
  R.add(getFromIndex(tp_1));
else
  for $i = 1$ *to* n **do**
    mhtSPO.put(s_i, tp_i);
    mhtOPS.put(o_i, tp_i);
  end
  stars = {};
  chains = {};
  for $i = 1$ *to* n **do**
    if *mhtSPO.get(s_i).size() $>$ 2* **then**
      expandedStar = expandSubject(mhtSPO.get(s_i));
      register(expandedStar, 'star', expandedStar.hop);
      stars.add(expandedStar);
    else if *mhtOPS.get(o_i).size() $>$ 2* **then**
      expandedStar = expandObject(mhtOPS.get(o_i));
      register(expandedStar, 'star', expandedStar.hop);
      stars.add(expandedStar);
    else
      expandedChain = expandChain(tp_i);
      if *expandedChain.horizon==1* **then**
        R.add(indexSPO.get(s_i));
        R.add(indexOPS.get(o_i));
      else
        register(expandedChain, 'chain', expandedChain.hop);
        chains.add(expandedChain);
  end
  R.add(readFromDocument(stars));
  R.add(readFromGraph(chains));
return R;

Algorithm 3: Workload-based triple querying

The input SPARQL queries are analyzed by the *Query Evaluator* component. It, in turn, classifies a query into *simple*, *star*, *chain* or *complex*.

A query is called *simple* if it does not involves a join in any triple component, like `SELECT ?x WHERE { A p1 ?x .}`, `SELECT ?x WHERE { ?x p1 B .}` or `SELECT ?x WHERE { A x? B .}`. This type of query is out of our scope, but for sake of completeness, we solve this query using the indexes S-PO and O-PS.

If the query is classified as *complex*, it is decomposed into one or more *simple*, *star* or *chain* subqueries. The *Query Evaluator* then reports to the *WAc* in order to keep the workload metrics up-to-date, and accesses the *Dictionary* to get the partitions where the triples for these queries are potentially located.

In the following, the subqueries are forwarded to the *Query Processor*. Each star-shaped (O-O or S-S join) query is converted to a query over a document database. For instance, the O-O star-shaped query *Q1* in the following is converted to the access method *D1* (MongoDB NoSQL database syntax) and the S-S star-shaped *Q2* is converted to the access method *D2*. The `$exists` function of MongoDB filters the JSON documents that have all the predicates of each query, moreover in M2 we filter also by the subject M.

Q1: `SELECT ?x WHERE {x? p5 y? . z? p2 y? .}`
Q2: `SELECT ?x WHERE {x? p9 y? . x? p10 M .}`
D1: `db.partition1.find({p5:{$exists:true}, p2:{$exists:true}}})`
D2: `db.partition1.find({p9:{$exists:true}, object:M}})`

The *chain* queries (O-S and S-O joins) are converted to queries over a graph database. For example, given the query *Q3* in the following, with O-S joins, WA-RDF translates it to the query *G1* according to the Cypher[9] query language of the Neo4J NoSQL database.

Q3: `SELECT ?x WHERE { x? p1 y? . y? p2 z? . z? p3 w? . }`
G1:
```
MATCH (f:Fragment)
WHERE ANY(item IN f.p WHERE item = p1 AND
item = p2 AND item = p3)
RETURN p
```

The processing of joins occurs when a query as a whole cannot be executed on a single partition, and it needs to be decomposed into a set of subqueries, being each subquery evaluated separately and joined at the *WA-RDF* node. For instance, if the query Q4 in the following is not able to be completed only querying the partitions P1 or P2 alone. In this case, the component *Query Decomposer* divides it into subqueries SQ5 and SQ6, issues it to the partitions P1 and P2, respectively, and joins the result sets by matching the predicate *p5* (the boundary connection between P1 and P2).

Q4: `SELECT ?x WHERE {x? p2 y?.y? p3 z?.x? p5 w?.w? p9 k?.M p11 k?.}`
SQ5: `SELECT ?x WHERE {x? p2 y?. y? p3 z?. x? p5 w?.}`
SQ6: `SELECT ?x WHERE {x? p5 w?. w? p9 k?. M p11 k?.}`

A complex query is a combination of the *star* and *chain* patterns, potentially connected by simple queries. Query *Q5* in the following is an example, where the

[9] https://neo4j.com/developer/cypher-query-language/

BGP `x? p1 y? . y? p2 z? . z? p3 w?` holds a *chain* pattern, the BGP `z? p5 ?k` is a simple query, and the BGP `k? p6 G . k? p7 I . k? p8 H` holds a *star* pattern. In this case, the decomposition process works as follows: *(i)* it first sorts the triple patterns by subject and object; *(ii)* if it is identified a subset with two or more patterns with the same subject or object, it is considered a *star* subquery, like the subquery *P1* in the following. Then, chains are identified in the remaining query patterns, *i.e.*, *(iii)* for each triple pattern, we navigate from object to subject creating chains, and we pick up the longest chain and consider this a *chain* subquery, like subquery *P2*.

Q5: `SELECT ?x WHERE { x? p1 y? . y? p2 z? .`
`z? p3 w? . z? p5 ?k . k? p6 G . k? p7 I . k? p8 H }`
P1: `{k? p6 G . k? p7 I . k? p8 H }`
P2: `{x? p1 y? . y? p2 z? . z? p3 w?}`
P3: `{z? p5 ?k}`

WA-RDF repeat step *(iii)* until there are no more chains, or there are only simple patterns, like the subquery *P3*. Each *star* and *chain* subquery is processed separately, and the join of the results (along with the simple patterns) is performed at the WA-RDF node. In case of ambiguity, *i.e.*, a pattern that is presented in more than one query type, we consider the following priority: *(1)* subject-based *star* query; *(2)* object-based *star* query; *(3)* the longest *chain* query; and *(4) simple* queries. The star queries are processed with high priority for two reasons: star queries are most common, and the MongoDB translation permits that we query mostly the document keys, what lets queries over documents much faster when compared to queries over graphs.

4.3.1 Querying details

In order to reduce the intermediate results during a query execution it is important to be assertive in the querying process by returning only fragments that contain the desired query answer. Moreover, ideally, the fragments retrieved from the NoSQL databases should not have repeated triples. Thus, WAc profoundly influences on querying and caching.

A query is processed by the WA-RDF component *Query Processor (QProc)*, as shown in Figure 17. Its design goal is to avoid joins between partitions, reduce the unnecessary intermediate results and dynamically choose the best NoSQL node to query. *QProc* is formed by a $Decomposer$, which uses WAc information to decompose the query into star and chain queries. The $Decomposer$ passes these queries to $Planner$, which generates different plans for this queries execution and send these plans to the $Optimizer/Executor$. Each query is tested in parallel against the $Cache$ by the $Optimizer/Executor$. For the queries that are not in the $Cache$,

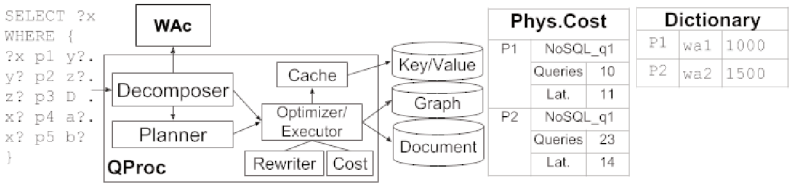

Figure 17 – Overview of QProc

the $Optimizer/Executor$ estimates the $Cost$ and, if necessary, rewrite the query ($Rewritter$). Finally, the rewritten query is executed in the NoSQL databases. For each answered query, the $Optimizer/Executor$ updates the cost of this action ($Phys.Cost$ in Figure 17).

For each SPARQL query, WA-RDF decomposes it into star and chain queries. For each decomposed query, it checks if this query matches any triple pattern registered on WAc. If so, WA-RDF knows what data partitions potentially store the fragments by getting the wa_{id} from the $Dictionary$. Otherwise, if the query matches no triple patterns, WA-RDF returns an empty result. If multiple partitions are found, $QProc$ has to join the results. Also, if a query is simple, *i.e.*, it does not contain star or chain patterns, or it is decomposed into multiple simple queries (there are no joins between the triple patterns), this query can be responded only by accessing the S-PO and O-PS indexes maintained by the *Indexer* component.

The processing of joins occurs when WA-RDF cannot execute a query on a single partition. In this case, the query must be decomposed into a set of subqueries, being each subquery evaluated separately and further joined. For example, supposing that query *Q1* (the same query in Figure 17) in the following is not able to be completed accessing only one partition, WA-RDF divides it into subqueries *S1Q1* and *S2Q1* and issues them to the partitions that hold the data ($P1$ and $P2$, for example - see Figure 17). Next, it joins the result sets by matching the subqueries by the triple `x? p1 y?` (the connection between the partitions).

Q1: `SELECT ?x WHERE {x? p1 y?. y? p2 z?. z? p3 D. x? p4 a?. x? p5 b?}`
S1Q1: `SELECT ?x WHERE {x? p1 y?. y? p2 z?. z? p3 D.}`
S2Q1: `SELECT ?x WHERE {x? p1 y?. x? p4 a?. x? p5 b?}`

Query optimization is made along the $QProc$ execution. Firstly, the $Decomposer$ tries to find all the star shapes present in a query q_i. If it finds only one star or chain shape, it forwards q_i directly to the $Optimizer/Executor$. However, if it detects star and chains, or more than one star or chain in q_i, it forwards q_i to the $Planner$ component. The $Planner$ tries to define options to solve the query and forwards it to the $Optimizer/Executor$. For each query, the $Optimizer/Executor$ tries to run it on the

$Cache$. WA-RDF also consider WAc information to manage caching. When it is time to evict data from the $Cache$, it searches $QMap$ to find out and remove the least common fragments. If the query is not found, the $Optimizer/Executor$ asks for the $Cost$ component to find out the less costly plan, and if it works, it asks for the $Rewriter$ component to modify it in order to improve the query processing.

The main strategy of the $Optimizer/Executor$ is to foster the early execution of triples with low selectivity to reduce the number of intermediate results. The selectivity of a triple pattern is an estimation of the percentage of accessed data. This information can be obtained with the aid of the $Dictionary$. As shown in Figure 17, the $Dictionary$ maintains, for each typical workload, the number of triples for each partition. For instance, the selectivity of $wa1$ is 1000/54500, and 1500/54500 for $wa2$ (54500 is the number of triples present in the Dictionary). When WA-RDF receives $q1$, it processes $wa1$ first. Secondly, WA-RDF considers the historical latency and the number of queries running for each NoSQL query in the table $PhysicalCost$ maintained by the $Cost$ component. This component also maintains a *matrix of joins* between fragments. This matrix is based on the work of Chawla, Singh, and Pilli (CHAWLA; SINGH; PILLI, 2017), but, in our case, it contains joins of fragments instead of joins of triples. The cost matrix is updated after each query. Finally, WA-RDF performs the join based on the cost of each partial query.

With the fragmentation approach, WA-RDF has to assure that the client will not receive additional triples. This checking is made after all resulting fragments come from the partitions. Algorithm 4 is executed to remove unnecessary triples of the result set. Its input is the list of fragments $f = \{f_1, ..., f_n\}$ and the user query, and the output is the final result set of triples R. The algorithm also follows the *map-reduce* paradigm. During the map phase, all triples of each fragment that are not desired are removed by matching to the query. During the $reduce$ phase, the triples are deduplicated. Finally, the result set is returned.

Input: f = $\{f_1, f_2, ..., f_n\}$, q
Output: R = $\{t_1, t_2, ..., t_n\}$
R = f
.map(match: (f, q))
.reduce((t_1, t_2), if $=>$!t_1.equals(t_2));
return R;

Algorithm 4: Map-reduce-based fragment cleaning

In order to reduce intermediate results, every time a query is responded, WA-RDF also tries to merge fragments. This necessity came from the storage problems of the first middleware version (*Rendezvous*), where the storage would grow exponentially. The merging process drastically reduces storage size. It occurs when a fragment has only a small difference in their triples than another existing one. In our current version,

this difference between the fragments must be less than 30%, but WA-RDF is flexible to set other thresholds, and this is a subject for future research.

On defining a merge percentage threshold, we avoid to create large fragments because for every new triple added it would unlikely that a next triple be merged. For instance, suppose that the query presented in Figure 18 (1) returns the fragments illustrated in Figure 18 (2) from a document NoSQL database. In this case, a new document is generated by the union of the triples from both fragments, as shown in Figure 18 (3), and the two previous fragments are deleted. A queue manages this process, which permits that, instead of recreating the whole database, we refragment only the "warm" portions of the database without impacting the rest of the WA-RDF processes.

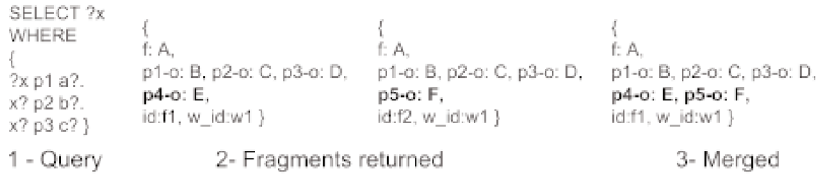

Figure 18 – Merging of Fragments

A merging occurs, for example, if a fragment `f2 = {A p1 B, B p2 C, C p3 D, B p4 E}` is created and a fragment `f1 = {A p1 B, X p2 C, C p3 D, B p4 E}` already exists. In this case, the graph database runs an update over $f1$ (`X p2 C` is replaced by `B p2 C`), and $f2$ is discarded. In the case of a document database, WA-RDF updates the document with the diff triples.

4.3.2 Cache Management

Two levels of cache are managed by WA-RDF. The first level is stored in the memory of each WA-RDF server. The second level of cache is stored in the Redis database. Figure 19 shows a scenario that exemplifies these cache levels. As presented in Section 4.2.3, the data in both cases is stored in terms of triple parts (subject, predicate, object) using the IRM. The triple parts in the cache is identified in the same manner of the IRM, *i.e.*, using the ID provided by Apache Jena.

Suppose that, after the processing of the query at the time *t1* by WA-RDF server 1, the values in the local and remote cache contain A, p1, B in the form SPO, OPS and POS, i.e., this triple indexed by its subject (A, t1), predicate (p1, t1), and object (B, t1). So, the query at time *t2* is able to be responded by the local cache of WA-RDF server 1 by intersecting the result queries to the cache for A and p1, i.e., the result of the query for A is {(A, p1, B)} and the result of the query for p1 is {(A, p1, B), (B, p1, C)}, so the final result is {(A, p1, B)}.

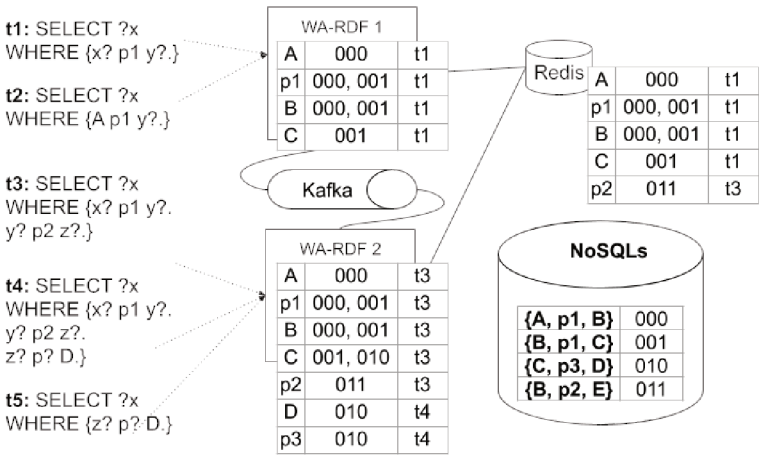

Figure 19 – Distributed Cache Management

Now suppose the WA-RDF server 2 receives the query at time *t3*. It checks its local cache, but finds only the desired data at the remote Redis cache by matching the results from p1 and p2. When the query at time *t4* arrives to WA-RDF server 2, it solves the triple patterns `x? p1 y?. y? p2 z?.` from the cache and query the NoSQL databases only for the triple pattern `z? p? D` to look for triples with the D object. Finally, when the query at time *t5* is issued, WA-RDF server 2 is able to solve it with the data on its local cache.

As exemplified before, the result of a less restrictive query (with more query variables or a smaller number of joins) can be usually reused to answer a more restrictive query. The values of the cache are evicted when a insertion, update or deletion is executed. In this case, all the values of the modified triple are removed from the cache. In order to manage the communication between the local caches, an *Apache Kafka* queue is used. The local cache can also be evicted when the local server memory or the Redis cache space is near to be exhausted. In this case, the least recently used triples are removed first. As explained in the next section, a query response always verify for pending insertion, deletion and update in order to guarantee that the values in the cache are trustful and no inconsistencies will be generated.

4.3.3 GeoSPARQL

Rendezvous was considered a storage layer in a project of our research group related to the management of semantic trajectories of moving objects (MELLO et al., 2019) (see more details in Section 5.2.3). During the experimental evaluation, the Rendezvous parser was enhanced to support queries in the GeoSPARQL[10] standard, which are mapped to MongoDB queries[11]. The Rendezvous parser is also available at

[10] https://www.opengeospatial.org/standards/geosparql
[11] https://docs.mongodb.com/manual/reference/operator/query-geospatial/

GeoSPARQL	MongoDB	
geof:intersection	geoIntersects	
geof:distance	near	

Table 8 – Mapping GeoSPARQL into MongoDB

the last WA-RDF version.

At insertion time, a polygon or a trajectory can include a geometry in the WTK format[12]. The current RDF triples can contain a Point, a Line or a Polygon. During a query, WA-RDF translates GeoSPARQL functions into MongoDB functions, as presented by Table 8.

4.4 PARTITIONING

Partitioning and replication processes are responsible to efficiently distribute the fragments. The main data source for these processes is the Dictionary. Algorithms 5 and 6 start after WA-RDF determines the *database routing*, i.e., after the definition of the partitions where a fragment can be stored.

For every new fragment that has to be inserted, the Algorithm 5 is executed. Given a new fragment f_{new}, each one of its triples t_i is queried in the dictionary (lines 1 and 2) through the *queryDictionary()* function, which returns a list of partitions (P_{t_i} = $\{p_1...p_m\}$) and cardinalities (C_{t_i} = $\{c_1...c_m\}$) for each triple. In line 4, the new predicates are added to a temporary hash map to be added in the replication algorithm. For each predicate found in the Dictionary, the cardinality is summed in the hash map where the key is the partition identification and the value is the sum of the cardinalities (in line 8). In line 9, the triples are added to the partitions that have to be inserted to the dictionary. Using the values of the addTriples hash, in line 12, the algorithm adds the triples that have subject in a partition and object in another.

Moreover, it adds the triples that crosses the partitions, i.e., triples that have the subject in a partition and the object in another partition. At the end of this loop, this hash contains the sum of cardinalities for each partition (lines 3 to 5). Finally, the algorithm returns a map of the partitions ordered by the highest cardinality, including the list of the triples that are connected to this partition. The partitions are ordered by the highest cardinality because the higher the cardinality is the more connected the fragment will probably be to the rest of the partition fragments.

Figure 20 shows an example scenario for this task. Given a new fragment to be stored, WA-RDF issues queries to the Dictionary for each fragment triple {(F, p6, A), (A, p7, D), (A, p1, B)}. In this case, the answer for the triple (F, p6, A) is *(P1, 10)* and *(P2, 10)*. Also, the answer for (A, p1, B) is *(P2, 15)*. So, the sum of cardinalities is 25 for

[12] https://www.opengeospatial.org/standards/wkt-crs

Input: f_{new} = $\{t_1, t_2, ..., t_n\}$
Output: Result set R = $\{p_1, p_2, ..., p_m\}$, where p_m=$\{c_m, [t_1, ..., t_o]\}$
for $i = 1$ *to* n **do**
    P, C = queryDictionary(t_i);
    if m==*0* **then**
        tempHashMaps.addNewPredicate(t_i);
    end
    else
        for $j = 1$ *to* m **do**
            tempHashMaps.sumCardinality(p_j, c_j);
            tempHashMaps.addTriples(p_j, t_i);
        end
    end
    tempHashMaps.addBorderTriples(t_i);
end
R = tempHashMaps.orderByPartitionCardinality();
return R;

Algorithm 5: Partitioning Algorithm

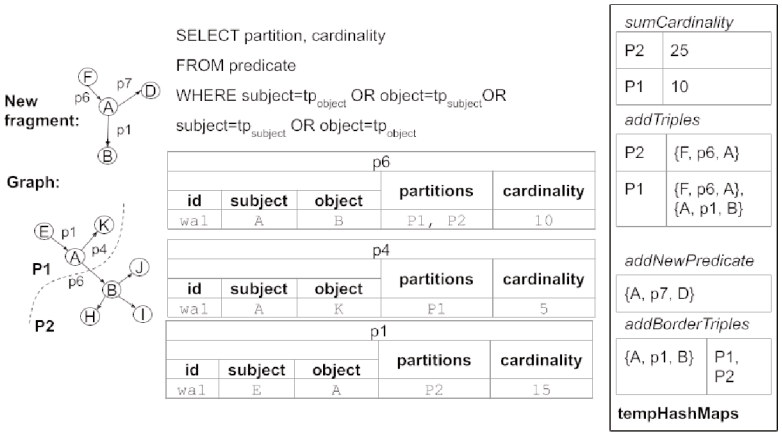

Figure 20 – Data Partitioning Scenario

partition P2 and 10 for the partition P1. When the predicate of a triple does not exist, a new table is created in the Dictionary. This is the case for the triple (A, p7, D).

Sometimes, the partitioning process must lead with a situation where more than one partition can benefit of a fragment. There are two situations when a fragment is replicated: *(i)* when a partition is too small (in the current version, if the threshold is less than 100 fragments) and there is at least one connected triple; *(ii)* when all the triples of a fragment is connected to a partition. In the first case, the rational is to increase the usage of the partitions. In the second case, the idea is to increase the connectivity of the sub-graph persisted in each partition.

From that, the Algorithm 6 has the following objectives:

- To keep the graph of each partition as connected as possible, by choosing the

bigger cardinality;

- To evenly distribute the fragments replicating the fragment, if more than one partition have the same number of triples connected;
- To replicate the border of the partition by using a *replication parameter* set globally for WA-RDF by a administrator.

In short, Algorithm 6 receives the new fragment to be stored and the *tempHashMaps* as the input, and returns a list of partitions where the fragment must be stored. In line 2, the algorithm finds the maximum value for the cardinalities. In lines 3 to 6, the new triple causes a new table to be created in the Dictionary. In lines 7-12 the algorithm iterates over the triples to be added and checks if, for any partition, the fragment is equally or more connected to the highest cardinality partition. If so, this partition also receives the fragment, *i.e.*, the fragment is replicated into this partition.

Input: f_{new} = $\{t_1, t_2, ..., t_n\}$, tempHashMaps
Output: Result set R = $\{P_1, P_2, .., P_n$

```
Partitions.add(max(tempHashMaps.sumCardinality));
NewTriples = tempHashMaps.addNewPredicate;
for i = 1 to n do
    createNewTableDictionary(NewTriple[i].predicate);
end
Triples = tempHashMaps.addTriples;
for i = 1 to n do
    if Triples[i].size>=tempHashMaps.addTriples.get(Partition[0]).size then
        Partitions.add(Triples[i].getPartition());
    end
end
return Partitions;
```

Algorithm 6: Replication algorithm

In the example of Figure 20, the biggest cardinality is the partition *P2*. However, the partition *P1* is more connected to the fragment. So, the fragment is replicated into both partitions *P1* and *P2*. Moreover, a new table in the Dictionary is created for the predicate *p7* with a row for the subject *A* and object *D*.

4.4.1 Dictionary Design

The Dictionary is the component of the middleware responsible for maintaining the distribution of the fragments over different partitions. The Dictionary is stored into Cassandra database. The main reason for choosing a columnar database is the possibility to model it as a *family of columns*, as exemplified in Figure 21.

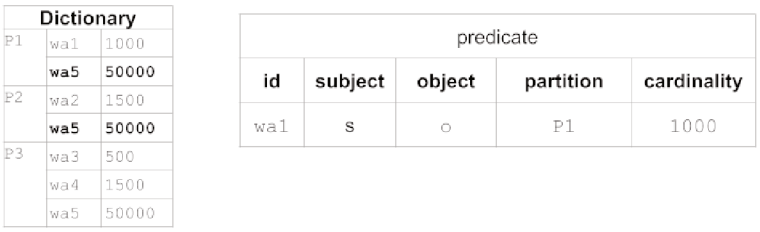

Figure 21 – Dictionary Design

For each triple pattern stored in the database, the dictionary keeps a *table* in Cassandra that has as name the predicate (represented in IRM format). Each table, in turn, has columns that hold the workload identification, the triple subject and object, the partition were the fragment is stored and the cardinality of the workload.

NoSQL document databases also hold a nested structure, so it could be employed in a similar way to keep the Dictionary. However, there are a few caveats that prevented this choice: *(i)* the lack of naming on a document; *(ii)* it is not possible to have a direct access to a specific document, as for columnar tables named by a predicate; and *(iii)* in order to replicate the subject and object query in a document, we would have to store a list for all the subjects and objects and for every new triple, and it would have to be updated. MongoDB, for example, would not be suitable for this type of update-heavy workload[13].

Such a dictionary modeling is suitable for fast discovery of which partitions maintain triples that match a given triple pattern `tp` during a manipulation operation. Specifically, the following query template over the Cassandra database is posed:

```
SELECT partition, cardinality FROM predicate
WHERE id.subject = tp.subject OR id.object = tp.subject
OR id.subject = tp.object OR id.object = tp.object
```

4.5 UPDATE AND DELETE

Data fragmentation can also occur during an update or delete of a triple. For the update, the WA-RDF strategy is to create, in the background, a new fragment and remove the old one. To avoid inconsistency during this process, WA-RDF manages an updated/deletion list that assures that the fragment triples are updated or deleted before they are sent to the client, as illustrated in Figure 22 (1). In this case, the triple `A p1 B` was updated to `A p1 Z`. The update process has two phases, as also shown in Figure 22. Firstly, WA-RDF matches this triple to the WAc triple patterns. In the following, it removes the fragments that have this triple (Figure 22 (2)). Secondly, it runs an insertion

[13] *https://benchmark-docs.readthedocs.io/en/latest/benchmarks/mongodb-ycsb-ct720-jul2015.1.html*

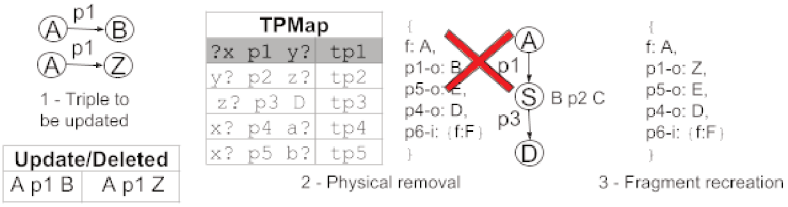

Figure 22 – Triple Update

of the new value of the triple, i.e., it recreates its corresponding fragment (Figure 22 (3)). Chapter 5 shows that the update time is not prohibitive.

The deletion process is similar to the update. In this case, WA-RDF creates a new fragment that holds all the existing triples in the old fragment except the deleted triple.

4.5.1 Pending updates and delete

Changes on the dataset, triggered by insertion, deletion or update operations, can generate inconsistencies that must be managed without adding a big burden on the data management infrastructure. The main related challenges are the following:

- To avoid high latency in the query processing;
- To minimize the insertion, deletion or update operation latency;
- To keep the response of WA-RDF consistent even if the dataset has small and temporal inconsistencies.

In order to provide low latency, two technologies were considered in the middlware design: a *Kafka queue*, which is used to control the pending operation, and the *Redis database* to store temporary values. The design rational is that the dataset managed by WA-RDF can be inconsistent for some seconds, but the results returned by the middleware have always to be consistent.

During an insertion, the new triple is firstly added in Redis by considering using the format defined in Section 4.2.3. After that, a *fragmentation creation* message is queued in Kakfa and asynchronously, the Algorithm 2 is executed. In the meantime, for every query received, WA-RDF checks if any of its triple patterns matches a temporary triple of Redis. If so, the temporary triples is added to the result dataset returned by the middleware.

For update and deletion operations, the mechanism is very similar. During an update, the old value of the triple is added as the key of an entry on Redis, where the new value of triple is the value. Thus, before returning a query result set, WA-RDF checks if the old value is present and exchanges it for the new value. Deletions follow a

similar reasoning, but instead of exchanging a value, the deleted triple is removed from the result set.

Physically, each triple is marked with a prefix for the pending operations, n: for new, u: for update and d: for delete. Thus, it is possible to use Redis for storing both pending operations and the cache.

Fragment merging is different because it is executed at a fragment level instead of a triple level. After the answer of a query, the fragments returned from the NoSQL database(s) are added to a Kafka queue in order to be checked against a similarity threshold. If the fragments are similar, a new fragment merging the similar fragments is created and the merged fragments are physically deleted. This process also affects the Dictionary and the Cache. Thus, the old fragments are only deleted after all the components be updated.

Finally, it is important to mention that these checkings are performed in parallel with the query processing. Thus, the results of the queries for pending operations are usually processed faster than the result of the query itself because pending information is available at Redis, which is a fast access database.

4.5.2 Asynchronous Processing

Before the latest version of WA-RDF, three important capabilities were missing: data movement and replication, as well as update and delete operations. Their support were delayed because the middleware did not hold an asynchronous processing component. Due to the thesis time and scope limitations, we considered an existing Big Data processing component: *Apache Kafka*[14].

Apache Kafka is a real-time streaming messaging system and protocol built around a *publish-subscribe* protocol. In this protocol, producers publish data to feeds for which consumers are subscribed to. Kafka is connected to the components Fragmenter/Mapper and Partitioner of the WA-RDF architecture, as shown in Figure 23.

Other options could be considered, like *RabbitMQ* (DOBBELAERE; ESMAILI, 2017), but Apache Kafka is currently the more scalable framework of this nature. Moreover, WA-RDF is not the only triplestore using this technology. *Strider* (REN; CURÉ, 2017), for instance, uses the capabilities of Kafka to create a streaming solution based on RDF data.

4.6 RUNNING EXAMPLE

This section presents a running example to better understand how WA-RDF works. We consider here an application that maintains data about trajectories of moving objects, and the RDF graph of Figure 24. Suppose we have an user called *John Smith*

[14] https://kafka.apache.org/

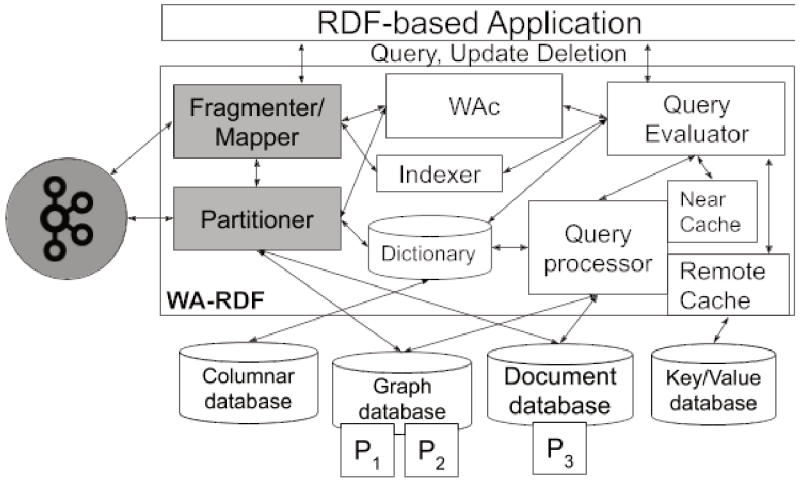

Figure 23 – WA-RDF with Apache Kafka for asynchronous processing

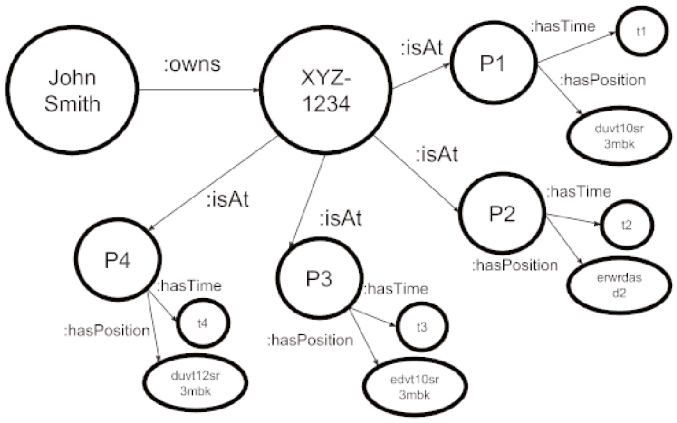

Figure 24 – Example of RDF Data for a Moving Object Trajectories Application

starts using his car with license plate *XYZ-1234* at his house. Thus, in this first moment (time *t1*), this data have to be stored. Figure 25 presents the state of the middleware before and after the insertion of the triples t_1 = *('John Smith', :owns, 'XYZ-1234')*, t_2 = *('XYZ-1234', :isAt, 'Point 1')*, t_3 = *('Point 1', hasTime, 'time1')*, t_4 = *('Point 1', :hasPosition, 'duvt10sr3mbk'* [15]*)*.

Each one of these triples are initially stored in a Document database as separated JSON documents with the format t_n = *{subject, predicate, object}*. After the insertion of all the triples, the RDF graph, the Dictionary, the indexes and the Document database are updated. The WAc and the Graph database remain empty. The WAc is represented here by two datasets that register star and chain shapes (*DS Star* and *DS Chain*, respectively). As stated before, the Dictionary is responsible for registering which are the triples stored in each partition. In this case, we are assuming only one

[15] We are using *geohash* (NIEMEYER, 2008) for describing coordinates of the geographical points.

t1 - Before	Insert triples [t_1,... t_4]
Graph	Empty
Dictionary	Empty
DS Star	Empty
DS Chain	Empty
Index SPO	Empty
Index OPS	Empty
Document database	Empty
Graph database	Empty

t1 - After	
Graph	{'John Smith', :owns, 'XYZ-1234'}, ..., {'Point 1', :hasPosition, 'duvt10sr3mbk'}
Dictionary	P1 = [t_1,... t_4]
DS Star	Empty
DS Chain	Empty
Index SPO	'John Smith' = {t1}, 'XYZ-1234' = {t2}, 'Point 1' = {t3, t4}
Index OPS	'XYZ-1234' = {t1}, 'Point 1' = {t2}, 'time1' = {t3}, 'duvt10sr3mbk' = {t4}
Document database	[{'subject':'John Smith', 'predicate':'owns', 'object':'XYZ-1234', _id:t1}, ...]
Graph database	Empty

Figure 25 – Triples insertion at time t1

t2 - After	SELECT ?y WHERE { 'XYZ-123' :isAt x?, x? :hasTime y?. x? :hasPosition z?. } ORDER BY ?y
Graph	{'John Smith', :owns, 'XYZ-1234'}, ..., {'Point 1', :hasPosition, 'duvt10sr3mbk'}
Dictionary	P1 = [t_1,... t_4]
DS Star	[{ _ :isAt ?x, ?x :hasTime _. _ :hasPosition ?x. }]
DS Chain	Empty
Index SPO	'John Smith' = {t1}, 'XYZ-1234' = {t2}, 'Point 1' = {t3, t4}
Index OPS	'XYZ-1234' = {t1}, 'Point 1' = {t2}, 'time1' = {t3}, 'duvt10sr3mbk' = {t4}
Document database	[{'subject':'John Smith', 'predicate':'owns', 'object':'XYZ-1234', _id:t1}, ...]
Graph database	Empty

Figure 26 – Query at time t2

partition *P1*.

When *John* turns his car on, an navigation system, for example, searches his position to display the car in the map. It generates the following SPARQL query at a time *t2*: `SELECT ?z WHERE { 'XYZ-123' :isAt x? . x? :hasTime y?. x? :hasPosition z?.} ORDER BY ?y`. As presented in Figure 26, this query is stored in the WAc. We remove the actual values of the subjects and objects, storing only the predicates to keep the shape of the query. As the triples are stored as separated documents in the Document Database, WA-RDF executes three queries: *(i)* get all documents such the subject is *'XYZ-123'* and the predicate is *:isAt*; *(ii)* for each object found in *(i)*: get the documents with the predicate *:hasTime*; *(iii)* for each object found in *(i)*: get the documents with the predicate *:hasPosition*. In this case, the first position returned is *'duvt10sr3mbk'*.

As *John Smith* moves towards his office, the GPS of his car updates the position, at a time *t3*, with the triples t_5 = *('XYZ-1234', :isAt, 'Point 2')*, t_6 = *('Point 2', hasTime,*

t3 - After	Insert triples [t_5, t_6, t_7]
Graph	{'John Smith', :owns, 'XYZ-1234'}, ..., {'Point 1', :hasPosition, 'duvt10sr3mbk'}, {'Point 2', :hasPosition, 'erwrdasd2'}
Dictionary	P1 = [t_1,... t_7]
DS Star	[{ _ :isAt ?x, ?x :hasTime _. _ :hasPosition ?x. }]
DS Chain	Empty
Index SPO	'John Smith' = {t1}, 'XYZ-1234' = {t2, **t5**}, 'Point 1' = {t3, t4}, **'Point 2' = {t6, t7}**
Index OPS	'XYZ-1234' = {t1}, 'Point 1' = {t2}, 'time1' = {t3}, 'duvt10sr3mbk' = {t4}, 'Point 2' = {t5}
Document database	[{ 'subject':'XYZ-1234', 'predicate':isAt, 'object': [{ 'subject':'Point 2', 'predicate':hasTime, object:'time2'}, { 'subject':'Point 2', 'predicate':hasPosition, object:'erwrdasd2' }], ...]
Graph database	Empty

Figure 27 – Triples insertion at time t3

'time2'), t_7 *= ('Point 2', :hasPosition, 'erwrdasd2')*. Now, as we have workload information (the query executed at time *t2*), the triples provoke the generation of a fragment, as present in Figure 27, in the Document database row, the row DS Star will receive a entry, and a new entry will be added to the index SPO.

Consider now that at a time *t4* the application queries all the points of all the users in the last hour to display the user's avatar in the application map. The following query is then executed: `SELECT ?x WHERE {?u :owns ?c . ?c :isAt x?}`. As presented in the left part of Figure 28, this query is stored in the WAc as a chain query because it has a chain shape. The query is executed in the Document database, and several joins must be executed to return all the points.

t4 - After	SELECT ?x WHERE \{ ?u :owns ?c . ?c :isAt x? \}
Graph	{'John Smith', :owns, 'XYZ-1234'}, ..., {'Point 1', :hasPosition, 'duvt10sr3mbk'}, ..., {'Point 2', :hasPosition, 'erwrdasd2'}
Dictionary	P1 = [t_1,... t_7]
DS Star	[{ _ :isAt ?x, ?x :hasTime _. _ :hasPosition ?x. }]
DS Chain	**[{ ?u :owns ?c . ?c :isAt x?}]**
Index SPO	'John Smith' = {t1}, 'XYZ-1234' = {t2, t5}, 'Point 1' = {t3, t4}, 'Point 2' = {t6, t7}
Index OPS	'XYZ-1234' = {t1}, 'Point 1' = {t2}, 'time1' = {t3}, 'duvt10sr3mbk' = {t4}, 'Point 2' = {t5}
Document database	[{ ... }]
Graph database	Empty

t5 - After	Insert triples [t_8, t_9, t_{10}]
Graph	{'John Smith', :owns, 'XYZ-1234'},..., {'Point 2', :hasPosition, 'erwrdasd2'},..., {'Point 3', :hasPosition, edvt10sr3mbk}
Dictionary	P1 = [t_1,... t_{10}]
DS Star	[{ _ :isAt ?x, ?x :hasTime _. _ :hasPosition ?x. }]
DS Chain	**[{ ?u :owns ?c . ?c :isAt x?}]**
Index SPO	'John Smith' = {t1},..
Index OPS	'XYZ-1234' = {t1},...
Document database	[{ ... }]
Graph database	CREATE (n:RDFNode { subject: 'John Smith', predicate: ':owns' , object: 'XYZ-1234'}) CREATE (n:RDFNode { subject: 'XYZ-1234', predicate: ':isAt , object: 'Point 3'}})

Figure 28 – Chain shape query at time t4 and triples insertion at time t5

After 5 minutes (time *t5*), points continue to be inserted through WA-RDF, and

the middleware receives the same query of time *t4*. So, when the application updates *John Smith* position with the triples t_8 = *('XYZ-1234', :isAt, 'Point 3')*, t_9 = *('Point 3', hasTime, 'time3')*, t_10 = *('Point 3', :hasPosition, 'edvt10sr3mbk)*, WA-RDF stores them in the graph database because the WAc contains now chain-shaped fragments (right part of Figure 28).

4.7 FINAL REMARKS

This chapter presents the thesis proposal: the WA-RDF middleware. It first introduces its architecture and, in the following, details its novel strategies for RDF data manipulation based on a workload-aware approach. Additionally, RDF mapping to the NoSQL data models, RDF data fragmentation and partitioning are discussed. Finally, a running example shows how WA-RDF behaves in an application with trajectory data specified in RDF.

This set of strategies makes WA-RDF a original triplestore based on a middleware that deals with all NoSQL data models and provides all RDF data manipulation operations. Next chapter presents the most important decisions regarding WA-RDF design and implementation.

This chapter gives details of the design and implementation of the middleware proposed for this thesis. The ideas and algorithms presented in this chapter are very important to the thesis results.

We can organize the design and implementation decisions into four groups: *(i)* the architectural decisions, guided by iterations over the state-of-the-art, such as semantic technologies over a polystore abstraction, mapping, partitioning and data replication; *(ii)* architectural decisions, guided by limits over the triplestore solutions, such as the asynchronous processing, the processing components, the map-reduce fragmentation and cleaning, and the usage of pending insertion, deletion and update; *(iii)* implementations-time decisions, like the internal resource mapping, the GeoSPARQL implementation, the database routing and the shape identification; *(iv)* decisions made with the intention of exploring the maximum of the NoSQL databases technology, like the dictionary and the caching algorithm.

These decisions were not explored in published papers yet. These ideas will be summarized and organized as a future paper about WA-RDF.

5 EXPERIMENTAL EVALUATIONS

This chapter presents several evaluations regarding the proposed middleware. First of all, a brief qualitative analysis compares the reuse of WA-RDF in an e-commerce application against *OrientDB*, listed in *DB-engines ranking*[1] as the most used multimodel NoSQL database. Also, a quantitative analysis is performed using the more modern RDF/SPARQL benchmark, called WatDiv. Finally, a comparison in a application in the Semantic Trajectory domain is presented, comparing this thesis results with the state-of-the-art in this domain.

In the following, we detail a set of experiments that evaluates the performance of WA-RDF.

5.1 QUALITATIVE ANALYSIS

On considering the same data model of Figure 2, we developed a data layer including insert and query operations for an e-commerce application using both WA-RDF and OrientDB. On using a Java client, Table 1 shows that when the data layer uses OrientDB, it was necessary almost twice as Java classes as the client using WA-RDF. The code for this comparison is available on the authors' GitHub[2].

Work	WA-RDF	OrientDB
Classes	1	4
Lines of code	90	176
Calls for create User and Product	2	5
Calls for queries	1	4

Table 9 – Qualitative comparison of WA-RDF and OrientDB

More coding is required by OrientDB because it exposes to the programmer all the complexity of multiple models. Although this is a preliminary study, it shows how a single abstraction facilitates software development and maintenance.

5.2 PERFORMANCE EVALUATION

Several experiments were executed during the thesis development. During the initial tests, the benchmark used was the *Lehigh University Benchmark (LUBM)* (GUO; PAN; HEFLIN, 2005), very considered by related works. The results on using this benchmark were published in our SBBD paper (SANTANA; SANTOS MELLO, 2017b) and DEXA paper (SANTANA; SANTOS MELLO, 2019b). Now, we consider in this chapter the *WatDiv*, a more modern and used benchmark (published almost 10 years

[1] https://db-engines.com/en/ranking
[2] https://github.com/lhzsantana/orientdb-wardf

after LUBM) that provides a diverse dataset in terms of the RDF graph and the SPARQL queries. Besides, we evaluate in this chapter the application of our middleware in a domain of moving object trajectories.

5.2.1 WatDiv Benchmark

The WatDiv benchmark (ALUÇ; HARTIG, et al., 2014) was developed as an evolution of LUBM, holding a more diversified dataset. It offers an RDF dataset in the e-commerce domain, having concepts like Product, Purchase and Retailer.

The used dataset has three sizes: 10 million triples[3], 100 million triples[4], and 1 billion triples[5]. The workload consists of 20 query templates, which generates almost 10,000 queries at each experiment[6]. These queries are approximately 40% simple, 40% star and 20% chain or a combination of both shapes.

Figure 29 shows several experimental results based on the usage of the WatDiv benchmark. Figure 29(a) compares our previous middleware version (Rendezvous), which uses the columnar NoSQL database *Cassandra*, with our current version (WA-RDF), which uses Neo4j as the graph NoSQL database. Each one of the 20 query templates were executed 5 times (they are presented in the X axis in a compressed way). It shows that WA-RDF is more than 300ms faster on average, mainly because Rendezvous had to perform multiple calls to $Cassandra$. This change speeds up the overall architecture in around 180ms.

We also evaluate the performance when using key/value, document and graph NoSQL databases. The key/value database main problem is the need for executing multiple calls during query response, and the unnecessary returned triples, mainly for the limitations on filtering only by the key. The document database is not efficient to solve queries with multiple chain subqueries given its hierarchical data access. The graph database is slow for the simple and star queries, also due the multiple calls. This result shows that using document and graph databases achieves the best mix in terms of performance.

Figure 29(b) shows the average times for update and delete operations. They were calculated from the time between the request and the return message of the operation. The workload for the update and delete was created by modifying the triples to be inserted. During the deletion test, it was inserted all the triples and then they were deleted one-by-one. In this experiment, it was not possible to compare WA-RDF with related work because they do not support both of these operations. For the *1GB* (i), *10GB* (ii) and *100GB* (iii) dataset sizes, the graph shows that the deletion time is uniform (around 1200ms on average), but the update time increases as the dataset grows

[3] https://dsg.uwaterloo.ca/watdiv/watdiv.10M.tar.bz2
[4] https://dsg.uwaterloo.ca/watdiv/watdiv.100M.tar.bz2
[5] https://dsg.uwaterloo.ca/watdiv/watdiv.1000M.tar.bz2
[6] https://dsg.uwaterloo.ca/watdiv/stress-workloads.tar.gz

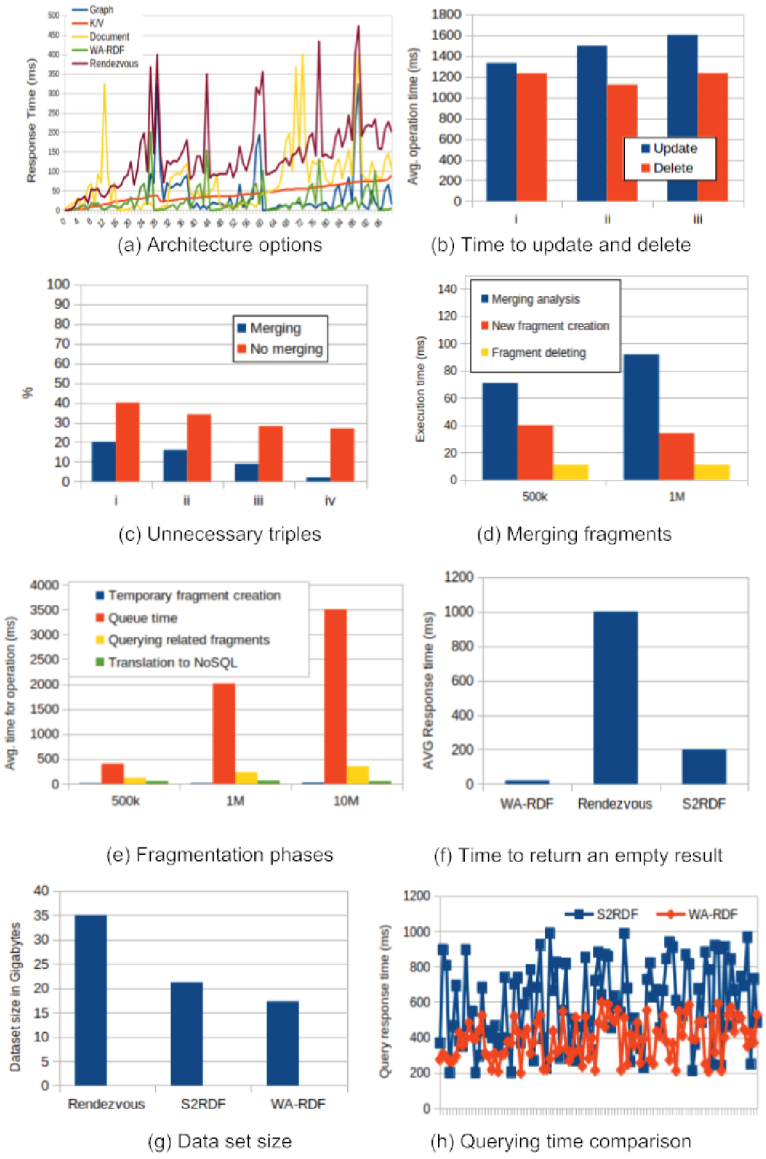

Figure 29 – Experiments with WA-RDF using WatDiv benchmark

(from 1300ms to 1600ms on average). This time increase is due to bigger fragments that can be created with a bigger dataset. It makes the fragment creation process takes longer as the dataset grows.

Figure 29(c) shows how much fragment triples were not necessary during the processing of four different query shapes. It is considered here scenarios with and without the merging process. In case (i), the triples are all complex, and the percentage of unnecessary triples was 20%. In case (ii), there are chain queries, and 16% was discarded. In case (iii), there are star queries, and 9% was not necessary, and in case (iv), where all queries were solved in the cache, only 2% was unnecessary. Without the merging process, around 35% of the triples were unnecessary for all the tests. It highlights the importance of having a proper access plan, mainly for the chain queries,

and the benefit of merging the fragments.

Figure 29(d) shows the spent time to merge fragments. The time calculation starts after a query response until the old fragment is deleted. The experiments were run for sizes of 500 thousand and 1 million triples. We show here the spent time for the three main tasks accomplished by the merging process. The only task of this process that is influenced by the size of the dataset is the merging analysis (when WA-RDF compares the queries), mainly due to the memory usage. Even so, the processing time is not prohibitive.

Figure 29(e) presents the time spent by each fragmentation creation phase: temporary fragment creation, queue time, querying related fragments and translation to the target NoSQL database. The times were collected for the dataset sizes of 500 thousand, 1 million and 10 million triples. The queue time is heavily influenced by the dataset size, as well as the time spent on querying the fragments to expand the query. However, the time spent in the 10M dataset is not prohibitive and scales linearly w.r.t. the dataset size. As expected, there is a directly proportional relation between dataset size and processing time for creating the fragment.

The next experiments compare WA-RDF with its previous version as well as the most famous and fast baseline (S2RDF). Figure 29(f) shows the average time to return an empty result. WA-RDF is much faster because it does not need to access the NoSQL databases if the workload information is not present in the Dictionary. Figure 29(g) presents the dataset sizes generated for WA-RDF, Rendezvous and S2RDF. It is considered here a dataset with a raw size of around 13GB. As illustrated, Rendezvous uses around 35GB, S2RDF about 20GB and WA-RDF slightly more than 18GB. This result is exclusively due to the merging process that avoids unnecessary replication of triples. Figure 29(h) compares the average query processing time for WA-RDF against the S2RDF baseline for 100 queries (each one of the 20 query templates executed 5 times). It shows the superior performance of WA-RDF for the great majority of the queries. It is possible to see that WA-RDF average execution was around 400ms while S2RDF average was around 600ms. However, S2RDF presented pretty larger standard deviations (200ms to 1000ms) when compared to our proposal (200ms to 600ms).

Finally, Figure 30 compares WA-RDF to Rendezvous using the columnar NoSQL database Cassandra to store chain fragments, as shown in Santana and Mello, 2017 (SANTANA; SANTOS MELLO, 2017a). WA-RDF is on average more than 300ms faster in chain queries, and around 180ms average overall. This improvement is due to the necessity of multiple calls to Cassandra in the previous version.

5.2.2 Comparison with a Industry Multimodel Database

This experiment evaluates the performance of WA-RDF against a industry multi-model database: *OrientDB*. The purpose here is to compare our polyglot middleware

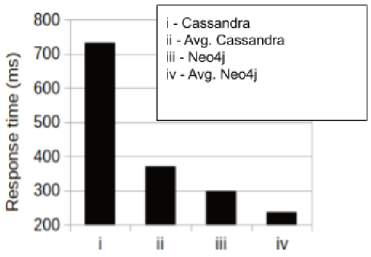

Figure 30 – Rendezvous versus WA-RDF

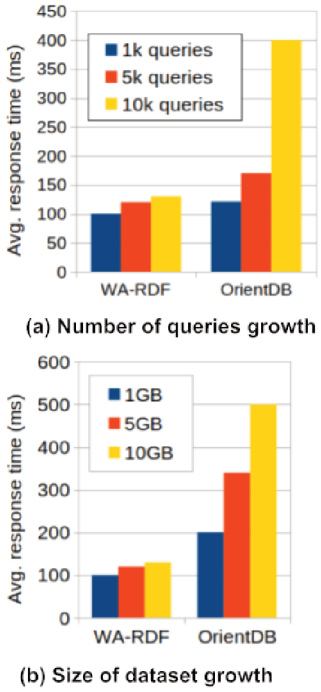

Figure 31 – WA-RDF and OrientDB comparison

with another polular polyglot solution. The tests consider OrientDB in a cluster configuration with four nodes, and the WatDiv dataset was adapted to OrientDB format.

Figure 31 (a) shows that OrientDB has difficulty in maintaining the same response time when the frequency of the queries increase. This result is mainly due to the caching features implemented in Redis. In turn, Figure 31 (b) shows that WA-RDF response time scales sublinearly to the dataset growth, while OrientDB presents a linear scale. This result is particularly true when we have a query that joins pieces of data from different data models. In this case, the WA-RDF fragmentation capabilities help on maintaining the response time almost the same independent of the amount of data stored in the NoSQL databases.

WA-RDF code, the dataset, and the OrientDB client can be found in the authors'

GitHub[7].

5.2.3 Middleware Application in a Semantic Trajectory Domain

This section evaluates the usage of our middleware as a data layer solution for managing trajectory data modeled in RDF format. In particular, we consider here trajectory data whose points can be enriched with semantic information (*semantic trajectories*). The management of semantic trajectories is also an open issue in the research area of Geographic Databases. This experiment contributed to a TGIS journal paper of our research group that proposes a new data model for semantic trajectories and considers our first version middleware (Rendezvous) as the data management layer (MELLO et al., 2019).

Our focus here is on comparing the query performance of Rendezvous against a state-of-art spatiotemporal database system for semantic trajectories proposed by Guting et al. (GÜTING; VALDÉS; DAMIANI, 2015), called *Secondo* (VALDÉS; DAMIANI; GÜTING, 2013). We consider here a dataset called BerlinMOD[8]. BerlinMOD is a benchmark for spatiotemporal database management systems, created by the Secondo team, which is able to generate semantic trajectories of vehicles. The benchmark provides more than 25 types of spatial and/or temporal and/or semantic queries. Some examples of BerlinMOD query types are: *(i)* What are the pairs of vehicles of type ”truck” whose trajectories have ever been as close as ”10m” or less to each other? (a spatio-semantic query); *(ii)* What vehicle of type ”passenger” has a trajectory that reached a point ”(X,Y)” before all the trajectories of vehicles of the same type during the time interval ”[ts, te]”? (a spatiotemporal semantic query). On using the BerlinMOD generator, we created a 19.45 GB dataset with around 53 million trajectories.

The dataset was converted to RDF and the queries to SPARQL. The total size of the converted dataset is around 60 million triples. The SPARQL queries were transformed into 6 simple, 5 star, 7 chain, and 7 complex queries.

We ran experiments considering both Rendezvous and Secondo as distributed infrastructures in the cloud in order to obtain better performance for large data volumes. For this experiment, Rendezvous uses MongoDB 3.4.3 and Neo4J 3.2.5. All the distributed data nodes are Amazon m3.xlarge spot instances[9] with 7.5 GB of memory and 1 x 32 SSD capacity. For all the experiments, we define nodes that represent MongoDB + Neo4J servers, and the Rendezvous servers were also installed on each node. In order to provide an equivalent test environment, we installed Secondo, following its tutorial[10], in a cluster with the same size of the Rendezvous installation (the same number of nodes of Amazon m3.xlarge spot instances). In both cases, all the queries

[7] https://github.com/lhzsantana/wa-rdf
[8] http://dna.fernuni-hagen.de/secondo/BerlinMOD/BerlinMOD.html
[9] https://aws.amazon.com/ec2/instance-types/
[10] http://dna.fernuni-hagen.de/secondo/DSecondo/DSECONDO/Website/index.html

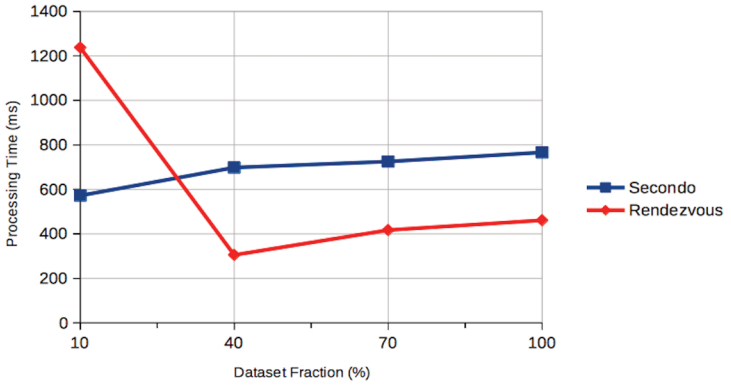

Figure 32 – Comparison of Query Performance over BerlinMOD between Secondo and Rendezvous

were issued from a server in the same network, so the latency between the client and the clusters was inexpressive.

We divide the experiments into several rounds (Round 1 to Round 4). We first insert 10% of the dataset and ran all the queries in order to initiate the workload-awareness (we call it Rendezvous *warm up*). Then, in the following, now with an existing workload-awareness, we ran again all the queries four times over increasing fractions of the dataset (40%, 70% and 100%) and get the average processing times. The results are shown in Figure 32.

As can be seen in this Figure, we obtained an average query processing time that outperforms Secondo from Round 2 on. We present a worse performance only at Round 1 because Rendezvous is not aware of the typical query workload at this time and it spends an extra time to analyze the query shapes as well as to store and index trajectory data usually accessed by these queries in an efficient way for further retrieval. Once aware of the typical workload, Rendezvous ran two times faster than Secondo in average.

This first set of experiments was carried out over four Amazon nodes. In order to evaluate if the performance is positively affected by an increase in the number of nodes, we duplicate them (eight nodes) maintaining the same setup (memory, storage and servers) on each of them. After that, we executed again all the queries four times and got the average processing time on each round. The results are shown in Figure 33.

Figure 33 reveals that our storage solution takes more advantage of the increase in the distributed infrastructure than Secondo. Both approaches spent less time to process the queries, but Rendezvous processing time is faster and does not increase so much with the increase of the dataset if compared to Secondo. This is justified mainly by the efficient query execution strategy followed by Rendezvous, which benefits from the increase in the number of nodes to parallelize the processing of the parts of a query.

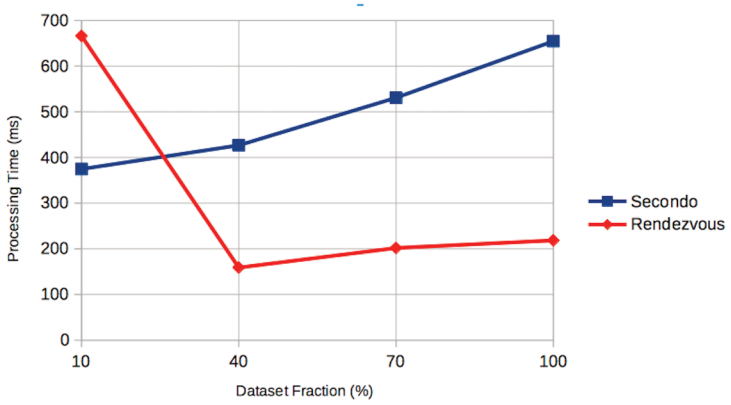

Figure 33 – Comparison of Query Performance between Secondo and Rendezvous over a Larger Number of Nodes

The efficient storage into NoSQL databases and workload-aware indexing scheme also contribute to this better performance.

This quantitative evaluation highlights that our solution for storing and querying semantic trajectories is promising since it demonstrates a good performance for processing queries over increasing fractions of the dataset, and a better scalability when compared to the baseline. In fact, our middleware had demonstrated to be suitable to applications whose data grows exponentially, which may be the case of applications that deal with semantic trajectories, since the add of new machines to process an increasing number of trajectories improves its query processing time performance.

5.3 FINAL REMARKS

This chapter presents several experimental evaluations with the intention to verify the efficiency of the proposed middleware mainly w.r.t. to its performance against well-known competitors as well as its consideration as a data layer in a recent application domain, as is the case of semantic trajectories. For all evaluations we obtained very good results, which suggest that our solution is very promising.

The relational databases were not part of the studies because it is out of the scope. It only fits in the architecture as a replacement for the columnar database in the Dictionary. However, the columnar databases are a better choice since the Dictionary can have *null* values and needs a nested structure, like a column family, which is not available in relational databases.

It is worth mentioning that we would like to compare our middleware with other related works, but we could not find a valid implementation of them. Next chapter is dedicated to the conclusion of this thesis.

6 CONCLUSION

This thesis presents *WA-RDF*, a triplestore that explores the frequent SPARQL query shapes to define the best NoSQL database to store parts of an RDF graph. WA-RDF is proposed in terms of a middleware architecture that rises advances in several aspects of RDF data management, including data fragmentation, partitioning, workload-awareness and RDF-to-NoSQL mapping.

The results of this thesis fully assures the hypothesis presented in Chapter 1. The experimental evaluations demonstrate that the performance of RDF-based applications can benefit of a dynamic workload-aware triplestore. In WA-RDF, multiple NoSQL databases can coexist in a triplestore in a seamless way to the RDF-based application. A middleware connecting an RDF-based application to NoSQL databases is a promising architectural solution for developing a new triplestore, since with a middleware support it is possible to design and add several efficient processing strategies without any impact in the application layer that uses the data. In the case of WA-RDF, it includes workload-aware fragmentation, partitioning and replication; intelligent workload fitted mapping layers; and support of efficient data processing frameworks, like *Apache Spark* and *Apache Kafka*, to provide RDF data manipulation operations, including RDF data update and deletion.

We argue that the main contribution of this thesis to the state-of-the-art is a *new reference architecture* for storage and manipulation of large RDF graphs. As specific contributions, we have new strategies for RDF data fragmentation, mapping, partitioning, caching and storage. A workload-awareness approach is the basis for all of these specific contributions. According to the typical shape of SPARQL queries, it defines RDF fragments and decides the best NoSQL database (document and/or graph) to map and store them in order to improve query performance and provides scalability.

Partitioning is considered when the RDF dataset is bigger than each server capability. In this case, we define a replication boundary to avoid cross-server joins and speed up the query response time. If WA-RDF notices that a fragmentation can benefit of replication, it copies the data from a server to another. It is important to observe the treatment of update and deletion operations by the middleware, which are uncommon features in the researched triplestores. At a first glance, to support these operations should be avoided as they could bring refragmentation of big portions of the graph. However, on delaying the physical data change (the strategy adopted by WA-RDF), we can consider the same replication procedure to update fragments in background.

In order to evaluate the thesis proposal, several rounds of experiments were accomplished. The comparison with *OrientBD*, a multimodel database, demonstrates how a polyglot persistence solution can facilitate the development of applications that deal with RDF data. Moreover, on using the *WatDiv* benchmark, WA-RDF was more

scalable than its competitors, including the faster baseline *S2RDF*, both in face of frequent queries and bigger datasets. Finally, we present the application of the proposed middleware in the domain of trajectory data, with very good results.

6.1 LIMITATIONS OF THE THESIS

The results of this thesis, although representing important advances to the state-of-the-art, have limitations due to the choices made during its development. We consider as main limitations:

- A more deep analysis of SPARQL queries in order to expand the potentiality of the workload-awareness strategy. Only two types of query shapes were explored, and other parts of a SPARQL query command, like projection and grouping, were not considered;
- The lack of experiments to find out the limit of scalability of the middlware;
- The development of an *in-memory* triplestore was limited mainly by time. However, this thesis could provide manners to use a workload-awareness strategy on the top of *Apache Ignite* or other in-memory infrastructures.

6.2 FUTURE WORKS

Future works for this thesis include the discarded research paths, but also includes new opportunities that are been enabled by the advance of related technologies such as new NoSQL databases, the dawn of NewSQL engines, and machine learning techniques available as frameworks integrated with Big Data tools. Some possibilities are given in the following.

- **Consideration of other NoSQL databases**

  There is a number of NoSQL databases that could be used instead of MongoDB, Neo4j, Redis and Cassandra. In special, OrientDB could be evaluated as a substitute for Neo4j.

- **A Pure in-memory triplestore**

  On developing a pure in-memory triplestore on top of Apache Ignite or Apache Spark could reduce the latency in the communication with the main storage. Many of the developed strategies of this thesis could be reused in such a project, like:

  - Workload monitoring, even with low-latency;
  - Graph data modeling. For instance, Apache Spark holds a *GraphX* component that has many graph capabilities. Another example is the *Redis Graph*[1].

[1] https://oss.redislabs.com/redisgraph/

  - Both Apache Spark and Apache Ignite support multiple data models. Another thesis could explore this capability by considering the low-latency delivered by these frameworks.

- **Comparison with other related works**

  There are several other triplestores that were not compared against our solution, and a detailed analysis of them could benefit the evolution of WA-RDF.

- **To better explore SPARQL query patterns**

  The workload of SPARQL queries is not limited to star and chain queries. Moreover, specific domains can have very specific workload patterns, like snow flakes and cyclic onea. A machine learning procedure could be used to discover new patterns among the issued queries. This support could provide better results in terms of performance and reduce the need for data replication.

- **Machine learning techniques on top of the Workload-awareness component**

  The Workload-aware component could benefit of a machine learning algorithm to classify the workload and find out the best way to store RDF data. It could bring benefits as fine-grained precision on the fragment creation.

- **Consideration of NewSQL databases**

  NewSQL databases is a very new category of scalable databases that could be considered in a next version of WA-RDF.

6.3 PUBLICATIONS

As stated before, this thesis had produced or contributed to several papers that were published in conferences or journals related to the Database area by our research group. We present here the paper title and the vehicle name. More details are given in the thesis references:

- Smart Crawler: Using Committee Machines for Web Pages Continuous Classification. XXI Brazilian Symposium on Multimedia and the Web (**WebMedia 2015**). (ZAMBOM SANTANA; SANTOS MELLO; ROISENBERG, 2015);

- Workload-Aware RDF Partitioning and SPARQL Query Caching for Massive RDF Graphs stored in NoSQL Databases. XXXII Brazilian Symposium on Databases (**SBBD 2017**). (SANTANA; SANTOS MELLO, 2017b);

- A Middleware for Polyglot Persistence of RDF Data into Multiple NoSQL Databases. XXX International Conference on Information Reuse and Integration for Data Science (**IRI 2019**). (SANTANA; SANTOS MELLO, 2019a);

- Querying in a Workload-aware Triplestore based on NoSQL Databases. XXX International Conference on Database and Expert Systems Applications (**DEXA 2019**). (SANTANA; SANTOS MELLO, 2019b);
- Workload-awareness in a NoSQL-based Triplestore. XXIII European Conference on Advances in Databases and Information Systems (**ADBIS 2019**). (SANTANA; SANTOS MELLO, 2019c);
- MASTER: A Multiple Aspect View on Trajectories. Transactions in GIS Journal (**TGIS**). (MELLO et al., 2019);
- Persistence of RDF Data into NoSQL: A Survey and a Unified Reference Architecture. Submitted to IEEE Transactions on Knowledge and Data Engineering (**TKDE**).

REFERENCES

ABADI, Daniel J et al. SW-Store: a vertically partitioned DBMS for Semantic Web data management. **The VLDB Journal—The International Journal on Very Large Data Bases**, Springer-Verlag New York, Inc., v. 18, n. 2, p. 385–406, 2009.

ABRAMOVA, Veronika; BERNARDINO, Jorge. NoSQL Databases: MongoDB vs Cassandra. In: PROCEEDINGS of the International C* Conference on Computer Science and Software Engineering. Porto, Portugal: ACM, 2013. (C3S2E '13), p. 14–22. DOI: 10.1145/2494444.2494447. Disponível em: http://doi.acm.org/10.1145/2494444.2494447.

ALUÇ, Güneş; HARTIG, Olaf, et al. Diversified stress testing of RDF data management systems. In: SPRINGER. INTERNATIONAL Semantic Web Conference. [S.l.: s.n.], 2014. p. 197–212.

ALUÇ, Güneş; ÖZSU, M Tamer; DAUDJEE, Khuzaima. Workload matters: Why RDF databases need a new design. **Proceedings of the VLDB Endowment**, VLDB Endowment, v. 7, n. 10, p. 837–840, 2014.

ARANDA-ANDÚJAR, Andrés et al. AMADA: web data repositories in the amazon cloud. In: ACM. PROCEEDINGS of the 21st ACM international conference on Information and knowledge management. [S.l.: s.n.], 2012. p. 2749–2751.

ATRE, Medha; SRINIVASAN, Jagannathan; HENDLER, James. Bitmat: A main-memory bit matrix of RDF triples for conjunctive triple pattern queries. In: CEUR-WS. ORG. PROCEEDINGS of the 2007 International Conference on Posters and Demonstrations-Volume 401. [S.l.: s.n.], 2008. p. 1–2.

BELLINI, Pierfrancesco; NESI, Paolo. Performance assessment of rdf graph databases for smart city services. **Journal of Visual Languages & Computing**, Elsevier, v. 45, p. 24–38, 2018.

BERNERS-LEE, Tim; HENDLER, James; LASSILA, Ora, et al. The semantic web. **Scientific american**, New York, NY, USA: v. 284, n. 5, p. 28–37, 2001.

BOUHALI, Raouf; LAURENT, Anne. Exploiting RDF Open Data Using NoSQL Graph Databases. In: SPRINGER. IFIP International Conference on Artificial Intelligence Applications and Innovations. [S.l.: s.n.], 2015. p. 177–190.

BROEKSTRA, Jeen et al. Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In: SPRINGER. INTERNATIONAL Semantic Web Conference. [S.l.: s.n.], 2002. p. 54–68.

BRUNOZZI, Simone. Big Data and NoSQL with Amazon DynamoDB. In: PROCEEDINGS of the 2012 Workshop on Management of Big Data Systems. San

Jose, California, USA: ACM, 2012. (MBDS '12), p. 41–42. DOI: 10.1145/2378356.2378369. Disponível em: http://doi.acm.org/10.1145/2378356.2378369.

BUGIOTTI, Francesca et al. Invisible Glue: Scalable Self-Tunning Multi-Stores. In: VII Biennial Conference on Innovative Data Systems Research. [S.l.: s.n.], 2015. Disponível em: http://cidrdb.org/cidr2015/Papers/CIDR15%5C_Paper7.pdf.

CALDAROLA, Enrico Giacinto; PICARIELLO, Antonio; CASTELLUCCIA, Daniela. Modern Enterprises in the Bubble: Why Big Data Matters. **SIGSOFT Softw. Eng. Notes**, ACM, New York, NY, USA, v. 40, n. 1, p. 1–4, Feb. 2015. ISSN 0163-5948. DOI: 10.1145/2693208.2693228. Disponível em: http://doi.acm.org/10.1145/2693208.2693228.

CATTELL, Rick. Scalable SQL and NoSQL data stores. **ACM Sigmod Record**, ACM, v. 39, n. 4, p. 12–27, 2011.

CERANS, Karlis et al. Graphical Schema Editing for Stardog OWL/RDF Databases using OWLGrEd/S. In: OWL: Experiences and Directions Workshop. [S.l.: s.n.], 2012.

CHANG, WW; MILLER, B. **AllegroGraph RDF-Triplestore Evaluation**. [S.l.], 2009. Disponível em: https://franz.com/agraph/cresources/white_papers/Adobe-Report_9-09.pdf.

CHAWLA, Tanvi; SINGH, Girdhari; PILLI, Emmanuel S. A shortest path approach to SPARQL chain query optimisation. In: IEEE. 2017 International Conference on Advances in Computing, Communications and Informatics (ICACCI). [S.l.: s.n.], 2017. p. 1778–1778.

CHOI, Hyunsik; SON, Jihoon, et al. SPIDER: a system for scalable, parallel/distributed evaluation of large-scale RDF data. **Proceeding of the 18th ACM conference on Information and knowledge management**, February 2016, p. 2087–2088, 2009. DOI: 10.1145/1645953.1646315. Disponível em: http://portal.acm.org/citation.cfm?id=1646315.

CHOI, Pilsik; JUNG, Jooik; LEE, Kyong-Ho. RDFChain: chain centric storage for scalable join processing of RDF graphs using MapReduce and HBase. In: CEUR-WS. ORG. PROCEEDINGS of the 2013th International Conference on Posters & Demonstrations Track-Volume 1035. [S.l.: s.n.], 2013. p. 249–252.

COSSU, Matteo et al. PRoST: Distributed Execution of SPARQL Queries Using Mixed Partitioning Strategies. In: XXI International Conference on Extending Database Technology. [S.l.: s.n.], 2018. p. 469–472.

CUDRÉ-MAUROUX, Philippe et al. NoSQL databases for RDF: An empirical evaluation. **Lecture Notes in Computer Science (including subseries Lecture**

Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 8219 LNCS, PART 2, p. 310–325, 2013. ISSN 3029743. DOI: 10.1007/978-3-642-41338-4{_}20.

CUESTA, Carlos E; MARTINEZ-PRIETO, Miguel A; FERNÁNDEZ, Javier D. Towards an architecture for managing big semantic data in real-time. In: SPRINGER. EUROPEAN Conference on Software Architecture. [S.l.: s.n.], 2013. p. 45–53.

DEAN, Jeffrey; GHEMAWAT, Sanjay. MapReduce: simplified data processing on large clusters. **Communications of the ACM**, ACM, v. 51, n. 1, p. 107–113, 2008.

DOBBELAERE, Philippe; ESMAILI, Kyumars Sheykh. Kafka Versus RabbitMQ: A Comparative Study of Two Industry Reference Publish/Subscribe Implementations: Industry Paper. In: PROCEEDINGS of the 11th ACM International Conference on Distributed and Event-based Systems. Barcelona, Spain: ACM, 2017. (DEBS '17), p. 227–238. DOI: 10.1145/3093742.3093908. Disponível em: http://doi.acm.org/10.1145/3093742.3093908.

DUGGAN, Jennie et al. The bigdawg polystore system. **ACM Sigmod Record**, ACM, v. 44, n. 2, p. 11–16, 2015.

FILALI, Imen et al. A survey of structured P2P systems for RDF data storage and retrieval. In: TRANSACTIONS on large-scale data-and knowledge-centered systems iii. [S.l.]: Springer, 2011. p. 20–55.

GALLEGO, Mario Arias et al. An empirical study of real-world SPARQL queries. In: USEWOD workshop. [S.l.: s.n.], 2011.

GRAY, Ian et al. Architecture-Awareness for Real-Time Big Data Systems. In: ACM. PROCEEDINGS of the 21st European MPI Users' Group Meeting. [S.l.: s.n.], 2014. p. 151.

GU, Rong; HU, Wei; HUANG, Yihua. Rainbow: A distributed and hierarchical RDF triple store with dynamic scalability. In: IEEE. BIG Data (Big Data), 2014 IEEE International Conference on. [S.l.: s.n.], 2014. p. 561–566.

GUO, Yuanbo; PAN, Zhengxiang; HEFLIN, Jeff. LUBM: A benchmark for OWL knowledge base systems. **Web Semantics: Science, Services and Agents on the World Wide Web**, Elsevier, v. 3, n. 2, p. 158–182, 2005.

GÜTING, Ralf Hartmut; VALDÉS, Fabio; DAMIANI, Maria Luisa. Symbolic trajectories. **ACM Transactions on Spatial Algorithms and Systems**, ACM, v. 1, n. 2, p. 7, 2015.

HARRIS, Steve; LAMB, Nick; SHADBOLT, Nigel. 4store: The design and implementation of a clustered RDF store. In: 5TH International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2009). [S.l.: s.n.], 2009. p. 94–109.

HARTH, Andreas; DECKER, Stefan. Optimized index structures for querying rdf from the web. In: IEEE. WEB Congress, 2005. LA-WEB 2005. Third Latin American. [S.l.: s.n.], 2005. 10–pp.

HOSE, Katja; SCHENKEL, Ralf. WARP: Workload-aware replication and partitioning for RDF. In: IEEE. DATA Engineering Workshops (ICDEW), 2013 IEEE 29th International Conference on. [S.l.: s.n.], 2013. p. 1–6.

HU, Chunming et al. ScalaRDF: a Distributed, Elastic and Scalable In-Memory RDF Triple Store, 2016.

HUANG, Jiewen; ABADI, Daniel J; REN, Kun. Scalable SPARQL querying of large RDF graphs. **Proceedings of the VLDB Endowment**, v. 4, n. 11, p. 1123–1134, 2011.

HUANG, Weiming; RAZA, Syed Amir, et al. Assessment and Benchmarking of Spatially Enabled RDF Stores for the Next Generation of Spatial Data Infrastructure. **ISPRS International Journal of Geo-Information**, Multidisciplinary Digital Publishing Institute, v. 8, n. 7, p. 310, 2019.

KAOUDI, Zoi; MANOLESCU, Ioana. RDF in the clouds: a survey. **The VLDB Journal**, Springer, v. 24, n. 1, p. 67–91, 2015.

KAUR, Karamjit; RANI, Rinkle. Managing data in healthcare information systems: many models, one solution. **Computer**, IEEE, v. 48, n. 3, p. 52–59, 2015.

KHADILKAR, Vaibhav et al. Jena-HBase: A distributed, scalable and efficient RDF triple store. **CEUR Workshop Proceedings**, v. 914, n. 2, p. 85–88, 2012. ISSN 16130073.

KIRYAKOV, Atanas et al. The Features of Bigowlim that Enabled the BBCs World Cup Website. In: WORKSHOP on Semantic Data Management. [S.l.: s.n.], 2010.

KOBASHI, Hiromichi et al. Cerise: an RDF store with adaptive data reallocation. In: ACM. PROCEEDINGS of the 13th Workshop on Adaptive and Reflective Middleware. [S.l.: s.n.], 2014. p. 1.

KOLOMIČENKO, Vojtěch; SVOBODA, Martin; MLÝNKOVÁ, Irena Holubová. Experimental Comparison of Graph Databases. In: PROCEEDINGS of International Conference on Information Integration and Web-based Applications & Services. Vienna, Austria: ACM, 2013. (IIWAS '13), 115:115–115:124. DOI: 10.1145/2539150.2539155. Disponível em: http://doi.acm.org/10.1145/2539150.2539155.

KUHLENKAMP, Jörn; KLEMS, Markus; RÖSS, Oliver. Benchmarking Scalability and Elasticity of Distributed Database Systems. **Proc. VLDB Endow.**, VLDB Endowment,

v. 7, n. 12, p. 1219–1230, Aug. 2014. ISSN 2150-8097. Disponível em: http://dl.acm.org/citation.cfm?id=2732977.2732995.

LADWIG, Günter; HARTH, Andreas. CumulusRDF: linked data management on nested key-value stores. In: THE 7th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS 2011). [S.l.: s.n.], 2011. p. 30.

LERNER, Reuven M. At the Forge: CouchDB. **Linux J.**, Belltown Media, Houston, TX, v. 2010, n. 195, July 2010. ISSN 1075-3583. Disponível em: http://dl.acm.org/citation.cfm?id=1883478.1883486.

LIBKIN, Leonid; REUTTER, Juan; VRGOČ, Domagoj. Trial for RDF: adapting graph query languages for RDF data. In: ACM. PROCEEDINGS of the 32nd ACM SIGMOD-SIGACT-SIGAI symposium on Principles of database systems. [S.l.: s.n.], 2013. p. 201–212.

LIU, Xiufeng; IFTIKHAR, Nadeem; XIE, Xike. Survey of Real-time Processing Systems for Big Data. In: PROCEEDINGS of the 18th International Database Engineering & Applications Symposium. Porto, Portugal: ACM, 2014. (IDEAS '14), p. 356–361. DOI: 10.1145/2628194.2628251. Disponível em: http://doi.acm.org/10.1145/2628194.2628251.

LUO, Yongming et al. Storing and indexing massive RDF datasets. In: SEMANTIC search over the web. [S.l.]: Springer, 2012. p. 31–60.

MA, Zongmin; CAPRETZ, Miriam AM; YAN, Li. Storing massive Resource Description Framework (RDF) data: a survey. **The Knowledge Engineering Review**, Cambridge University Press, v. 31, n. 4, p. 391–413, 2016.

MAHMOUDINASAB, Hooran; SAKR, Sherif. AdaptRDF: adaptive storage management for RDF databases. **International Journal of Web Information Systems**, Emerald Group Publishing Limited, v. 8, n. 2, p. 234–250, 2012.

MAMMO, Mulugeta; BANSAL, Srividya K. Presto-RDF: SPARQL Querying over Big RDF Data. In: SPRINGER. AUSTRALASIAN Database Conference. [S.l.: s.n.], 2015. p. 281–293.

MARZ, Nathan; WARREN, James. **Big Data: Principles and best practices of scalable realtime data systems**. [S.l.]: Manning Publications Co., 2015.

MCGLOTHLIN, James P; KHAN, Latifur R. RDFKB: efficient support for RDF inference queries and knowledge management. In: ACM. PROCEEDINGS of the 2009 International Database Engineering & Applications Symposium. [S.l.: s.n.], 2009. p. 259–266.

MCGLOTHLIN, JAMES; KHAN, L. RDFJoin: A scalable data model for persistence and efficient querying of RDF datasets. **Database**, 2009.

MELLO, Ronaldo dos Santos et al. MASTER: A multiple aspect view on trajectories. **Transactions in GIS**, 2019. DOI: 10.1111/tgis.12526. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/tgis.12526. Disponível em: https://onlinelibrary.wiley.com/doi/abs/10.1111/tgis.12526.

MICHEL, Franck; FARON-ZUCKER, Catherine; MONTAGNAT, Johan. A generic mapping-based query translation from SPARQL to various target database query languages. In: 12TH International Conference on Web Information Systems and Technologies (WEBIST'16). [S.l.: s.n.], 2016.

MICHEL, Franck; FARON-ZUCKER, Catherine; MONTAGNAT, Johan. **Mapping-based SPARQL access to a MongoDB database**. 2016. PhD thesis – CNRS.

MULAY, Kunal; KUMAR, P Sreenivasa. SPOVC: a scalable RDF store using horizontal partitioning and column oriented DBMS. In: ACM. PROCEEDINGS of the 4th International Workshop on Semantic Web Information Management. [S.l.: s.n.], 2012. p. 8.

NEUMANN, Thomas; WEIKUM, Gerhard. The RDF-3X engine for scalable management of RDF data. **The VLDB Journal—The International Journal on Very Large Data Bases**, Springer-Verlag New York, Inc., v. 19, n. 1, p. 91–113, 2010.

NIEMEYER, Gustavo. **Geohash**. [S.l.: s.n.], 2008. Disponível em: http://geohash.org/.

PAPAILIOU, Nikolaos; DIMITRIOS TSOUMAKOS, et al. Graph-Aware , Workload-Adaptive SPARQL Query Caching. **Sigmod**, p. 1777–1792, 2015. ISSN 7308078. DOI: 10.1145/2723372.2723714.

PAPAILIOU, Nikolaos; KONSTANTINOU, Ioannis; TSOUMAKOS, Dimitrios; KARRAS, Panagiotis, et al. H 2 RDF+: High-performance distributed joins over large-scale RDF graphs. In: IEEE. BIG Data, 2013 IEEE International Conference on. [S.l.: s.n.], 2013. p. 255–263.

PAPAILIOU, Nikolaos; KONSTANTINOU, Ioannis; TSOUMAKOS, Dimitrios; KOZIRIS, Nectarios. H2RDF: adaptive query processing on RDF data in the cloud. In: ACM. PROCEEDINGS of the 21st International Conference on World Wide Web. [S.l.: s.n.], 2012. p. 397–400.

PAVLO, Andrew; ASLETT, Matthew. What's really new with NewSQL? **ACM Sigmod Record**, ACM, v. 45, n. 2, p. 45–55, 2016.

PERERA, Srinath; SUHOTHAYAN, Sriskandarajah. Solution patterns for realtime streaming analytics. In: ACM. PROCEEDINGS of the 9th ACM International Conference on Distributed Event-Based Systems. [S.l.: s.n.], 2015. p. 247–255.

PHAM, M. Self-organizing structured RDF in MonetDB. In: IEEE. DATA Engineering Workshops (ICDEW), 2013 IEEE 29th International Conference on. [S.l.: s.n.], 2013. p. 310–313.

PHAM, Minh-Duc; BONCZ, Peter. Exploiting Emergent Schemas to make RDF systems more efficient. In: SPRINGER. INTERNATIONAL Semantic Web Conference. [S.l.: s.n.], 2016. p. 463–479.

PUNNOOSE, Roshan; CRAINICEANU, Adina; RAPP, David. Rya: a scalable RDF triple store for the clouds. In: ACM. PROCEEDINGS of the 1st International Workshop on Cloud Intelligence. [S.l.: s.n.], 2012. p. 4.

RAMAKRISHNAN, Raghu; GEHRKE, Johannes. **Database management systems**. [S.l.]: Osborne/McGraw-Hill, 2000.

REN, Xiangnan; CURÉ, Olivier. Strider: A hybrid adaptive distributed RDF stream processing engine. In: SPRINGER. INTERNATIONAL Semantic Web Conference. [S.l.: s.n.], 2017. p. 559–576.

SADALAGE, Pramod J; FOWLER, Martin. **NoSQL distilled: a brief guide to the emerging world of polyglot persistence**. [S.l.]: Pearson Education, 2012.

SAHOO, Satya S et al. A survey of current approaches for mapping of relational databases to RDF. **W3C RDB2RDF Incubator Group Report**, p. 113–130, 2009.

SAKR, Sherif; AL-NAYMAT, Ghazi. Relational processing of RDF queries: a survey. **ACM SIGMOD Record**, ACM, v. 38, n. 4, p. 23–28, 2010.

SANTANA, Eduardo Felipe Zambom; CHAVES, Ana Paula, et al. Software platforms for smart cities: Concepts, requirements, challenges, and a unified reference architecture. **arXiv preprint arXiv:1609.08089**, 2016.

SANTANA, Luiz Henrique Zambom; SANTOS MELLO, Ronaldo dos. A Middleware for Polyglot Persistence of RDF Data into NoSQL Databases. In: 20TH International Conference on Information Reuse and Integration for Data Science (IRI). [S.l.]: IEEE, 2019. p. 237–244. DOI: 10.1109/IRI.2019.00046.

SANTANA, Luiz Henrique Zambom; SANTOS MELLO, Ronaldo dos. Querying in a Workload-aware Triplestore based on NoSQL Databases. In: 30TH International Conference on Database and Expert Systems Applications (DEXA). [S.l.: s.n.], 2019.

SANTANA, Luiz Henrique Zambom; SANTOS MELLO, Ronaldo dos. Workload-Abellini2018performanceware RDF Partitioning and SPARQL Query Caching for Massive RDF Graphs stored in NoSQL Databases. In: XXXII Simpósio Brasileiro de Banco de Dados. [S.l.: s.n.], 2017. p. 184–195.

SANTANA, Luiz Henrique Zambom; SANTOS MELLO, Ronaldo dos. Workload-Aware RDF Partitioning and SPARQL Query Caching for Massive RDF Graphs stored in NoSQL Databases. In: XXXII Simpósio Brasileiro de Banco de Dados. [S.l.: s.n.], 2017. p. 184–195. Disponível em: http://sbbd.org.br/2017/wp-content/uploads/sites/3/2018/02/p184-195.pdf.

SANTANA, Luiz Henrique Zambom; SANTOS MELLO, Ronaldo dos. Workload-awareness in a NoSQL-based Triplestore. In: 23TH International Conference on Advances in Databases and Information Systems (ADBIS). [S.l.: s.n.], 2019.

SCHÄTZLE, Alexander; PRZYJACIEL-ZABLOCKI, Martin; BERBERICH, Thorsten, et al. S2X: graph-parallel querying of RDF with graphX. In: SPRINGER. VLDB Workshop on Big Graphs Online Querying. [S.l.: s.n.], 2015. p. 155–168.

SCHÄTZLE, Alexander; PRZYJACIEL-ZABLOCKI, Martin; DORNER, Christopher, et al. Cascading map-side joins over HBase for scalable join processing. **SSWS+ HPCSW**, v. 59, 2012.

SCHÄTZLE, Alexander; PRZYJACIEL-ZABLOCKI, Martin; NEU, Antony, et al. Sempala: interactive SPARQL query processing on hadoop. In: SPRINGER. INTERNATIONAL Semantic Web Conference. [S.l.: s.n.], 2014. p. 164–179.

SCHÄTZLE, Alexander; PRZYJACIEL-ZABLOCKI, Martin; SKILEVIC, Simon, et al. S2RDF: RDF querying with SPARQL on spark. **Proceedings of the VLDB Endowment**, VLDB Endowment, v. 9, n. 10, p. 804–815, 2016.

SHAO, Bin; WANG, Haixun; LI, Yatao. The trinity graph engine. **Microsoft Research**, p. 54, 2012.

SHARP, John et al. Data access for highly-scalable solutions: Using SQL, NoSQL, and polyglot persistence. Microsoft patterns & practices, 2013.

SRIVASTAVA, Kriti; SHEKOKAR, Narendra. A Polyglot Persistence approach for E-Commerce business model. In: IEEE. 2016 International Conference on Information Science (ICIS). [S.l.: s.n.], 2016. p. 7–11.

STEIN, Raffael; ZACHARIAS, Valentin. Rdf on cloud number nine. In: 4TH Workshop on New Forms of Reasoning for the Semantic Web: Scalable and Dynamic. [S.l.: s.n.], 2010. p. 11–23.

STOREY, Veda C; SONG, Il-Yeol. Big data technologies and management: What conceptual modeling can do. **Data & Knowledge Engineering**, Elsevier, 2017.

SUMBALY, Roshan et al. Serving Large-scale Batch Computed Data with Project Voldemort. In: PROCEEDINGS of the 10th USENIX Conference on File and Storage Technologies. San Jose, CA: USENIX Association, 2012. (FAST'12), p. 18–18. Disponível em: `http://dl.acm.org/citation.cfm?id=2208461.2208479`.

TOMASZUK, Dominik. Document-oriented triplestore based on RDF/JSON. **Studies in Logic, Grammar and Rhetoric,(22 (35))**, p. 130, 2010.

TOSHNIWAL, Ankit et al. Storm@ twitter. In: ACM. PROCEEDINGS of the 2014 ACM SIGMOD international conference on Management of data. [S.l.: s.n.], 2014. p. 147–156.

VALDÉS, Fabio; DAMIANI, Maria Luisa; GÜTING, Ralf Hartmut. Symbolic Trajectories in SECONDO: Pattern Matching and Rewriting. In: MENG, Weiyi et al. (Eds.). **Database Systems for Advanced Applications**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. p. 450–453.

WEISS, Cathrin; KARRAS, Panagiotis; BERNSTEIN, Abraham. Hexastore: sextuple indexing for semantic web data management. **Proceedings of the VLDB Endowment**, VLDB Endowment, v. 1, n. 1, p. 1008–1019, 2008.

WIESE, Lena. Polyglot Database Architectures=Polyglot Challenges. In: PROCEEDINGS of the Lernen, Wissen, Adaption (LWA). [S.l.: s.n.], 2015. p. 422–426.

ZAHARIA, Matei. **An architecture for fast and general data processing on large clusters**. [S.l.]: Morgan & Claypool, 2016.

ZAMBOM SANTANA, Luiz Henrique; SANTOS MELLO, Ronaldo dos; ROISENBERG, Mauro. Smart Crawler: Using Committee Machines for Web Pages Continuous Classification. In: PROCEEDINGS of Brazilian Symposium on Multimedia and the Web. Manaus, Brazil: ACM, 2015. (WebMedia '15). Disponível em: `http://dx.doi.org/10.1145/2820426.2820437`.

ZENG, Kai et al. A distributed graph engine for web scale RDF data. In: VLDB ENDOWMENT, 4. PROCEEDINGS of the VLDB Endowment. [S.l.: s.n.], 2013. p. 265–276.

APPENDIX A – RDF TRIPLES FOR THE MAPPING EXPERIMENTS

A.1 LEHIGH UNIVERSITY BENCHMARK(LUBM)

Insert a university (LUBM):

University0.edu rdf:type ub:University

t1 - [http://University0.edu, http://www.w3.org/2001/vcard-rdf/3.0FN, "University0.edu"]

Insert a department for the university (LUBM):

Department0.University0.edu rdf:type ub:Department

Department0.University0.edu ub:subOrganizationOf University0.edu

t2 - [http://Department0.University0.edu, http://www.w3.org/2001/vcard-rdf/3.0FN, "Department0.University0.edu"]

t3 - [http://Department0.University0.edu, http://www.w3.org/2001/vcard-rdf/3.0subOrganizationOf, "http://University0.edu"]

Insert a professor for the department (LUBM):

Professor0 ub:worksFor Department0.University0.edu

t4 - [http://Professor0, http://www.w3.org/2001/vcard-rdf/3.0FN, "Professor0"]

t5 - [http://Professor0, http://www.w3.org/2001/vcard-rdf/3.0worksFor, "http://Department0.University0.edu"]

APPENDIX B – CONFIGURATION APPENDIX

The instructions to install a WA-RDF cluster are given in the following. For each server:

1. Download the code from https://github.com/lhzsantana/wa-rdf;
2. In the root of project run *mvn compile*, in order to download the dependencies;
3. Change the configurations in .properties file, as explained below;
4. Run *nohup ./gradlew -Pprod -Pswagger*.

The following properties must be filled:

- Cassandra URL. Servers using the same Cassadra will share the dictionary;
- MongoDB URL;
- Neo4j URL;
- Redis URL. Servers using the same Cassadra will share the pending operations and cache;
- Spark URL;
- Kafka URL.